Revisit Batch Normalization: New Understanding and Refinement via Composition Optimization

Xiangru Lian
Department of Computer Science
University of Rochester
admin@mail.xrlian.com

Ji Liu
Department of Computer Science
University of Rochester
ji.liu.uwisc@gmail.com

Abstract

Batch Normalization (BN) has been used extensively in deep learning to achieve faster training process and better resulting models. However, whether BN works strongly depends on how the batches are constructed during training, and it may not converge to a desired solution if the statistics on the batch are not close to the statistics over the whole dataset. In this paper, we try to understand BN from an optimization perspective by providing an explicit objective function associated with BN. This explicit objective function reveals that: 1) BN, rather than being a new optimization algorithm or trick, is creating a different objective function instead of the one in our common sense; and 2) why BN may not work well in some scenarios. We then propose a refinement of BN based on the compositional optimization technique called Full Normalization (FN) to alleviate the issues of BN when the batches are not constructed ideally. The convergence analysis and empirical study for FN are also included in this paper.

1 Introduction

Batch Normalization (BN) [Ioffe and Szegedy, 2015] has been used extensively in deep learning [Szegedy et al., 2016, He et al., 2016, Silver et al., 2017, Huang et al., 2017, Hubara et al., 2017] to accelerate the training process. During the training process, a BN layer normalizes its input by the mean and variance computed within a mini-batch. Many state-of-the-art deep learning models are based on BN such as ResNet [He et al., 2016, Xie et al., 2017] and Inception [Szegedy et al., 2016, 2017]. It is often believed that BN can mitigate the exploding/vanishing gradients problem [Cooijmans et al., 2016] or reduce internal variance [Ioffe and Szegedy, 2015]. Therefore, BN has become a standard tool that is implemented almost in all deep learning solvers such as Tensorflow [Abadi et al., 2015], MXnet [Chen et al., 2015], Pytorch [Paszke et al., 2017], etc.

Despite the great success of BN in quite a few scenarios, people with rich experiences with BN may also notice some issues with BN. Some examples are provided below.

BN fails/overfits when the mini-batch size is 1 as shown in Figure 1 We construct a simple network with 3 layers for a classification task. It fails to learn a reasonable classifier on a dataset with only 3 samples as seen in Figure 1.

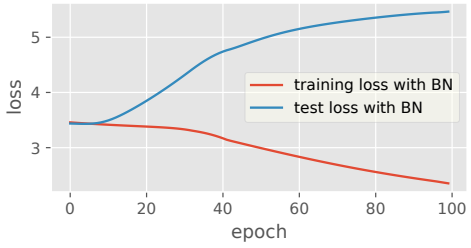


Figure 1: (**Fail when batch size is 1)** Given a dataset comprised of three samples: $[0, 0, 0]$ with label 0, $[1, 1, 1]$ with label 1 and $[2, 2, 2]$ with label 2, use the following simple network including one batch normalization layer, where the numbers in the parenthesis are the dimensions of input and output of a layer: linear layer $(3 \to 3) \Rightarrow$ batch normalization $\Rightarrow$ relu $\Rightarrow$ linear layer $(3 \to 3) \Rightarrow$ nll loss. Train with batch size 1, and test on the same dataset. The test loss increases while the training loss decreases.

Proceedings of the 22[nd] International Conference on Artificial Intelligence and Statistics (AISTATS) 2019, Naha, Okinawa, Japan. PMLR: Volume 89. Copyright 2019 by the author(s).

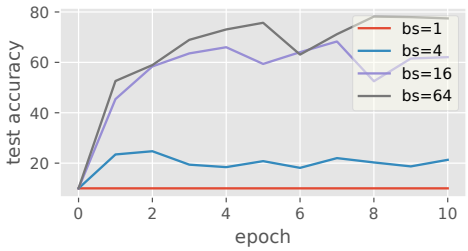


Figure 2: (**Sensitive to the size of mini-batch**) The test accuracy for ResNet18 on CIFAR10 dataset trained for 10 epochs with different batch sizes. The smaller the batch size is, with BN layers in ResNet, the worse the convergence result is.

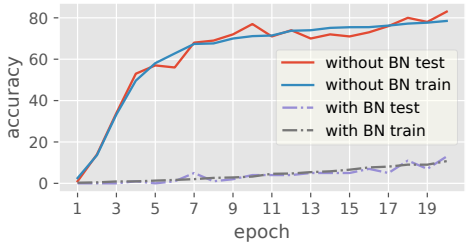


Figure 3: (**Fail if data are with large variation**) The training and test accuracy for a simple logistic regression problem on synthesized dataset with mini-batch size 20. We synthesize 10,000 samples where each sample is a vector of 1000 elements. There are 1000 classes. Each sample is generated from a zero vector by firstly randomly assigning a class to the sample (for example it is the i-th class), and then setting the i-th element of the vector to be a random number from 0 to 50, and finally adding noise (generated from a standard normal distribution) to each element. We generate 100 test samples in the same way. A logistic regression classifier should be able to classify this dataset easily. However, if we add a BN layer to the classifier, the model no longer converges.

BN's solution is sensitive to the mini-batch size as shown in Figure 2 The test conducted in Figure 2 uses ResNet18 on the Cifar10 dataset. When the batch size changes, the neural network with BN training tends to converge to a different solution. Indeed, this observation is true even for solving convex optimization problems. It can also be observed that the smaller the batch size, the worse the performance of the solution.

BN fails if data are with large variation as shown in Figure 3 BN breaks convergence on simple convex logistic regression problems if the variance of the dataset is large. Figure 3 shows the first 20 epochs of such training on a synthesized dataset. This phenomenon also often appears when using distributed training algorithms where each worker only has its local dataset, and the local datasets are very different.

Therefore, these observations may remind people to ask some fundermental questions:

1. Does BN always converge and/or where does it converge to?
2. Using BN to train the model, why does sometimes severe over-fitting happen?
3. Is BN a trustable "optimization" algorithm?

