CDF Normalization for Controlling the Distribution of Hidden Nodes

Anonymous Author(s)
Affiliation
Address
email

Abstract

Batch Normalizaiton (BN) is a normalization method for deep neural networks that has been shown to accelerate training. While the effectiveness of BN is undisputed, the explanation of its effectiveness is still being studied. The original BN paper attributes the success of BN to reducing internal covariate shift, so we take this a step further and explicitly enforce a Gaussian distribution on hidden layer activations. This approach proves to be ineffective, demonstrating that reducing internal covariate shift in and of itself does not help training.

1 Introduction

Deep Neural Networks have increased in popularity over the last several years, particular in fields like computer vision where the complexity of data requires deeper networks [8, 11, 13, 3]. Training these deep networks is challenging; many methods have been proposed to boost training performance, including optimization methods [12, 6], learning rate schedulers [9], and regularization methods [5]. One very popular regularization method is Batch Normalization (BN) [4], which standardizes the activations of convolutional and linear layers before applying an activation function. This simple normalization technique allows for more stable training with higher learning rates, and has become standard across many fields of deep learning.

In [4], BN's effectiveness is attributed to a reduction in internal covariate shift, which is defined as the shift in the distribution of layer activations due to changes in earlier layer's parameters during training. By making each activation have zero mean and unit variance, BN ensures that changes to network parameters earlier in the network do not effect the first and second moments of network activations later in the network. Further, the authors argue that the outputs of linear and convolution layers are "more Gaussian", in which case controlling the first and second moments of the output distributions actually controls the whole distribution, since the distribution of Gaussian random variables is completely determined by the mean and variance.

However, the explanations for BN's effectiveness continue to be studied [10, 1]. Interestingly, in [10] the authors argue that BN accelerates training not because of a reduction in internal covariate shift, but because of a smoother loss landscape. To show this, the authors add random noise after BN to explicitly induce internal covariate shift, and they find that his "noisy" BN is just as effective as regular BN. Additionally, the authors find that adding such random noise without using BN prevents the network from training entirely. This suggests that reducing internal covariate shift is not actually why BN accelerates training, since training is accelerated even when internal covariate shift is present.

In this paper, we aim to further demonstrate that reducing internal covariate shift in itself does not accelerate training. In show this, we explicitly enforce a standard Gaussian distribution upon hidden layer activations. We do this by first applying an estimated cumulant distribution function (CDF) to each hidden layer activation, followed by transforming to a standard Gaussian distribution via the

Under review at ICBINB Workshop @ NeurIPS 2021. Do not distribute.

standard Gaussian inverse CDF. In theory, this would completely eliminate internal covariate shift, since the inputs to subsequent layers are always enforced to follow the same distribution.

2 Method

In this section, we describe our method transforming hidden layer activations to standard Gaussians, which we will call CDF Normalization (CDFN). Let $X_1, \ldots, X_p$ be random variables representing the p-dimensional output of a hidden layer in the network. Assume first that we have an estimate $\hat{F}_i, i = 1, \ldots, p$ for the CDF of our random variables, then $\hat{F}_1(X_1), \ldots, \hat{F}_p(X_p)$ are approximately standard uniform random variables. Next, let ϕ be the CDF of some other distribution (in our case, the standard Gaussian), then $\phi^{-1}\left[\hat{F}_1(X_1)\right], \ldots, \phi^{-1}\left[\hat{F}_p(X_p)\right]$ are all random variables with distributions given by ϕ. This gives us a framework for approximately transforming data at this hidden layer to any distribution we desire.

We now turn our attention to how to calculate the CDF estimates $\hat{F}_1, \ldots, \hat{F}_p$. Following BN, it would be ideal to make our CDF estimates only using the current mini-batch so that the normalization can flow through the gradient backpropagation. However, as we will discuss in section 4, this turns out to be challenging. Instead, we use a running collection of data from previous mini-batches to estimate the CDFs for the current mini-batch. We compute the empirical CDF using a running collection of n data samples, creating a step function, and we further linearly interpolate this step function to transform to a piece-wise linear function. This transformation creates a function whose gradient is not always 0, allowing for successful backpropagation of gradients. The process of linear interpolation is illustrated in Figure 3 in the Appendix. Last, the running collection of data is updated using a momentum parameter $m \in [n - N, n]$: sample m points from the current collection without replacement, and sample the remaining $n - m$ points from the current mini-batch. The overall CDFN process is summarized in Algorithm 1.

