# Deep learning

# 6.4. Batch normalization

François Fleuret

https://fleuret.org/dlc/

We saw that maintaining proper statistics of the activations and derivatives was a critical issue to allow the training of deep architectures.

It was the main motivation behind Xavier's weight initialization rule.

A different approach consists of explicitly forcing the activation statistics during the forward pass by re-normalizing them.

**Batch normalization** proposed by Ioffe and Szegedy (2015) was the first method introducing this idea.

"Training Deep Neural Networks is complicated by the fact that **the distribution of each layer's inputs changes during training, as the parameters of the previous layers change.** This slows down the training by requiring lower learning rates and careful parameter initialization /.../"

(Ioffe and Szegedy, 2015)

Batch normalization can be done anywhere in a deep architecture, and forces the activations' first and second order moments, so that the following layers do not need to adapt to their drift.



---

**Notes**

The motivation for batchnorm as explained here is that if the statistics of the activations are not controlled during training, a layer will have to adapt to the changes of the activations computed by the previous layers in addition to making changes to its own output to reduce the loss.

During training batch normalization **shifts and rescales according to the mean and variance estimated on the batch.**

Processing a batch jointly is unusual. Operations used in deep models can virtually always be formalized per-sample.

During test, it simply shifts and rescales according to the empirical moments estimated during training.

If $x_b \in \mathbb{R}^D, b = 1, \ldots, B$ are the samples in the batch, we first compute the empirical per-component mean and variance **on the batch**

$$\hat{m}_{batch} = \frac{1}{B} \sum_{b=1}^{B} x_b$$

$$\hat{v}_{batch} = \frac{1}{B} \sum_{b=1}^{B} (x_b - \hat{m}_{batch})^2$$

from which we compute normalized $z_b \in \mathbb{R}^D$, and outputs $y_b \in \mathbb{R}^D$

$$\forall b = 1, \ldots, B, \ z_b = \frac{x_b - \hat{m}_{batch}}{\sqrt{\hat{v}_{batch} + \epsilon}}$$

$$y_b = \gamma \odot z_b + \beta.$$

where $\odot$ is the Hadamard component-wise product, and $\gamma \in \mathbb{R}^D$ and $\beta \in \mathbb{R}^D$ are parameters to optimize.



---

**Notes**

$\epsilon$ deals with numerical issues when the variance gets to zero.
The terminology for a batch normalization is the same as for linear layers: $\gamma$ are the "weights" and $\beta$ the "bias".

During inference, batch normalization shifts and rescales independently each component of the input $x$ according to statistics estimated during training:

$$y = \gamma \odot \frac{x - \hat{m}}{\sqrt{\hat{v} + \epsilon}} + \beta.$$

Hence, during inference, batch normalization performs **a component-wise affine transformation**, and it processes samples independently.

As for dropout, the model behaves differently during train and test.

As dropout, batch normalization is implemented as separate modules that process input components independently.

```
>>> bn = nn.BatchNorm1d(3)
>>> with torch.no_grad():
...     bn.bias.copy_(torch.tensor([2., 4., 8.]))
...     bn.weight.copy_(torch.tensor([1., 2., 3.]))
...
Parameter containing:
tensor([2., 4., 8.], requires_grad=True)
Parameter containing:
tensor([1., 2., 3.], requires_grad=True)
>>> x = torch.randn(1000, 3)
>>> x = x * torch.tensor([2., 5., 10.]) + torch.tensor([-10., 25., 3.])
>>> x.mean(0)
tensor([-9.9669, 25.0213,  2.4361])
>>> x.std(0)
tensor([1.9063, 5.0764, 9.7474])
>>> y = bn(x)
>>> y.mean(0)
tensor([2.0000, 4.0000, 8.0000], grad_fn=<MeanBackward2>)
>>> y.std(0)
tensor([1.0005, 2.0010, 3.0015], grad_fn=<StdBackward1>)
```



**Notes**

To illustrate batch normalization, we create such a layer and set the values of $\gamma$ and $\beta$, stored respectively in `bn.weight` and `bn.bias`.
We then create a batch of 1000 samples in $\mathbb{R}^3$ in which the first dimension follows $\mathcal{N}(-10, 2)$, the second $\mathcal{N}(25, 5)$, and the third $\mathcal{N}(3, 10)$.
We finally pass this batch through the batch normalization layer, and see that the output batch is rescaled to have the mean and standard deviation as specified with $\beta$ and $\gamma$.

