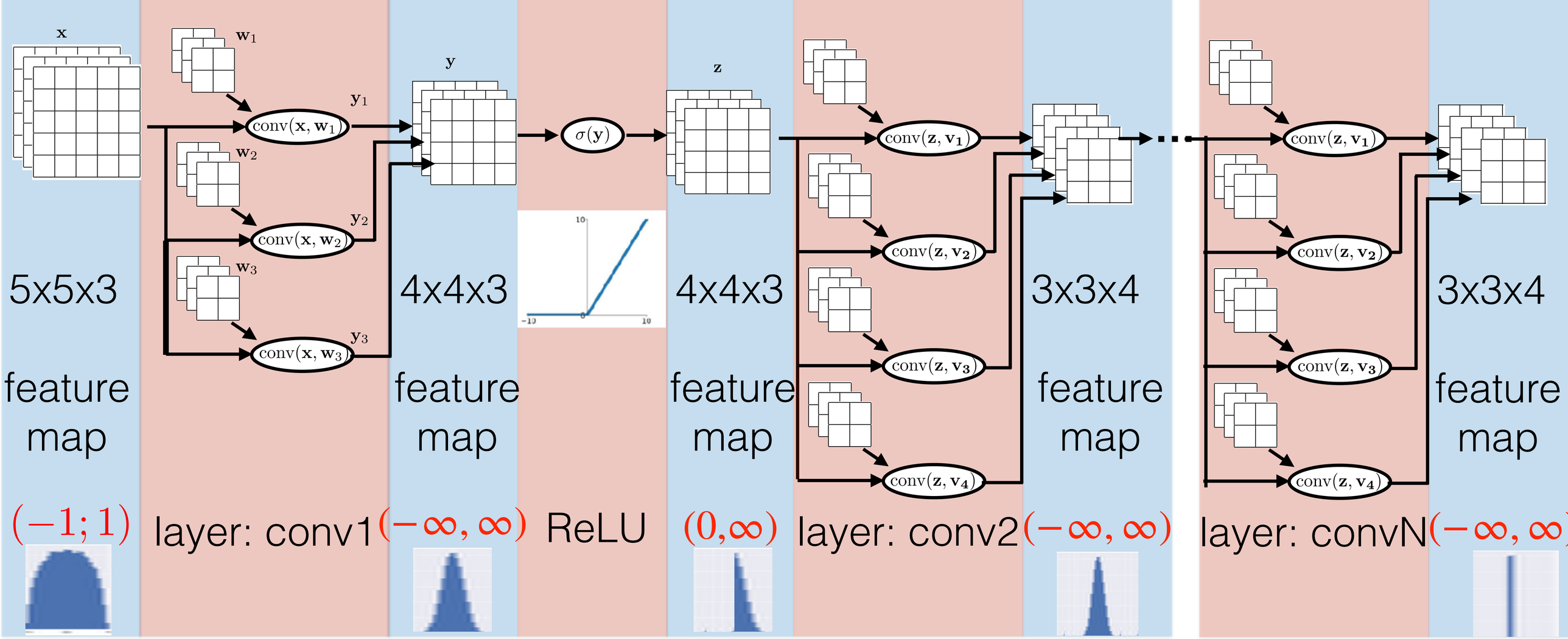
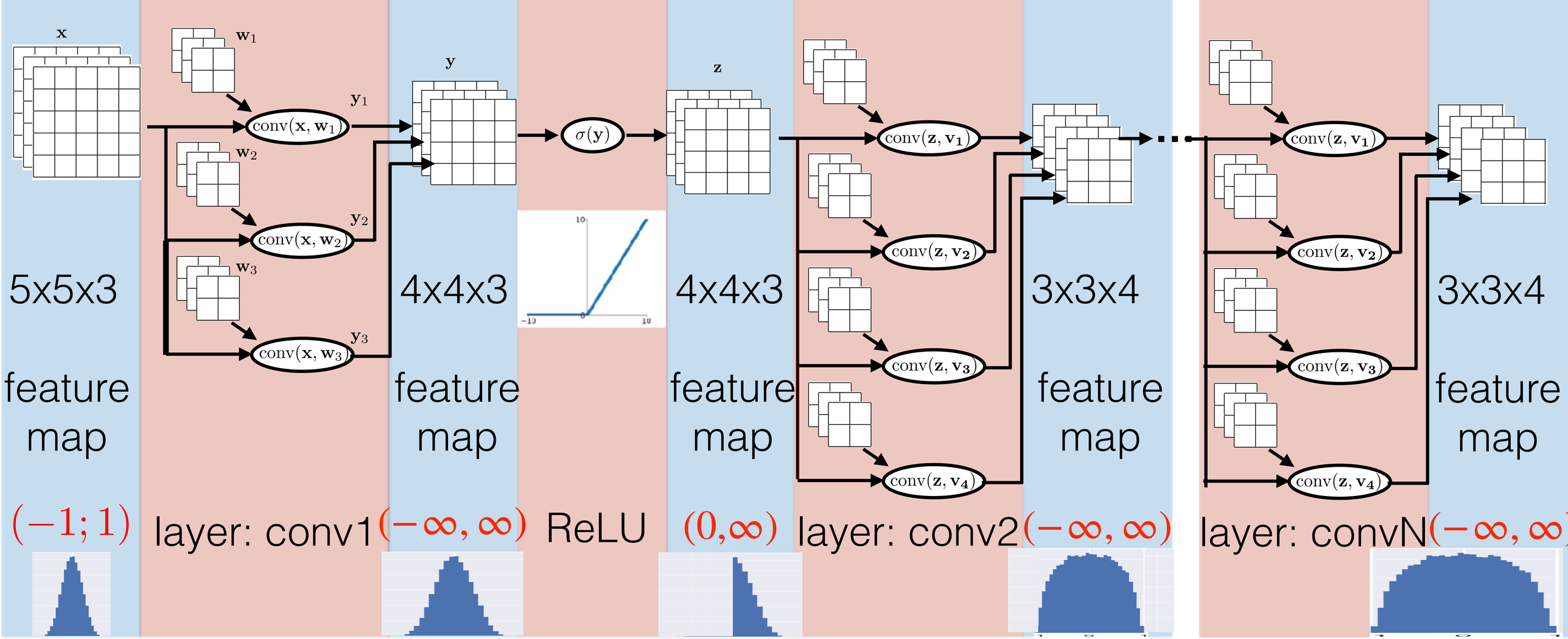
Elementary layers and their issues

Karel Zimmermann
Czech Technical University in Prague
Faculty of Electrical Engineering, Department of Cybernetics



Small weights => values converging to zeros



Small weights => values converging to zeros

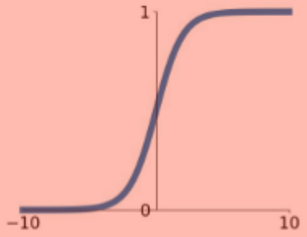
Large weights => values exploding to infinity

Gradient is also convolution => easily diminish or explode

Activation functions

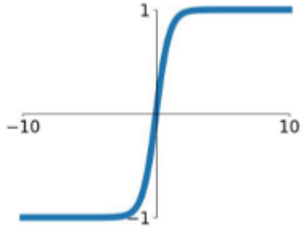
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



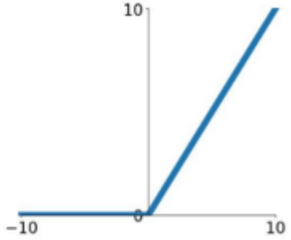
tanh

$$\tanh(x)$$



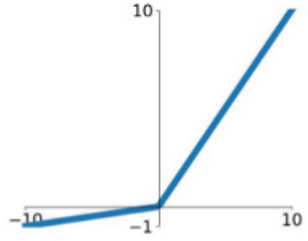
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

