An Overview of Normalization Methods in Deep Learning

Normalization in deep learning has always been a hot topic. Getting normalization right can be a crucial factor in getting your model to train effectively, but this isn't as easy as it sounds. For example, when you fine-tune ResNet, how should you treat the batch normalization layers? What happens to batch normalization layers in multi-GPU environments? What if you change the batch size during training?

Unless you have a good grasp of normalization, these questions are difficult to answer. This is why I previously covered batch normalization, weight normalization, and layer normalization as well as why they are effective on this blog. Though I did my best to provide a holistic view of normalization at that time, I've learned much more since then and there have been many important developments such as group normalization and batch-instance normalization.

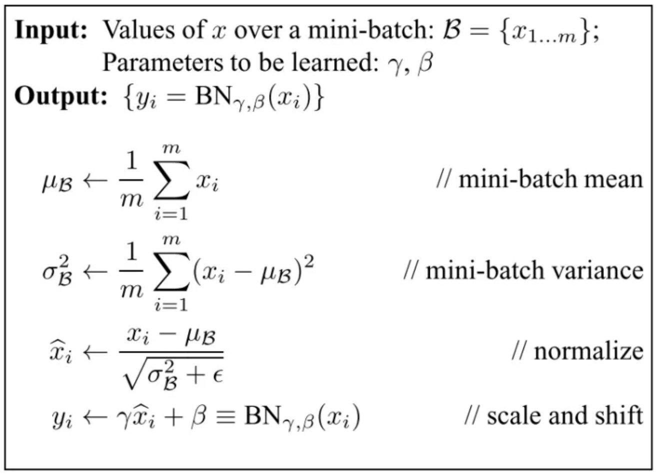
Therefore, I've decided to write a whole new compilation post about normalization, why its effective, and what its variants are. This post will cover the basics of batch normalization and other normalization methods such as weight normalization, group normalization, instance normalization, and layer normalization.

1. What is Batch Normalization and Why is it Effective?

Let's start by reviewing (or introducing) batch normalization, the method that started it all. Batch normalization is a normalization method that normalizes activations in a network across the mini-batch. For each feature, batch normalization computes the mean and variance of that feature in the mini-batch. It then subtracts the mean and divides the feature by its mini-batch standard deviation.

This restricts the activations to have 0 mean and unit standard deviation. There are many potential problems with this. What if we were using relu activations? What if increasing the magnitude of the weights made the network perform better? To address these potential problems, batch normalization adds two additional learnable parameters: the mean and magnitude of the activations. Batch normalization **rescales** the normalized activations and adds a constant. This means that the expressiveness of the network does not change.

Let's summarize this by looking at the equations behind batch normalization



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

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

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

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

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

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

Simple right? Take a careful look at these equations, because we will be encountering many similar equations later on, and it helps to connect each equation with its intuitive meaning.

As you will probably know, batch normalization took the deep learning community by storm, significantly improving the performance of CNNs on image classification and multiple other tasks.

So why does batch normalization work so well? Well, we don't really know. As this paper mentions, the original explanation of "internal covariate shift" is not very precise, so at best it is a heuristic explanation, and at worst it might be completely wrong. A recent paper actually claims that the success of batch normalization has nothing to do with internal covariate shift! The figure below shows the performance of models trained with and without batch normalization on the left and the change in layer input distributions during training on the right. The difference in performance is clear, but there does not seem to be much difference in terms of the change in layer input distributions.

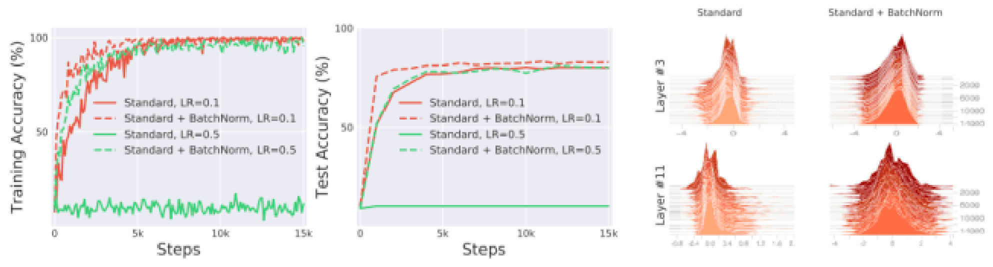


Figure 1: Comparison of (a) training (optimization) and (b) test (generalization) performance of a standard VGG network trained on CIFAR-10 with and without BatchNorm (details in Appendix A). There is a consistent gain in training speed in models with BatchNorm layers. (c) Even though the gap between the performance of the BatchNorm and non-BatchNorm networks is clear, the difference in the evolution of layer input distributions seems to be much less pronounced. (Here, we sampled activations of a given layer and visualized their distribution over training steps.)