As for any other module, we have to compute the derivatives of the loss $\mathscr{L}$ with respect to the inputs values and the parameters.

**For clarity, since components are processed independently, in what follows we consider a single dimension and do not index it.**

We have

$$\hat{m}_{batch} = \frac{1}{B}\sum_{b=1}^{B} x_b$$

$$\hat{v}_{batch} = \frac{1}{B}\sum_{b=1}^{B} (x_b - \hat{m}_{batch})^2$$

$$\forall b = 1, \dots, B, \ z_b = \frac{x_b - \hat{m}_{batch}}{\sqrt{\hat{v}_{batch} + \epsilon}}$$

$$y_b = \gamma z_b + \beta.$$

From which

$$\forall b = 1, \dots, B, \ \frac{\partial \mathscr{L}}{\partial z_b} = \gamma \frac{\partial \mathscr{L}}{\partial y_b}$$

$$\frac{\partial \mathscr{L}}{\partial \gamma} = \sum_b \frac{\partial \mathscr{L}}{\partial y_b}\frac{\partial y_b}{\partial \gamma} = \sum_b \frac{\partial \mathscr{L}}{\partial y_b} z_b$$

$$\frac{\partial \mathscr{L}}{\partial \beta} = \sum_b \frac{\partial \mathscr{L}}{\partial y_b}\frac{\partial y_b}{\partial \beta} = \sum_b \frac{\partial \mathscr{L}}{\partial y_b}.$$

**Every sample in the batch impacts the moment estimates, hence all the outputs**, which makes the derivative with respect to an input complicated.

$$\frac{\partial \mathscr{L}}{\partial \hat{v}_{batch}} = -\frac{1}{2}\left(\hat{v}_{batch} + \epsilon\right)^{-3/2} \sum_{b=1}^{B} \frac{\partial \mathscr{L}}{\partial z_b}(x_b - \hat{m}_{batch})$$

$$\frac{\partial \mathscr{L}}{\partial \hat{m}_{batch}} = -\frac{1}{\sqrt{\hat{v}_{batch} + \epsilon}} \sum_{b=1}^{B} \frac{\partial \mathscr{L}}{\partial z_b}$$

$$\forall b = 1, \ldots, B, \quad \frac{\partial \mathscr{L}}{\partial x_b} = \frac{\partial \mathscr{L}}{\partial z_b} \frac{1}{\sqrt{\hat{v}_{batch} + \epsilon}} + \frac{2}{B} \frac{\partial \mathscr{L}}{\partial \hat{v}_{batch}} (x_b - \hat{m}_{batch}) + \frac{1}{B} \frac{\partial \mathscr{L}}{\partial \hat{m}_{batch}}$$

In standard implementations, test $\hat{m}$ and $\hat{v}$ are estimated with a moving average during train, to avoid the need for an additional pass through the samples.

Results on ImageNet's LSVRC2012:

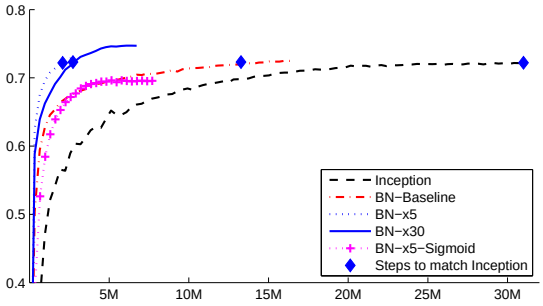



Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

| Model | Steps to 72.2% | Max accuracy |
|---|---|---|
| Inception | $31.0 \cdot 10^6$ | 72.2% |
| *BN-Baseline* | $13.3 \cdot 10^6$ | 72.7% |
| *BN-x5* | $2.1 \cdot 10^6$ | 73.0% |
| *BN-x30* | $2.7 \cdot 10^6$ | 74.8% |
| *BN-x5-Sigmoid* | | 69.8% |

Figure 3: *For Inception and the batch-normalized variants, the number of training steps required to reach the maximum accuracy of Inception (72.2%), and the maximum accuracy achieved by the network.*

(Ioffe and Szegedy, 2015)

The authors state that with batch normalization

- samples have to be shuffled carefully,
- the learning rate can be greater,
- dropout and local normalization are not necessary,
- $L^2$ regularization influence should be reduced.



