Batch Normalization and Floating-point Numbers

Min Hyuk "Daniel" Jang*

Contents

1 Introduction

Artificial intelligence is a rare example of a human invention which humans have not yet been able to fully understand. It is for this reason that artificial intelligence becomes an object of science.

There are many mysteries about artificial intelligence. A prominent one is why batch normalization works. Stated in less ambiguous terms, the question at hand is: "What does the action of batch normalization have to do with the various positive effects it has on training deep learning models, as observed initially in [IS15], and, subsequently, in practice?" This paper investigates this issue. We first put forth a hypothesis that the reason why batch normalization helps with training is because it allows the subsequent layer to learn from a distribution that is "optimal". We will describe in detail what makes an activation's distribution optimal later. We will then demonstrate the results of a series of experiments, and provide an analysis to these.

*College of Liberal Studies, Seoul National University

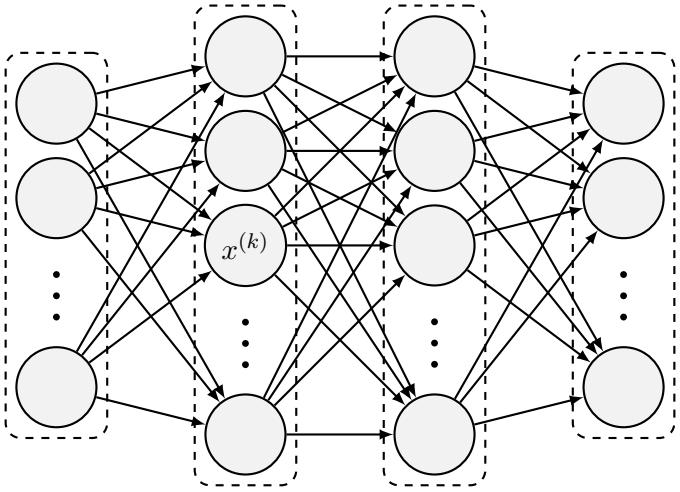

Figure 1: **nntikz**

2 Preliminaries

2.1 Batch Normalization

When we want to fit a deep neural network to some dataset, we usually use mini-batch training. Mini-batch training is a method in which training occurs in mini-batches; a mini-batch is a subset of the training dataset, usually sampled randomly. The model is evaluated on this mini-batch, and an optimization algorithm such as gradient descent updates the parameters of the model so that, simply put, it can "do better next time."

Batch normalization is defined in the context of mini-batch training neural networks. When a neural network is fed a mini-batch of N elements, an activation $x^{(k)}$ in a neural network will take on different (or perhaps the same) values for each element in the mini-batch:

$$x_1^{(k)}, x_2^{(k)}, \ldots, x_N^{(k)}.$$

We can consider the distribution of these values; we may even compute its mean $\mathbb{E}[x^{(k)}]$ and its variance $\mathrm{Var}[x^{(k)}]$. We may also consider normalizing this distribution:

$$\frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}.$$

This is the essence of batch normalization. To batch-normalize an activation is to ensure its distribution over the batch has mean 0 and standard deviation 1.

However, Ioffe and Szegedy, the inventors of batch normalization, feared (rightly) that this normalization could possibly diminish the "representation power" of the activation. It may be that the model simply cannot learn and express what it wants to if we force all activations to have mean 0 and standard deviation 1. Hence, they proposed introducing two more parameters, for each batch normalization, so that the model has the option, if need

be, to train using the distribution that that activation would've taken on if we hadn't done any batch normalization:

$$\hat{x}^{(k)} := \mathrm{BN}(x^{(k)}) := \gamma^{(k)} \cdot \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)} + \epsilon]}} + \beta^{(k)}. \tag{1}$$

Namely, $\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$ and $\beta^{(k)} = \mathbb{E}[x^{(k)}]$ implies $\hat{x}^{(k)} = x^{(k)}$, thereby recovering the original values and hence the original distribution. We say in this case that the batch normalization is an identity transform. A very small number ϵ is added to $\mathrm{Var}[x^{(k)}] \geq 0$ to prevent division by zero.

We may choose as many activations of the neural network to batch-normalize as we please. In practice, neural networks are neatly organized in layers, and we batch-normalize all activations of a layer. Therefore, we effectively *insert* an entire *layer* of batch normalization.

