Understanding the backward pass through Batch Normalization Layer

Posted on February 12, 2016

At the moment there is a wonderful course running at Standford University, called CS231n - Convolutional Neural Networks for Visual Recognition (http://cs231n.stanford.edu/), held by Andrej Karpathy, Justin Johnson and Fei-Fei Li. Fortunately all the course material (http://cs231n.stanford.edu/syllabus.html) is provided for free and all the lectures are recorded and uploaded on Youtube (https://www.youtube.com/playlist?list=PLkt2uSq6rBVctENoVBg1TpCC7OQi31AlC). 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 (http://arxiv.org/abs/1502.03167), 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: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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

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 to much time on explaining Batch Normalization. If you want to learn more about it, the paper (http://arxiv.org/abs/1502.03167) is very well written and here

(https://youtu.be/gYpoJMlgyXA?list=PLkt2uSq6rBVctENoVBg1TpCC7OQi31AlC&t=3078) 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 trainingsset.

Backpropagation

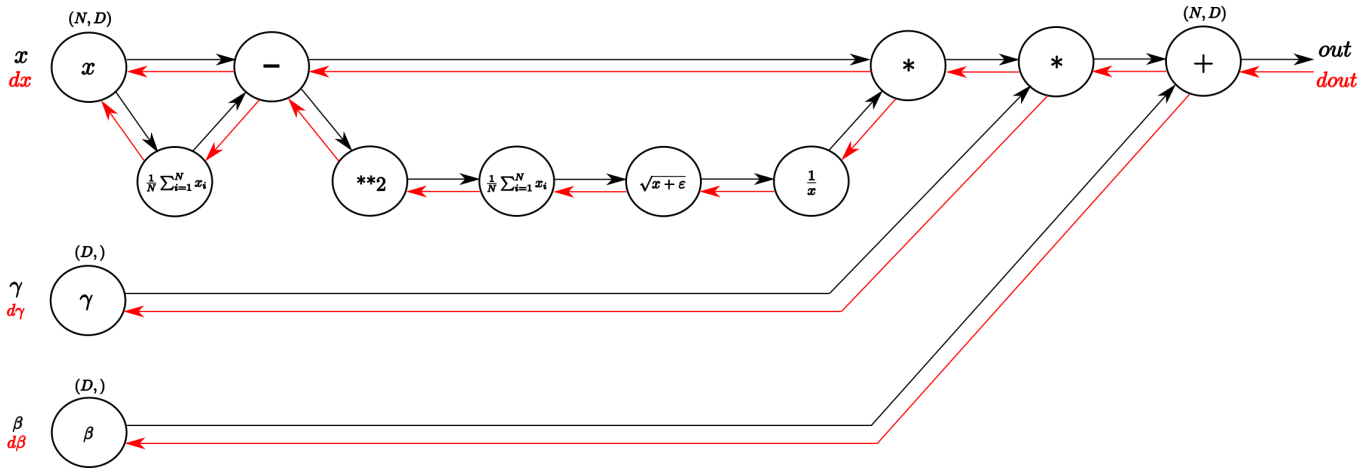
In this blog post I don't want to give a lecture in Backpropagation and Stochastic Gradient Descent (SGD). For now I will assume that whoever will read this post, has some basic understanding of these principles. For the rest, let me quote Wiki:

> *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 gamma and beta as vectors. From right to left, following the red arrows flows the backward pass which distributes the gradient from above layer to gamma and beta 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_normalized`. The last two blobs on the right perform the squashing by multiplying with the input `gamma` and finally adding `beta`. Et voilà, we have our Batch-Normalized output.

A vanilla implementation of the forwardpass might look like this:

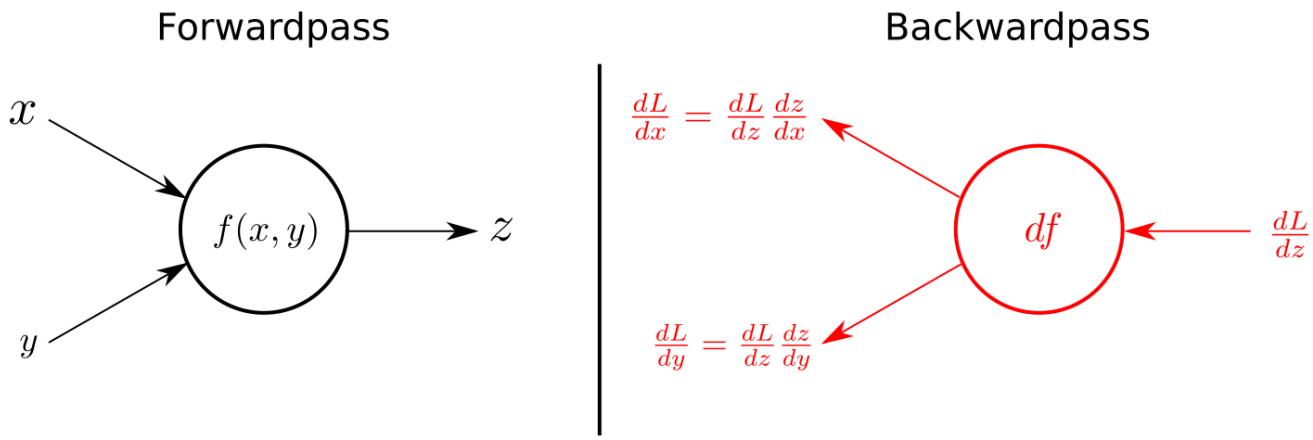
```
def batchnorm_forward(x, gamma, beta, eps):

  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 sqrtwar
  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 (https://en.wikipedia.org/wiki/Chain_rule) is. As a little refresh follows one figure that exemplifies the use of chain rule for the backward pass in computational graphs.



