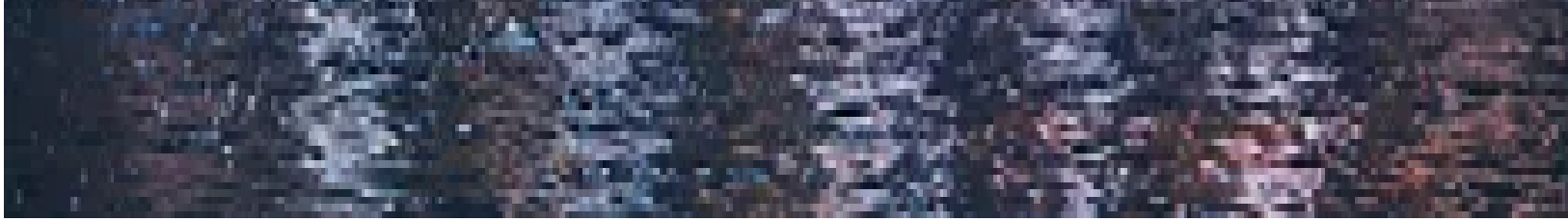
Batch normalization in 3 levels of understanding

What do we know about it so far: From a 30 seconds digest to a comprehensive guide

Johann Huber · Follow

Published in Towards Data Science · 28 min read · Nov 6, 2020



Credit : jabari timothy on Unsplash

There is a lot of content about Batch Normalization (BN) on the internet. Yet, many of them are defending an outdated intuition about it. I spent a lot of time putting all this scattered information together to build a good intuition about this fundamental method, I thought a step-by-step walkthrough could be useful to some readers.

In particular, this story aims to bring :

- An **updated explanation** of **Batch Normalization** through **3 levels of understanding** : in 30 seconds, 3 minutes, and a comprehensive guide ;
- Cover the **key elements** to have in mind to get the most out of BN ;
- **A simple implementation** of BN layer using Pytorch in Google Colab**,** reproducing MNIST-based experiments from the official paper (feel free to play around with the notebook) ;
- Key insights to understand **why BN is still poorly understood** (even after reading explanations from high quality authors !).

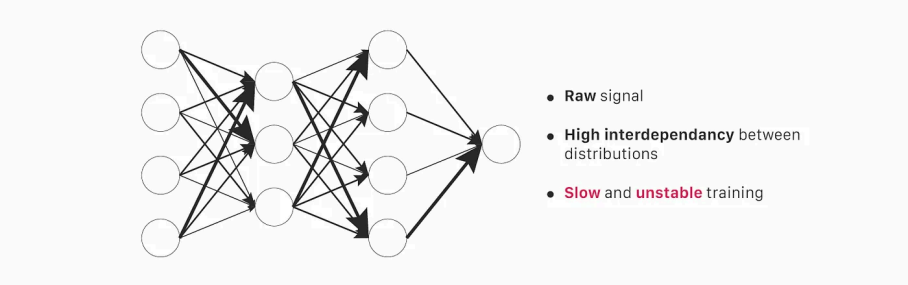
Let's jump into it !

Overview

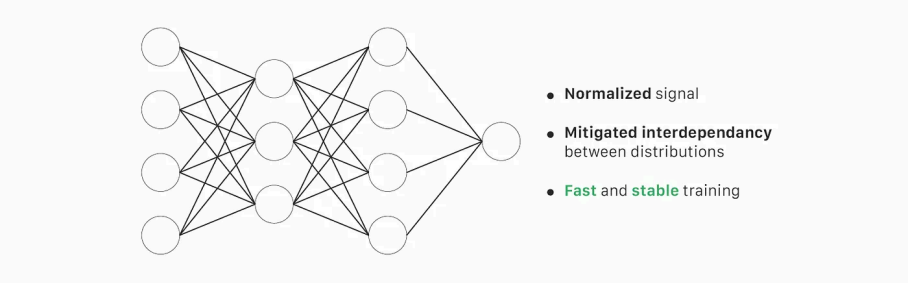
A) In 30 seconds

Batch-Normalization (BN) is an algorithmic method which makes the training of Deep Neural Networks (DNN) **faster** and **more stable**.

It consists of **normalizing activation vectors from hidden layers** using the first and the second statistical moments (mean and variance) of the current batch. This normalization step is applied right before (or right after) the nonlinear function.



Multilayer Perceptron (MLP) **without batch normalization (BN)** | Credit : author - Design : Lou HD



Multilayer Perceptron (MLP) **with batch normalization (BN)** | Credit : author - Design : Lou HD

All the current deep learning frameworks have already implemented methods which apply batch normalization. It is usually used as a module which could be inserted as a standard layer in a DNN.

Remark : For those who prefer read code than text, I wrote a simple implementation of Batch Normalization in this repo.

B) In 3 minutes

B.1) Principle

Batch normalization is computed differently during the training and the testing phase.

B.1.1) Training

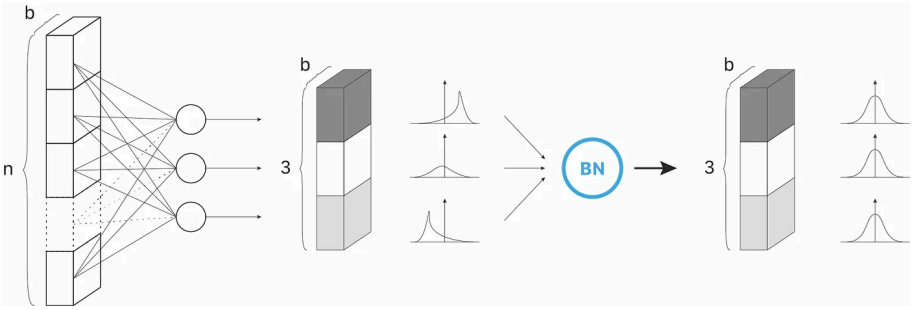
At each hidden layer, Batch Normalization transforms the signal as follow :

$$(1)\ \ \mu = \frac{1}{n}\sum_i Z^{(i)} \qquad (2)\ \ \sigma^2 = \frac{1}{n}\sum_i (Z^{(i)} - \mu)^2$$

$$(3)\ \ Z^{(i)}_{norm} = \frac{Z^{(i)} - \mu}{\sqrt{\sigma^2 - \epsilon}} \qquad (4)\ \ \check{Z} = \gamma * Z^{(i)}_{norm} + \beta$$

The BN layer first determines the **mean μ** and **the variance σ^2** of the activation values across the batch, using (1) and (2).

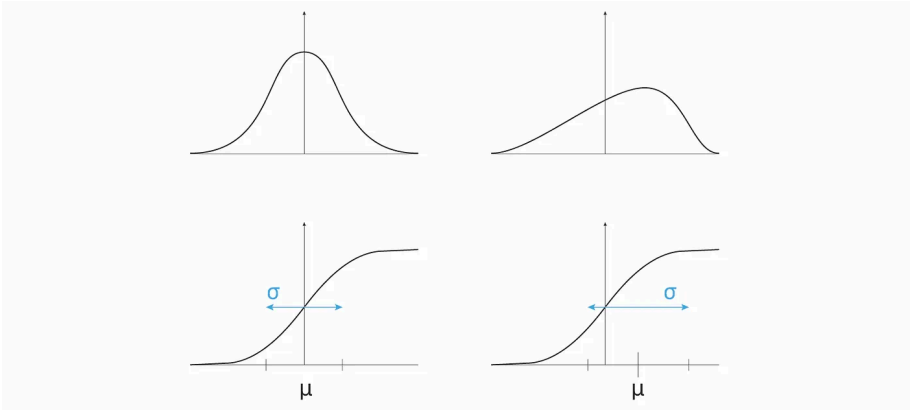
It then **normalizes the activation vector Z^(i)** with (3). That way, each neuron's output follows a standard normal distribution across the batch. (ε is a constant used for numerical stability)



Batch Normalization first step. Example of a 3-neurons hidden layer, with a batch of size b. Each neuron follows a standard normal distribution. | Credit : author - Design : Lou HD

It finally calculates the **layer's output Ẑ(i)** by applying a linear transformation with γ and β, two trainable parameters (4). Such step allows the model to choose the optimum distribution for each hidden layers, by adjusting those two parameters :

- γ allows to adjust the standard deviation ;
- β allows to adjust the bias, shifting the curve on the right or on the left side.