Although we don't have a definitive answer as to why batch normalization works, we do have a few intuitive explanations.

According to the book Deep Learning by Ian Goodfellow, batch normalization can be understood from the perspective of **high-order interactions**. I explain more about this in this post, but the intuition goes like this:

- In a neural network, changing one weight affects subsequent layers, which then affect subsequent layers, and so on.
- This means changing one weight can make the activations fly all over the place in complex ways.
- This forces us to use lower learning rates to prevent the gradients from exploding, or – if we use sigmoid activations – to prevent the gradients from disappearing.
- Batch normalization to the rescue! Batch normalization reparameterizes the network to make optimization easier.
- With batch normalization, we can control the magnitude and mean of the activations **independent** of all other layers.
- This means weights don't fly all over the place (as long as we have sensible initial means and magnitudes), and we can optimize much easier.

In the paper I mentioned earlier, the authors claim that batch normalization works because it makes the loss surface **smoother**. Concretely, it bounds the magnitude of the gradients much more tightly (more formally, it improves the β-smoothness or the Lipschitzness of the function). For a bit more intuition into why this might help optimization, please refer to my post on Hessians and second-order optimization.

Either way, the overall intuition is that batch normalization makes the loss surface "easier to navigate", making optimization easier, enabling the use of higher learning rates, and improving model performance across multiple tasks. So overall, batch normalization is great. But is it perfect? Not quite. Understanding these pitfalls and potential solutions is crucial to being a deep learning practitioner, so let's delve into the pitfalls of batch normalization.

2. Why not Batch Normalization?

In normalization, we ideally want to use the **global** mean and variance to standardize our data. Computing this for each layer is far too expensive though, so we need to approximate using some other measures. In batch normalization, this measure is the mean/variance of the **mini batch**. At first glance, this seems perfectly okay. There might be some noise, but it's not a bad estimate, right?

Well, not always. Here are two scenarios where this might not be a good idea.

1. Small batch size

If the batch size is 1, the variance is 0 so batch normalization cannot be applied. Slightly larger mini-batch sizes won't have this problem, but small mini-batches make our estimates very noisy and can negatively impact training, meaning batch normalization imposes a certain lower bound on our batch size.

2. Recurrent connections in an RNN

In an RNN, the recurrent activations of each time-step will have different statistics. This means that we have to fit a separate batch normalization layer for each time-step. This makes the model more complicated and – more importantly – it forces us to store the statistics for each time-step during training.

The question then is: how else can we estimate the global mean/variance to normalize our data? Here are a couple of proposed methods.

3. What are the Alternatives?

1. Weight Normalization

In weight normalization, instead of normalizing the activations directly, we **normalize the weights of the layer**.

Weight normalization re-parameterizes the weights $\mathbf{w}$ of any layer in the neural network in the following way:

$$\mathbf{w} = \frac{g}{||\mathbf{v}||}\mathbf{v}$$

Similar to batch normalization, weight normalization **separates the norm of the weight vector from its direction without reducing expressiveness**. This has a similar effect to dividing the inputs by the standard deviation in batch normalization.

As for the mean, the authors of the paper proposed using a method called "mean-only batch normalization" together with weight normalization. Basically, they subtract out the mean of the minibatch but do not divide by the standard deviation. Compared to the standard deviation, the mean apparently has a "gentler" noise thanks to the law of large numbers, so weight normalization can still work in settings with a smaller minibatch size.

The experimental results of the paper show that weight normalization combined with mean-only batch normalization achieves the best results on CIFAR-10, an image classification dataset.

2. Layer Normalization

Layer normalization is a method developed by Geoffery Hinton. Compared to weight normalization, layer normalization is slightly harder to grasp intuitively.

To understand layer normalization, recall that a mini-batch consists of multiple examples with the same number of features. Mini-batches are matrices – or tensors if each input is multi-dimensional – where one axis corresponds to the batch and the other axis – or axes – correspond to the feature dimensions.

Batch normalization normalizes the input features across the batch dimension. The key feature of layer normalization is that it **normalizes the inputs across the features**.