In this paper, we aim at understanding the BN algorithm from a rigorous optimization perspective to show

1. BN always converges, but not solving either the objective in our common sense, or the optimization originally motivated.
2. The result of BN heavily depends on how to construct batches, and it can overfit predefined batches. BN treats the training and the inference differently, which makes the situation worse.
3. BN is not always trustable, especially when the batches are not constructed with randomly selected samples or the batch size is small.

Besides these, we also provide the explicit form of the original objective BN aims to optimize (but does not in fact), and propose a Multilayer Compositional Stochastic Gradient Descent (MCSGD) algorithm based on the compositional optimization technology to solve the original objective. We prove the convergence of the proposed MCSGD algorithm and empirically study its effectiveness to refine the state-of-the-art BN algorithm.

2 Related work

We first review *traditional normalization* techniques. LeCun et al. [1998] showed that normalizing the input dataset makes training faster. In deep neural networks, normalization was used before the invention of BN. For example Local Response Normalization (LRU) [Lyu and Simoncelli, 2008, Jarrett K et al., 2009, Krizhevsky et al., 2012] which computes the statistics for the neighborhoods around each pixel.

We then review *batch normalization* techniques. Ioffe and Szegedy [2015] proposed the Batch Normalization (BN) algorithm which performs normalization along the batch dimension. It is more global than LRU and can be done in the middle of a neural network. Since during inference there is no "batch", BN uses the running average of the statistics during training to perform inference, which introduces unwanted bias between training and testing. While this paper tries to study why BN fails in some scenarios and how to fix it, Santurkar et al. [2018], Bjorck et al. [2018], Kohler et al. [2018] provided new understandings about why BN accelerates training by analyzing simplified neural

networks or the magnitude of gradients. Ioffe [2017] proposed Batch Renormalization (BR) that introduces two extra parameters to reduce the drift of the estimation of mean and variance. However, Both BN and BR are based on the assumption that the statistics on a mini-batch approximates the statistics on the whole dataset.

Next we review the *instance based normalization* which normalizes the input sample-wise, instead of using the batch information. This includes Layer Normalization [Ba et al., 2016], Instance Normalization [Ulyanov et al., 2016], Weight Normalization [Salimans and Kingma, 2016] and a recently proposed Group Normalization [Wu and He, 2018]. They all normalize on a single sample basis and are less global than BN. They avoid doing statistics on a batch, which work well when it comes to some vision tasks where the outputs of layers can be of handreds of channels, and doing statistics on the outputs of a single sample is enough. However we still prefer BN when it comes to the case that the single sample statistics is not enough and more global statistics is needed to increase accuracy.

Finally we review the *compositional optimization* which our refinement of BN is based on. Wang and Liu [2016] proposed the compositional optimization algorithm which optimizes the nested expectation problem $\min \mathbb{E}f(\mathbb{E}g(\cdot))$, and later the convergence rate was improved in Wang et al. [2016]. The convergence of compositional optimization on nonsmooth regularized problems was shown in Huo et al. [2017] and a variance reduced variant solving strongly convex problems was analyzed in Lian et al. [2016].

3 Review the BN algorithm

In this section we review the Batch Normalization (BN) algorithm [Ioffe and Szegedy, 2015].

BN is usually implemented as an additional layer in neural networks. In each iteration of the training process, the BN layer normalizes its input using the mean and variance of each channel of the input batch to make its output having zero mean and unit variance. The mean and variance on the input batch is expected to be similar to the mean and variance over the whole dataset. In the inference process, the layer normalizes its input's each channel using the saved mean and variance, which are the running averages of mean and variance calculated during training. This is described in Algorithm 1[1]. With BN, the input at the BN layer is normalized so that the next layer in the network accepts inputs that are easier to train on. In practice it has been observed in many applications that the speed of training is improved by using BN.

[1] A linear transformation is often added after applying BN to compensate the normalization.

Algorithm 1 Batch Normalization Layer

TRAINING

Require: Input $\mathbf{B}_{\text{in}}$, which is a batch of input. Estimated mean μ and variance ν, and averaging constant α.

1:

$$\mu \leftarrow (1-\alpha)\cdot\mu + \alpha\cdot \boldsymbol{mean}(\mathbf{B}_{\text{in}}),$$
$$\nu \leftarrow (1-\alpha)\cdot\nu + \alpha\cdot \boldsymbol{var}(\mathbf{B}_{\text{in}}).$$

▷ $\boldsymbol{mean}(\mathbf{B}_{\text{in}})$ and $\boldsymbol{var}(\mathbf{B}_{\text{in}})$ calculate the mean and variance of $\mathbf{B}_{\text{in}}$ respectively.

2: **Output**

$$\mathbf{B}_{\text{out}} \leftarrow \frac{\mathbf{B}_{\text{in}} - \boldsymbol{mean}(\mathbf{B}_{\text{in}})}{\sqrt{\boldsymbol{var}(\mathbf{B}_{\text{in}}) + \epsilon}}.$$

▷ ϵ is a small constant for numerical stability.

INFERENCE

Require: Input $\mathbf{B}_{\text{in}}$, estimated mean μ and variance ν. A small constant ϵ for numerical stability.

1: **Output**

$$\mathbf{B}_{\text{out}} \leftarrow \frac{\mathbf{B}_{\text{in}} - \mu}{\sqrt{\nu + \epsilon}}.$$

4 Training objective of BN

In the original paper proposing BN algorithm [Ioffe and Szegedy, 2015], authors do not provide the explicit optimization objective BN targets to solve. Therefore, many people may naturally think that BN is an optimization trick, that accelerates the training process but still solves the original objective in our common sense. Unfortunately, this is not true! The actual objective BN solves is different from the objective in our common sense and also nontrivial. In this section, we derive the objective function which is actually optimized when BN layers are used. Then we discuss how this objective is different from the objective without BN layers and when the BN's objective is less desired.