Benefits of γ and β parameters. Modifying the distribution (on the top) allows us to use different regimes of the nonlinear functions (on the bottom). | Credit : author — Design : Lou HD

Remark : The reasons explaining BN layers effectiveness are subjected to misunderstanding and errors (even in the original article). A recent paper [2] disproved some erroneous hypotheses, improving the community's understanding of this method. We will discuss that matter in section C.3 : "Why does BN work ?".

At each iteration, the network computes the mean μ and the standard deviation σ corresponding to the current batch. Then it trains γ and β *through* gradient descent, using an Exponential Moving Average (EMA) to give more importance to the latest iterations.

B.1.2) Evaluation

Unlike the training phase, **we may not have a full batch to feed into the model during the evaluation phase**.

To tackle this issue, we compute (μ_pop , **σ**_pop), such as :

- μ_pop : estimated mean of the studied population ;
- **σ**_pop : estimated standard-deviation of the studied population.

Those values are computed using all the (μ_batch , **σ**_batch) determined during training, and directly fed into equation (3) during evaluation (instead of calling (1) and (2)).

Remark : We will discuss that matter in depth in section C.2.3 : "Normalization during evaluation".

B.2) In practice

In practice, we consider the batch normalization as a standard layer, such as a perceptron, a convolutional layer, an activation function or a dropout layer.

Each of the popular frameworks already have an implemented Batch Normalization layer. For example :

Pytorch : torch.nn.BatchNorm1d, torch.nn.BatchNorm2d, torch.nn.BatchNorm3d

Tensorflow /
Keras : tf.nn.batch_normalization, tf.keras.layers.BatchNormalization

All of the BN implementations allow you to set each parameters independently. However, **the input vector size is the most important one**. It should be set to :

- How many neurons are in the current hidden layer (for MLP) ;
- How many filters are in the current hidden layer (for convolutional networks).

Take a look at the online docs of your favorite framework and read the BN layer page : is there anything specific to their implementation ?

B.3) Overview of results

Even if we don't understand all the underlying mechanisms of Batch Normalization yet (see C.3), there's something everyone agrees on : **it works**.

To get a first insight, let's take a look on the official article's results [1] :

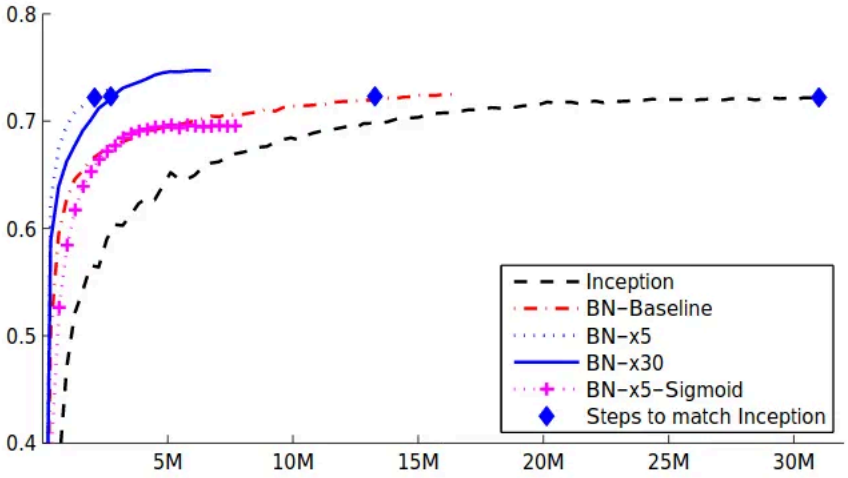


Figure 1 : **How BN affects training.** Accuracy on the ImageNet (2012) validation set, w.r.t. the number of trained iterations. Five networks are compared : "Inception" is the vanilla Inception network [3], "BN-X" are Inception network with BN layers (for 3 differents learning rates : x1, x5, x30 the Inception optimum one, "BN-X-Sigmoid" is an Inception network with BN layers, were all ReLU nonlinearities are replaced by sigmoid. | Source : [1]

The results are clear-cut : BN layers **make the training faster**, and **allow a wider range of learning rate** without compromising the training convergence.

Remark : At this point, you should know enough about BN layers to use them. However, you will need to dig deeper if you want to get the most out of Batch Normalization !

What's the link between Batch Normalization and this picture ? | Credit : Danilo Alvesd on Unsplash

C) Understanding Batch Normalization (BN)

C.1) Implementation

I've reimplemented the Batch Normalization layer with Pytorch to reproduce the original paper results. The code is available on this github repository.

I recommend that you take a look at some of the implementations of BN layer available online. It is very instructive to see how it is coded in your favorite framework !

C.2) BN layer in practice

Before diving into the theory, let's start with what's certain about Batch Normalization.

In this section, we'll see:

- How does BN impact **training performances** ? Why is this method so important in Deep Learning nowadays ?
- What are the BN **side-effects** we must be aware of ?
- **When** and **how** should we use BN ?

C.2.1) Results from the original article

As previously stated, BN is widely used because it almost always **makes deep learning models perform much better**.

The official article [1] carried out 3 experiments to show their method effectiveness.

At first, they have trained a classifier on the MNIST dataset (handwritten digits). The model consists of 3 fully connected layers of 100 neurons each, all activated by sigmoid function. They have trained this model twice (with and without BN layers) for 50 000 iterations, using stochastic gradient descent (SGD), and the same learning rate (0.01). Notice that BN layers were put right before the activation function.

You can easily reproduce those results without GPU, it's a great way to get more familiar with this concept !

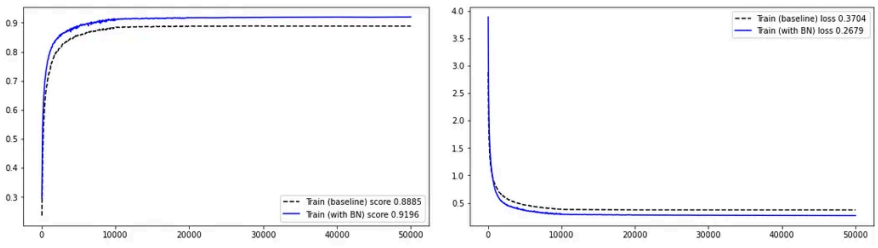


Figure 2 : **How does BN affects the training of a simple Multi-Layer-Perceptron (MLP)** | Left : Training accuracy w.r.t. iterations | Right : Training loss w.r.t iterations | Credit : author

Looks good ! Batch Normalization increases our network performances, regarding both the loss and the accuracy.