Batch normalization, as mentioned, was invented in 2015 by Sergey Ioffe and Christian Szegedy [IS15]. Their motivation was that, if an activation was not batch-normalized, subsequent layers in the model would have a hard time adjusting—at every step of the training process—to its fluctuating distribution. The original authors called this phenomenon "internal covariate shift." Thus, the original answer to "why batch normalization worked" was that it mitigated internal covariate shift.

Interestingly, a few years later, Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Ma¸dry were able to experimentally disprove this claim [San+19]. They trained VGG-16 models on CIFAR-10 in the following three ways:

(I) without batch normalization,

(II) with batch normalization, and

(III) with batch normalization *and time-varying, non-*0*-mean, non-*1*-variance noise* inserted after each batch normalization layer

through which they discovered that (II) and (III) trained at roughly the same rate, and better than (I). In essence, not much changed after we re-introduced the internal covariate shift that batch normalization was supposedly mitigating. This led many to believe that internal covariate shift was not an issue to begin with and batch normalization was solving another problem, or internal covariate *was* indeed a formidable issue and batch normalization was just solving another problem. Either way, the link between internal covariate shift and batch normalization was no more.

There has been surprisingly little progress concerning the workings of batch normalization since [San+19]'s rebuttal. If anything, there was progress in the other direction: Yang et al. showed that at initialization, batch normalization actually made gradients explode [Yan+19].

2.2 Floating-Point Representation

On the other hand, people have been eager to try out different ways of representing real numbers in computers for the purposes of accelerating their workloads or making them more accurate (or both).

The traditional way of representing real numbers in computers has been according to the IEEE 754 standard. Briefly put, the standard assigns to each real number its nearest *floating-point representation*, which is a real number of the following form:

$$(-1)^{\mathsf{sign}} \times 2^{\mathsf{exponent}} \times \left(1 + \sum_{i=0}^{23} b_{23-i} 2^{-i}\right)$$

Here, $\mathsf{sign}, b_i \in \{0, 1\}$ $(i = 0, \ldots, 23)$, $-126 \leq \mathsf{exponent} \leq 127$.

These floating-point representations can be stored in, say, a register of a electronic computer quite neatly as shown in Figure 2.

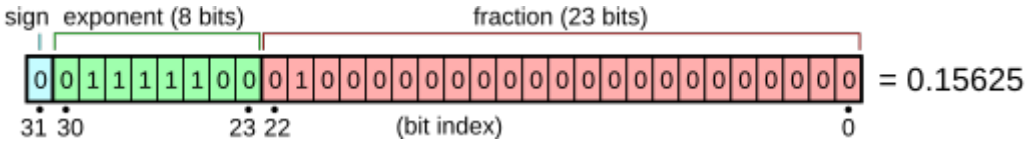

Figure 2: Wikimedia Commons. *File:Float example.svg — Wikimedia Commons, the free media repository*. [Online; accessed 22-October-2025]. 2025. URL: %5Curl%7Bhttps://commons.wikimedia.org/w/index.php?title=File:Float_example.svg&oldid=1034537282%7D

Clearly, floating-point representations can only approximate real numbers. This issue is exasperated when operations need to be done with numbers far away from the origin: the only way IEEE 754 can express a real number of large magnitude is through a large exponent, making the distance between two consecutive floating-point representations—and consequently any errors—unexpectedly large.

The usual solution is to translate any computations you wish to be high-precision to be near zero—in other words, to normalize. This is the starting point of our hypothesis.

3 Hypothesis

When we normalize a distribution, we usually expect it to have mean 0 and standard deviation 1. But this does not have to be the case: we may as well as normalize to a mean of m and a standard deviation of σ. We can relax our notion of normalization.

The normalization of batch normalization is no different. We may as well as batch-normalize to a mean of m and a standard deviation of σ of our liking.

When working with real numbers, the above relaxation is completely meaningless. However, in computers, due to the skewed distribution of floating-point representations, we cannot rule out the possibility that a different mean and standard deviation may yield different results.