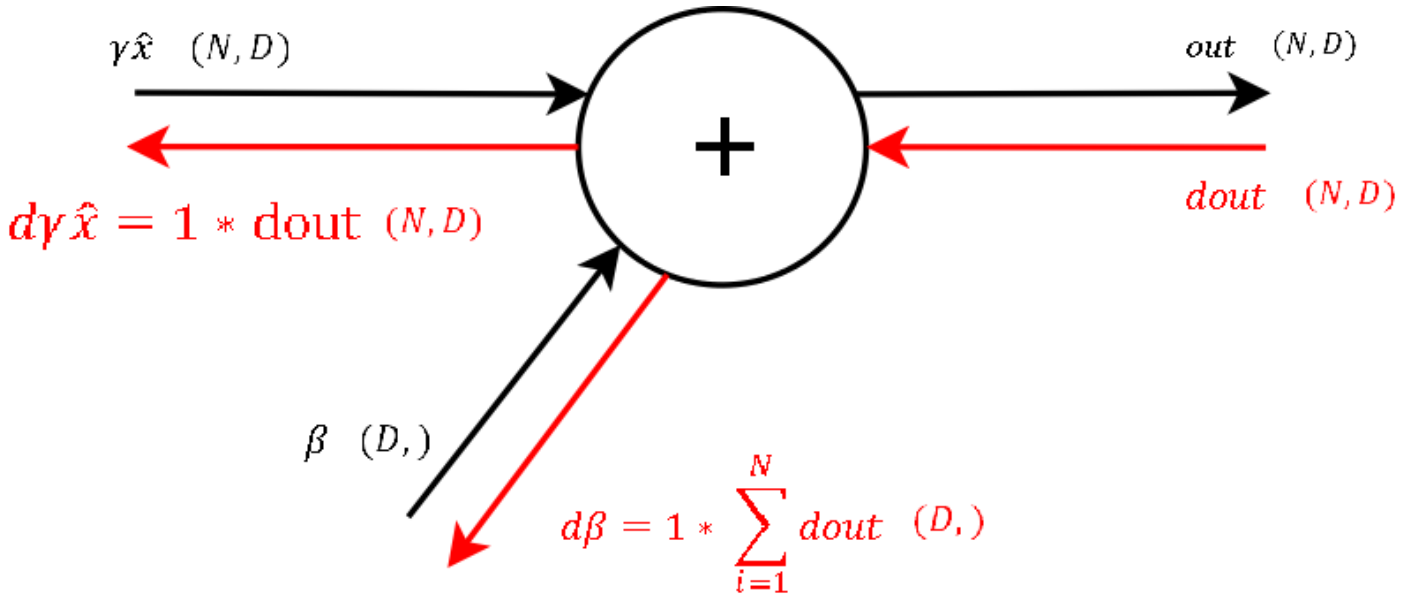
The forwardpass on the left in calculates ž as a function f(x,y)`using the input variables x̀`and ỳ (This could literally be any function, examples are shown in the BatchNorm-Graph above). The right side of the figures shows the backwardpass. Receiving ∂L/dż, the gradient of the loss function with respect to ž from above, the gradients of x̀`and ỳ 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 derivatives of some basic functions are listed in the course material (http://cs231n.github.io/optimization-2/#sigmoid). 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 know 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.

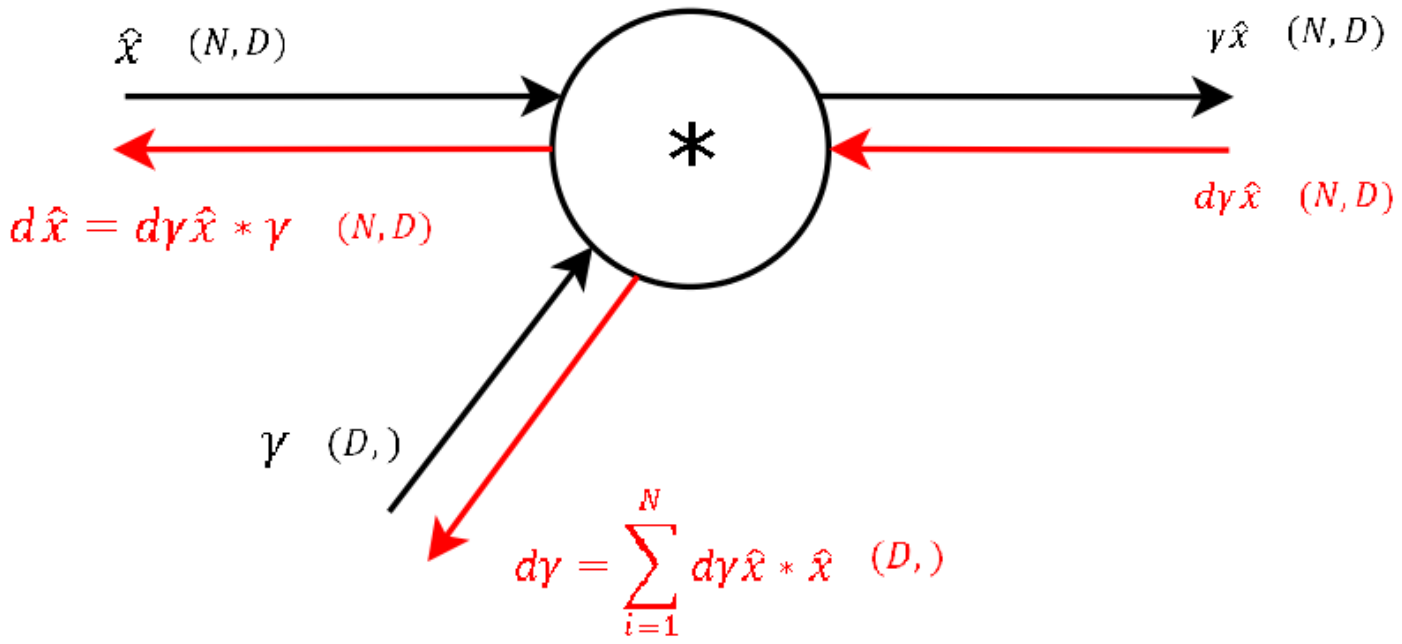
Step 9



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

Recall that the derivative 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`. For our final loss evaluation, we sum the gradient of all samples