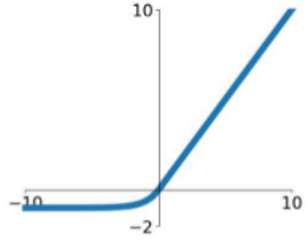


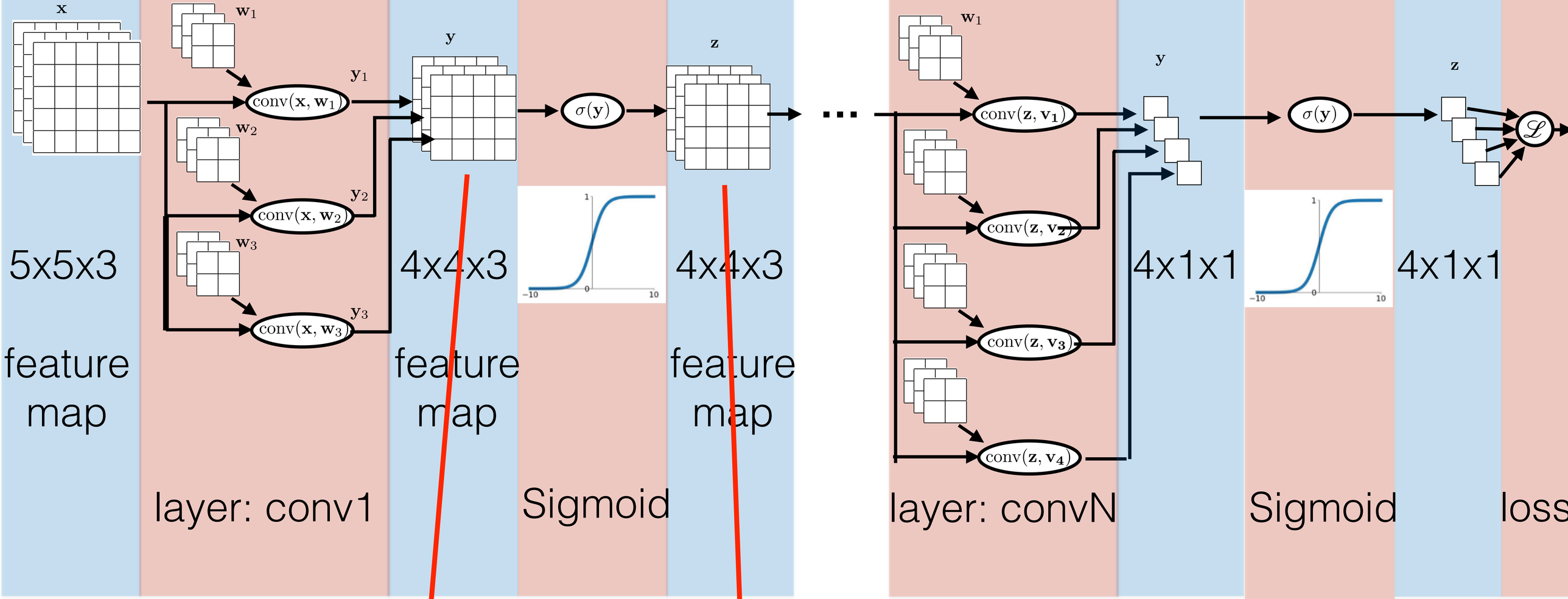
Maxout

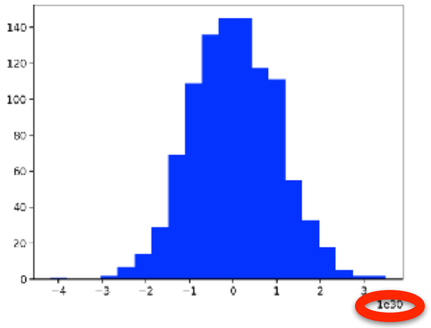
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

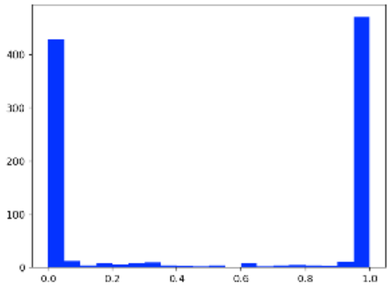
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



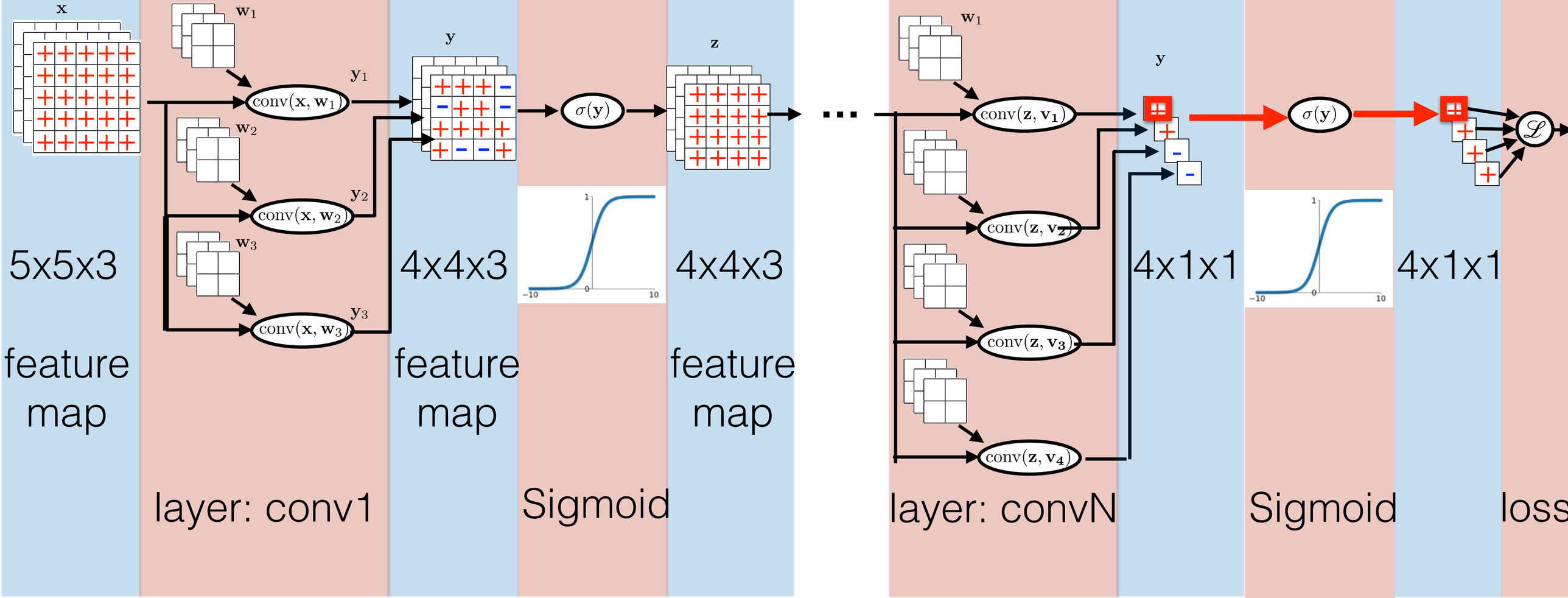






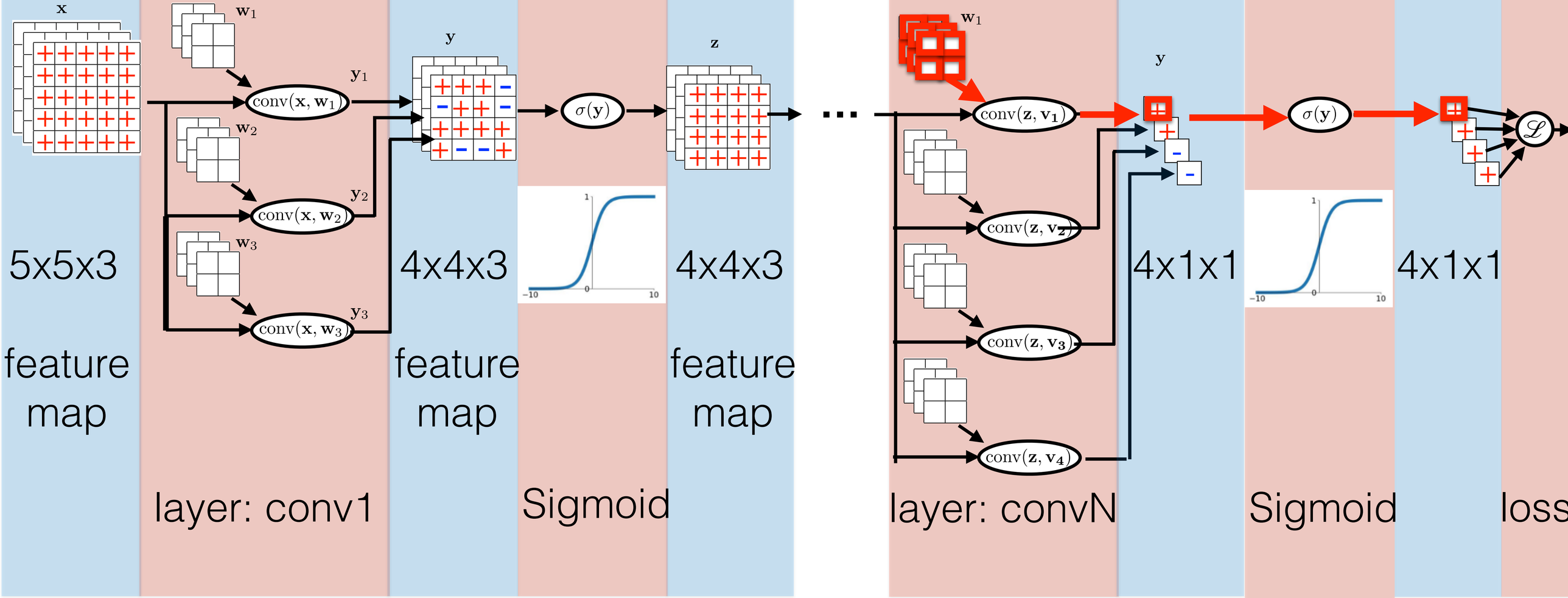
Large weights suppresses gradient

Small weights suppresses gradient



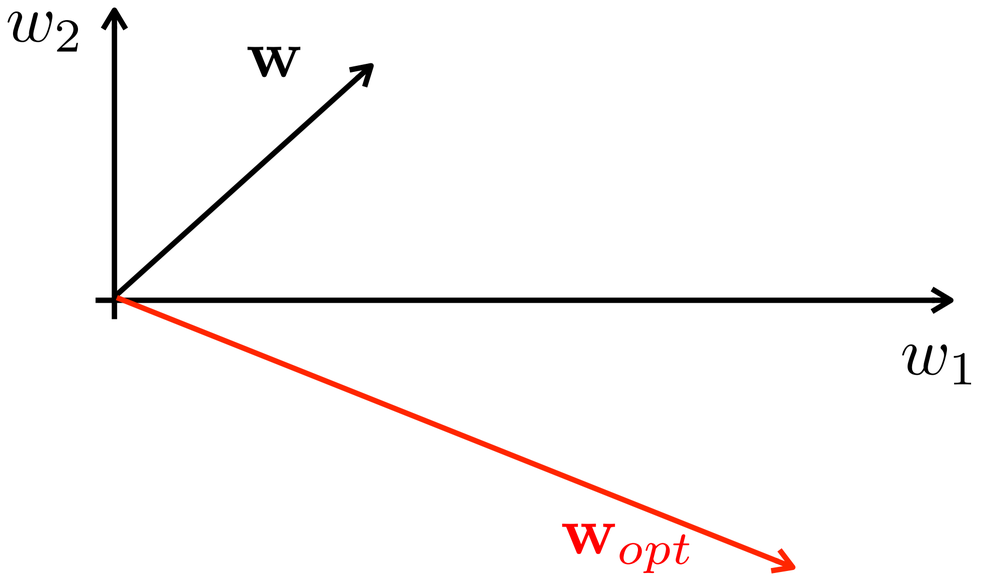
$$\mathbf{g} = \frac{\partial \sigma(\mathbf{y})}{\partial \mathbf{y}} \cdot \mathbf{p}$$