We hypothesize that this is what batch normalization tries to mitigate. Batch normalization helps training not because it makes the activations' distributions steady (i.e. free from internal covariate shift) but because it actively funnels them into an "optimal" distribution that provides enough floating-point representations that the model can use to learn and express, but doesn't make the activations overly large (relative to the other weights of the model) as to destabilize training. Concisely,

Hypothesis. *Batch normalization*

- *saves training time, and*
- *increases accuracy*

because it normalizes the distribution of the activation to one that:

- *is not too sparse, and*
- *whose values are not too large.*

4 Experiments

We conduct our experiments by training a certain deep neural network on a certain dataset, just as in [San+19]. We chose the smallest widely-used version of ResNet [**resnet**], ResNet-18. For our dataset, we chose the ILSVRC2012 dataset [**imagenet**]. Experiments, which were programmed in Python using PyTorch, were run on a GPU cluster with Kubernetes, in parallel. A Git(Hub) repository containing the setup can be found at `https://github.com/jangdan/batchnorm`.

5 Results

Considering the (rough) symmetry of IEEE 754, we opted not to experiment with different means. Hence, all results which follow are trained with batch normalization mean 0.

We were lucky in that the optimum (at least for ResNet-18) seemed to be a reasonable number. Namely, it seems to be in the interval $(1, 4)$. In our tests, batch normalization with a standard deviation of 2 generally ranked highest in all the metrics we were gathering—these being training loss, validation loss, top-1 accuracy, and top-5 accuracy. The standard-deviation-1 and -4 experiments followed. We also had a control run, which used the unmodified version of PyTorch's implementation of batch normalization. We note the strictly decreasing performance of batchnorms of standard deviation 0.5, 0.25, 0.125, 0.0625, in that order. This observation is in line with our proposition; we imagine the distribution of the activations are being funneled to too narrow an interval near zero, outright reducing the number of floating point representations the model can use to learn and express (Figure 3).

We searched with powers of two to minimize the variability of the outcomes, since the exponent part of IEEE 754 is a power of two; we'd ideally be changing only the exponent of our (deterministic) training runs.

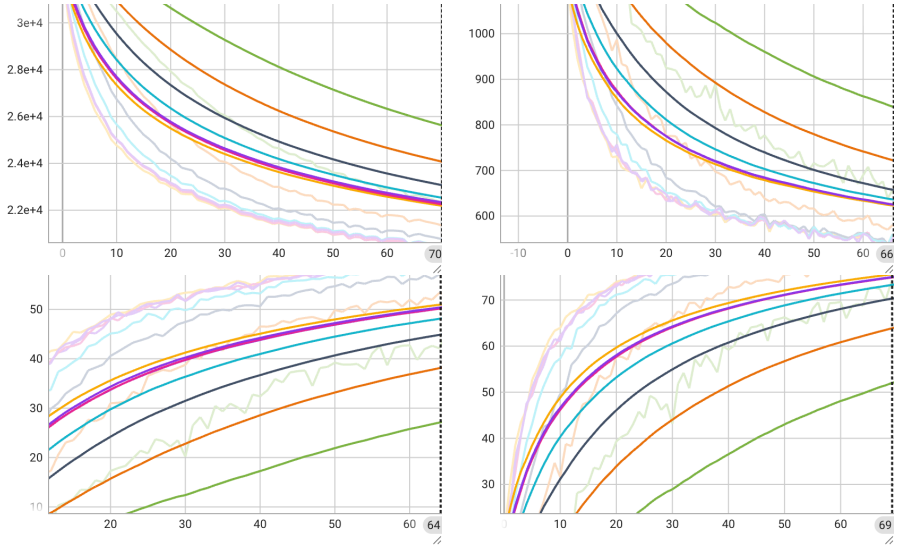

Figure 3: Graphic results of our initial search. The x axis represents the number of training epochs, or passes through the training dataset. From top left, training loss, validation loss, top-1-accuracy, top-5-accuracy. The graphs show data for standard deviation 0.0625, 0.125, 0.25, 0.5, 1, 2, and 4, but the colors are a bit hard to discern; we note the "rankings" of the respective experiments instead: For training loss & validation loss: 2 1 control 4 0.5 0.25 0.125 0.0625; For top-1 accuracy: 2 4 1 control 0.5 0.25 0.125 0.0625; For top-5 accuracy: 2 4 control 1 0.5 0.25 0.125 0.0625.