in the batch. Through this operation, we also get a vector of gradients with the correct shape for `beta`. So after the first step of backpropagation we already got the gradient for one learnable parameter: `beta`

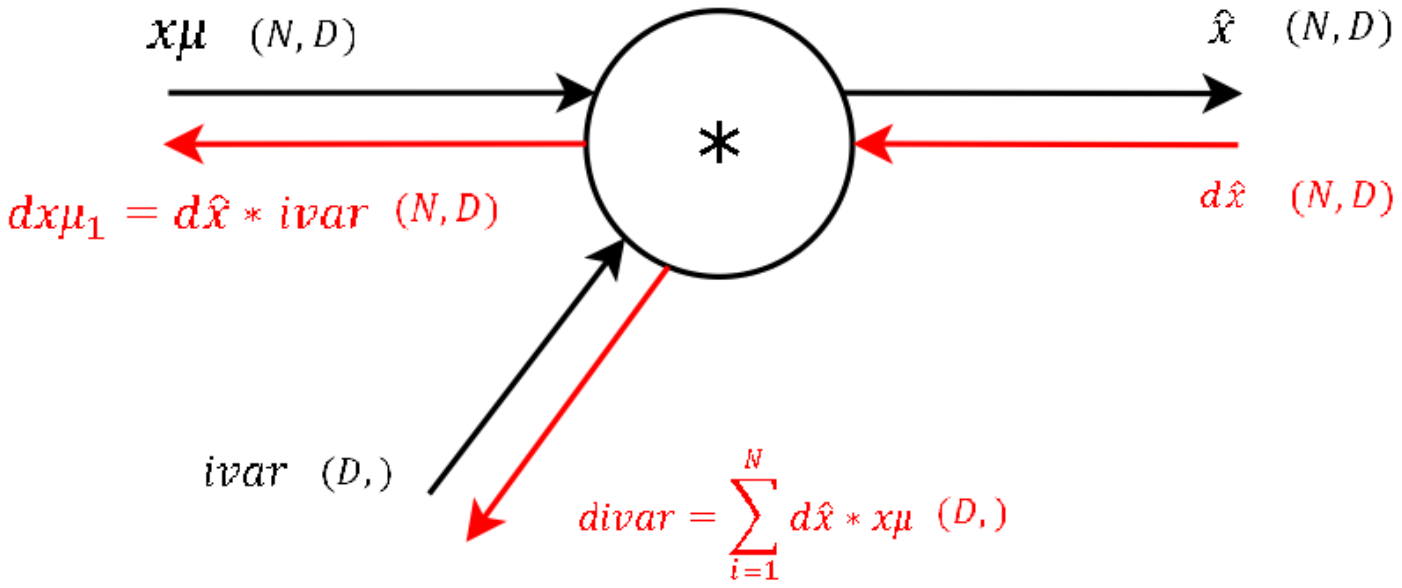
Step 8



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 derivative 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 aboves 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`. 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.

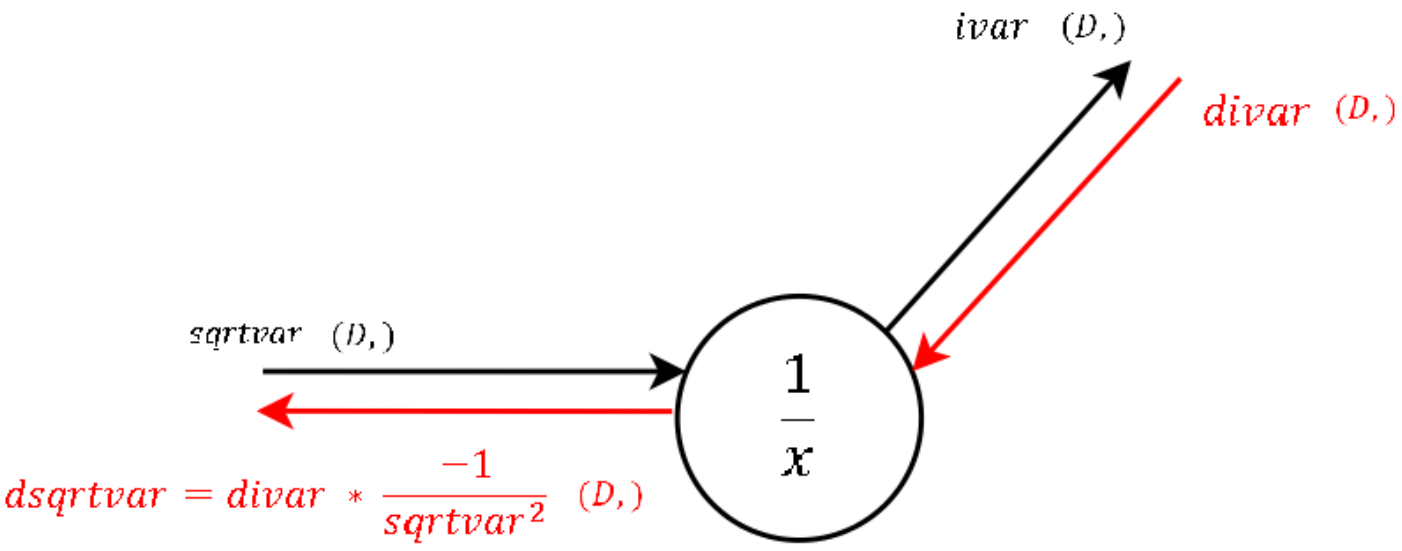
Step 7



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 lets not waste much time and continue. The two needed variables `xmu` and `ivar` for this step are also stored `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!)

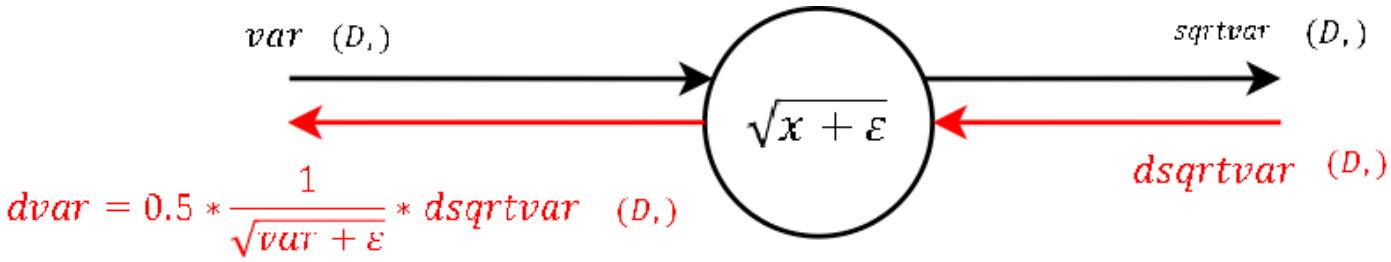
Step 6



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. `sqrtvar` is also one of the variables passed in `cache`.