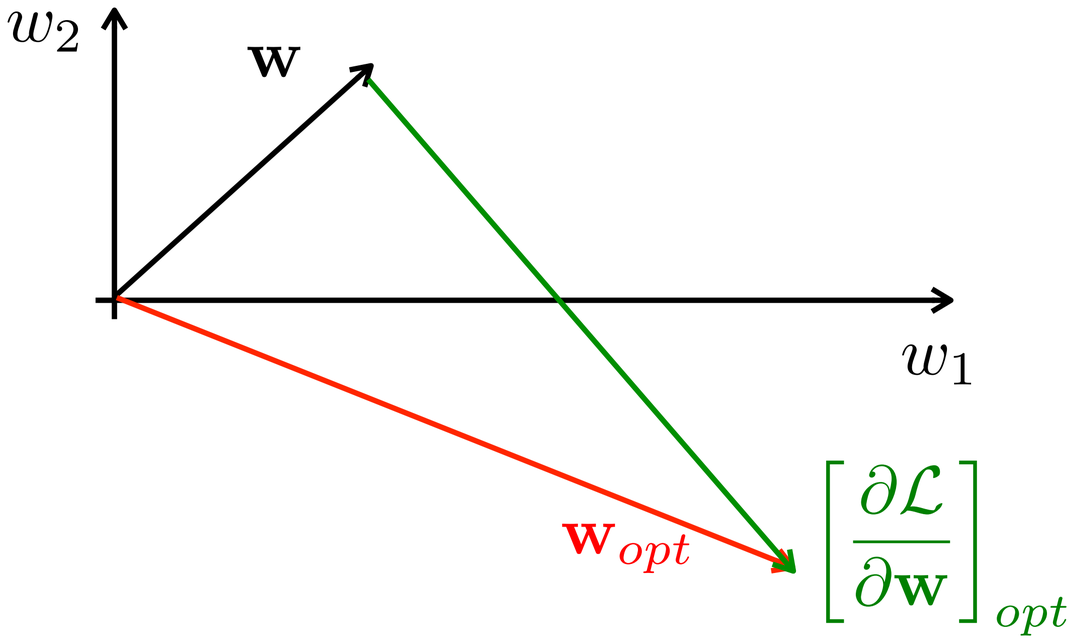
>0 >0 >0

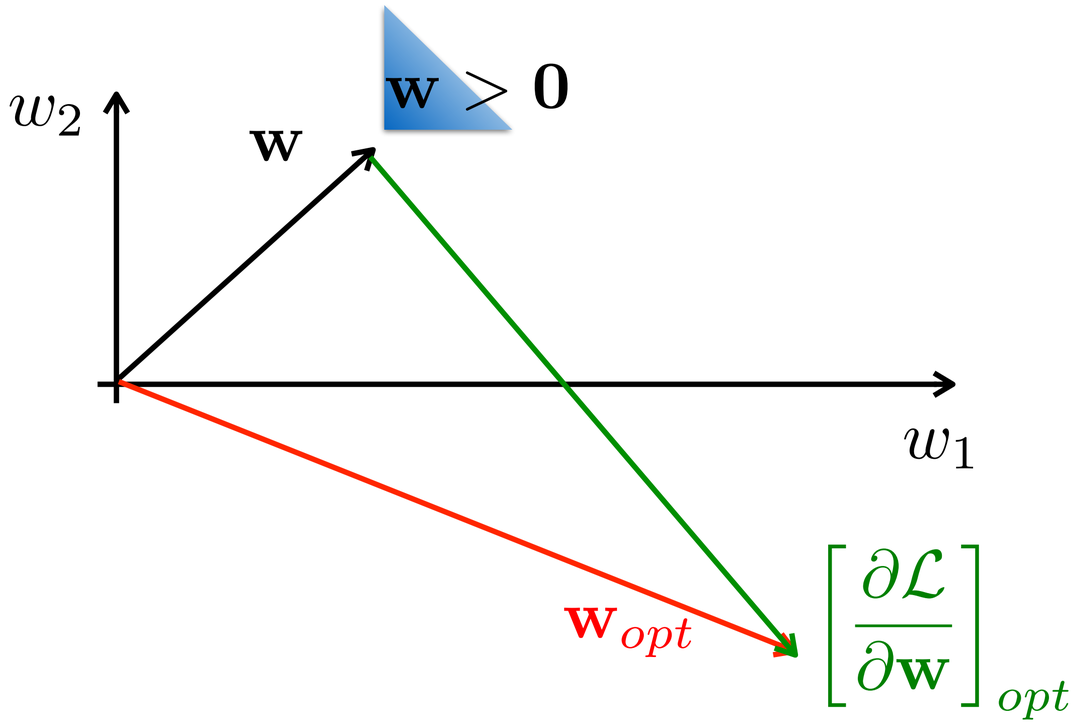


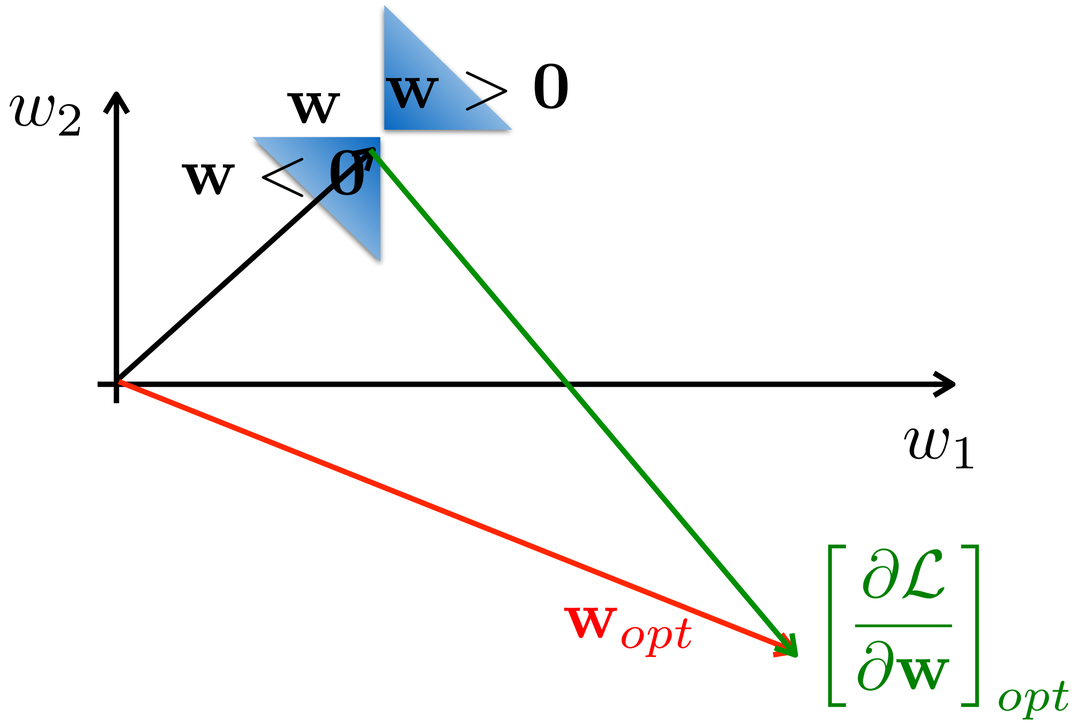
$$\frac{\partial \text{conv}(\mathbf{z}, \mathbf{w}_1)}{\partial \mathbf{w}_1} \approx \text{conv}(\mathbf{g}, \mathbf{z}) \qquad \mathbf{g} = \frac{\partial \sigma(\mathbf{y})}{\partial \mathbf{y}} \cdot \mathbf{p}$$

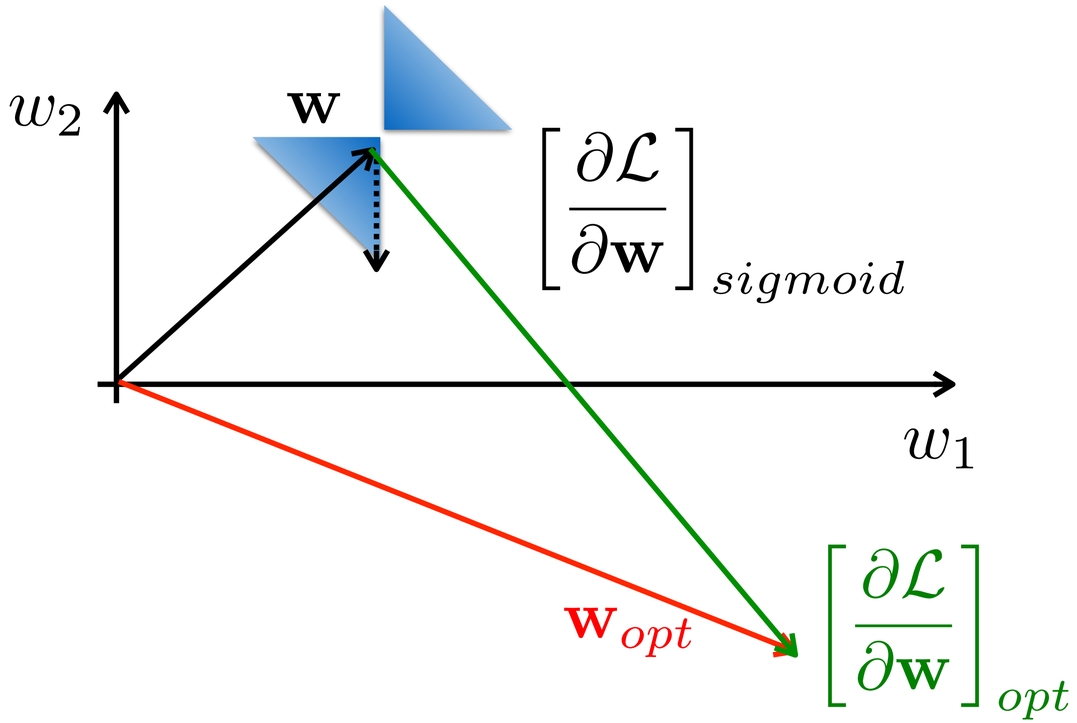
Gradient to **all** weights is positive	>0	>0	>0	>0
Gradient to **all** weights is negative	<0	<0	>0	<0