The 2nd experiment consists of taking a look at the activation values in the hidden layers. Here are the plots corresponding to the last hidden layer (right before the nonlinearity) :

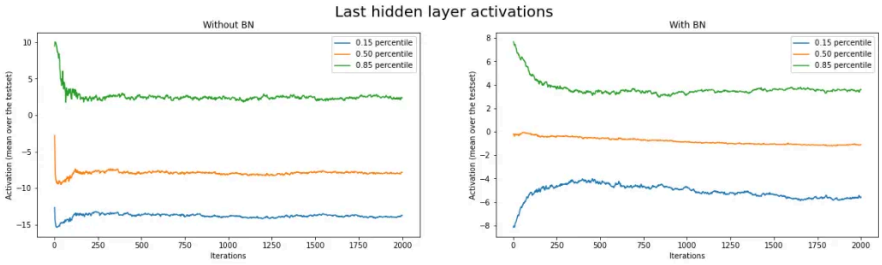


Figure 3 : **Batch Normalization impact on the activation values of the last hidden layer** | Credit : author

Without Batch Normalization, the activated values fluctuate significantly during the first iterations. On the contrary, activation curves are smoother when BN is used.

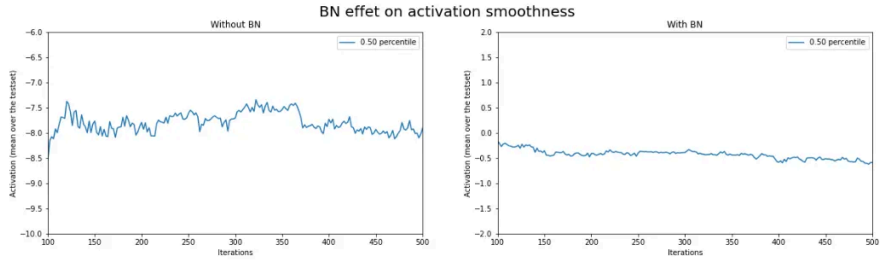


Figure 4 : **Batch Normalization impact on hidden layers activation** | Model with BN have a smoother activation curve than the one without BN | Credit : author

Also, the signal is less noisy when adding BN layers. It seems to make the convergence of the model easier.

This example does not show all the benefits of the Batch Normalization.

The official article carried out a third experiment. They wanted to compare the model performances while adding BN layers on a larger dataset : ImageNet (2012). To do so, they trained a powerful neural network (at that time) called Inception [3]. Originally, this network doesn't have any BN layers. They added some and trained the model by modifying its learning rate (x1, x5, x30 former optimum). They also tried to replace every ReLU activation function by sigmoid in another network. Then, they compared the performances of those new networks with the original one.

Here is what they got :

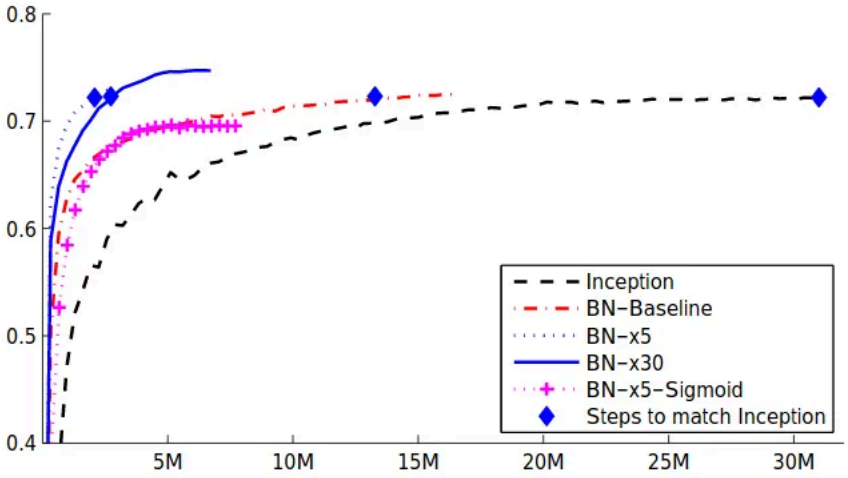


Figure 5 : Batch normalization impact on training (ImageNet) | "Inception" : original network [3] ; "BX-Baseline" : same as Inception with BN, same learning rate (LR) ; "BX-x5" : same as Inception with BN, LR x5 ; "BX-x30" : same as Inception with BN, LR x30 ; "BN-x5-Sigmoid" : same as Inception with BN, LR x5 and sigmoid instead of ReLU | Source : [1]

What we can conclude from those curves :

- Adding BN layers leads to **faster and better convergence** (where better means higher accuracy) ;

On such a large dataset, the improvement is much more significant than the one observed on the small MNIST dataset.

- Adding BN layers allows us to use **higher learning rate (LR) without compromising convergence** ;

The authors have successfully trained their Inception-with-BN network using a **30 times higher learning rate** than the original one. Notice that a 5 times larger LR already makes the vanilla network diverge !

That way, it is much easier to find an "acceptable" learning rate : the interval of LR which lies between underfitting and gradient explosion is much larger.

Also, a higher learning rate helps the optimizer to avoid local minima convergence. Encouraged to explore, the optimizer will **more easily converge on better solutions**.

- The **sigmoid-based model reached competitive results** with ReLU-based models

We need to take a step back and **look at the bigger picture**. We can clearly see that we can get slightly better performances with ReLU-based models than with sigmoid ones, but that's not what matters here.

To show why this results is meaningful, let me rephrase what Ian Goodfellow (inventor of GANs [6], author of the famous "Deep learning" handbook) said about it :

> *Before BN, we thought that it was almost impossible to efficiently train deep models using sigmoid in the hidden layers. We considered several approaches to tackle training instability, such as looking for better initialization methods. Those pieces of solution were heavily heuristic, and way too fragile to be satisfactory.* ***Batch Normalization makes those unstable networks trainable*** *; that's what this example shows.*
>
> *— Ian Goodfellow (rephrased from : source)*

Now we understand why BN had such an **important impact on the deep learning field**.

Those results give **an overview of Batch Normalization benefits on network performances**. However, there are some side effects you should have in mind to get the most out of BN.

C.2.2) Regularization, a BN side effect

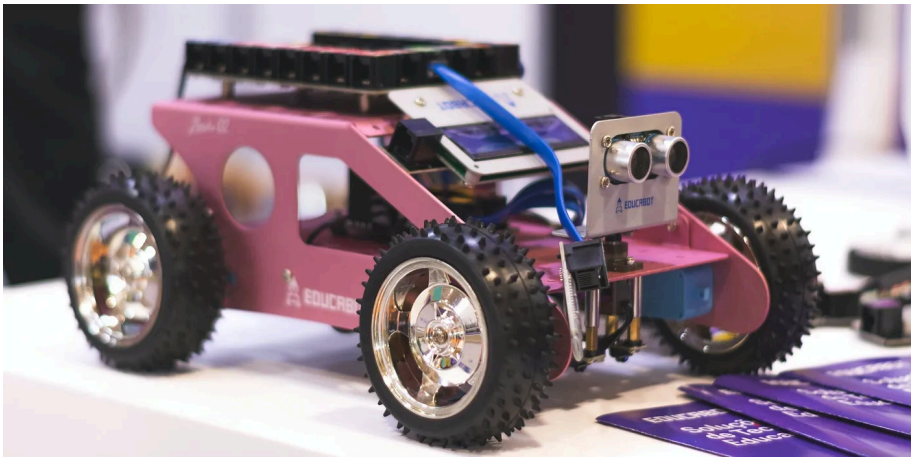
BN relies on batch first and second statistical moments (mean and variance) to normalize hidden layers activations. The output values are then strongly tied to the current batch statistics. Such transformation adds some noise, depending on the input examples used in the current batch.

Adding some noise to avoid overfitting ... sounds like a **regularization process**, doesn't it ? ;)

In practice, **we shouldn't rely on batch normalization to avoid overfitting**, because of orthogonality matters. Put simply, we should always make sure that one module addresses one issue. Relying on several modules to deal with different problems makes the development process much more difficult than needed.