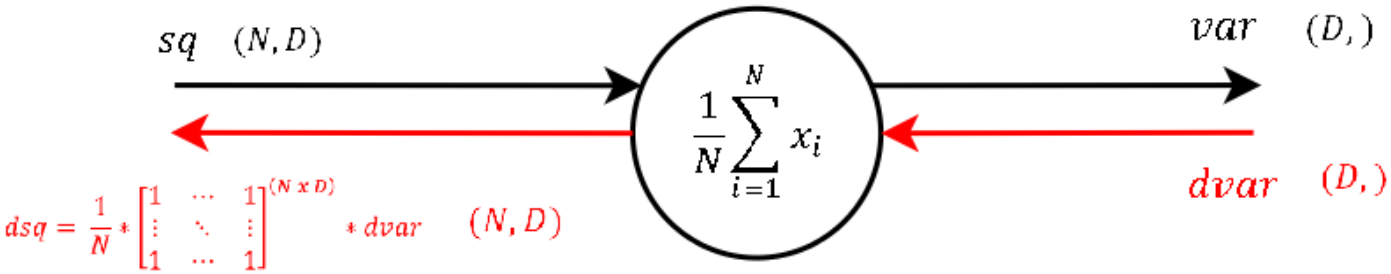
Step 5



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

The calculation of the derivative 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!

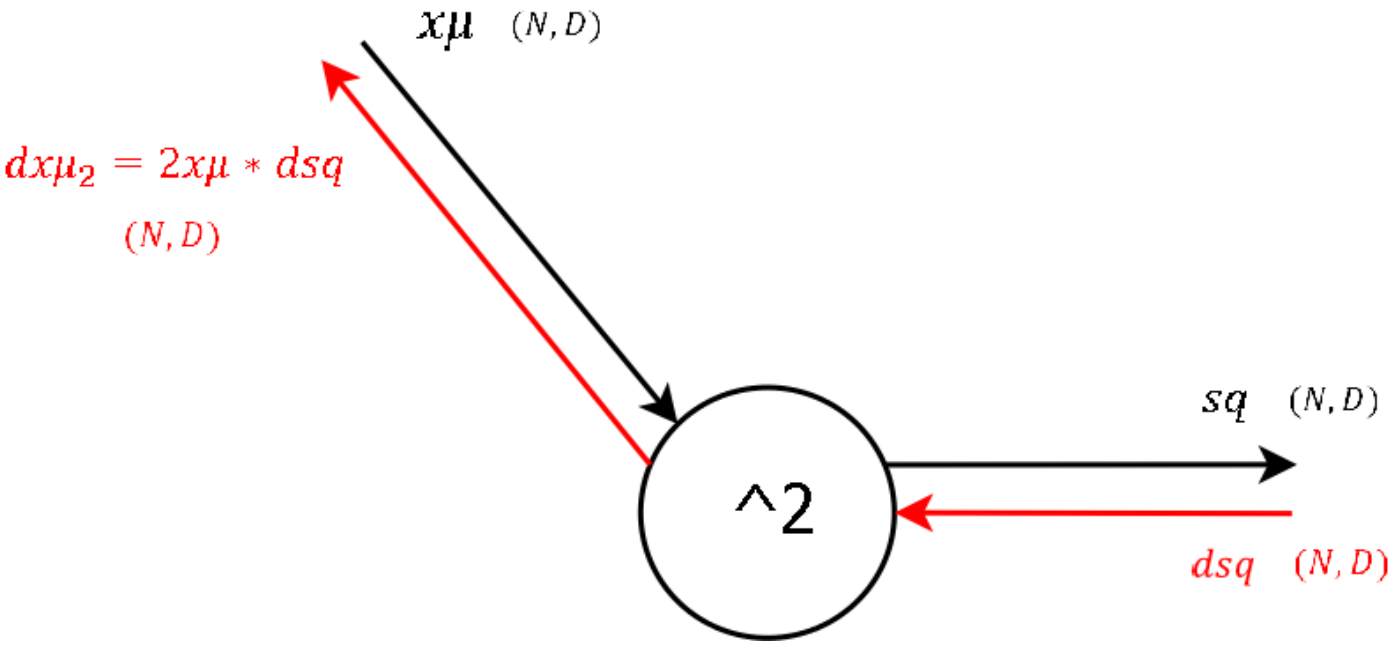
Step 4



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

The calculation of the derivative 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.

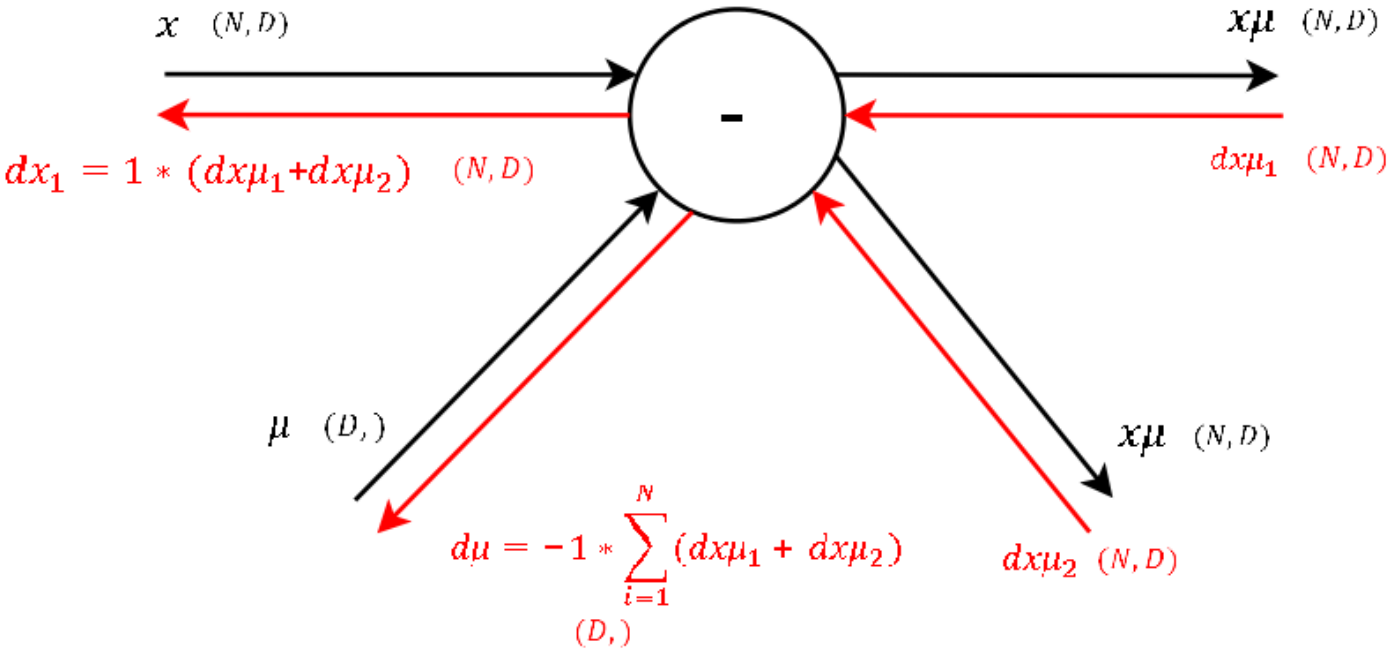
Step 3



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 regarding the derivative of the local gradient.