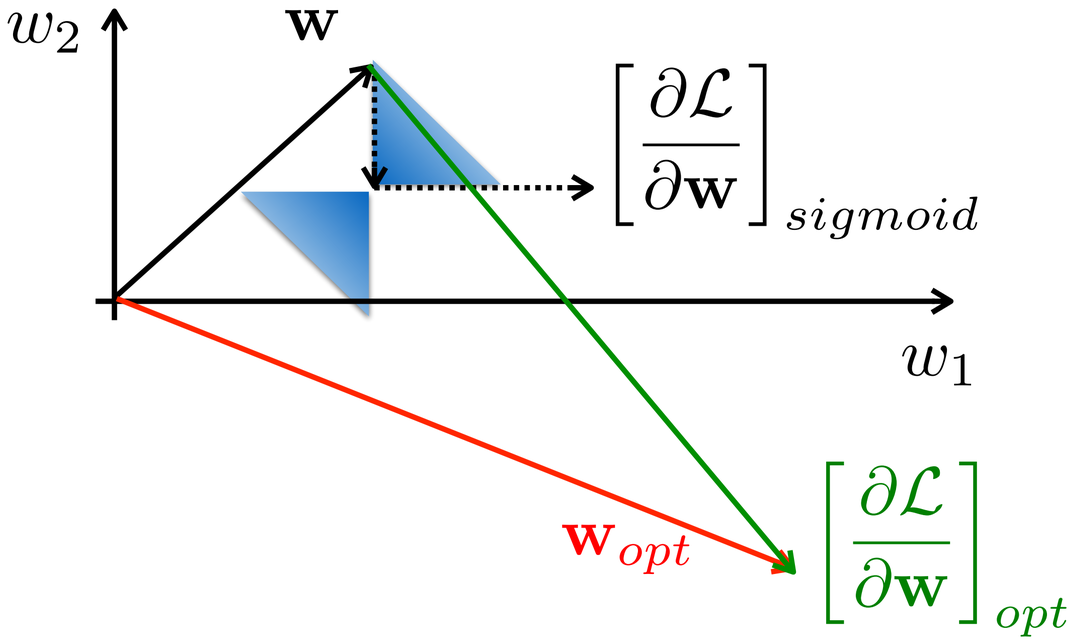

w_2
$\mathbf{w}$
w_1
$\mathbf{w}_{opt}$

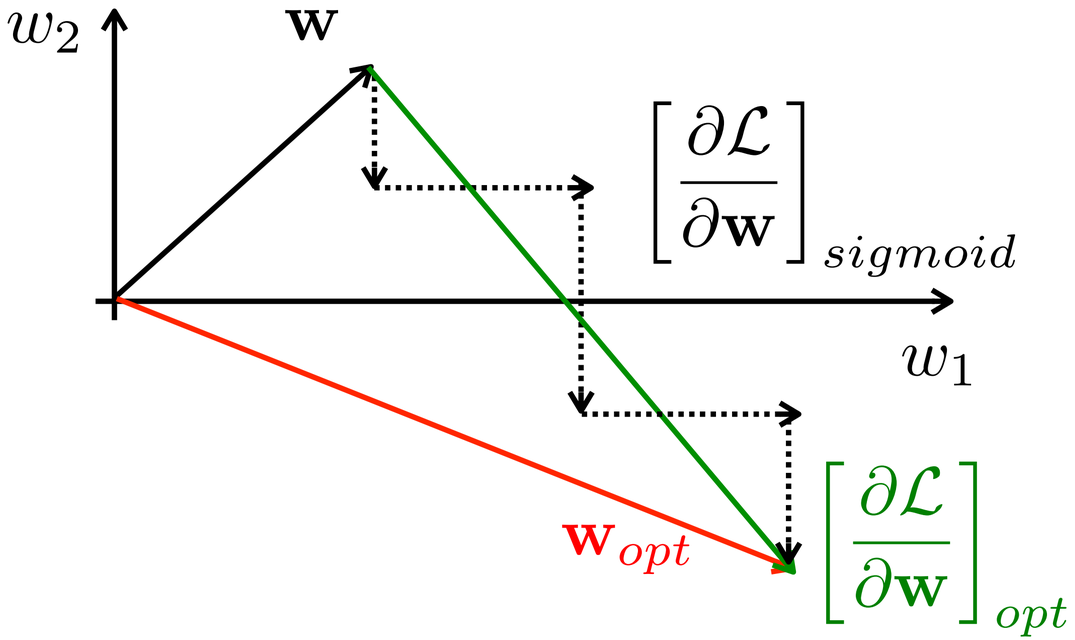

w_2
$\mathbf{w}$
w_1
$\mathbf{w}_{opt}$
$\left[\frac{\partial \mathcal{L}}{\partial \mathbf{w}}\right]_{opt}$

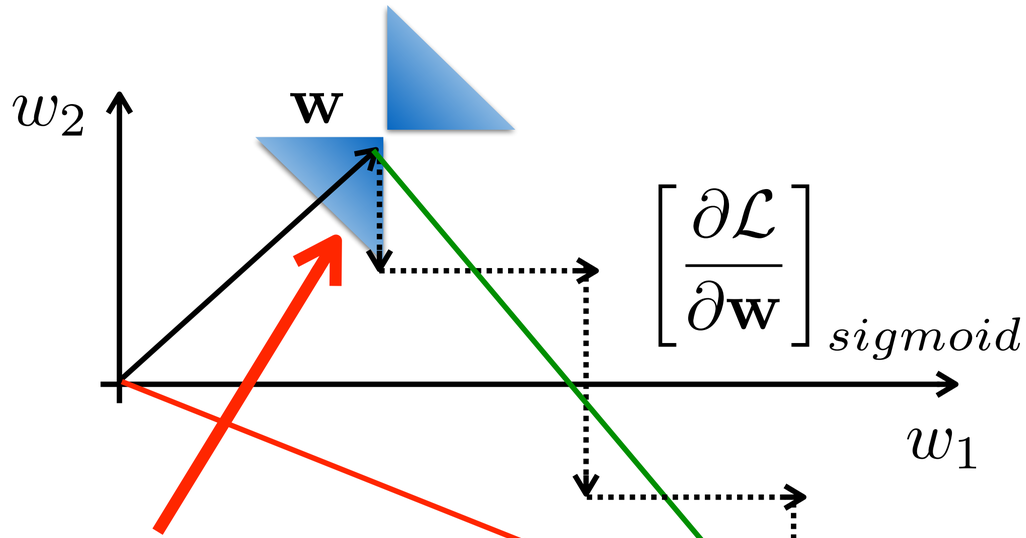

$\mathbf{w} > \mathbf{0}$
w_2
$\mathbf{w}$
w_1
$\mathbf{w}_{opt}$
$\left[\frac{\partial \mathcal{L}}{\partial \mathbf{w}}\right]_{opt}$


w_2
$\mathbf{w} > \mathbf{0}$
$\mathbf{w}$
$\mathbf{w} < \mathbf{0}$
w_1
$\mathbf{w}_{opt}$
$\left[\frac{\partial \mathcal{L}}{\partial \mathbf{w}}\right]_{opt}$

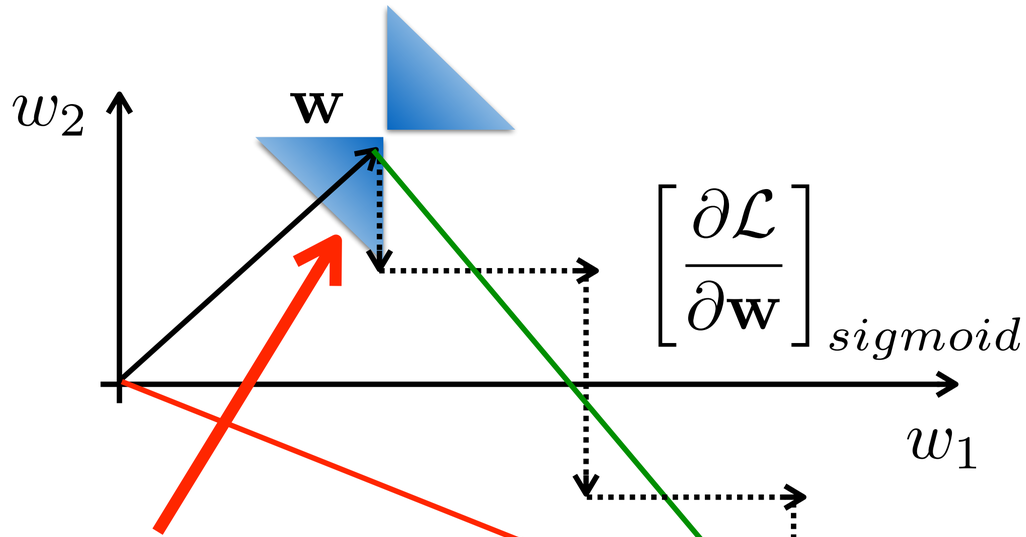

w_2
$\mathbf{w}$
$\left[\frac{\partial \mathcal{L}}{\partial \mathbf{w}}\right]_{sigmoid}$
w_1
$\mathbf{w}_{opt}$
$\left[\frac{\partial \mathcal{L}}{\partial \mathbf{w}}\right]_{opt}$


w_2
$\mathbf{w}$
$\left[\frac{\partial \mathcal{L}}{\partial \mathbf{w}}\right]_{sigmoid}$
w_1
$\mathbf{w}_{opt}$
$\left[\frac{\partial \mathcal{L}}{\partial \mathbf{w}}\right]_{opt}$


w_2
$\mathbf{w}$
$\left[\frac{\partial \mathcal{L}}{\partial \mathbf{w}}\right]_{sigmoid}$
w_1
$\mathbf{w}_{opt}$
$\left[\frac{\partial \mathcal{L}}{\partial \mathbf{w}}\right]_{opt}$



how big fraction is the blue region in 10-dim space?