Rigorous mathematical description of BN layer To define the objective of BN in a precise way, we need to define the normalization operator $f_W^{B,\sigma}$ that maps a *function* to a *function* associating with a mini-batches B, an activation function σ, and parameters W. Let $g(\cdot)$ be a function $g(\cdot) = [g_1(\cdot) \quad g_2(\cdot) \quad \cdots \quad g_n(\cdot) \quad 1]^\top$ where $g_1(\cdot), \cdots, g_n(\cdot)$ are functions mapping a vector to a number. The operator $f_W^{B,\sigma}$ is defined by

$$\underbrace{f_W^{B,\sigma}\overbrace{(g)}^{f_W^{B,\sigma}\text{'s argument is a function } g}}_{f_W^{B,\sigma}(g) \text{ is another function}}(\cdot) := \sigma\left(W\begin{pmatrix} \frac{g_1(\cdot)-\mathbf{mean}(g_1,B)}{\sqrt{\mathbf{var}(g_1,B)}} \\ \frac{g_2(\cdot)-\mathbf{mean}(g_2,B)}{\sqrt{\mathbf{var}(g_2,B)}} \\ \vdots \\ \frac{g_n(\cdot)-\mathbf{mean}(g_n,B)}{\sqrt{\mathbf{var}(g_n,B)}} \\ 1 \end{pmatrix}\right) \quad (1)$$

where $\mathbf{mean}(r, B)$ is defined by $\mathbf{mean}(r, B) := \frac{1}{|B|}\sum_{b\in B} r(b)$. and $\mathbf{var}(r, B)$ is defined by $\mathbf{var}(r, B) = \mathbf{mean}(r^2, B) - \mathbf{mean}(r, B)^2$.

Note that the first augment of $\mathbf{mean}(\cdot,\cdot)$ and $\mathbf{var}(\cdot,\cdot)$ is a function, and the second augment is a set of samples. A m-layer's neural network with BN can be represented by a function

$$F^B_{\{W_i\}_{i=1}^m}(\cdot) := f^{B,\sigma_m}_{W_m}(f^{B,\sigma_{m-1}}_{W_{m-1}}(\cdots(f^{B,\sigma_1}_{W_1}(I))))(\cdot) \quad \text{or}$$
$$f^{B,\sigma_m}_{W_m} \circ f^{B,\sigma_{m-1}}_{W_{m-1}} \circ \cdots \circ f^{B,\sigma_1}_{W_1}(I)(\cdot)$$

where $I(\cdot)$ is the identical mapping function from a vector to itself, that is, $I(x) = x$.

BN's objective is very different from the objective in our common sense Using the same way, we can represent a fully connected network in our common sense (without BN) by $F_{\{W_i\}_{i=1}^m}(\cdot) := \bar{f}^{\sigma_m}_{W_m} \circ \bar{f}^{\sigma_{m-1}}_{W_{m-1}} \circ \cdots \circ \bar{f}^{\sigma_1}_{W_1}(I)(\cdot)$ where operator $\bar{f}^{\sigma}_{W}$ is defined by

$$\bar{f}^{\sigma}_{W}(g)(\cdot) := \sigma(Wg(\cdot)). \quad (2)$$

Besides of the difference of operator function definitions, their ultimate objectives are also different. Given the training date set $\mathscr{D}$, the objective without BN (or the objective in our common sense) is defined by

$$\min_{\{W_j\}_{j=1}^m} \quad \frac{1}{|\mathscr{D}|}\sum_{(\mathbf{x},y)\in\mathscr{D}} l(F_{\{W_j\}_{j=1}^m}(\mathbf{x}), y), \quad (3)$$

where $l(\cdot,\cdot)$ is a predefined loss function, while the objective with BN (or the objective BN is actually solving) is

$$(\mathbf{BN}) \quad \min_{\{W_j\}_{j=1}^m} \quad \frac{1}{|\mathscr{B}|}\sum_{B\in\mathscr{B}}\frac{1}{|B|}\sum_{(\mathbf{x},y)\in B} l(F^B_{\{W_j\}_{j=1}^m}(\mathbf{x}), y), \quad (4)$$

where $\mathscr{B}$ is the set of batches.

Therefore, the objective of BN could be very different from the objective in our common sense in general.

BN could be very sensitive to the sample strategy and the minibatch size The super set $\mathscr{B}$ has different forms depending on how to define mini-batchs. For example,

- People can choose b as the size of minibatch, and form $\mathscr{B}$ by $\mathscr{B} := \{B \subset \mathscr{D} : |B| = b\}$. Choosing $\mathscr{B}$ in this way implicitly assumes that all nodes can access the same dataset, which may not be true in practice.
- If data is distributed ($\mathscr{D}_i$ is the local dataset on the i-th worker, disjoint with others, satisfying $\mathscr{D} = \mathscr{D}_1 \cup \cdots \cup \mathscr{D}_n$), a typical $\mathscr{B}$ is defined by $\mathscr{B} = \mathscr{B}_1 \cup \mathscr{B}_2 \cup \cdots \cup \mathscr{B}_n$ with $\mathscr{B}_i$ defined by $\mathscr{B}_i := \{B \subset \mathscr{D}_i : |B| = b\}$.
- When the mini-batch is chosen to be the whole dataset, $\mathscr{B}$ contains only one element $\mathscr{D}$.

After figure out the implicit objective BN optimizes in (4), it is not difficult to have following key observations

- The BN objectives vary a lot when different sampling strategies are applied. This explains why the convergent solution could be very different when we change the sampling strategy;
- For the same sample strategy, BN's objective function also varies if the size of minibatch gets changed. This explains why BN could be sensitive to the batch size.

The observations above may seriously bother us how to appropriately choose parameters for BN, since it does not optimize the objective in our common sense, and could be sensitive to the sampling strategy and the size of minibatch.