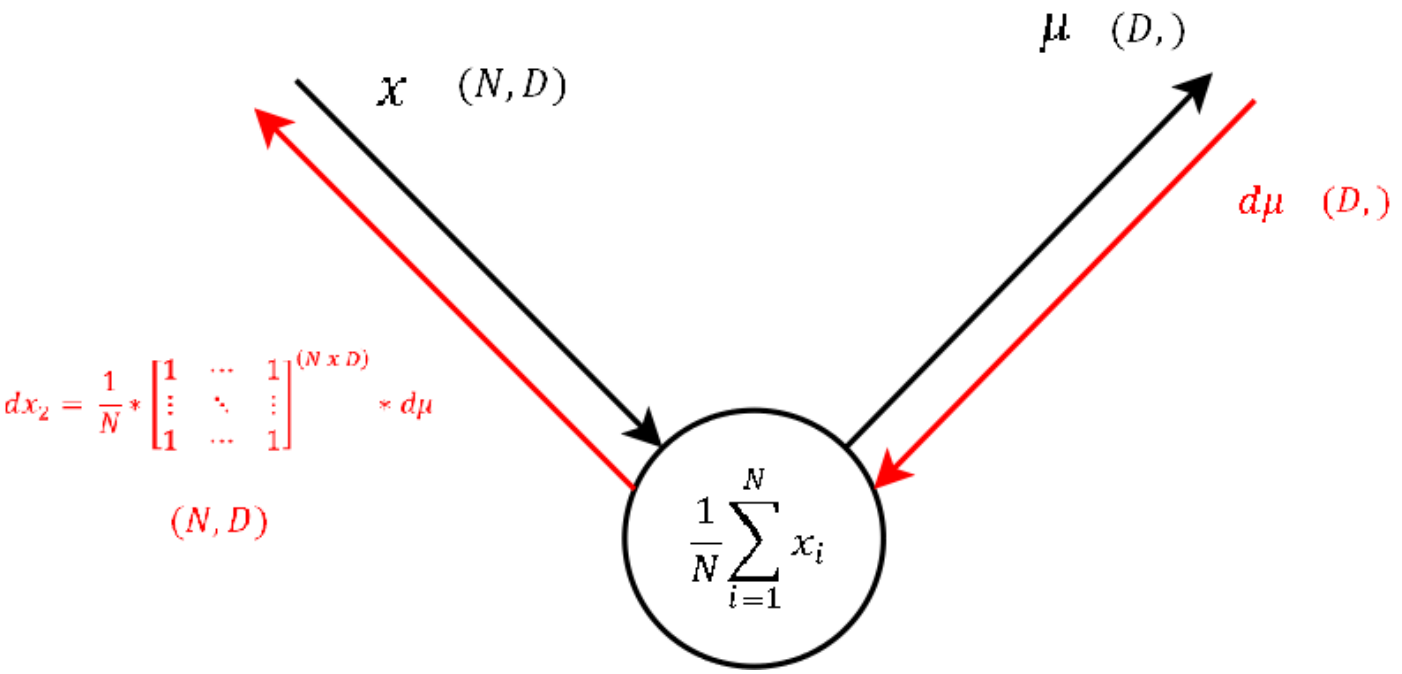
Step 2



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 ȵ for n in [1...N]`during the forward pass.

Okay lets see. One of the definitions of backprogatation 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 `mu` we have to sum up the gradients over the dimension `N` (as we did before for `gamma` and `beta`).