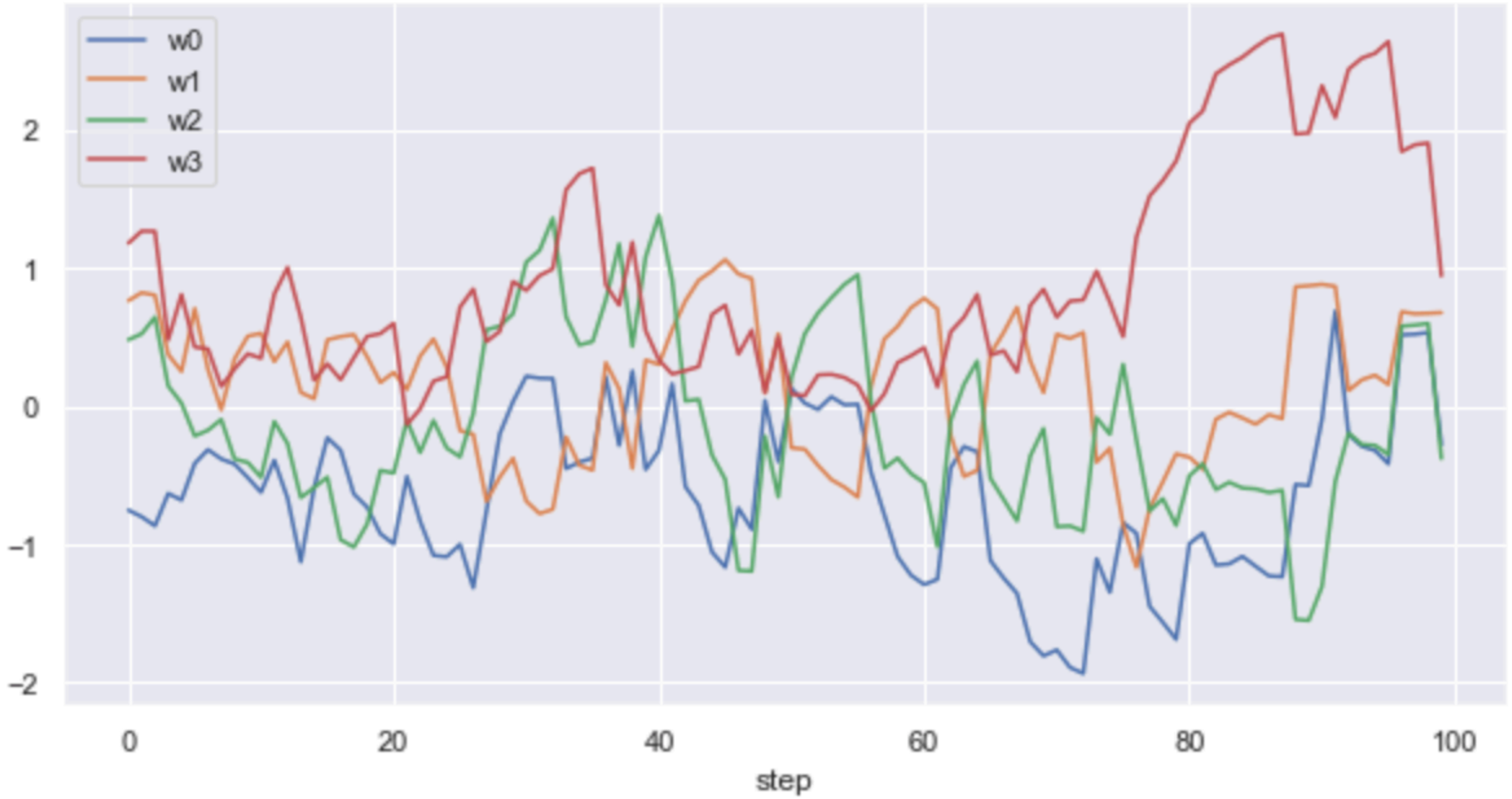
2/(2^10)

how big fraction is the blue region in 10-dim space?

- what happens when sigmoid input is only positive?

sigmoid activation function

- what happens when sigmoid input is only positive?

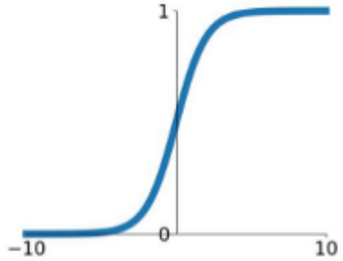


tanh activation function

Activation functions

Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$

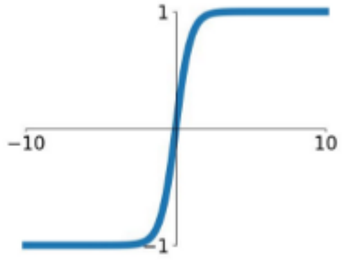


- zero gradient when saturated
- not zero-centered (pos. output)=>zig-zag
- computationally expensive

PyTorch: `nn.Sigmoid()`

tanh

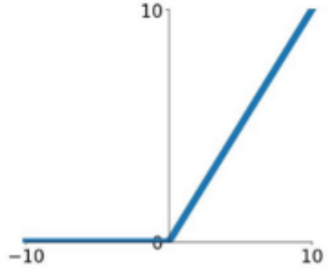
$\tanh(x)$



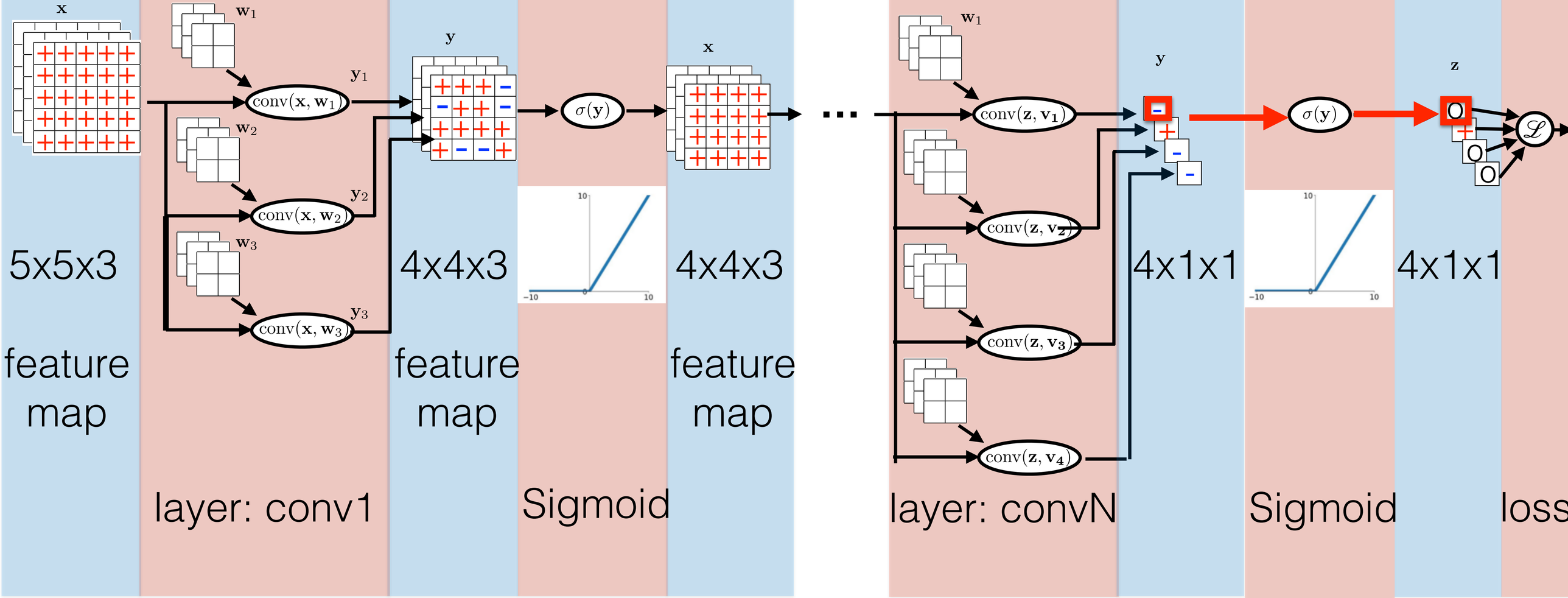
- zero gradient when saturated
- ~~not zero-centered (only positive ouputs)~~
- computationally expensive
- PyTorch: `nn.Tanh()`