BN's objective (4) **with** $\mathscr{B} = \{\mathscr{D}\}$ **has the same optimal value as the original objective** (3) When the batch size is equal to the size of the whole dataset in (4), the objective becomes

$$(\mathbf{FN}) \quad \min_{\{W_j\}_{j=1}^m} \quad \frac{1}{|\mathscr{D}|}\sum_{(\mathbf{x},y)\in\mathscr{D}} l(F^{\mathscr{D}}_{\{W_j\}_{j=1}^m}(\mathbf{x}), y), \quad (5)$$

which differs from the original objective (3) only in the first argument of l. Noting the only difference between (1) and (2) when B is a constant $\mathscr{D}$ is a linear transformation which can be absorbed into W:

$$f^{\mathscr{D},\sigma}_W(g)(\cdot) = \sigma\left(W\begin{pmatrix} \frac{g_1(\cdot)-\mathbf{mean}(g_1,\mathscr{D})}{\sqrt{\mathbf{var}(g_1,\mathscr{D})}} \\ \frac{g_2(\cdot)-\mathbf{mean}(g_2,\mathscr{D})}{\sqrt{\mathbf{var}(g_2,\mathscr{D})}} \\ \vdots \\ \frac{g_n(\cdot)-\mathbf{mean}(g_n,\mathscr{D})}{\sqrt{\mathbf{var}(g_n,\mathscr{D})}} \\ 1 \end{pmatrix}\right)$$

$$= \sigma\left(\underbrace{W\begin{pmatrix} \frac{1}{\sqrt{\mathbf{var}(g_1,\mathscr{D})}} & & & \frac{-\mathbf{mean}(g_1,\mathscr{D})}{\sqrt{\mathbf{var}(g_1,\mathscr{D})}} \\ & \frac{1}{\sqrt{\mathbf{var}(g_2,\mathscr{D})}} & & \frac{-\mathbf{mean}(g_2,\mathscr{D})}{\sqrt{\mathbf{var}(g_2,\mathscr{D})}} \\ & & \ddots & \vdots \\ & & \frac{1}{\sqrt{\mathbf{var}(g_n,\mathscr{D})}} & \frac{-\mathbf{mean}(g_n,\mathscr{D})}{\sqrt{\mathbf{var}(g_n,\mathscr{D})}} \\ & & & 1 \end{pmatrix}}_{=:W'}\begin{pmatrix} g_1(\cdot) \\ g_2(\cdot) \\ \vdots \\ g_n(\cdot) \\ 1 \end{pmatrix}\right)$$

$$=\sigma\left(W'\left(g_1(\cdot)\ g_2(\cdot)\cdots\ g_n(\cdot)\ 1\right)^\top\right).$$

If we use this W' as our new W, (3) and (4) has the same form and the two objectives should have the same optimal value, which means when $\mathscr{B}=\{\mathscr{D}\}$ adding BN does not hurt the expressiveness of the network. However, since each layer's input has been normalized, in general the condition number of the objective is reduced, and thus easier to train.

5 Solving full normalization objective (5) via compositional optimization

The BN objective contains the normalization operator which mostly reduces the condition number of the objective that can accelerate the training process. While the FN formulation in (5) provides a more stable and trustable way to define the BN formation, it is very challenging to solve the FN formulation in practice. If follow the standard SGD used in solving BN to solve (5), then every single iteration needs to involve the whole dataset since $B=\mathscr{D}$, which is almost impossible in practice. The key difficulty to solve (5) lies on that there exists "expectation" in each layer, which is very expensive to compute even we just need to compute a stochastic gradient if use the standard SGD method.

To solve (5) efficiently, we follow the spirit of compositional optimization to develop a new algorithm namely, Multilayer Compositional Stochastic Gradient Descent (Algorithm 2). The proposed algorithm does not have any requirement on the size of minibatch. The key idea is to estimate the expectation from the current samples and historical record.

5.1 Formulation

To propose our solution to solve (5), let us define the objective in a more general but neater way in the following. When $\mathscr{B}=\{\mathscr{D}\}$, (4) is a special case of the following general objective:

$$\min_w f(w):=\mathbb{E}_\xi\left[F_1^{w_1,\mathscr{D}}\circ F_2^{w_2,\mathscr{D}}\circ\cdots\circ F_n^{w_n,\mathscr{D}}\circ I(\xi)\right], \tag{6}$$

where ξ represents a sample in the dataset $\mathscr{D}$, for example it can be the pair $(\mathbf{x},y)$ in (4). w_i represents the parameters of the i-th layer. w represents all parameters of the layers: $w:=(w_1,\ldots,w_n)$. Each F_i is an operator of the form:

$$F_i^{w_i,\mathscr{D}}(g)(\cdot):=F_i(w_i;g(\cdot);\mathbb{E}_{\xi\in\mathscr{D}}e_i(g(\xi))),$$

where e_i is used to compute statistics over the dataset. For example, with $e_i(x)=[x,x^2]$, the mean and variance of the layer's input over the dataset can be calculated, and $F_i^{w_i,\mathscr{D}}$ can use that information, for example, to perform normalization.

There exist compositional optimization algorithms [Wang and Liu, 2016, Wang et al., 2016] for solving (6) when $n=2$, but for $n>2$ we still do not have a good algorithm to solve it. We follow the spirit to extend the compositional optimization algorithms to solve the general optimization problem (6) as shown in Algorithm 2. See Section 6 for an implementation when it comes to normalization.