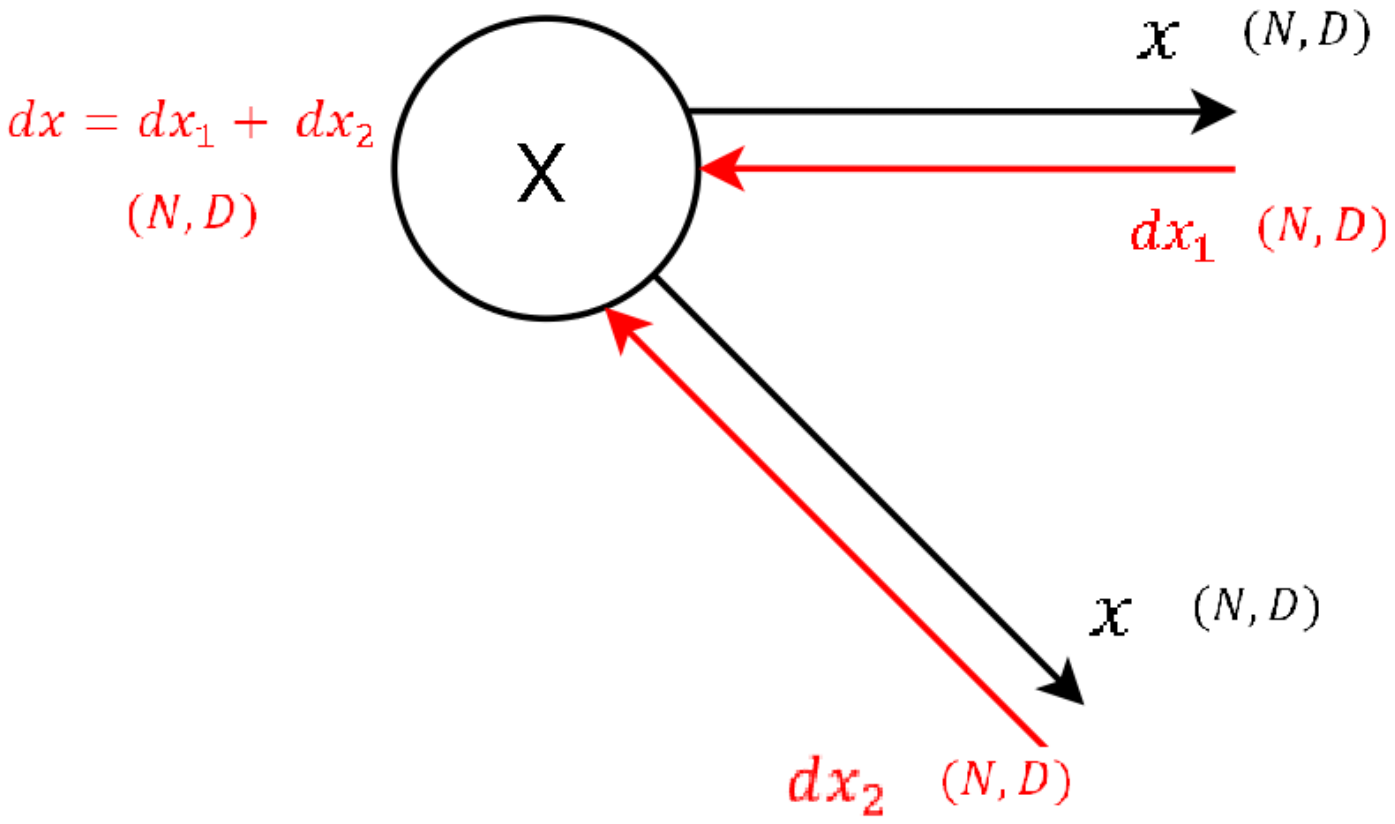
Step 1



The function of this node is exactly the same as of step 4. Only that during the forward pass the input was x̀ - the input to the BatchNorm-Layer and the output here is m̀u, 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 implemantation 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)
  dgammax = dout #not necessary, but more understandable

  #step8
  dgamma = np.sum(dgammax*xhat, axis=0)
  dxhat = dgammax * 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 (http://cthorey.github.io./backpropagation/) 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 to 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!)

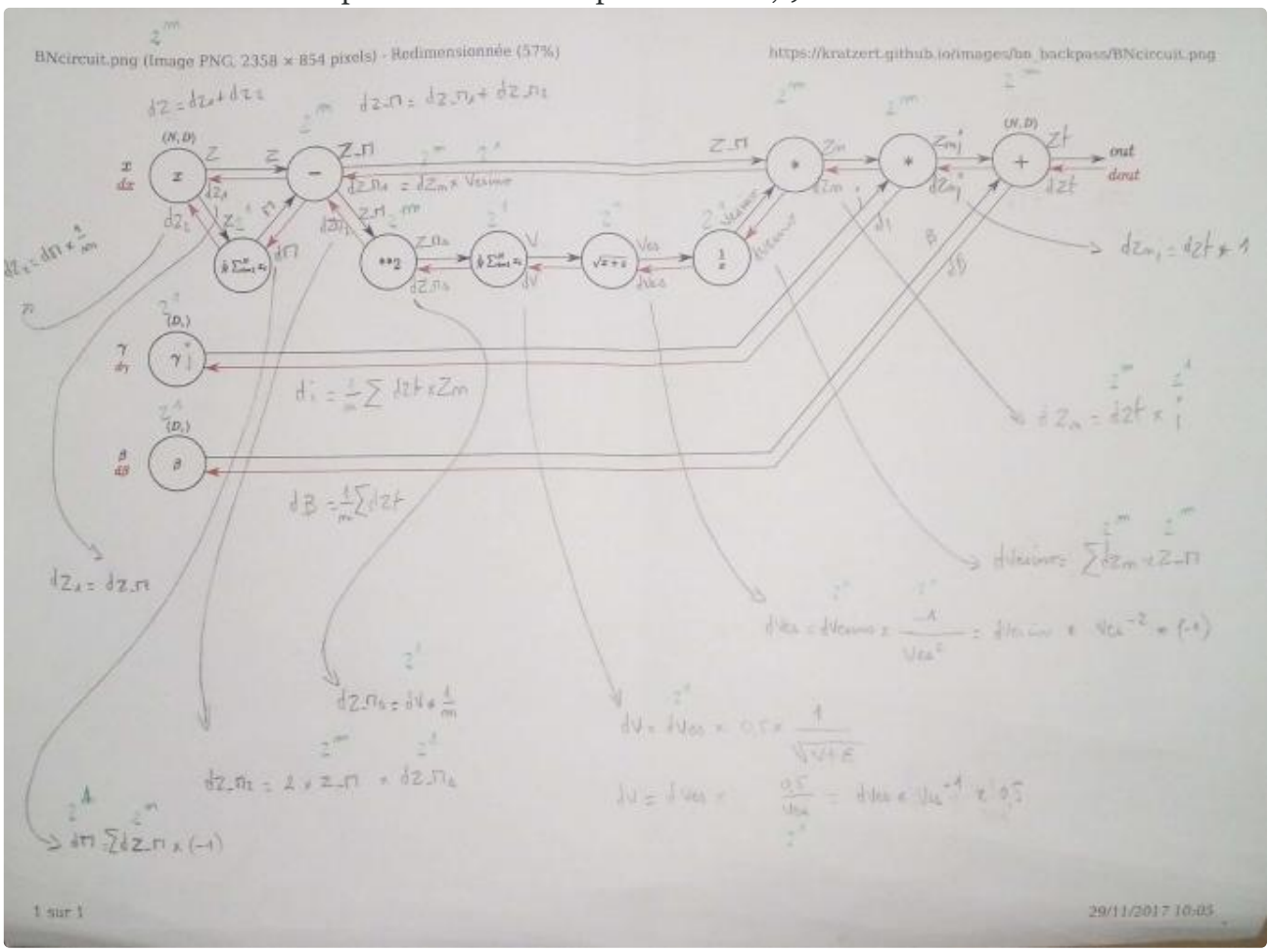
And as we have already passed the deadline for the second assignment, I might upload my code during the next days on github.