Algorithm 1: CDFN Procedure. For more details on Linear Interpolation step, see Figure 3

Input : Current mini-batch $x_1, \ldots, x_N$, running collection $y_1, \ldots, y_n$, momentum m
Output : Normalized mini-batched $x'_1, \ldots, x'_N$
ECDF $\leftarrow$ Func(a) : $\frac{1}{n}\sum_{i=1}^{n} I(y_1 \leq a)$
$\hat{F} \leftarrow$ LinInterp(ECDF)
for *i = 1, . . . , N* **do**
  $x'_i = \hat{F}(x_i)$
end
$y_1, \ldots, y_n =$ sample$(y_1, \ldots, y_n, m) +$ sample$(x_1, \ldots, x_N, n - m)$
return $x'_1, \ldots, x'_N$

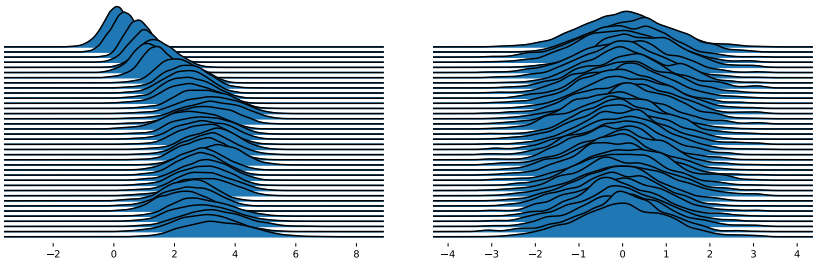
Note that we have described CDFN for 2D inputs of size (N, p), yet hidden layers may be of higher dimensions, for example the output of a 2D convolution layer which is a 4D tensor of size (N, C, H, W). In this case, CDFN is adapted similarly to BN: a CDF is estimated for each index of the second dimension (channel dimension) using values from all other dimensions.

3 Experiments

To test the CDFN procedure, we train a Convolutional Neural Network (CNN) on the CIFAR-10 image classification dataset [7]. Our CNN architecture is similar to a VGG style network [11]:

$$\begin{aligned}\text{ConvBlock}(3, 32) \rightarrow \text{ConvBlock}(32, 64) \rightarrow \text{MaxPool} \rightarrow \\ \text{ConvBlock}(64, 128) \rightarrow \text{ConvBlock}(128, 128) \rightarrow \text{MaxPool} \rightarrow \\ \text{ConvBlock}(128, 256) \rightarrow \text{ConvBlock}(256, 256) \rightarrow \text{MaxPool} \rightarrow \text{FinalLinear}\end{aligned}$$

A convolution block ConvBlock(n_{in}, n_{out}) is a sequential layer of a 2D convolutional layer with n_{in} in channels, n_{out} out channels, 3×3 kernel size, and 1 padding, followed by a normalization layer and a ReLU activation. We train this network using the Adam optimizer [6] with a batch size of 128.



(a) Before CDF Norm (b) After CDF Norm

Figure 1: Evolution of the distribution of one activation from the ConvBlock$(128, 256)$ layer before and after CDFN. Each horizontal line corresponds to one training iteration (mini-batch update), evolving from back to front.

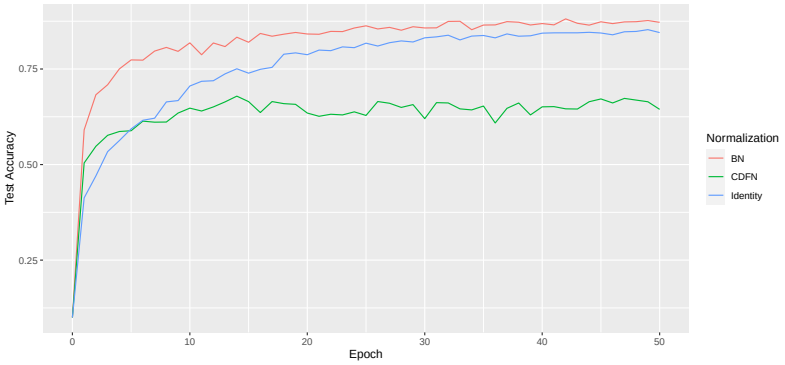


Figure 2: Test accuracies for different normalization techniques on the CIFAR-10 dataset.