ReLU

$\max(0, x)$



- ~~zero gradient when saturated~~ *(dead ReLU!)*
- not zero-centered (only positive ouputs)
- ~~computationally expensive~~
- PyTorch: `nn.ReLu()`
- backprop: $\frac{\partial \max(0,x)}{\partial x} = \begin{cases} 0 & x < 0 \\ 1 & \text{otherwise} \end{cases}$



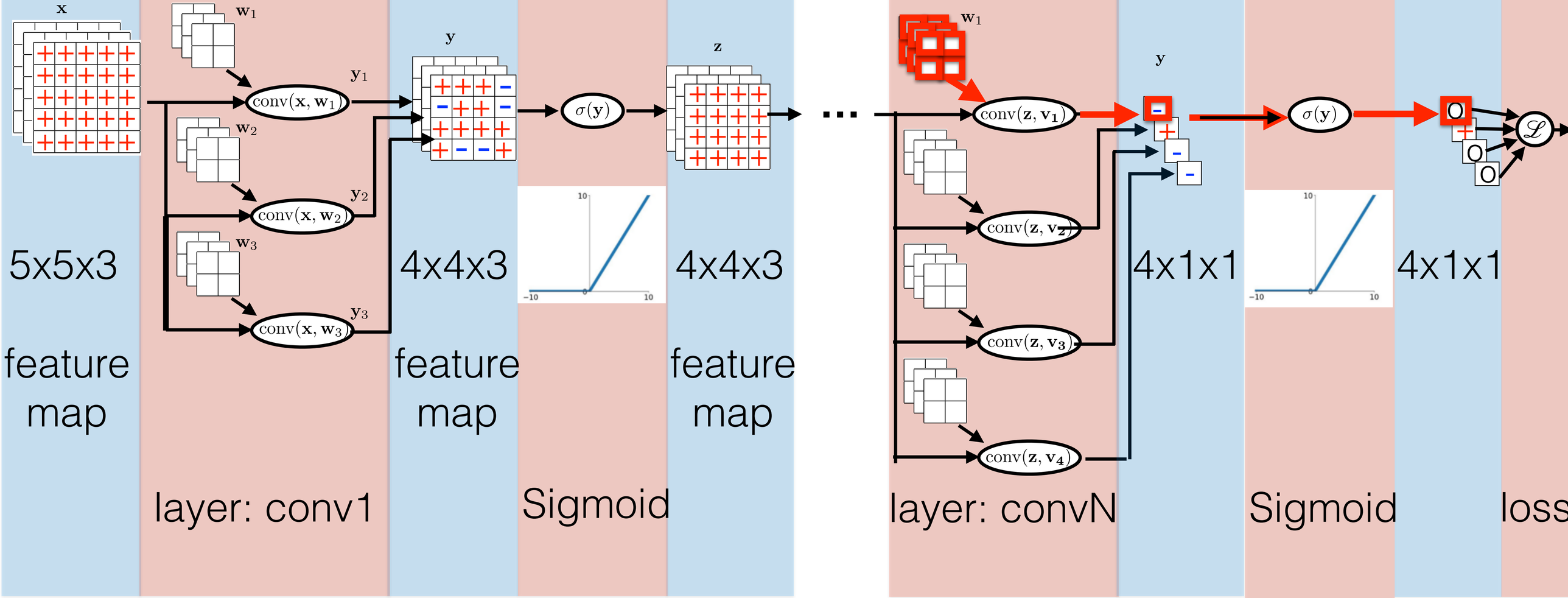
$$\mathbf{g} = \frac{\partial \sigma(\mathbf{y})}{\partial \mathbf{y}}$$

$\mathbf{p}$

$=0$

>0

<0



$$\frac{\partial \text{conv}(\mathbf{z}, \mathbf{w}_1)}{\partial \mathbf{w}_1} \approx \text{conv}(\mathbf{g}, \mathbf{z}) \qquad \mathbf{g} = \frac{\partial \sigma(\mathbf{y})}{\partial \mathbf{y}} \qquad \mathbf{p}$$

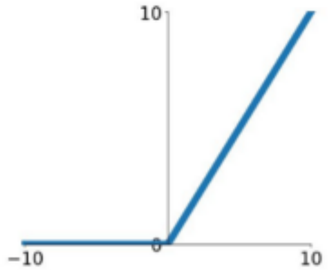
$=0$ $\qquad$ $=0$ $\qquad$ >0

<0

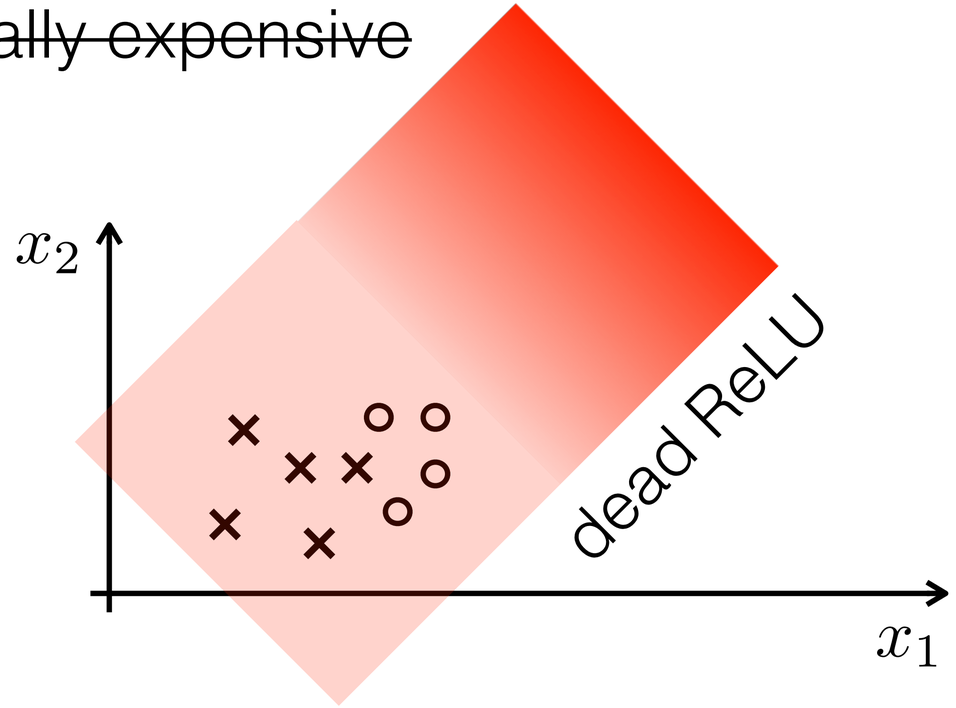
Activation functions

ReLU

$\max(0, x)$



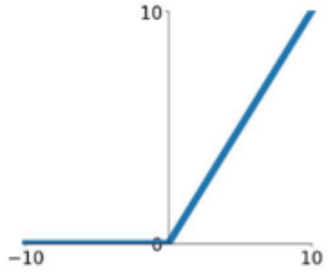
- ~~zero gradient when saturated~~ *(partially => dead ReLU!)*
- not zero-centered (only positive ouputs)
- ~~computationally expensive~~



Several ReLU variants

ReLU

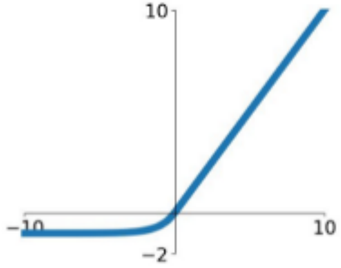
$\max(0, x)$



- PyTorch: `nn.ReLu()`
- backprop: $\frac{\partial \max(0,x)}{\partial x} = \begin{cases} 0 & x < 0 \\ 1 & \text{otherwise} \end{cases}$

ELU

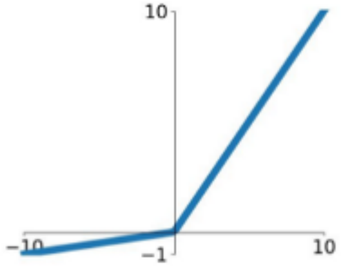
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



- Suppress discontinuity around zero
- PyTorch: `nn.LeakyReLU(alpha=1)`

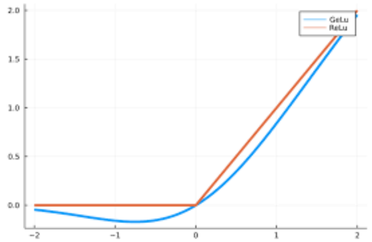
Leaky ReLU

$\max(0.1x, x)$



- Does not suffer from dead ReLUs
- backprop: $\frac{\partial \max(0.1x,x)}{\partial x} = \begin{cases} 0.1 & x < 0 \\ 1 & \text{otherwise} \end{cases}$

GELU



- Suppress discontinuity around zero
- Does not suffer from dead ReLUs
- Improved functionality in transformers

Summary so far observed issues

- **Values** in feature maps can easily **explode, diminish** or get **biased**
- **Gradients** can easily **explode** or **diminish** or get **biased**
- We need to keep reasonable values - **what are reasonable values?**