NEXT POST → (/2017/02/24/FINETUNING-ALEXNET-WITH-TENSORFLOW.HTML)

Name

aboul • 2 years ago

Thanks ! the flowchart help me so much to implement in R ;-)



5 ^ | v • Reply • Share ›

fkratzert Mod ➚ aboul • 2 years ago

Looks like my sketches, when I first sat down to derive the graph for my self. Good work!

^ | v • Reply • Share ›

Patrick Langechuan LIU ➚ fkratzert • 2 years ago • edited

Hi Frederik,

Thanks for the great explanation in this post, but I had some trouble understanding the backprop in step 9, specifically why an np.sum() is used rather than np.mean() to calculate dbeta. My intuition is, during the addition step with numpy broadcasting to every row, any error in beta is duplicated n_row times, and thus dbeta should be the average across rows of dout.

From what I can see in aboul's handwriting in the flow chart above, he used average rather than sum. I was wondering if you could shed some light onto this issue. Thanks!

Patrick

==========
Edit: Never mind. I found the explanation on the link you gave at the end of the post http://cthorey.github.io/ba... in the section "We can therefore chain the gradient of the loss with respect to the input h_ij by the gradient of the loss with respect to ALL the outputs y_kl which reads ...". This is related to the notion of "total derivative" https://en.wikipedia.org/wi....

In addition, I agree with John McIain's explanation above that a more appropriate explanation about why the broadcast items are added up but are not averaged is that the final cost is the summed across different rows,

1 ^ | v • Reply • Share ›

vijendra rana • 3 years ago

Thanks for sharing :)

4 ^ | v • Reply • Share ›

fkratzert Mod ➚ vijendra rana • 3 years ago

you're welcome ;)

1 ^ | v • Reply • Share ›

Rahul Devanarayanan • a year ago

Hi Frederik,

I'm having trouble understanding your derivation for dBeta and dGamma. Why do you take the sum of the gradient over all the backprop inputs for that node? Shouldn't it be the average of all the backprop inputs for that node?

To my knowledge, that's how dTheta terms are calculated in other layers of the neural network as well.

2 ^ | v • Reply • Share ›

JohnMclain • 2 years ago • edited

hi ,
I believe there is something wrong with your explanation about why the broadcast terms are added up. For example beta are summed in columns.

"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)."

I think the reason is that in the end loss different training examples are summed for the final loss evaluation. Not the reason you give.

2 ^ | v • Reply • Share ›

fkratzert Mod ➚ JohnMclain • 2 years ago

Sorry, John, for replying so late. I somehow haven't seen your post but was now directed

Sorry Jonn for replying so late, I somehow haven t seen your post but was now directed through another comment to your comment again. And well I think you are totally right (I adapted the passages in step 9 and 8 of the backprop, you might take a look and give me your opinion on the updated version).

1 ^ | v • Reply • Share ›

Harvey Qiu ➚ fkratzert • 2 years ago • edited

Hi, I have been struggling with getting this part for a while. I ended up writing out the full expression of dL/d\beta (or any broadcasting part) to arrive at the same result (quite some work for one step, phew).

I think the reason for adding up rows is the multivariate-calculus rule which says "gradients flow into different branches of the graph should be summed". Even if the loss function at the end is not a sum, we still should sum the gradient from different branches that flow into one variable.

As a concrete example, in step 9, a term of dL/d\beta should be:

$$\frac{dL}{d\beta_i} = \sum_{m,n} \frac{\partial L}{\partial \tilde{z}_{m,n}} \frac{\partial \tilde{z}_{m,n}}{\partial \beta_i}$$

where z_tilde is the output in your notation. Obviously only the i-th column of the output is involved in the forward path so this derivative is the sum of the i-th column of dout.

However, I do have a question. I found myself in need of doing this full-out derivation basically every time which is quite non-trivial. What's your thinking track that leads to the expression without all the element-wise writting-out?

^ | v • Reply • Share ›

fkratzert Mod ➚ Harvey Qiu • 2 years ago

Hey Harvey,
sorry for responding so late. I think the reason you give is absolutely correct but I'm not sure if I get your last question. You mean why I started to disassemble everything in small steps? I saw it in one of the lectures Andrej Kaparthy gave during the CS231n class I linked at the top. He did the same to explain the backward pass an gradient calculation and I think it is a nice way to make things easier ;)

^ | v • Reply • Share ›

Jorge • 2 years ago

Thanks fkratzert for your effort !!
I verified your code passes Andrew NG's Gradient Checking.
Cheers!!

1 ^ | v • Reply • Share ›

fkratzert Mod → Jorge • 2 years ago

Thanks man, good to hear ;)

1 ^ | v • Reply • Share ›

Anand Saha • 3 years ago

Thank you! You have a knack of simplifying complexity :-)

1 ^ | v • Reply • Share ›

fkratzert Mod → Anand Saha • 3 years ago

Thanks, this is really what I tried. I had a hard time to work myself through this topic, back when I followed the cs231n online and set many hours in front of a paper an tried to derive myself everything on my own. At the end I broke everything down to this really simple steps.

^ | v • Reply • Share ›

禹邵 賴 → fkratzert • 10 months ago • edited

But still wrong. Still not give a precise explanation about the sigma of beta term.

^ | v • Reply • Share ›

Majic Ji • 3 years ago

Thanks. I got the intuition of how to derive the derivative of np.sum along one axis.

1 ^ | v • Reply • Share ›

fkratzert Mod → Majic Ji • 3 years ago

Glad I could help you.

^ | v • Reply • Share ›

jschaeff • 3 years 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 propogating 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!

1 ^ | v • Reply • Share ›

fkratzert Mod → jschaeff • 3 years 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?

1 ^ | v • Reply • Share ›

jschaeff ➚ fkratzert • 3 years 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

1 ^ | v • Reply • Share ›

fkratzert Mod ➚ jschaeff • 3 years 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.

1 ^ | v • Reply • Share ›

jschaeff ➚ fkratzert • 3 years ago

Got it. Thanks again for your clear explanations.

