EE-559 – Deep learning

6.4. Batch normalization

François Fleuret
https://fleuret.org/ee559/
Sun Sep 30 08:42:14 UTC 2018

RESEARCH INSTITUTE
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

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

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.

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)

> "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.

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

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

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

If $x_b \in \mathbb{R}^D, b = 1, \dots, B$ 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 $y_b \in \mathbb{R}^D$

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

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

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.**

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.**

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

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

```
>>> x = torch.empty(1000, 3).normal_()
>>> x = x * torch.tensor([2., 5., 10.]) + torch.tensor([-10., 25., 3.])
>>> x.mean(0)
tensor([ -9.9555,  24.9327,   3.0933])
>>> x.std(0)
tensor([ 1.9976,  4.9463,  9.8902])
>>> 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.])
Parameter containing:
tensor([ 1.,  2.,  3.])
>>> y = bn(x)
>>> y.mean(0)
tensor([ 2.0000,  4.0000,  8.0000])
>>> y.std(0)
tensor([ 1.0005,  2.0010,  3.0015])
```

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

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

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

Since **each input in the batch impacts all the outputs of the batch**, the derivative of the loss with respect to an input is quite complicated.

Since **each input in the batch impacts all the outputs of the batch**, the derivative of the loss with respect to an input is quite complicated.

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

Since **each input in the batch impacts all the outputs of the batch**, the derivative of the loss with respect to an input is quite complicated.

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

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

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

Since **each input in the batch impacts all the outputs of the batch**, the derivative of the loss with respect to an input is quite complicated.

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

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

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

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

Since **each input in the batch impacts all the outputs of the batch**, the derivative of the loss with respect to an input is quite complicated.

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

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

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

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

In standard implementation, $\hat{m}$ and $\hat{v}$ for test are estimated with a moving average during train, so that it can be implemented as a module which does not need an additional pass through the samples during training.

Results on ImageNet's LSVRC2012:

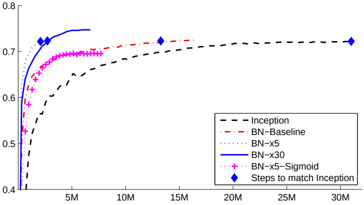


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

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)

Results on ImageNet's LSVRC2012:

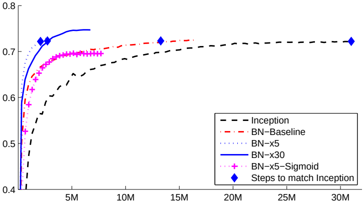


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

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

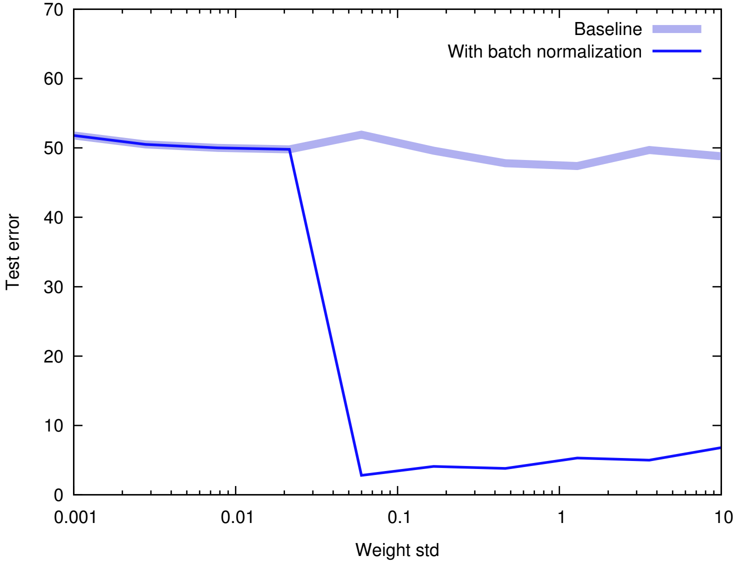
- samples have to be shuffled carefully,
- the learning rate can be greater,
- dropout and local normalization are not necessary,
- L^2 regularization influence should be reduced.

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, nc = 32, depth = 16):
    modules = []

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

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

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

    return nn.Sequential(*modules)
```

We try different standard deviations for the weights

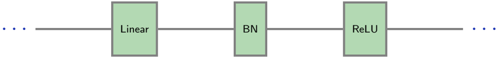
```
with torch.no_grad():
    for p in model.parameters(): p.normal_(0, std)
```


Baseline
With batch normalization
Test error
Weight std
70
60
50
40
30
20
10
0
0.001
0.01
0.1
1
10

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."

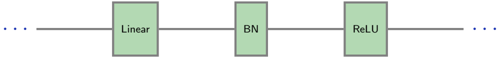
(Ioffe and Szegedy, 2015)



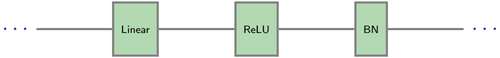
The position of batch normalization relative to the non-linearity is not clear.

> "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 γ and a β 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

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

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

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

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.

The end

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.