3.1 Reducing Internal Covariate Shift

The goal of using CDFN is to eliminate internal covariate shift in the network, and thus we should check empirically that this in fact happening. For this, we train our CNN for 10 epochs, during which we kept track of the distribution of a random feature channel from the output of each convolutional layer before and after applying CDFN. The results of this experiment are shown in Figure 1. We see that even though the distribution of the activation is evolving during the training process, the CDFN normalization procedure is successfully transforming them to approximate Gaussian distributions that remain consistent through training. This supports the claim that CDFN is in fact eliminating internal covariate shift in the network.

3.2 Accuracy Results

Now that we know that CDFN does in fact eliminate internal covariate shift, we can ask how training the network with CDFN compares to training with BN as well as without any normalization. We do this by plotting the test accuracies for each epoch during training of our CIFAR-10 network with no normalization, BN, and CDFN. We use the same learning rate of 10^{-4} to maintain a fair convergence speed comparison even though we know BN allows for larger learning rates. The accuracies of such training runs are shown in Figure 2. As expected, BN converges faster while achieving similar final accuracy compared to using no normalization. On the other hand, while the initial convergence of CDFN is similar to using no normalization, the final accuracy achieved is much lower, as CDFN achieves a final accuracy of less than 70% compared to final accuracies over 80% for the other two methods. All in all, we see that CDFN is not condusive to effective training.

4 Analysis

Even though BN and CDFN are both normalization techniques which standardize hidden layer activations, BN accelerates training while CDFN hinders training. What is different between BN and CDFN which leads to this severe difference in performance? The critical difference is that BN uses only the current mini-batch to perform normalization, while CDFN uses data from other mini-batches acululated through training. This difference is key, because using just the current mini-batch allows the normalization to flow through the gradient backpropagation. To illustrate this point, consider one activation node mini-batch $x \in \mathbb{R}^N$. BN will transform this as

$$y = \text{BN}(x) = \frac{x - \bar{x}}{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2 + \epsilon}}$$

so during backpropagation, computing the gradient $\frac{\partial y}{\partial x}$ accounts for the normalization. Comparatively, CDFN does the transformation

$$y = \text{CDFN}(x) = \phi^{-1}\left[\hat{F}(x)\right]$$

where ϕ and $\hat{F}$ are the inverse standard Gaussian CDF and the approximated CDF following Algorithm 1. When computing the gradient $\frac{\partial y}{\partial x}$ here, we use the chain rule

$$\frac{\partial}{\partial x}\phi^{-1}\left[\hat{F}(x)\right] = \left(\phi^{-1}\right)'\left(\hat{F}(x)\right) \cdot \hat{F}'(x)$$

The gradient of $\hat{F}$ with respect to x is simply some constant that does not depend on x since $\hat{F}$ is a piece-wise linear function formed from previous mini-batches independently of x. Similarly, $\left(\phi^{-1}\right)'\left(\hat{F}(x)\right)$ does not involve any other terms than x itself, so the gradient does not account for any normalization. Thus, it is possible that the optimizer will make gradient steps that are counteractive to the normalization from CDFN, similarly to the discussion about whitening in [4]. This provides both a potential reason why CDFN is not effective and also highlights how important it is for the normalization to flow through the gradient as in BN.

At this point, it is natural to wonder whether it is possible to adapt CDFN to use only the current mini-batch as in BN. It turns out that applying an estimated CDF from only the current mini-batch is challenging. The natural first attempt would be to compute the empirical CDF using the current mini-batch and proceed exactly as in CDFN. The problem is that applying an empirical CDF to the same data it is built on amounts to doing an argsort operation, and this operation is not differentiable and thus breaks the computational graph. We can get around this problem, we can "soften" the empirical CDF calculation so that it is differentiable. One way to do this is to replace the indicator functions in the empirical CDF with sigmoid functions. Another approach would be to use some kind of softened argmax operator such as [2]. We find that both of these approaches are extremely slow when applied across an entire mini-batch, especially when normalizing over multiple dimensions as with CNNs. We also tried applying these to random subsets of the current mini-batch, and this is both still slow and also results in poor training.