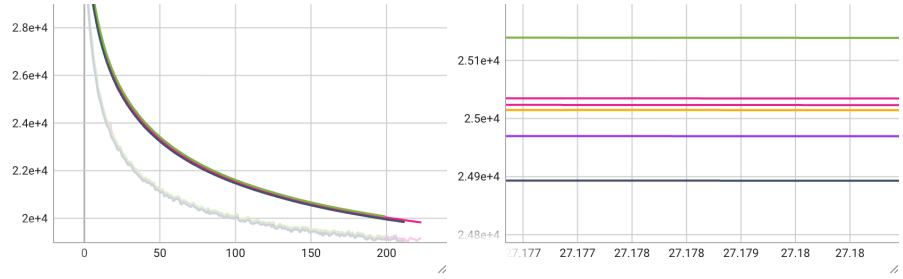

Figure 4: Graphic results of our non-powers-of-two search. The x axis again represents the number of training epochs. Left: training loss; Right: a closeup of training loss (same experiment). Again, the colors are a bit hard to discern; hence we note the "ranking" instead: 4 5 7 6 8 9.

We also attempted an inconclusive trial of non-powers of two. For this, we used standard deviations of 5, 6, 7, 9, and laid the data out with runs of $\sigma = 4, 8$ (Figure 4).

This trial's results are a bit harder to understand, but the performance roughly gets worse as the standard deviation increases, an observation again in line with our proposition. Compared to the powers-of-two case, increasing the standard deviation of normalization in increments of one does not correspond to an outright constant increase in the number of number representations the model can use, and furthermore each increment changes the inner distribution of floating point representations. A more comprehensive analysis of the distribution of IEEE 754 numbers must be performed for further quantitative understanding.

Finally, we analyze the terminal statistics after training for 200 epochs [1].

Experiment	training loss	val. loss	acc1 (%)	acc5 (%)
control	18,998.0938	491	65.836	86.868
0.0625	19,942.9277	512	59.804	83.52
0.125	19,586.6836	503	62.352	84.722
0.25	19,285.6055	506	64.4	86.132
0.5	19,080.3223	**490**	65.758	86.856
1	**18,982.6816**	499	**65.828**	**86.96**
2	18,996.6387	506	64.272	85.778
4	19,061.5352	491	65.694	86.884
8	19,160.4609	484	65.69	86.772
16	19,320.5762	500	65.136	86.416

At the end, the statistics seem to indicate that a standard deviation of 1 is (mostly) optimal. This is expected, as batch normalization has learnable parameters—eventually, the γ and β should allow the activation distributions to converge to the optimum.

[1] The data here is from a separate run, not from the first trial.

We note that these experiments have only been run for resnet18, prompting a demand for many more experiments not only with larger ResNet models but also with different architectures.

In both trials, we see that the experiments follow roughly the same trajectory. This is expected, as we are merely changing the standard deviation, and considering the factor of $\gamma^{(k)}$ in Equation 1 all models should eventually correct for any initially set standard deviation (if it were optimal to do so). Furthermore, we had made sure that our deep learning framework would not train probabilistically; we fixed all known-to-be-used pseudorandom generator's seeds to 0, among other standard procedures detailed in the PyTorch documentation.

6 Conclusion

In this paper, we laid out a simple proposition: that the reason batch normalization benefits training is because of its interaction with the very realm of representing real numbers on a computer. Namely, we hypothesized that batch normalization works because it normalizes the distribution of activations i.e. "the number range" to an optimal one. Further work is required in many different directions for a complete analysis of this previously overlooked interaction. The reason why batchnorm (of zero mean and unit standard deviation) worked till now was indeed merely by chance (as many good things are), because 1 happened to be close to the true optimum, which we believe is in the interval $(1, 4)$, at least for the case of ResNet-18.

Acknowledgements

I would like to acknowledge the hard work of Bacchus, the server administrator group of the Department of Computer Science and Engineering at Seoul National University, whose well-maintained computing machines I ran my experiments on.

References

[IS15] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG]. URL: https://arxiv.org/abs/1502.03167.

[San+19] Shibani Santurkar et al. *How Does Batch Normalization Help Optimization?* 2019. arXiv: 1805.11604 [stat.ML]. URL: https://arxiv.org/abs/1805.11604.

[Wik25] Wikipedia contributors. *Bfloat16 floating-point format — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Bfloat16_floating-point_format&oldid=1315813313. [Online; accessed 22-October-2025]. 2025.

[Yan+19] Greg Yang et al. *A Mean Field Theory of Batch Normalization*. 2019. arXiv: 1902.08129 [cs.NE]. URL: https://arxiv.org/abs/1902.08129.