Still, it is interesting to be aware of the regularization effect, as it could explains some unexpected behavior from a network (especially during sanity checks).

Remark : The greater the batch size, the lesser the regularization (as it reduces noise impact).



What if we want to deploy a batch normalized model on such a terrific embedded system ? | Credit : Marília Castelli on Unsplash

C.2.3) Normalization during evaluation

There are two cases where a model could be called in **evaluation mode** :

- When doing **cross-validation** or **test**, during model training and development ;
- When **deploying** the model.

In the first case, we could apply Batch Normalization using current batch statistics for convenience. In the second one, **using the same approach makes no sense**, because we do not necessarily have an entire batch to predict.

Let's see the example of a robot with an embedded camera. We could have a model that uses the current framework to predict the positions of any upcoming obstacles. So we want to compute inference on a single frame (i.e. one rgb image) per iteration. If the training batch size is N, **what should we choose for the (N - 1) other inputs expected by the model to do a forward propagation ?**

Remember that for each BN layer, (β, γ) were trained using a normalized signal. So **we need to determine (μ, σ)** in order to have meaningful results.

A solution would be to choose arbitrary values to fulfil the batch. By feeding a first batch to the model, we will get a certain result for the image we are interested in. If we build a second batch with other random values, we will have different predictions for the same image. **Two different outputs for a single input is not a desirable model behavior**.

This trick is to define (μ_pop , σ_pop), which are respectively the estimated mean and standard deviation of the **targeted population**. Those parameters are calculated as the mean of all the (μ_batch, σ_batch) determined during training.

That's how we do it !

Remark : This trick might leads to instability during the evaluation phase : let's discuss it in the next section.

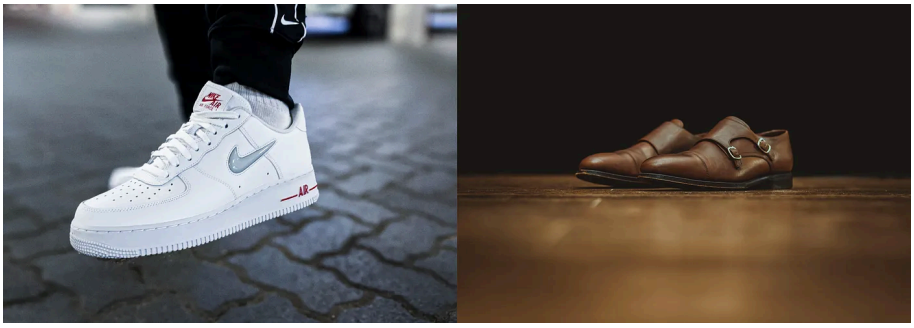
C.2.4) BN layer stability

Even though Batch Normalization works pretty well, it can sometimes cause stability issues. There are cases where BN layer makes activation values explode during the evaluation phase (making the model returns loss = NaN).

We have just mentioned how (μ_pop , σ_pop) are determined, in order to use them during evaluation : we compute the mean of all the

(μ_batch, σ_batch) calculated during training.

Let's consider a model which was trained only on images containing sneakers. What if we have derby-like shoes in the test set ?



If the input distribution varies too much from training to evaluation, the model could overeact to some signals, resulting in activaton divergence. | Credit : Grailify (left), and Jia Ye (right) on Unsplash

We suppose that the activation values of hidden layers will have significantly different distributions during training and evaluation — maybe too much. In this case, **the estimated (μ_pop, σ_pop) does not properly estimate the real population mean and standard deviation.** Applying (μ_pop, σ_pop) might push the activation values away from ($\mu = 0$, $\sigma = 1$), resulting in **overestimation of activation values**.

Remark : The shifting of distribution between training and testing set is called "covariate shift". We will talk about this effect again in section (C.3.).

This effect is increased by a well known property of BN : during training, activation values are normalized using their own values. During inference, the signal is normalized using (μ_pop, σ_pop), which have been computed during training. Thus, the **coefficients used for normalization don't take into account the actual activation values themselves.**

In general, a training set must be "similar enough" to the testing set : otherwise, it would be impossible to properly train a model on the targetted task. So in most cases, μ_pop and σ_pop should be a good fit for the testing set. If not, we may conclude that the training set is not large enough, or that its quality is not good enough for the targeted task.

But sometimes, it just happens. And there is not always a clean solution to this issue.

I have personally faced it once, during the Pulmonary Fibrosis Progression Kaggle competition. The training dataset consisted of metadata, and 3D scans of lungs associated to each patients. The content of those scans was complex and diverse, but we only had ~ 100 patients to split into train and validation sets. As a result, the convolutional networks I wanted to use for feature extractions returned NaN as soon as the model switched from training to evaluation mode. Pleasant to debug.

When you cannot easily get additional data to enhance your training set, you have to find a workaround. In the above case, I've manually forced the BN layers to compute (μ_batch, σ_batch), on the validation set too. (I agree, that's an ugly way to fix it, but I ran out of time. ;))

Adding BN layers to your network - assuming it cannot have negative impacts - is not always the best strategy !

C.2.5) Recurrent network and Layer normalization

In practice, it is widely admitted that :

- **For convolutional networks (CNN)** : **Batch Normalization (BN)** is better
- **For recurrent network (RNN)** : **Layer Normalization (LN)** is better

While BN uses the current batch to normalize every single value, LN uses all the current layer to do so. In other words, the normalization is performed using other features from a single example instead of using the same feature across all current batch examples. This solution seems more efficient for recurrent networks. Note that it is quite difficult to define a consistent strategy for those kinds of neurons, as they rely on multiplication of the same weight matrix several times. Should we normalize each step independently ? Or should we compute the mean across all steps, and then apply normalization recursively ? *(source of the intuition argument : here)*

I will not detail any further on that matter, as it's not the purpose of this article.

C.2.6) Before or after the nonlinearity ?

Historically, BN layer is positioned **right before** the nonlinear function, which was consistent with the authors objectives and hypothesis at that time :

In their article, they stated :

> *"We would like to ensure that, for any parameter values, the network always produces activations with the desired distribution."*
>
> *— Sergey Ioffe & Christian Szegedy (source : [1])*

Some experiments showed that positioning BN layers **right after** the nonlinear function leads to **better results**. Here is an example.

François Chollet, creator of Keras and currently engineer at Google, stated that :

> *"I haven't gone back to check what they are suggesting in their original paper, but I can guarantee that recent code written by Christian*

> *[Szegedy]* ***applies relu before BN****. It is still occasionally a topic of debate, though."*
>
> *— François Chollet (source)*

Still, many commonly used architecture of transfer learning apply BN before nonlinearity (ResNet, mobilenet-v2, ...).

Notice that the article [2] - which challenges hypotheses defended by the original article [1] to explain BN effectiveness (see C.3.3)) - puts the BN layer before the activation function, without bringing solid reason for it.

As far as I know, **this question is still discussed**.

Further reading : Here is an interesting reddit thread - even if some arguments are not convincing - mostly in favor of BN after activation.

Why the hell does my messy code work ? | Credit : Rohit Farmer on Unsplash

C.3) Why does Batch Normalization work ?

In most cases, Batch Normalization improves the performances of deep learning models. That's great. But we want to know **what's actually happening** inside the black box.

This is where things get a bit hairy.

The problem is : **we don't exactly know** what makes Batch Normalization work so well **yet**. A few hypotheses are often discussed within the DL community: we will explore them step by step.

Before diving into the discussion, here are what we'll see :

- The original paper [1] assumes that BN effectiveness is due to the **reduction** of what they call the **internal covariate shift** (ICS). A recent paper [2] refutes this hypothesis. (see C.3.1)
- Another hypothesis had replaced the first one with much more caution : BN **mitigates the interdependency between layers during training**. (see C.3.2)
- A recent paper from the MIT [2] stresses the impact of BN on the **optimization landscape smoothness**, making the training easier. (see C.3.3)

I bet that exploring those hypotheses will help you to build a strong intuition about Batch Normalization.