To simplify notation, given $w=(w_1,\ldots,w_n)$ we define:

$$\mathbf{e}_i(w;\xi):=e_i(F_{i+1}^{w_{i+1},\mathscr{D}}\circ F_{i+2}^{w_{i+2},\mathscr{D}}\circ\cdots\circ F_n^{w_n,\mathscr{D}}\circ I(\xi)).$$

We define the following operator if we already have an estimation, say $\hat{e}_i$, of $\mathbb{E}_{\xi\in\mathscr{D}}\mathbf{e}_i(w;\xi)$:

$$\hat{F}_i^{w_i,\hat{e}_i}(g)(\cdot):=F_i(w_i;g(\cdot);\hat{e}_i).$$

Given $w=(w_1,\ldots,w_n)$ and $\hat{e}=(\hat{e}_1,\ldots,\hat{e}_n)$, we define

$$\hat{\mathbf{e}}_i(w;\xi;\hat{e}):=e_i(\hat{F}_{i+1}^{w_{i+1},\hat{e}_{i+1}}\circ\hat{F}_{i+2}^{w_{i+2},\hat{e}_{i+2}}\circ\cdots\circ\hat{F}_n^{w_n,\hat{e}_n}\circ I(\xi)).$$

Algorithm 2 Multilayer Compositional Stochastic Gradient Descent (MCSGD) algorithm

Require: Learning rate $\{\gamma_k\}_{k=0}^K$, approximation rate $\{\alpha_k\}_{k=0}^K$, dataset $\mathscr{D}$, initial point $w^{(0)}:=(w_1^{(0)},\ldots,w_n^{(0)})$ and initial estimations $\hat{e}^{(0)}:=(\hat{e}_1^{(0)},\ldots,\hat{e}_n^{(0)})$.

1: **for** $k=0,1,2,\ldots,K$ **do**

2: For each i, randomly select a sample ξ_k, estimate $\mathbb{E}_\xi[\mathbf{e}_i(w^{(k)};\xi)]$ by

$$\hat{e}_i^{(k+1)}\leftarrow(1-\alpha_k)\hat{e}_i^{(k)}+\alpha_k\hat{\mathbf{e}}_i(w^{(k)};\xi_k;\hat{e}^{(k)}).^a$$

3: Ask the oracle ⑥ using $w^{(k)}$ and estimated means $\hat{e}^{(k+1)}$ to obtain the approximated gradient at $w^{(k)}$: $g^{(k)}$.

▷ See Remark 1 for discussion on the oracle.

4: $w^{(k+1)}\leftarrow w^{(k)}-\gamma_k g^{(k)}$.

5: **end for**

[a] This step is borrowed from compositional optimization to estimate the statistics across the whole dataset.

Remark 1 (MCSGD oracle)**.** *In MCSGD, the gradient oracle takes the current parameters of the model, and the current estimation of each $\mathbb{E}_\xi[\mathbf{e}_i(w^{(k)};\xi)]$, to output an estimation of a stochastic gradient.*

For example for an objective like (6)*, the derivative of the loss function w.r.t. the i-th layer's parameters is*

$$\begin{aligned}\partial_{w_i}f(w)=&\partial_{w_i}(\mathbb{E}_\xi[F_1^{w_1,\mathscr{D}}\circ F_2^{w_2,\mathscr{D}}\circ\cdots\circ F_n^{w_n,\mathscr{D}}\circ I(\xi)])\\=&\mathbb{E}_\xi\begin{bmatrix}[\partial_x(F_1^{w_1,\mathscr{D}}(x)(\xi))]_{x=F_2^{w_2,\mathscr{D}}\circ\cdots\circ F_n^{w_n,\mathscr{D}}\circ I}\\\cdot[\partial_x(F_2^{w_2,\mathscr{D}}(x)(\xi))]_{x=F_3^{w_3,\mathscr{D}}\circ\cdots\circ F_n^{w_n,\mathscr{D}}\circ I}\\\cdots\cdot[\partial_x(F_{i-1}^{w_{i-1},\mathscr{D}}(x)(\xi))]_{x=F_i^{w_i,\mathscr{D}}\circ\cdots\circ F_n^{w_n,\mathscr{D}}\circ I}\\\cdot[\partial_{w_i}(F_i^{w_i,\mathscr{D}}(x)(\xi))]_{x=F_i^{w_{i+1},\mathscr{D}}\circ\cdots\circ F_n^{w_n,\mathscr{D}}\circ I}\end{bmatrix},\end{aligned} \tag{7}$$

where for any $j \in [1, \dots, i-1]$:

$$
\begin{aligned}
&[\partial_x F_j^{w_j,\mathcal{D}}(x)(\xi)]_{x=F_{j+1}^{w_{j+1},\mathcal{D}}\circ\cdots\circ F_n^{w_n,\mathcal{D}}\circ I} \\
=&\begin{bmatrix} \partial_x F_j(w_j;x;y)+ \\ \partial_y F_j(w_j;x;y)\cdot \mathbb{E}_{\xi'\sim\mathcal{D}} D_j(\xi') \end{bmatrix} \\
&\text{with } \begin{array}{l} x = F_{j+1}^{w_{j+1},\mathcal{D}} \circ \cdots \circ F_n^{w_n,\mathcal{D}} \circ I(\xi), \\ y = \mathbb{E}_\xi[\mathbf{e}_j(w;\xi)], \\ D_j(\xi') = [\partial_z e_i(z)]_{z=F_{j+1}^{w_{j+1},\mathcal{D}}\circ\cdots\circ F_n^{w_n,\mathcal{D}}\circ I(\xi')} \end{array}.
\end{aligned}
$$

The oracle samples a ξ' for each layer and calculate this derivative with y set to an estimated value $\hat{e}_j$, and returns the stochastic gradient. The expectation of this stochastic gradient will be (7) with some error caused by the difference between the estimated $\hat{e}_j$ and the true expectation $\mathbb{E}_\xi[\mathbf{e}_j(w;\xi)]$. The more accurate the estimation is, the closer the expectation of this stochastic and the true gradient (7) will be.

In practice, we often use the same ξ' for each layer to save some computation resource, this introduce some bias the expectation of the estimated stochastic gradient. One such implementation can be found in Algorithm 3.

5.2 Theoretical analysis

In this section we analyze Algorithm 2 to show its convergence when applied on (6) (a generic version of the FN formulation in (5)). The detailed proof is provided in the supplementary material. We use the following assumptions as shown in Assumption 1 for the analysis.