5 Conclusion

Batch Normalizaiton (BN)[4] has the effect of reducing internal covariate shift in neural networks, yet [10] demonstrates that the effectiveness of batch norm is not related to this reduction in internal covariate shift. We provide a normalization technique, CDF Normalization (CDFN), which eliminates internal covariate shift and yet hinders training. We argue that CDFN not using the current mini-batch for normalization disrupts gradient flow, and yet doing CDFN using only the current mini-batch is challenging. We leave it to future work to find more efficient and effective ways to perform CDFN-style normalization using only the current mini-batch.

References

[1] J. Bjorck, C. Gomes, B. Selman, and K. Q. Weinberger. Understanding batch normalization. *arXiv preprint arXiv:1806.02375*, 2018.

[2] M. Blondel, O. Teboul, Q. Berthet, and J. Djolonga. Fast differentiable sorting and ranking. In *International Conference on Machine Learning*, pages 950–959. PMLR, 2020.

[3] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[4] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[5] A. Kadra, M. Lindauer, F. Hutter, and J. Grabocka. Regularization is all you need: Simple neural nets can excel on tabular data. *arXiv preprint arXiv:2106.11189*, 2021.

[6] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[7] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

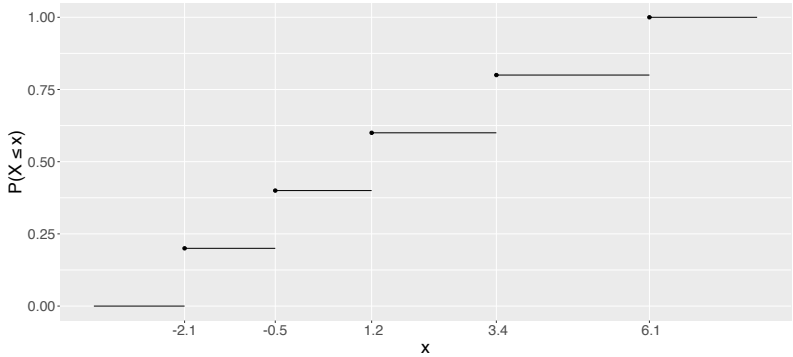
[9] I. Loshchilov and F. Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

[10] S. Santurkar, D. Tsipras, A. Ilyas, and A. Mądry. How does batch normalization help optimization? In *Proceedings of the 32nd international conference on neural information processing systems*, pages 2488–2498, 2018.

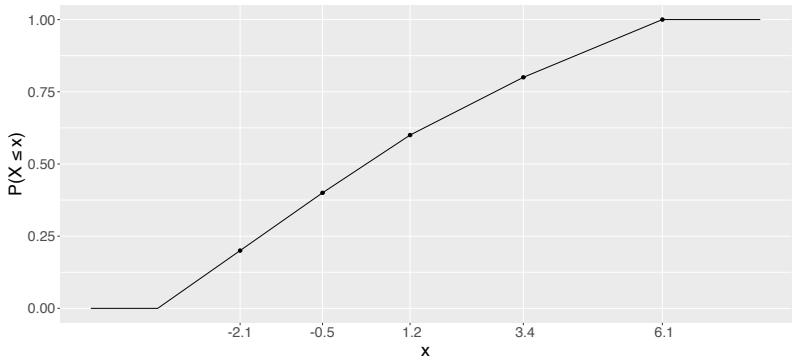
[11] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[12] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR, 2013.

[13] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.



(a) Empirical CDF from sample



(b) Linear Interpolation of ECDF

Figure 3: Demonstration of Linear Interpolation from an empirical CDF. In a) a sample of $\{-2.1, -0.5, 1.2, 3.4, 6.1\}$ is used to construct the ECDF, which then is smoothed by linear interpolation in b). In practice, values after interpolation are truncated between [0.001, 0.999] to avoid infinities.

Appendix

In Figure 3, we demonstrate the process of generating a piece-wise linear function that estimates the CDF of a given data sample. In the example in the figure, we take the sample $S = \{-2.1, -0.5, 1.2, 3.4, 6.1\}$ and start by plotting the empirical CDF of this sample as the step function $a \mapsto \frac{1}{n}\sum_{s \in S} I(a \leq s)$. To perform linear interpolation of this step function, we simply build a piece-wise linear function connecting the jump points from the step function. After the last point, the function levels off at 1, while before the first point the function continues linearly down to 0 before leveling off. In practice, we take the additional step of truncating the function between 0.001 and 0.999 to avoid computing the inverse CDF of 0 or 1 during the following step when we convert from uniform to Gaussian distribution.