Let's go !

C.3.1) Hypothesis n°1 — BN reduces the internal covariance shift (ICS)

Despite its fundamental impact on DNN performances, **Batch Normalization is still subject to misunderstanding**.

Confusions about BN are mostly due to a **wrong hypothesis supported by the original article** [1].

Sergey Ioffe & Christian Szegedy introduced BN as follow :

> *"We refer to the change in the distributions of internal nodes of a deep network, in the course of training, as **Internal Covariate Shift**. [...] We propose a new mechanism, which we call **Batch Normalization**, that takes a **step towards reducing internal covariate shift**, and in doing so dramatically accelerates the training of deep neural nets."*
>
> *— Sergey Ioffe & Christian Szegedy (source : [1])*

In other words, BN is efficient because it - partially - tackles the internal Covariate Shift issue.

This statement has been severely challenged by later works [2].

*Notation : From now, **ICS** refers to internal Covariate Shift.*

To understand what causes such a confusion, let's begin by discussing what covariate shift is, and how is it impacted by normalization.

What is covariate shift (distributional stability perspective) ?

[1]'s authors defined it clearly : **covariate shift** - in the distributional stability prospective - describes the **shifting of a model input distribution**. By extension, the **internal covariate shift** describes this

phenomenon when it happens in the **hidden layers** of a **deep neural network**.

Let's see why it could be an issue through an example.

Let's assume we want to train a classifier which could answers the following question : **Does this image contain a car ?** If we wanted to extract all the car images from a very large unlabeled dataset, such a model would save us a huge amount of time.

We would have an RGB image as input, some convolutional layers, followed by fully connected layers. The output should be a single value, fed into a logistic function to make the final value lies between 0 and 1 - describing the probability for the input image to contain a car.

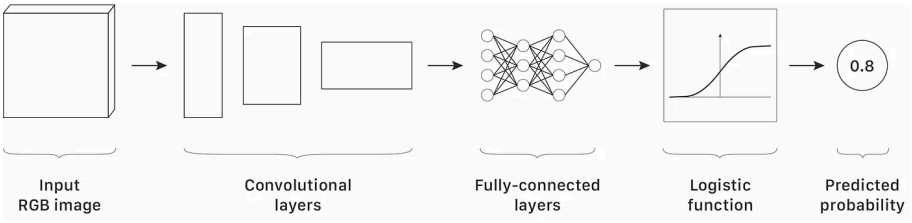


Diagram 5 : **A simple CNN classifier.** | Credit : author - Design : Lou HD

To train such a model, we would need a significant amount of labeled images.

Now, assuming we only have "common" cars for training. How would the model react if we ask it to classify a formula 1 car ?

As stated before, the covariate shift can make the network activations diverge (section C.2.4). Even if it doesn't, it would deteriorate overall performances ! | Credit : Dhiva Krishna (left) and Ferhat Deniz Fors (right) on Unsplash

In this example, there is a shift between the **training and testing distribution.** More broadly, a different car orientation, lightning, or whether condition would be enough to impact our model performances. Here, **our model does not generalize well.**

If we had plot the extracted features in the features space, we would have something like this :

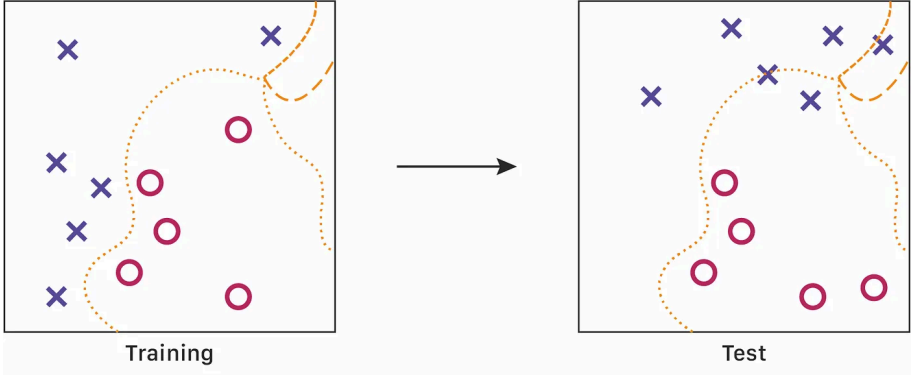


Diagram 6.a : Why do we need to normalize the model input value ? Case without normalization. During training, input values are scattered : the approximated function will be very accurate where there's a high density of points. On the contrary, it will be inaccurate and submitted to randomness where the density of points is low. (For example, the approximated curve could be one of the 3 lines drawn.) | Credit : author - Design : Lou HD

Let's assume the **crosses** describe features associated to **images which do not contain any car**, while **rings** describe **images containing a car**. Here, a single function could split the two ensembles. But the function is likely to be less accurate on the top-right part of the diagram : there are not enough points to determine a good function. This might lead the classifier to do many errors during evaluation.

To efficiently train our model, we would require many car images, in any possible context we could imagine. Even if this is still how we train our CNN nowadays, we want to make sure our model will **generalize well using as few examples as possible**.

The problem could be summarized as follow :

> *From the model point-of-view, training images are - statistically - too different from testing images.*
> *There is a **covariate shift**.*

We can face this issue using simpler models. It is well known that **linear regression** models are easier to optimize when the input values are normalized (i.e. making its distribution close to ($\mu = 0$, $\sigma = 1$)) : that's why **we usually normalize the input values of a model.**

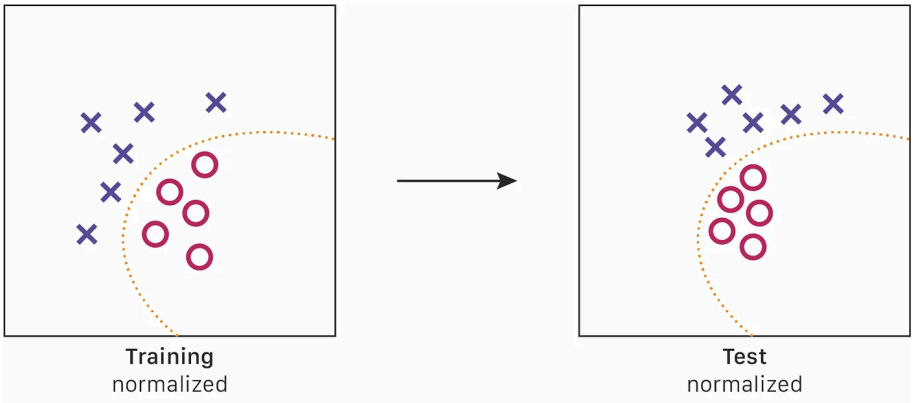


Diagram 6.b : **Why do we need to normalize the model input value ? Case with normalization.** Normalizing the input signal makes the points closer to each other in the feature space during training : it is now easier to find a well generalizing function. | Credit : author — Design : Lou HD

This solution was already well known before the publication of the BN paper. With BN, [1]'s authors wanted to extend this method to the hidden layers to help training.