**Notes**

On the left graph, the blue diamonds show when the different variants reach the asymptotic performance of the inception network. With batch normalization, the same performance is achieved after a fraction of the training steps, even when the sigmoid is used as a non-linearity.
Not only batch normalization trains faster, but it eventually reaches better performance.

Deep MLP on a 2d "disc" toy example, with naive Gaussian weight initialization, cross-entropy, standard SGD, $\eta = 0.1$.

```
def create_model(with_batchnorm, dimh = 32, nb_layers = 16):
    modules = []

    modules.append(nn.Linear(2, dimh))
    if with_batchnorm: modules.append(nn.BatchNorm1d(dimh))
    modules.append(nn.ReLU())

    for d in range(nb_layers):
        modules.append(nn.Linear(dimh, dimh))
        if with_batchnorm: modules.append(nn.BatchNorm1d(dimh))
        modules.append(nn.ReLU())

    modules.append(nn.Linear(dimh, 2))

    return nn.Sequential(*modules)
```

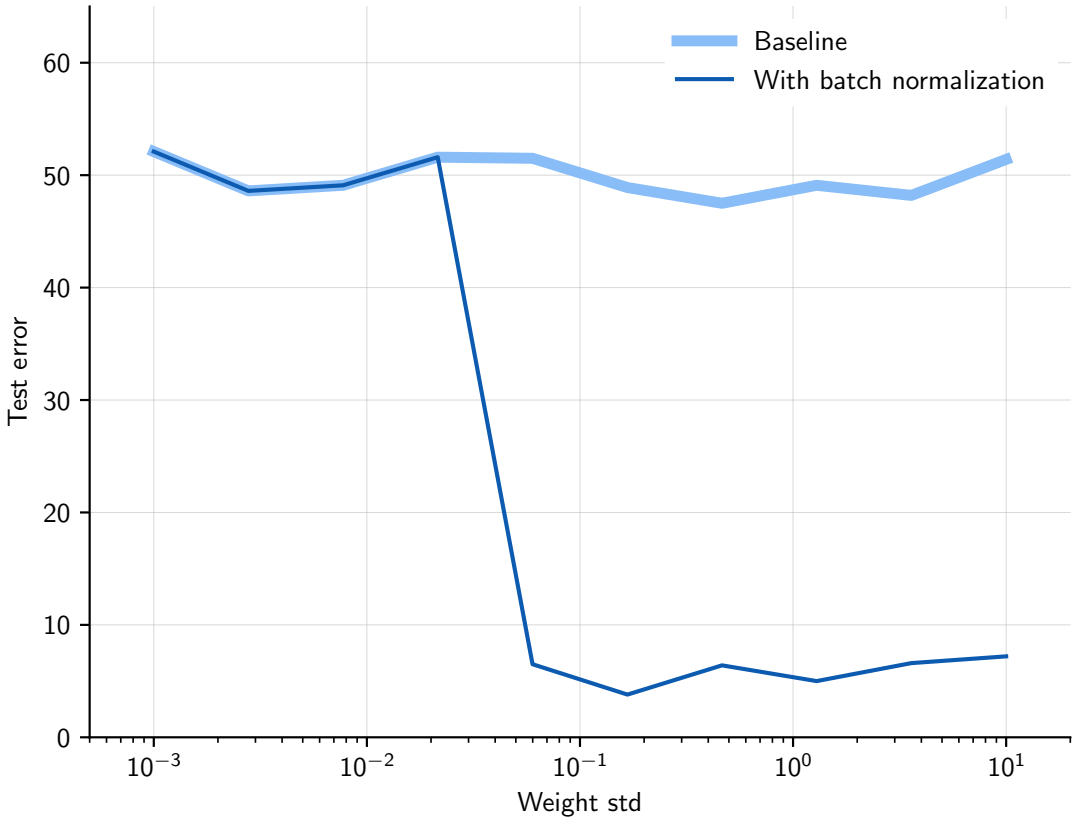
We try different standard deviations for the weights

```
with torch.no_grad():
    for p in model.parameters(): p.normal_(0, std)
```



**Notes**

We illustrate batch normalization with a 2D synthetic problem in which points inside a disk belong to class 1, and points outside belong to class 0.
Function `create_model` returns a MLP with a batch normalization module between each linear and ReLU modules if flag `with_batchnorm` is set. We keep in the experiment the default values for the number of layers and number of unit per layer.
The weights of the linear modules are initialized with a centered Gaussian noise, and not with the default normalizing PyTorch procedure that would compensate to some extent the absence of batch normalization.