The equations of batch normalization and layer normalization are deceptively similar:

Batch normalization

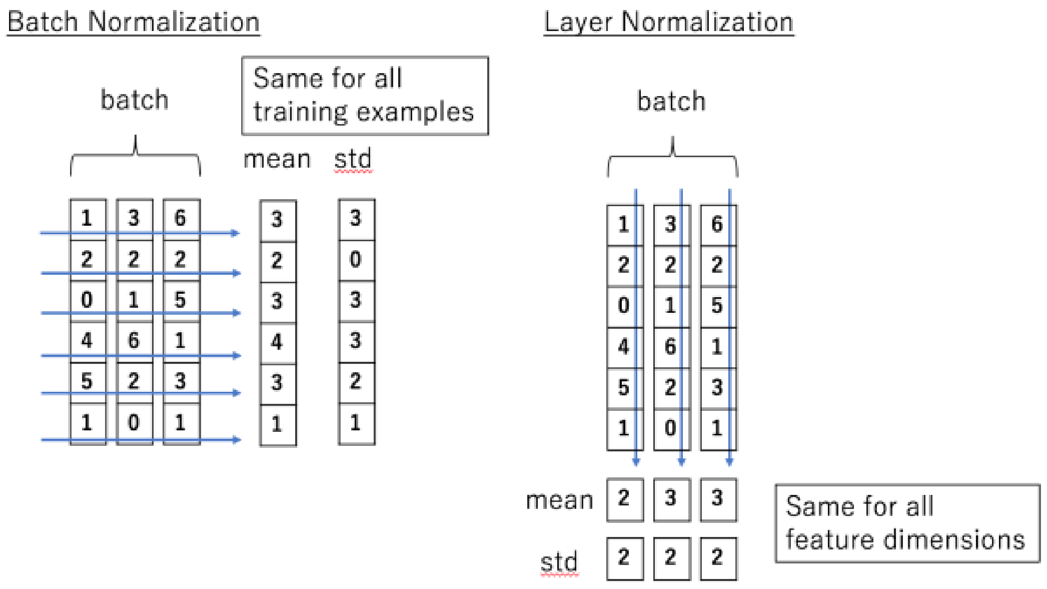
$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_{ij}$$
$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_{ij} - \mu_j)^2$$
$$\hat{x_{ij}} = \frac{x_{ij}-\mu_j}{\sqrt{\sigma_j^2+\epsilon}}$$

Layer normalization:

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_{ij}$$
$$\sigma_i^2 = \frac{1}{m}\sum_{j=1}^{m}(x_{ij} - \mu_i)^2$$
$$\hat{x_{ij}} = \frac{x_{ij}-\mu_i}{\sqrt{\sigma_i^2+\epsilon}}$$

where x_{ij} is the i,j-th element of the input, the first dimension represents the batch and the second represents the feature (I have modified the notation from the original papers to make the contrast clearer).

The difference becomes clearer when we visualize it:



In batch normalization, the statistics are computed *across the batch* and are the same for each example in the batch. In contrast, in layer normalization, the statistics are computed *across each feature* and are **independent of other examples**.

This means that layer normalization is not a simple reparameterization of the network, unlike the case of weight normalization and batch normalization, which both have the same expressive power as an unnormalized neural network. A detailed discussion of the differences this creates involves a lot of math and is beyond the scope of this article, so if you are interested please refer to the original paper. Experimental results show that layer normalization **performs well on RNNs**.

3. Instance Normalization

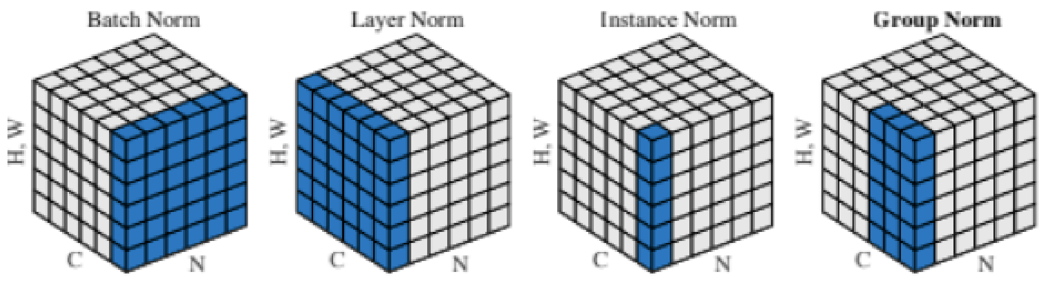
Instance normalization is similar to layer normalization but goes one step further: it computes the mean/standard deviation and normalize across **each channel in each training example**.

Originally devised for style transfer, the problem instance normalization tries to address is that the network should be agnostic to the **contrast** of the original image. Therefore, it is specific to images and not trivially extendable to RNNs.