Internal covariate shift deteriorates training : the original paper hypothesis

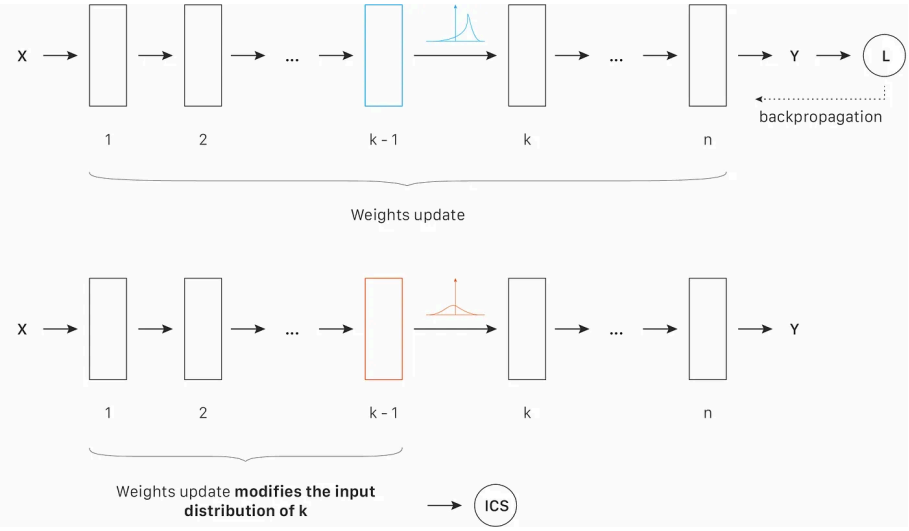


Diagram 7 : Internal covariate shift (ICS) principle in the distributional stability perspective (ICS_distrib). | Credit : author - Design : Lou HD

In our car classifier, we can see hidden layers as units which are activated when they identify some "conceptual" features associated with cars : it could be a wheel, a tire or a door. We can suppose that the previously described effect could happen inside hidden units. A wheel with a certain orientation angle will activate neurons with a specific distribution. Ideally, we want to **make some neurons react with a comparable distribution for any wheel orientations**, so the model could efficiently conclude on the probability for the input image to contain a car.

If there is a huge covariate shift in the input signal, **the optimizer will have trouble generalizing well**. On the contrary, if the input signal always follows a standard normal distribution, the optimizer will more easily generalize. With this in mind, [1]'s authors applied the strategy of normalizing the signal in the hidden layers. They assumed that forcing to (μ = 0, σ = 1) the intermediates signal distribution will **help the network generalization in the "conceptual" levels of features**.

Though, we do not always want standard normal distribution in the hidden units. It would reduce the model representativity :

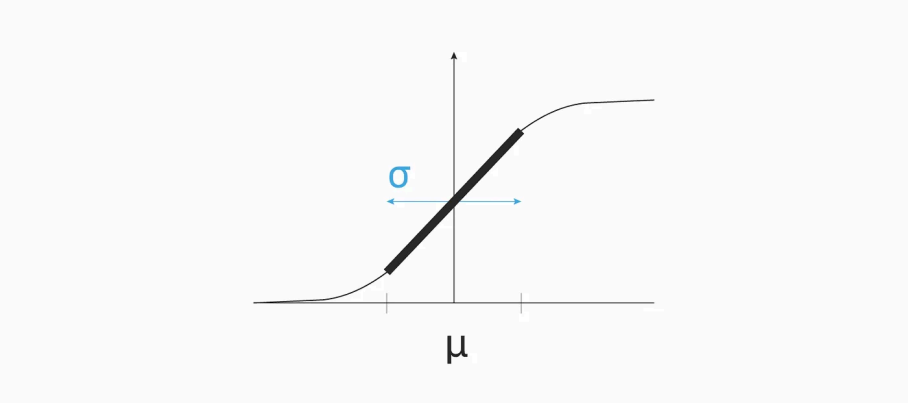


Diagram 8 : Why don't we always want a standard normal distribution in the hidden units. Here, the sigmoid function only works in its linear regime. | Credit : author - Design : Lou HD

The original article takes the sigmoid function as an example to show why normalization alone is an issue. If the input signal values lie between 0 and 1, **the nonlinear function would only work … in its linear regime**. Sounds problematic.

To tackle this issue, they added two trainable parameters β and γ, allowing the optimizer to define the optimum mean (using β) and standard deviation (using γ) for a specific task.

⚠**Warning : The following hypothesis is now outdated. A lot of great content about BN still claim it as the reason that makes the method works in practice. Yet, recent works severely challenged it.**

For a couple of years after the release of [1], the DL community explained BN effectiveness as follow :

> ***— Hypothesis 1*** *— —*
>
> *BN ➜* ***Normalization*** *of the input signal inside hidden units ➜ Adding two trainable parameters to* ***adjust the distribution*** *and get the most out of the nonlinearities ➜ Easier training*

Here, normalization to ($\mu = 0, \sigma = 1$) is what mostly explains BN's effectiveness. **This hypothesis has been challenged** (see section C.3.3), replaced by another hypothesis :

> ***— Hypothesis 2*** *— —*
>
> *BN ➜ Normalization of the input signal inside hidden units ➜* ***Reduces interdependency between hidden layers*** *(in a distribution stability perspective) ➜ Easier training*

There's is a slight but very important difference. Here, **the purpose of normalization is to reduce interdependency between layers** (in a distribution stability perspective), so the optimizer could choose the

optimum distribution by adjusting only two parameters ! Let's explore this hypothesis a little bit further.

What's the link between Batch Normalization and this picture ? | Credit : Danilo Alvesd on Unsplash

C.3.2) Hypothesis n°2 — BN mitigates interdependency between hidden layers during training

About this section : I could not find any solid evidence about the hypothesis discussed in this section. Therefore, I decided to mostly rely on Ian Goodfellow's explanations on that matter (in particular in this brilliant video).

Consider the following example :

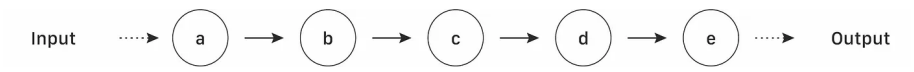


Diagram 9 : **A simple DNN**, which consists of linear transformations. | Inspired by Ian Goodfellow

Where (a), (b), (c), (d) and (e) are the sequential layers of the network. Here is a very simple example, where all layers are linked by linear transformations. Let's assume we want to train such model using SGD.

To update the weights of the layer (a), we need to calculate the gradient from the network's output :

$$grad(a) = grad(b) * grad(c) * grad(d) * grad(e)$$

We first consider a network without BN layers. From the above equation, we conclude that if all gradients are large, grad(a) will be very large. On the contrary, if all gradients are small, grad(a) will be almost negligible.

It is pretty easy to see how dependent are layers from each other by looking at the input distribution of hidden units : a modification of (a) weights will modify the input distribution of (b) weights, which will eventually modify the input signal of (d) and (e). This interdependency could be problematic for training stability : **if we want to adjust the input distribution of a**

specific hidden unit, we need to consider the whole sequence of layers.

However, SGD considers 1st order relationships between layers. So they don't take into account the higher degree relationships mentioned above !

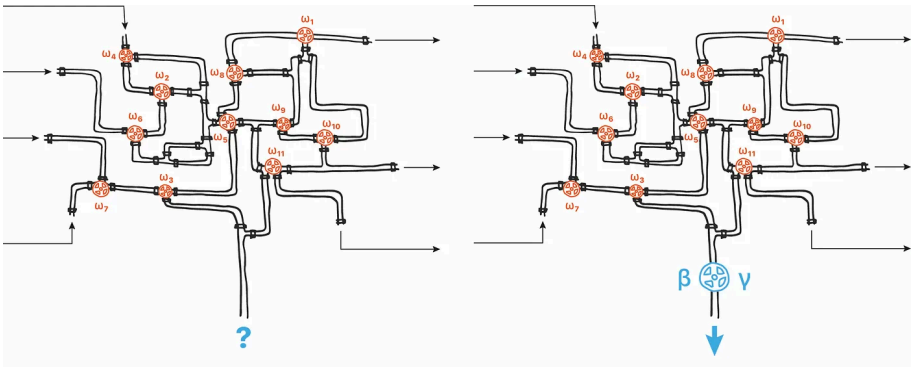


Diagram 10 : Hypothesis n°2 principle. BN layers make the signal regulation easier, by normalizing the signal inside each hidden unit, and allowing distribution adjustment using β and γ. BN acts like a valve which makes the flow control easier at some points, without deteriorating the underlying complexity of the network ! | Credit : author - Design : Lou HD

Adding BN layers significantly reduces the interdependence between each layers (in the distribution stability perspective) during training. **Batch Normalization acts like a valve which holds back the flow, and allows its regulation using β et γ.** It is then no longer necessary to take all parameters into account to have clues about distribution inside the hidden units.