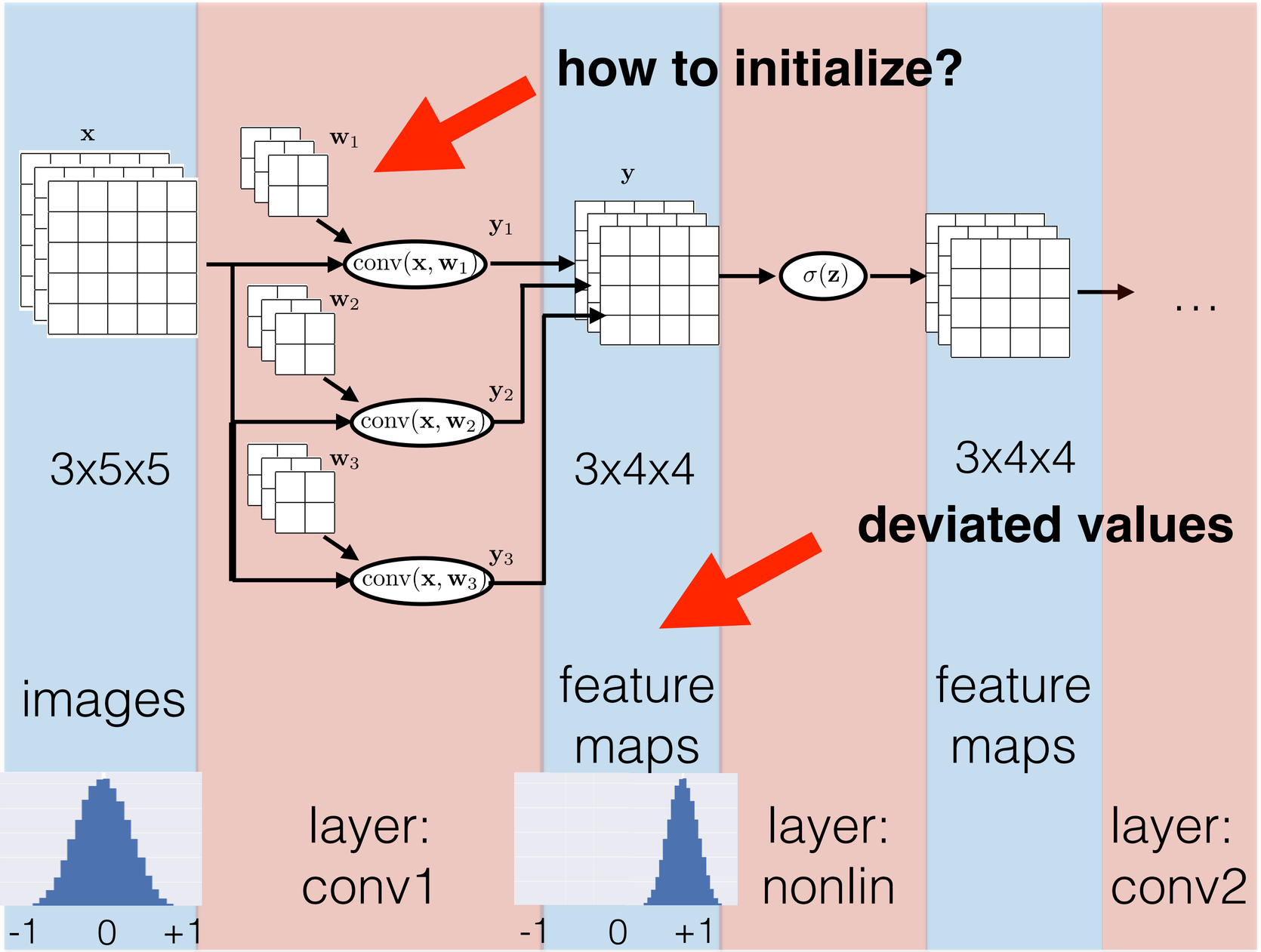
Initialization

Normalization

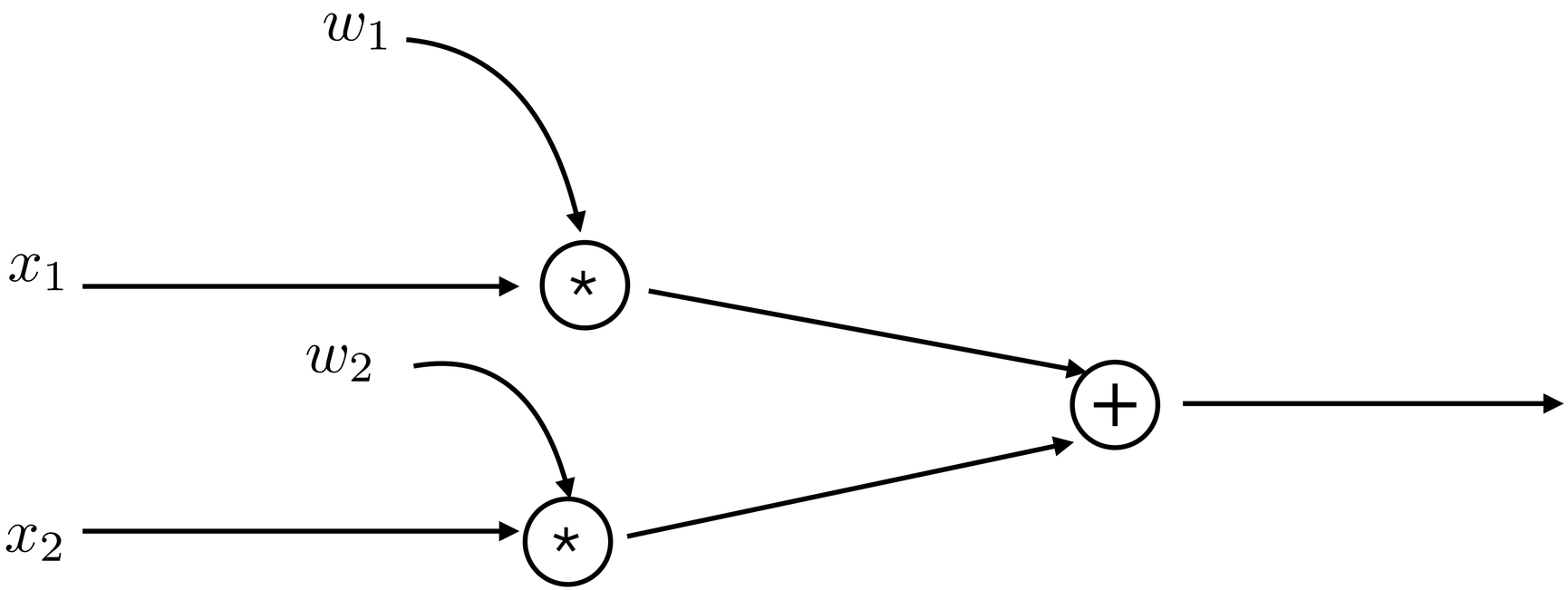
$$\mu_x = 0 \quad \text{var}(x) = 1$$

Batch normalization layer [Ioffe and Szegedy 2015]

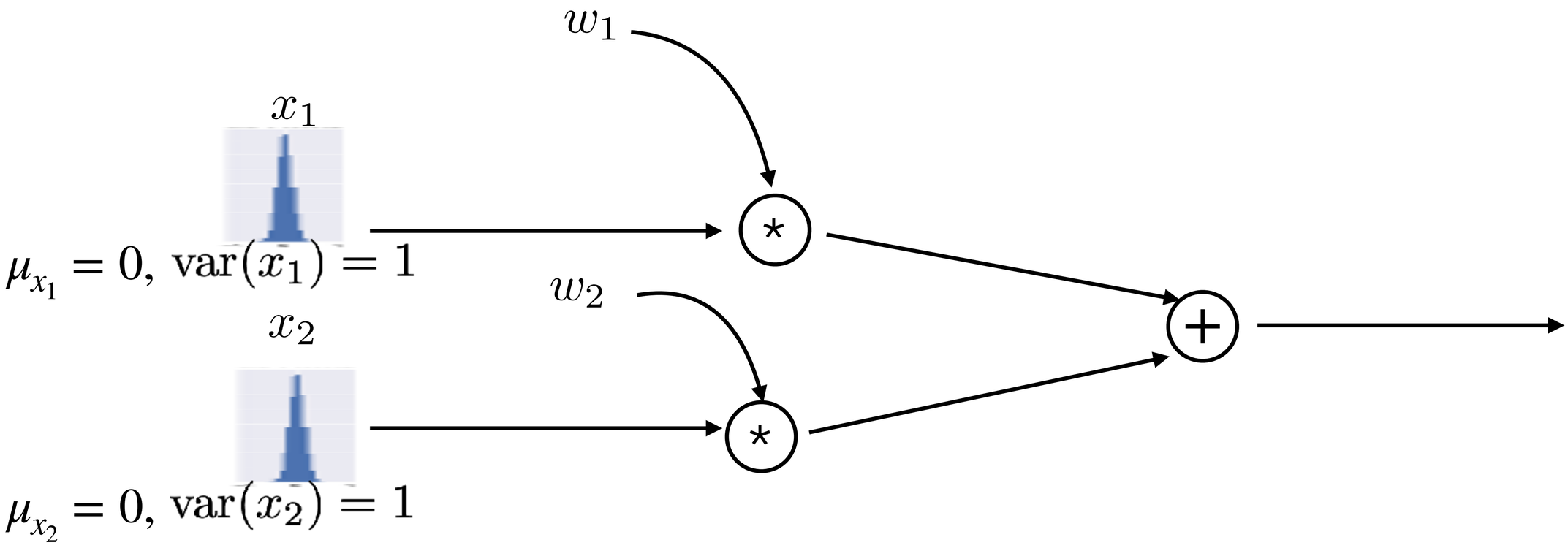
https://arxiv.org/pdf/1502.03167.pdf (over 6k citation)