Experimental results show that instance normalization performs well on style transfer when replacing batch normalization. Recently, instance normalization has also been used as a replacement for batch normalization in GANs.

4. Group Normalization

Group normalization – as its name suggests – computes the mean and standard deviation over **groups of channels** for each training example. In a way, group normalization is a combination of layer normalization and instance normalization. Indeed, when we put all the channels into a single group, group normalization becomes layer normalization and when we put each channel into a different group it becomes instance normalization.



Though layer normalization and instance normalization were both effective on RNNs and style transfer respectively, they were still inferior to batch normalization for image recognition tasks. Group normalization was able to achieve much closer performance to batch normalization with a batch size of 32 on ImageNet and outperformed it on smaller batch sizes. For tasks like object detection and segmentation that use much higher resolution images (and therefore cannot increase their batch size due to memory constraints), group normalization was shown to be a very effective normalization method.

Now the question is, why is group normalization so effective compared to layer normalization and instance normalization?

One of the implicit assumptions that layer normalization makes is that all channels are "equally important" when computing the mean. This assumption is not always true in convolution layers. For instance, neurons near the edge of an image and neurons near the center of an image will have very different activation statistics. This means that computing different statistics for different channels can give models much-needed flexibility. Channels in an image are not completely independent though, so being able to leverage the statistics of nearby channels is an advantage group normalization has over instance normalization.

5. Batch Renormalization

Batch renormalization is another interesting approach for applying batch normalization to small batch sizes. The basic idea behind batch renormalization comes from the fact that we do not use the individual mini-batch statistics for batch normalization during inference. Instead, we use a **moving average** of the mini batch statistics. This is because a moving average provides a better estimate of the true mean and variance compared to individual mini-batches.

Then why don't we use the moving average during training? The answer has to do with the fact that during training, we need to perform backpropagation. In essence, when we use some statistics to normalize the data, we need to **backpropagate through those statistics as well**. If we use the statistics of activations from previous mini-batches to normalize the data, we need to account for how the previous layer affected those statistics during backpropagation. If we ignore these interactions, we could potentially cause previous layers to keep on increasing the magnitude of their activations even though it has no effect on the loss. This means that if we use a moving average, we would need to store the data from **all previous mini-batches** during training, which is far too expensive.

In batch renormalization, the authors propose to use a moving average while also taking the effect of previous layers on the statistics into account. Their method is – at its core – a simple reparameterization of normalization with the moving average. If we denote the moving average mean and standard deviation as μ and σ and the mini-batch mean and standard deviation as μ_B and σ_B, the batch renormalization equation is:

$$\frac{x_i - \mu}{\sigma} = \frac{x_i - \mu_\mathcal{B}}{\sigma_\mathcal{B}} \cdot r + d, \quad \text{where } r = \frac{\sigma_\mathcal{B}}{\sigma}, \ \ d = \frac{\mu_\mathcal{B} - \mu}{\sigma}$$

In other words, we multiply the batch normalized activations by r and add d, where both r and d are computed from the minibatch statistics and moving average statistics. The trick here is to **not backpropagate** through r and d. Though this means we ignore some of the effects of previous layers on previous mini batches, since the mini batch statistics and moving average statistics should be the same on average, the overall effect of this should cancel out on average as well.

Unfortunately, batch renormalization's performance still degrades when the batch size decreases (though not as badly as batch normalization), meaning group normalization still has a slight advantage in the small batch size regime.

6. Batch-Instance Normalization

Batch instance normalization is an extension of instance normalization that attempts to account for differences in contrast and style in images. The problem with instance normalization is that it **completely** erases style information. Though this is beneficial in certain settings like style transfer, it can be problematic in settings like weather classification where the style (e.g. the brightness of the image) can be a crucial feature. In other words, the degree of style information that should be removed is dependent on the task at hand. Batch-instance normalization attempts to deal with this by **learning** how much style information should be used for each **task and feature map** (channel).

Denoting the batch normalized outputs and the instance normalized outputs as $\hat{x}^{(B)}$ and $\hat{x}^{(I)}$ each, the batch-instance normalized output can be expressed as:

$$\mathbf{y} = \left(\rho \cdot \hat{\mathbf{x}}^{(B)} + (1 - \rho) \cdot \hat{\mathbf{x}}^{(I)}\right) \cdot \gamma + \beta,$$

In other words, batch-instance normalization is just an interpolation between batch normalization and instance normalization. The interesting aspect of batch-instance normalization is that the balancing parameter ρ is **learned** through gradient descent.