Remark : The optimizer can then do larger weights modifications without deteriorating adjusted parameters in other hidden layers. It makes the hyperparameter tuning way easier !

This example puts aside the hypothesis which claims that BN effectiveness is due to the normalization of intermidiates signal distribution to ($\mu = 0$, $\sigma = 1$). Here, BN aims to **make the optimizer job easier**, allowing it to **adjust hidden layer distribution** with only **two parameters at a time**.

⚠ However, keep in mind that **this is mostly speculation**. Those discussions should be used as insights to build intuition about BN. We still don't exactly know why BN is efficient in practice !

In 2019, a team of researchers from MIT carried out some interesting experiments about BN [2]. **Their results severely challenge the hypothesis n°1** (still shared by many serious blog posts and MOOCS !).

We should have a look on this paper if we want to avoid "local minima hypotheses" about the BN impact on training... ;)

Alright … you better initialize well. | Credit : Tracy Zhang on Unsplash

C.3.3) Hypothesis n°3 — BN makes the optimisation landscape smoother

About this section : I've synthesized results from [2] which could help us to build a better intuition about BN. I wasnt able to be exhaustive, this paper is dense, I recommend you to read it thoroughly if you're interested in those concepts.

Let's jump straight into the 2nd experiment of [2]. Their goal was to **check the correlation between ICS, and the benefits of BN on training performances** (hypothesis n°1).

*Notation : We'll now refer to this covariate shift by **ICS_distrib**.*

To do so, the researchers have trained three VGG networks (on CIFAR-10) :

- The first one does not have any BN layer ;
- The second one does have BN layers ;
- The third one is similar to the second one, except from the fact that **they have explicitly added some ICS_distrib inside the hidden unit** right before the activation (by adding random bias & variance).

They measured the accuracy reached by each model, and the evolution of the distribution values w.r.t iterations. Here is what they got :

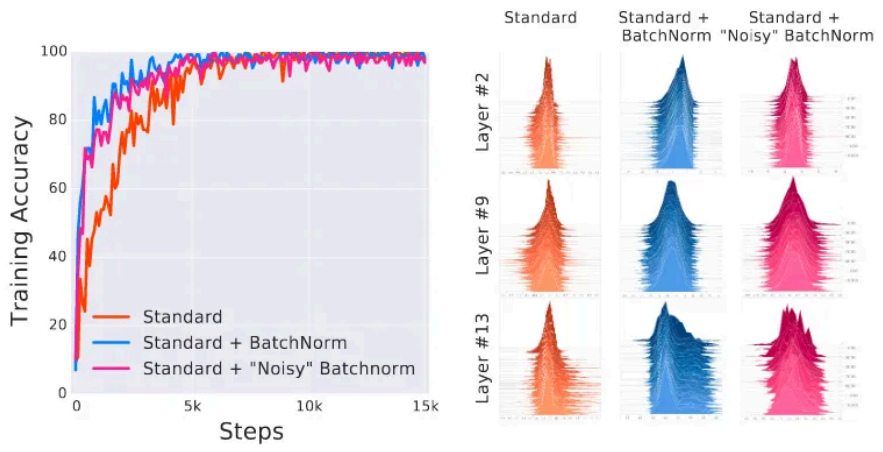


Diagram 6 : **BN on ICS_distrib |** Networks with BN are trained faster than the standard one ; explicitly adding ICS_distrib on a regulated network does not deteriorate BN benefits. | Source : [2]

We can see that the 3rd network has, as expected, a very high ICS. However, **the noisy network is still trained faster than the standard one.** Its reached performances are comparable to the ones obtained with a standard BN network. This result suggests that the **BN effectiveness is not related to ICS_distrib.** Oops !

We shouldn't discard the ICS theory too fast : if the BN effectiveness does not come from ICS_distrib, **it might be related to another definition of ICS**. After all, the intuition behind the hypothesis n°1 makes sense, doesn't it ?

The main issue with ICS_distrib is that its definition is related to input distribution of hidden units. So there is no direct link with the optimization problem on its own.

[2]'s authors proposed another definition of ICS :

Let's consider a fixed input X.

> *We define the **internal covariate shift** from an **optimization** perspective as the difference between the **gradient** computed on a hidden layer k after backpropagating the error **L(X)_it**, and the gradient computed on the same layer k from the loss **L(X)_it+1** computed after the **iteration = it** update of weights.*

This definition aims to **focus on the gradients** more than on the hidden layer input distributions, assuming that it could give us better clues on how ICS could have an impact on the **underlying optimization problem**.

*Notation: **ICS_opti** now refers to the ICS defined from an optimization perspective.*

In the next experiment, authors evaluate ICS_opti impact on training performances. To do so, they measure the variation of ICS_opti during training for a DNN with and without BN layers. To quantify the variation of gradient mentioned in the ICS_opti definition, they calculate :

- **L2 difference :** Do the gradients have a close norm before and after the weights update ? *Ideally : 0*
- **Cosine angle** : Do the gradients have a close orientation before and after the weights update ? *Ideally : 1*

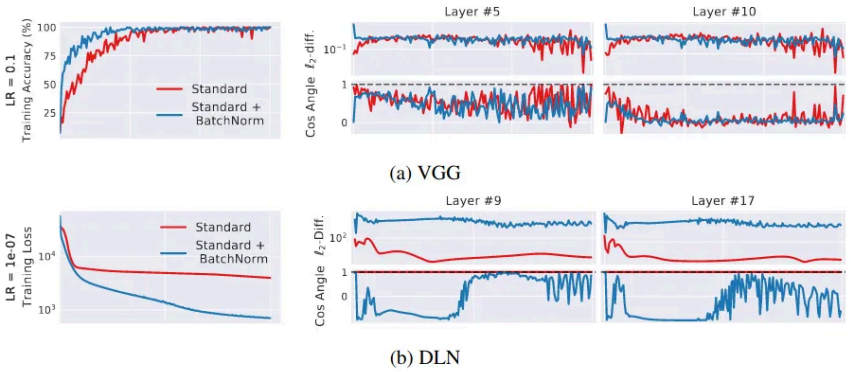


Figure 7 : BN impact on ICS_opti | L2 diff and cosine angles suggest that BN does not prevent ICS_opti (it seems to slightly increase it, somehow). | Source [2]

Results are a bit surprising : the network which relies on BN **seems to have a higher ICS_opti** than the standard network. Remember that the network with BN (blue curve) is trained faster than the standard network (red curve) !

ICS seems definitively not related to training performances...at least for the explored definition of ICS.

Somehow, Batch Normalization have **another impact on the network**, which makes convergence easier.

Now, let's study **how does BN affects the optimization landscape**. We may find clues there.