Assumption 1. *1. The gradients $g^{(k)}$'s are bounded: $\|g^{(k)}\| \leqslant \mathcal{G}, \forall k$ for some constant $\mathcal{G}$.*

2. The variance of all e_i's are bounded:

$$\mathbb{E}_\xi\|\mathbf{e}_i(w;\xi) - \mathbb{E}_\xi\mathbf{e}_i(w;\xi)\|^2 \leqslant \sigma^2, \forall i, w,$$

$$\mathbb{E}_\xi\|\mathbf{e}_i(w;\xi;\hat{e}) - \mathbb{E}_\xi\mathbf{e}_i(w;\xi;\hat{e})\|^2 \leqslant \sigma^2, \forall i, w, \hat{e}$$

for some constant σ.

3. The error of approximated gradient $\mathbb{E}[g^{(k)}]$ is proportional to the error of approximation for $\mathbb{E}[e_i]$:

$$
\begin{aligned}
&\|\mathbb{E}_{\xi_k}[g^{(k)}] - \nabla f(w^{(k)})\|^2 \\
\leqslant& L_g \sum_{i=1}^n \|\hat{e}_i^{(k+1)} - \mathbb{E}_{\xi_k}[\mathbf{e}_i(w^{(k)};\xi_k)]\|^2, \forall k
\end{aligned}
$$

for some constant L_g.

4. All functions and their first order derivatives are Lipschitzian with Lipschitz constant L.

5. The minimum of the objective $f(w)$ is finite.

6. γ_k, α_k are monotonically decreasing. $\gamma_k = O(k^{-\gamma})$ and $\alpha_k = O(k^{-a})$ for some constants $\gamma > a > 0$.

It can be shown under the given assumptions, the approximation errors will vanish shown in Lemma 1.

Lemma 1 (*Approximation error*)

Choose the learning rate γ_k and α_k in Algorithm 2 in the form defined in Assumption 1-6 with parameters γ and a. Under Assumption 1, the sequence generated in Algorithm 2 satisfies

$$
\begin{aligned}
&\mathbb{E}\|\hat{e}_i^{(k+1)} - \mathbb{E}_\xi\mathbf{e}_i(w^{(k)};\xi)\|^2 \\
\leqslant& \mathcal{E}(k^{-2\gamma+2a+\varepsilon} + k^{-a+\varepsilon}), \quad \forall i, \forall k, \qquad (8) \\
&\mathbb{E}\|\hat{e}_i^{(k+1)} - \mathbb{E}_\xi\mathbf{e}_i(w^{(k)};\xi)\|^2 \\
\leqslant& \left(1 - \frac{\alpha_k}{2}\right)\mathbb{E}\|\hat{e}_i^{(k)} - \mathbb{E}_\xi\mathbf{e}_i(w^{(k-1)};\xi)\|^2 \\
&+ \mathcal{C}(k^{-2\gamma+a+\varepsilon} + k^{-2a+\varepsilon}), \quad \forall i, \forall k. \qquad (9)
\end{aligned}
$$

for any ε satisfying $1 - a > \varepsilon > 0$, where $\mathcal{E}$ and $\mathcal{C}$ are two constants independent of k.

Then it can be shown that on (6) Algorithm 2 converges as seen in Theorem 2 and Corollary 3. It is worth noting that the convergence rate in Corollary 3 is slower than the $\frac{1}{\sqrt{K}}$ convergence rate of SGD without any normalization. This is due to the estimation error. If the estimation error is small (for example, the samples in a batch are randomly selected and the batch size is large), the convergence will be fast.

Theorem 2 (*Convergence*)

Choose the learning rate γ_k and α_k in Algorithm 2 in the form defined in Assumption 1-6 with parameters γ and a satisfying $a < 2\gamma - 1, a < 1/2$, and $\frac{\gamma_k L_g}{\alpha_{k+1}} \leqslant \frac{1}{2}$. Under Assumption 1, for any integer $K > 1$ the sequence generated by Algorithm 2 satisfies

$$\frac{\sum_{k=0}^K \gamma_k \mathbb{E}\|\partial f(w^{(k)})\|^2}{\sum_{k=0}^K \gamma_k} \leqslant \frac{\mathcal{H}}{\sum_{k=0}^K \gamma_k},$$

where $\mathcal{H}$ is a constant independent of K.

We next specify the choice of γ and a to give a more clear convergence rate for our MCSGD algorithm in Corollary 3.

Corollary 3

Choose the learning rate γ_k and α_k in Algorithm 2 in the form defined in Assumption 1-6, more specifically, $\gamma_k = \frac{1}{2L_g}(k+2)^{-4/5}$ and $\alpha_k = (k+1)^{-2/5}$. Under Assumption 1, for any integer $K > 1$ the sequence generated in Algorithm 2 satisfies

$$\frac{\sum_{k=0}^K \mathbb{E}\|\partial f(w^{(k)})\|^2}{K+2} \leqslant \frac{\mathcal{H}}{(K+2)^{1/5}},$$

where $\mathcal{H}$ is a constant independent of K.

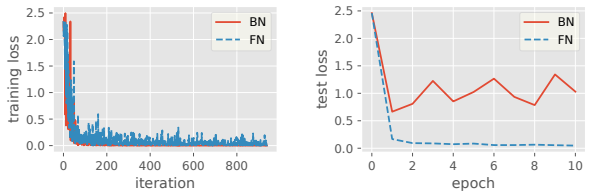


Figure 4: The training and testing error for the given model trained on MNIST with batch size 64, learning rate 0.01 and momentum 0.5 using BN or FN for the case where all samples in a batch are of a single label. The approximation rate α in FN is $(\frac{k}{20}+1)^{-0.4}$ where k is the iteration number.