1 ^ | v • Reply • Share ›

fkratzert Mod ➚ jschaeff • 3 years ago

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

1 ^ | v • Reply • Share ›

Sahil • 5 months ago • edited

I am loaded the trained checkpoint file for inference. I am extracted the beta, moving mean and moving variance and all weights from the model. In batch normalization I getting wrong result when I am manually calculating the output of batch_normalization. for example,

for given input x = 2.107175067067146301e-02
the extracted parameters,
beta = 0.04061257
moving mean = -0.0013569
moving variance = 4.48082483e-06
epsilon = 1e-07

and gamma is ignored. because I set training time scale = false so gamma is ignored.

When I calculate the output of batch normalization at inference time for given input x
x_hat = (x - moving_mean) / square_root_of(moving variance + epsilon)
= (0.021071750670671463 01 – (-0.0013569)) / √(0.0000044808248 3 + 0.0000001)
= 10.479277536
so x_hat is 10.479277536

see more

Reply • Share ›

Sean • 9 months ago

Hello. I'm a bit late to the party, but could you explain a bit more why we sum all the gradients when calculating for dbeta? Why aren't we doing a summation over dout for the other arrow?

Reply • Share ›

Hassan Saeed • 9 months ago

Thanks very much for your explanation it really help me understand this technique better , thanks THAAAANKS

Reply • Share ›

Dhawal Gupta • a year ago

Great blog, doing gods work, thank you for such a clear explanation.

Reply • Share ›

Karthikeyan Mg • a year ago

Good one!
Is there a video version of this blog in cs231n? If so can you link the specific video?

Reply • Share ›

fkratzert Mod ➚ Karthikeyan Mg • a year ago

You mean if they explain it in class somewhere? No they don't, at least not when I did the class by watching the videos. There is one point where Andrej explains chain rule and computational graphs, but never the backpass through the batchnorm layer

Reply • Share ›

Reply · Share ›

Michael Green • 2 years ago

Nice writeup. :)

Reply · Share ›

fkratzert Mod ➚ Michael Green • 2 years ago

Thanks :)

Reply · Share ›

Abhishek Nadgeri • 2 years ago

Thanks for the post I loved it :D

Reply · Share ›

fkratzert Mod ➚ Abhishek Nadgeri • 2 years ago

you are welcome. great that you liked it!

Reply · Share ›

Ryan Neph • 2 years ago

Thanks for the nice explanation. A small terminological suggestion: when you discuss the derivative /gradient of a function, it is widely accepted to use the term "derivative", whereas "derivation" instead refers to the process of arriving at a result.

Reply · Share ›

fkratzert Mod ➚ Ryan Neph • 2 years ago

Hey Ryan,
sorry for replying so late and thank you very much for your corrections. Since I'm no english native, corrections like this are essential for me. So thanks again and I fixed the words.

Reply · Share ›

Jae Duk Seo • 2 years ago

Wow this is such amazing post! Thank you so much!

Reply · Share ›

Vibhu Jawa • 2 years ago

Amazing stuff man, I was trying to understand computational graphs for RELU and your post made everything so easy to follow.
Loved the clarity of explanation.

Reply · Share ›

fkratzert Mod ➚ Vibhu Jawa • 2 years ago

Hey man, good to hear that my article could help you understanding computational graphs in general and that you were able to port it to a different operation! Good work

Reply · Share ›

Jae Duk Seo • 2 years ago

Amazing post, thank you for posting!

Reply • Share ›

Hyeungshik Jung • 2 years ago

What a lovely post ㅜㅜㅜ

Reply • Share ›

tapsi • 2 years ago • edited

I have a problem with "give me back my raw inputs" or the un-doing part. The network can easily learn some parameters for a shift-and-scale transform, but could this really undo batchnorm? The normalization is done per-batch (with different means and variances each batch). The learned shift-scale transform is not per-batch but global for the complete dataset.

Reply • Share ›

fkratzert Mod ➔ tapsi • 2 years ago

No you are right, I can't undo the batch normalization of each batch perfectly, since it learns the shift and scale parameters as representatives of the entire training data. I would say numerically it won't be the same, but I think the distribution of your data/the activations would be close to the original one, since the scale and shift parameters are learned from your entire training data.

Reply • Share ›

tapsi ➔ fkratzert • 2 years ago

Thanks. This goes down to batch size then. For a 100 samples batch, the variance and the difference from the global distribution can be very, very high. So the training is done with each neuron almost randomly pulled around (according the neighbour samples in the batch). This is reported to have a nice regularizing effect, but in my experiment also a big source of noise that damages training noticeably. I'm experimenting with using running stats instead of just the current batch.

1 Reply • Share ›

fkratzert Mod ➔ tapsi • 2 years ago

Would be nice to hear of your results, once you can show off something (also negative results please ;))

Reply • Share ›

kanhavishva • 2 years ago

Awesome work, helped a lot, thank you buddy..

Reply • Share ›

fkratzert Mod ➔ kanhavishva • 2 years ago

Thanks a lot. I'm still astonished how many people actually read this artical. Glad to see it could help you too

1 Reply • Share ›

Rakhil Immidisetti • 2 years ago

Why are we not averaging the sum of gradients of training examples in a mini-batch by the size of the mini-batch??

^ | v • Reply • Share ›

zhuzii • 2 years ago

Cool, thanks.

^ | v • Reply • Share ›

Writer • 3 years ago

Nice work on this derivation! I was able to implement a JavaScript implementation of batch norm backpropagation.

The description is at http://miabellaAI.net/ and the web app is here: http://ann.miabellaAI.net/

Here are some tips for those who took Andrew Ng's deep learning course:

1. Andrew Ng and his team offer a backpropagation chain based on D-by-N matrices, which is the transpose of what is presented here. This needs to be taken into account at every step.

2. The Python code I've seen at various places makes liberal use of broadcasting, which is when Python handles matrix-by-vector operations automatically. Depending on your platform, you might need to code this yourself, always keeping in mind the dimensions from point #1 above.

3. Gradient checking can be a pain to write, but it's the main way to make sure everything is hooked up correctly.

^ | v • Reply • Share ›

Samuel Pun • 3 years ago

Dear, thank you so very much for sharing this. It did saved my life. I have one stupid question hope you dont mind. I tried implementing your way of backward prop and compared it with other