Batch-instance normalization outperformed batch normalization on CIFAR-10/100, ImageNet, domain adaptation, and style transfer. In image classification tasks, the value of ρ tended to be close to 0 or 1, meaning many layers used either instance or batch normalization almost exclusively. In addition, layers tended to use batch normalization more than instance normalization, which fits the intuition proposed by the authors that instance normalization serves more as a method to eliminate unnecessary style variation. On style transfer – on the other hand – the model tended to use instance normalization more, which makes sense given style is much less important in style transfer.

The authors also found that in practice, using a higher learning rate for ρ improves performance.

One important contribution of batch-instance normalization is that it showed that models could learn to adaptively use different normalization methods using gradient descent. This raises the question: could models learn to use an even wider variety of normalization methods? This nicely leads us to our next normalization method.

7. Switchable Normalization

With all these shiny new normalization methods, a question naturally comes to mind: is batch normalization still the best normalization method out-of-the-box? What if we combine different normalization methods? What if the best normalization method actually differs depending on the depth of the layer?

This is the question asked by this paper. This paper proposed switchable normalization, a method that uses a weighted average of different mean and variance statistics from batch normalization, instance normalization, and layer normalization. Similar to batch-instance normalization, the weights were learned through backpropagation. The authors showed that switch normalization could potentially outperform batch normalization on tasks such as image classification and object detection.

Perhaps more interestingly, the paper showed that the statistics of instance normalization were used more heavily in earlier layers, whereas layer normalization was preferred in the later layers, and batch normalization being used in the middle. Smaller batch sizes lead to a preference towards layer normalization and instance normalization, as is expected.

8. Spectral Normalization

Though not designed to be a replacement for batch normalization, I thought I would give spectral normalization an honorable mention here since it gives us a very interesting look into normalization in deep learning in general. Spectral normalization was a method proposed to improve the training of GANs by limiting the Lipschitz constant of the discriminator. The Lipschitz constant – in case you are not familiar – is a constant L for a function f where for any x and y,

$$||f(x) - f(y)|| \leq L||x - y||$$

The authors restrict the Lipschitz constant by normalizing the weight matrices by their largest eigenvalue (or their spectral norm – hence the name). The largest eigenvalue is computed using the power method which makes the computational cost of this method very cheap. Compared to weight normalization, spectral normalization does not reduce the rank of the weight matrix. Experimental results show that spectral normalization improves the training of GANs with minimal additional tuning.

4. Additional Pitfalls of Batch Normalization

In addition to the issue of dependence on batch size mentioned above, here are a few more concerns that batch normalization raises:

1. Dependence of the loss between samples in a minibatch

When we introduce batch normalization, the loss value for each sample in a minibatch becomes dependent **on other samples in the minibatch.** For instance, if a sample causes a certain layer's activations to become much larger, this will make the mean and variance larger as well. This will change the activation for **all other samples in the minibatch** as well. Furthermore, the minibatch statistics will depend on the minibatch size as well (a smaller minibatch size will increase the random variation in the mean and variance statistics).

This isn't much of a problem when training a model on a single machine, but when we start to conduct distributed training, things can get ugly. As mentioned in this paper, we need to take extra care in choosing the batch size and learning rate in the presence of batch normalization when doing distributed training. If two different machines use different batch sizes, they will indirectly be optimizing **different loss functions**: this means that the value of γ that worked for one machine is unlikely to work for another machine. This is why the authors stressed that the batch size for each worker must be kept constant across all machines.

Moral of the story: remember that batch normalization makes the loss function dependent on the batch size.

2. Fine tuning

Suppose we fine-tune a ResNet50 by freezing all the layers except the last layer. An interesting question is whether we should use the mean and variance computed on the **original dataset** or use the mean and variance of the mini-batches. Though most frameworks use the mini-batch statistics, if we are using a different mini-batch size there will be a mismatch between the optimal batch normalization parameters and the parameters in the network. As this thread discusses, it might be better to use the statistics of the original dataset instead.

5. Conclusion

Normalization is now a staple in deep learning thanks to how it makes the optimization of deep neural networks much easier. Research on normalization in deep learning has come far, but this is still an active area of research with many exciting findings and new applications being discovered at a rapid rate.

Though recent papers have explored different normalization methods at different depths of the network, there are still many dimensions that can be explored. In this paper, the authors show that l_1 normalization performs better than batch normalization, suggesting that as we understand batch normalization better, we might be able to come up with more principled methods of normalization. This means that we might see new normalization methods use different statistics instead of changing what they compute the statistics over.