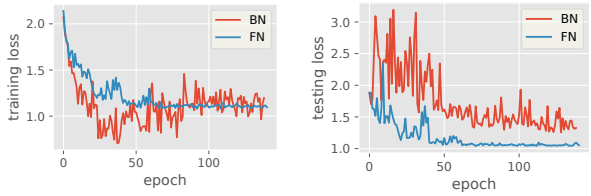


Figure 5: The training and testing error for the given model trained on CIFAR10 with batch size 64, learning rate 0.01 and momentum 0.9 using BN and FN for the case where all samples in a batch are of no more than 3 labels. The learning rate is decreased by a factor of 5 for every 20 epochs. The approximation rate α in FN is $(\frac{k}{5}+1)^{-0.3}$ where k is the iteration number.

Algorithm 3 Full Normalization (FN) layer

FORWARD PASS

Require: Learning rate γ, approximation rate α, input $\mathbf{B}_{\text{in}} \in \mathbb{R}^{b\times d}$, mean estimation μ, and mean of square estimation ν. ▷ b is the batch size, and d is the dimension of each sample.

1: **if** training **then**

$$\mu \leftarrow (1-\alpha)\mu + \alpha \textbf{\textit{mean}}(\mathbf{B}_{\text{in}}),$$
$$\nu \leftarrow (1-\alpha)\nu + \alpha \textbf{\textit{mean\_of\_square}}(\mathbf{B}_{\text{in}}).$$

2: **end if**

3: **return** layer output:

$$\mathbf{B}_{\text{output}} \leftarrow \frac{\mathbf{B}_{\text{in}} - \mu}{\sqrt{\max\{\nu-\mu^2, \epsilon\}}}.$$

▷ ϵ is a small constant for numerical stability.

BACKWARD PASS

Require: Mean estimation μ, mean of square estimation ν, the gradient at the output g_{out}, and input $\mathbf{B}_{\text{in}} \in \mathbb{R}^{b\times d}$.

1: Define

$$f(\mu, \nu, \mathbf{B}_{\text{in}}) := \frac{\mathbf{B}_{\text{in}} - \mu}{\sqrt{\max\{\nu-\mu^2, \epsilon\}}}.$$

2: **return** the gradient at the input:

$$g_{\text{in}} \leftarrow g_{\text{out}} \cdot \left(\partial_{\mathbf{B}_{\text{in}}} f + \frac{\partial_\mu f + 2\mathbf{B}_{\text{in}}\partial_\nu f}{bd}\right).$$

▷ $\partial_a f$ is the derivative of f w.r.t. a at $\mu, \nu, \mathbf{B}_{\text{in}}$.

6 Experiments

Experiments are conducted to validate the effectiveness of our MCSGD algorithm for solving the FN formulation. We consider two settings. The first one in Section 6.1 shows BN's convergence highly depends on the size of batches, while FN is more robust to different batch sizes. The second one in Section 6.2 shows that FN is more robust to different construction of mini-batches.

We use a simple neural network as the testing network whose architecture is shown in Figure 6. Steps 2 and 3 in Algorithm 2 are implemented in Algorithm 3. The *forward pass* essentially performs Step 2 of Algorithm 2, which estimates the mean and variance of layer inputs over the dataset. The *backward pass* essentially performs Step 3 of Algorithm 2, which gives the approximated gradient based on current network parameters and the estimations. Note that as discussed in Remark 1, for efficiency, in each iteration of this implementation we are using the same samples to do estimation in all normalization layers, which saves computational cost but brings additional bias.

6.1 Dependence on batch size

MNIST is used as the testing dataset with some modification by multiplying a random number in $(-2.5, 2.5)$ to each sample to make them more diverse. In this case, as shown in Figure 9, with batch size 1 and batch size 16, we can see the convergent points are different in BN, while the convergent results of FN are much more close for different batch sizes. Therefore, FN is more robust to the size of mini-batch.

6.2 Dependence on batch construction

We study two cases — shuffled case and unshuffled case. For the shuffled case, where the samples in a batch are randomly sampled, we expect the performance of FN matches BN's in this case. For the unshuffled case, where the batch contains only samples in just a few number of categories (so the mean and variance are very different from batch to batch). In this case we can observe that the FN outperforms BN.

Unshuffled case We show the FN has advantages over BN when the data variation is large among mini-batches. In the unshuffled setting, we do not shuffle the dataset. It means that the samples in the same mini-batch are mostly with the same labels. The comparison uses two datasets (MINST and CIFAR10):

- On MNIST, the batch size is chosen to be 64 and each batch only contains a single label. The convergence results are shown in Figure 4. We can observe

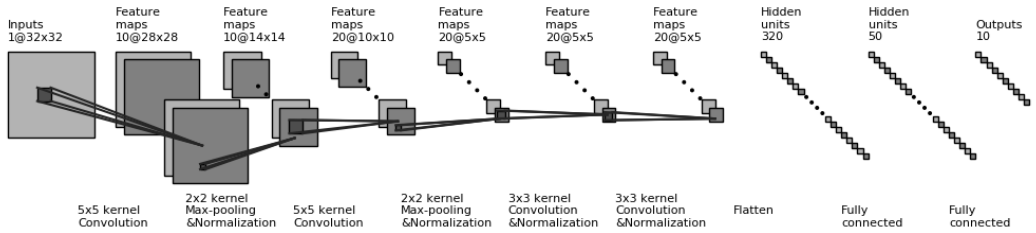


Figure 6: The simple neural network used in the experiments. The "Normalization" between layers can be BN or FN. The activation function is relu and the loss function is negative log likelihood.

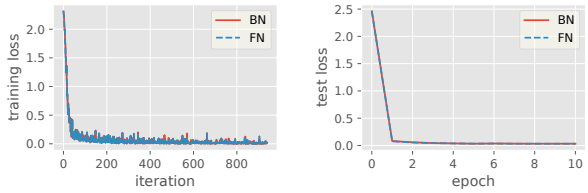


Figure 7: The training and testing error for the given model trained on MNIST with batch size 64, learning rate 0.01 and momentum 0.5 using BN or FN for the case where samples in a batch are randomly selected. The approximation rate α in FN is $(\frac{k}{20}+1)^{-0.4}$ where k is the iteration number.