Here is the last experiment covered in this story :

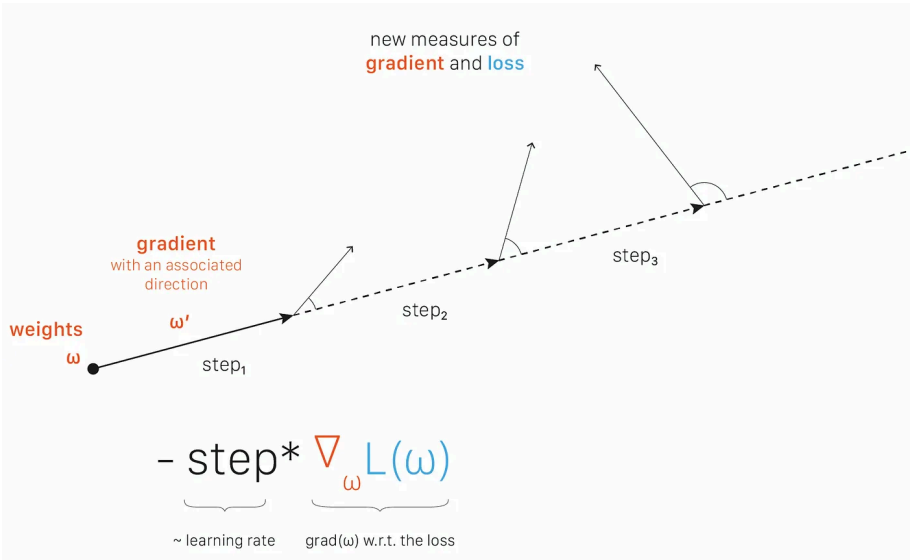


Figure 11 : Optimization landscape exploration in the gradient's direction. Experiment carried out in the paper [2] | Inspired by Andrew Ilyas - Design : Lou HD

From a single gradient, we update weights with different **optimization steps** (which act like a learning rate). Intuitively, we define a direction from a **certain point** (i.e. a **network configuration ω)** in the feature space, and then **explore the optimization landscape further and further in this direction**.

At each step, we measure the **gradient** and the **loss**. We can therefore compare different points of the optimization landscape with a starting point. If we measure large variations, the **landscape is very unstable** and the gradient is incertain : **big steps might deteriorate optimization**. On the contrary, if the measured variations are small, the **landscape is stable** and the gradient is trustworthy : **we can apply larger steps without compromising optimization**. In other words, we can use a **larger learning rate**, and make the **convergence faster** (a well known properties of BN).

Let's have a look on the results :

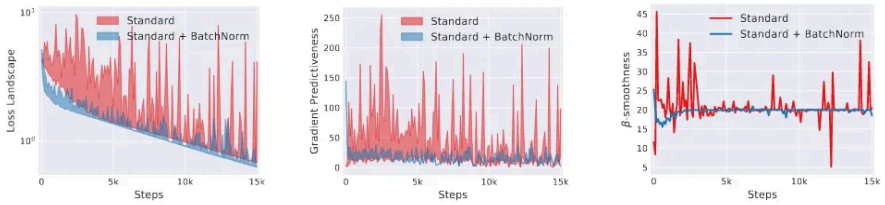


Figure 8 : BN impact on optimization landscape smoothing | Using BN significantly reduces gradient variations. | Source : [2]

We can clearly see that **the optimization landscape is way smoother with BN layers**.

We finally have results we can rely on to explain BN effectiveness : BN layer makes somehow the optimization landscape smoother. That makes the optimizer job easier : we can define a larger learning rate without being submitted to gradient vanishing (weights stuck on sudden flat surfaces) or to gradient explosion (weights fallen in an abrupt local minima).

We can know formulate a 3rd hypothesis, proposed by [2] :

> ***— Hypothesis 3 :*** *— — — — — — — — — — — — — — — — —*
>
> *BN ➜ Normalization of the input signal inside hidden units ➜ Makes the **optimization landscape smoother** ➜ Faster and more stable training*

It raises another question : **How does BN make the optimization landscape smoother ?**

[2]'s authors also explored those matters from a theoretical point of view. Their work is very instructive, helping to get a better grasp of the smoothing effect of Batch Normalization. In particular, they showed that BN makes the optimization landscape smoother while **preserving all of the minima of the normal landscape**. In other words, **BN reparametrizes the underlying optimization problem, making the training faster and easier** !

⚠ In additional studies, [2]'s authors observed that **this effect is not unique to BN**. They obtained similar training performances with other **normalization methods** (for example L1 or L2 normalization). Those observations suggest that BN effectiveness is mostly due to **serendipity**, leveraging on underlying mechanisms that we have not perfectly identified yet.

Now it's time to set a very high learning rate. | Credit : Finding Dan | Dan Grinwis on Unsplash

To conclude this section, this paper **severely challenges** the widely admitted idea that **BN effectiveness is mostly due to ICS reduction** (in a training stability distribution perspective, as well as in an optimization perspective). However, it stresses the impact of **BN smoothing effect on the optimization landscape**.

While this paper states a hypothesis about **BN impact on training speed**, it does not answer why **BN helps generalization**.

They briefly argue that **making the optimization landscape smoother could help the model to converge on flat minima**, which have **better generalizing properties**. No more details on that matter, though.

Their main contribution is to challenge the commonly admitted idea of BN effect on ICS — which is already significant !

C.4) Summary : Why does BN work ? What do we know so far

- **Hypothesis 1** : BN layer **mitigates the internal covariate shift** (ICS)

❌ **Wrong :** [2] showed that in practice, **there is no correlation** between ICS and training performances.

- **Hypothesis 2** : BN layer makes the optimizer job easier by allowing it to **adjust input distribution of hidden units with only 2 parameters**.

❓ **Maybe :** This hypothesis highlights the interdependency between parameters, making the optimization task harder. **No solid proof,** though.

- **Hypothesis 3** : BN layer **reparametrize the underlying optimization** problem, making it smoother and more stable.

❓ **Maybe :** Their results are quite recent. To my knowledge, they have not been challenged so far. They provide empirical demonstrations and pieces of theoretical justifications, but **some fundamental questions remain unanswered** (such as "how does BN help generalization ?").

Discussion : It seems to me that the last two hypotheses are compatible. Intuitively, we could see the hypothesis n°2 as a projection from a problem with many parameters, to many problems with a couple of parameters ; a kind of dimensionality reduction, which would helps generalization. Any ideas about it ?

Many questions remain open, and Batch Normalization is still a topic of research nowadays. Discussing those hypotheses still helps to get a better understanding of this commonly used method, discarding some erroneous statements we had in mind for a couple of years.

Those questions do not prevent us to leverage on the benefits of BN in practice, though !

Conclusion

Batch Normalization (BN) is **one of the most important advances** in the field of **Deep Learning** (DL) in recent years. Relying on two successive linear transformations, this method makes Deep Neural network (DNN) **training faster** and **more stable**.

The most widely admitted hypothesis about what makes BN efficient in practice is the **reduction** of **interdependence** between hidden layers during training. However, the normalization transformation impact of **optimization landscape smoothness** seems to be an important mechanism of BN **effectiveness**.

Many commonly used DNN rely on BN nowadays (ex : ResNet [4], EfficientNet [5], ...).

If you are interested in Deep Learning, you will for sure have to get familiar with this method !

Opened questions

Even if BN appears to be efficient in practice for years, many questions about its underlying mechanisms remain unanswered.

Here is a non-exhaustive list of opened questions about BN :

- How does BN helps **generalization** ?
- Is BN the **best normalization method** to make the optimization easier ?
- How do β **et** γ impact the optimization landscape smoothness ?
- Experiments carried out by [2] to explore optimization landscape focus on **short-term impact of BN on gradient** : they have measured the variation of gradient & loss from a **single iteration**, for several values of step. How does BN impact gradient on **the long run** ? Do weights interdependency have other interesting impact on the optimization landscape ?