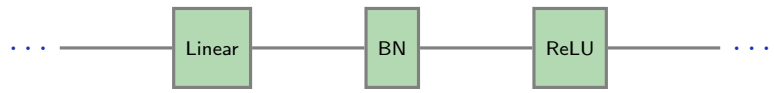



**Notes**

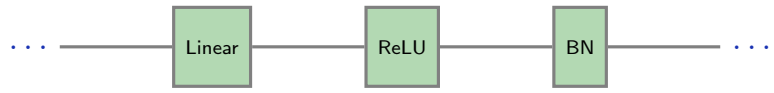
The graph shows the test error as a function of the standard deviation used for initialization of the weights.
The baseline curve shows that no matter what the standard deviation used for initialization, the network does not learn anything and the the test accuracy is 50% (balanced classes).
When the standard deviation becomes reasonably high, the network trained with batch normalization modules does almost perfect.
Batch normalization fixes very well inappropriate initialization of the weights, and beyond that, makes all the layers behave similarly and in a proper regime.

The position of batch normalization relative to the non-linearity is not clear.

> "We add the BN transform immediately before the nonlinearity, by normalizing $x = Wu + b$. We could have also normalized the layer inputs $u$, but since $u$ is likely the output of another nonlinearity, the shape of its distribution is likely to change during training, and constraining its first and second moments would not eliminate the covariate shift. In contrast, $Wu + b$ is more likely to have a symmetric, non-sparse distribution, that is 'more Gaussian' (Hyvärinen and Oja, 2000); normalizing it is likely to produce activations with a stable distribution. "
>
> (Ioffe and Szegedy, 2015)




However, this argument goes both ways: activations after the non-linearity are less "naturally normalized" and benefit more from batch normalization. Experiments are generally in favor of this solution, which is the current default.



As for dropout, using properly batch normalization on a convolutional map requires parameter-sharing.

The module `torch.BatchNorm2d` (respectively `torch.BatchNorm3d`) processes samples as multi-channels 2d maps (respectively multi-channels 3d maps) and normalizes each channel separately, with a $\gamma$ and a $\beta$ for each.

Another normalization in the same spirit is the **layer normalization** proposed by Ba et al. (2016).

Given a single sample $x \in \mathbb{R}^D$, it normalizes the components of $x$, hence normalizing activations across the layer instead of doing it across the batch

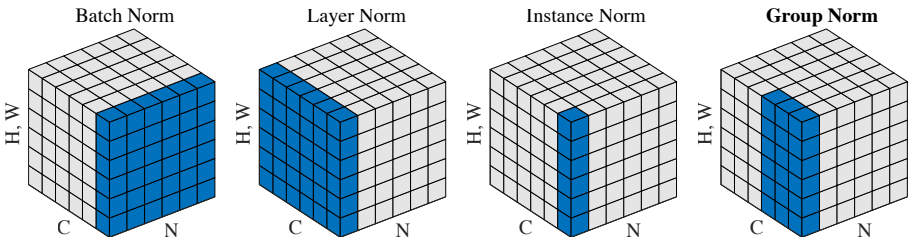
$$\mu = \frac{1}{D}\sum_{d=1}^{D} x_d$$

$$\sigma = \sqrt{\frac{1}{D}\sum_{d=1}^{D}(x_d - \mu)^2}$$

$$\forall d,\ y_d = \frac{x_d - \mu}{\sigma}$$

Although it gives slightly worst improvements than BN it has the advantage of behaving similarly in train and test, and processing samples individually.

These normalization schemes are examples of a larger class of methods.




(Wu and He, 2018)

## References


J. L. Ba, J. R. Kiros, and G. E. Hinton. **Layer normalization**. CoRR, abs/1607.06450, 2016.

A. Hyvärinen and E. Oja. **Independent component analysis: Algorithms and applications**. Neural Networks, 13(4-5):411–430, 2000.

S. Ioffe and C. Szegedy. **Batch normalization: Accelerating deep network training by reducing internal covariate shift**. In International Conference on Machine Learning (ICML), 2015.

Y. Wu and K. He. **Group normalization**. CoRR, abs/1803.08494, 2018.