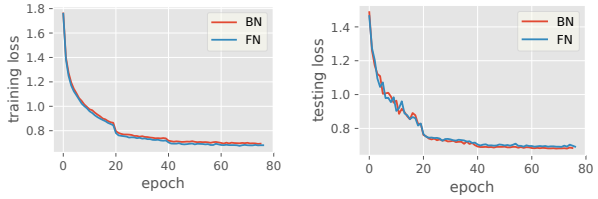
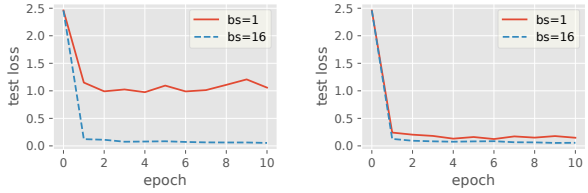


Figure 8: The training and testing error for the given model trained on CIFAR10 with batch size 64, learning rate 0.01 and momentum 0.9 using BN or FN for the case where samples in a batch are randomly selected. The learning rate is decreased by a factor of 5 for every 20 epochs. The approximation rate α in FN is $(\frac{k}{20}+1)^{-0.2}$ where k is the iteration number.



(a) With BN layers. (b) With FN layers.

Figure 9: The testing error for the given model trained on MNIST, where each sample is multiplied by a random number in $(-2.5, 2.5)$. The optimization algorithm is SGD with learning rate is 0.01 and momentum is 0.5. FN gives more consistent results under different batch sizes. The approximation rate α in FN is $(\frac{k}{20}+1)^{-0.4}$ where k is the iteration number.

from these figures that for the training loss, BN and FN converge equally fast (BN might be slightly faster). However since the statistics on any batch are very different from the whole dataset, the estimated mean and variance in BN are very different from the true mean and variance on the whole dataset, resulting in a very high test loss for BN.

- On CIFAR10, we observe similar results as shown in Figure 5. In this case, we restrict the number of labels in every batch to be no more than 3. Thus BN's performance on CIFAR10 is slightly better than on MNIST. We see the convergence efficiency of both methods is still comparable in term of the training loss. However, the testing error for BN is still far behind the FN.

Shuffled case For the shuffled case, where the samples in each batch are selected randomly, we expect BN to have similar performance as FN in this case, since the statistics in a batch is close to the statistics on the whole dataset. The results for MNIST are shown in Figure 7 and the results for CIFAR10 are shown in Figure 8. We observe the convergence curves of BN and FN for both training and testing loss match well.

7 Conclusion

We provide new understanding for BN from an optimization perspective by rigorously defining the optimization objective for BN. BN essentially optimizes an objective different from the one in our common sense. The implicitly targeted objective by BN depends on the sampling strategy as well as the minibatch size. That explains why BN becomes unstable and sensitive in some scenarios. The stablest objective of BN (called FN formulation) is to use the full dataset as the mini-batch, but it is very challenging to solve such formulation. To solve the FN objective, we follow the spirit of compositional optimization to develop MC-SGD algorithm to solve it efficiently.

Acknowledgment Xiangru Lian and Ji Liu are in part supported by NSF CCF1718513, IBM faculty award, and NEC fellowship.

References

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

N. Bjorck, C. P. Gomes, B. Selman, and K. Q. Weinberger. Understanding batch normalization. In *Advances in Neural Information Processing Systems*, pages 7705–7716, 2018.

T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

T. Cooijmans, N. Ballas, C. Laurent, Ç. Gülçehre, and A. Courville. Recurrent batch normalization. *arXiv preprint arXiv:1603.09025*, 2016.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *CVPR*, volume 1, page 3, 2017.

I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research*, 18: 187–1, 2017.

Z. Huo, B. Gu, and H. Huang. Accelerated method for stochastic composition optimization with nonsmooth regularization. *arXiv preprint arXiv:1711.03937*, 2017.

S. Ioffe. Batch renormalization: Towards reducing minibatch dependence in batch-normalized models. In *Advances in Neural Information Processing Systems*, pages 1942–1950, 2017.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

K. K. Jarrett K, A. Ranzato M, et al. What is the best multii stage architecture for object recognition? *Proceedings of the 2009 IEEE 12th International Conference on Computer Vision. Kyoto, Japan*, 2146:2153, 2009.

J. Kohler, H. Daneshmand, A. Lucchi, M. Zhou, K. Neymeyr, and T. Hofmann. Towards a theoretical understanding of batch normalization. *arXiv preprint arXiv:1805.10694*, 2018.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–50. Springer, 1998.

X. Lian, M. Wang, and J. Liu. Finite-sum composition optimization via variance reduced gradient descent. *arXiv preprint arXiv:1610.04674*, 2016.

S. Lyu and E. P. Simoncelli. Nonlinear image representation using divisive normalization. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

T. Salimans and D. P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in Neural Information Processing Systems*, pages 901–909, 2016.

S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry. How does batch normalization help optimization? In *Advances in Neural Information Processing Systems*, pages 2488–2498, 2018.

D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550 (7676):354, 2017.

C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, volume 4, page 12, 2017.

D. Ulyanov, A. Vedaldi, and V. Lempitsky. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022*, 2016.

M. Wang and J. Liu. A stochastic compositional gradient method using markov samples. In *Proceedings of the 2016 Winter Simulation Conference*, pages 702–713. IEEE Press, 2016.

M. Wang, J. Liu, and E. Fang. Accelerating stochastic composition optimization. In *Advances in Neural Information Processing Systems*, pages 1714–1722, 2016.

Y. Wu and K. He. Group normalization. *arXiv preprint arXiv:1803.08494*, 2018.

S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, pages 5987–5995. IEEE, 2017.