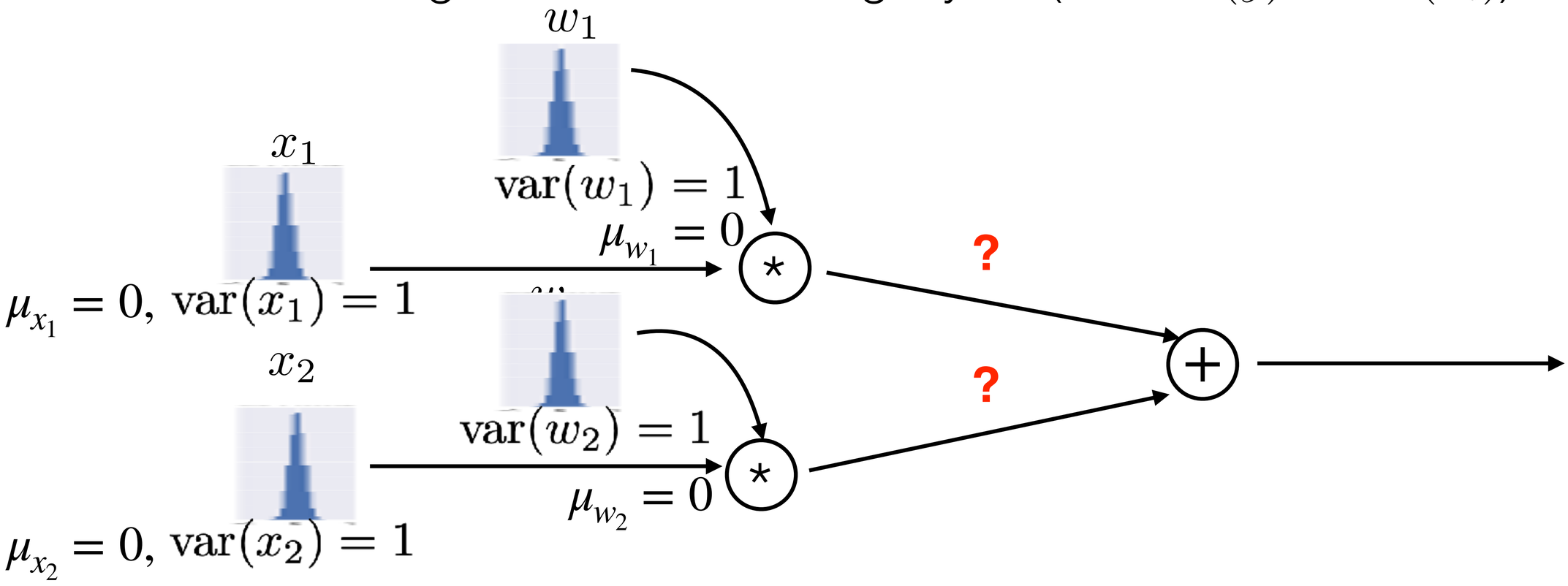
Preserve signal variance among layers (i.e. $\mathrm{var}(y) = \mathrm{var}(x_i)$)



Preserve signal variance among layers (i.e. $\mathrm{var}(y) = \mathrm{var}(x_i)$)

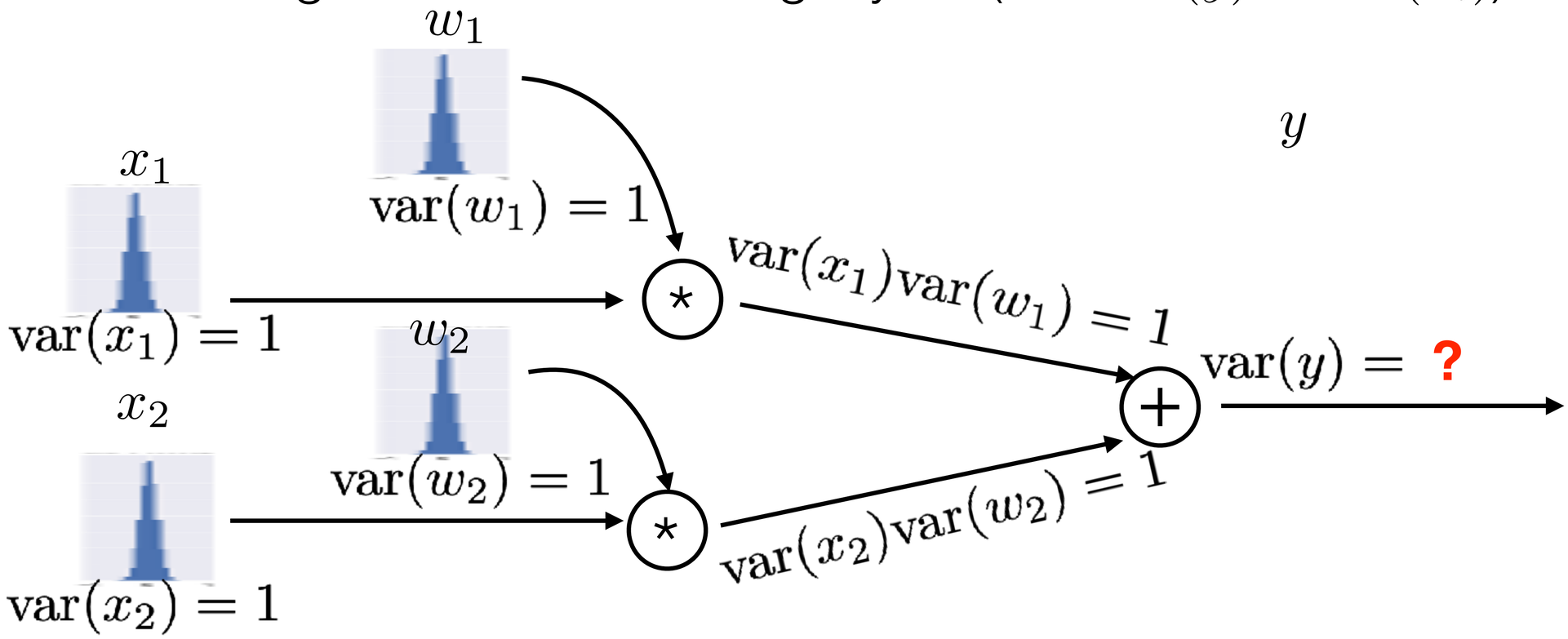


Preserve signal variance among layers (i.e. $\mathrm{var}(y) = \mathrm{var}(x_i)$)



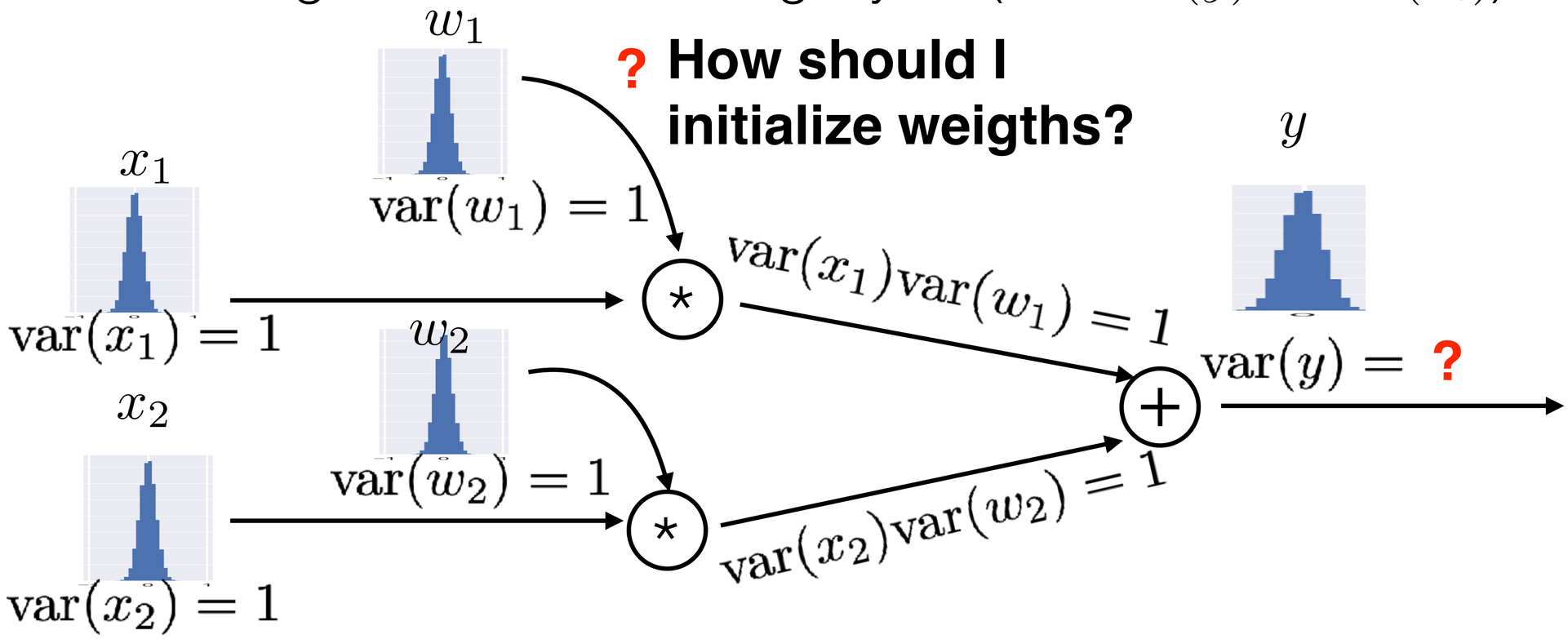
$$\mathrm{var}(x_1 w_1) = (\mathrm{var}(x_1) + \mu_{x_1}^2)(\mathrm{var}(w_1) + \mu_{w_1}^2) - \mu_{x_1}^2 \mu_{w_1}^2 = \mathrm{var}(x_1)\mathrm{var}(w_1) = 1$$

Preserve signal variance among layers (i.e. $\mathrm{var}(y) = \mathrm{var}(x_i)$)



$$\mathrm{var}(x_1 w_1) = (\mathrm{var}(x_1) + \mu_{x_1}^2)(\mathrm{var}(w_1) + \mu_{w_1}^2) - \mu_{x_1}^2 \mu_{w_1}^2 = \mathrm{var}(x_1)\mathrm{var}(w_1) = 1$$

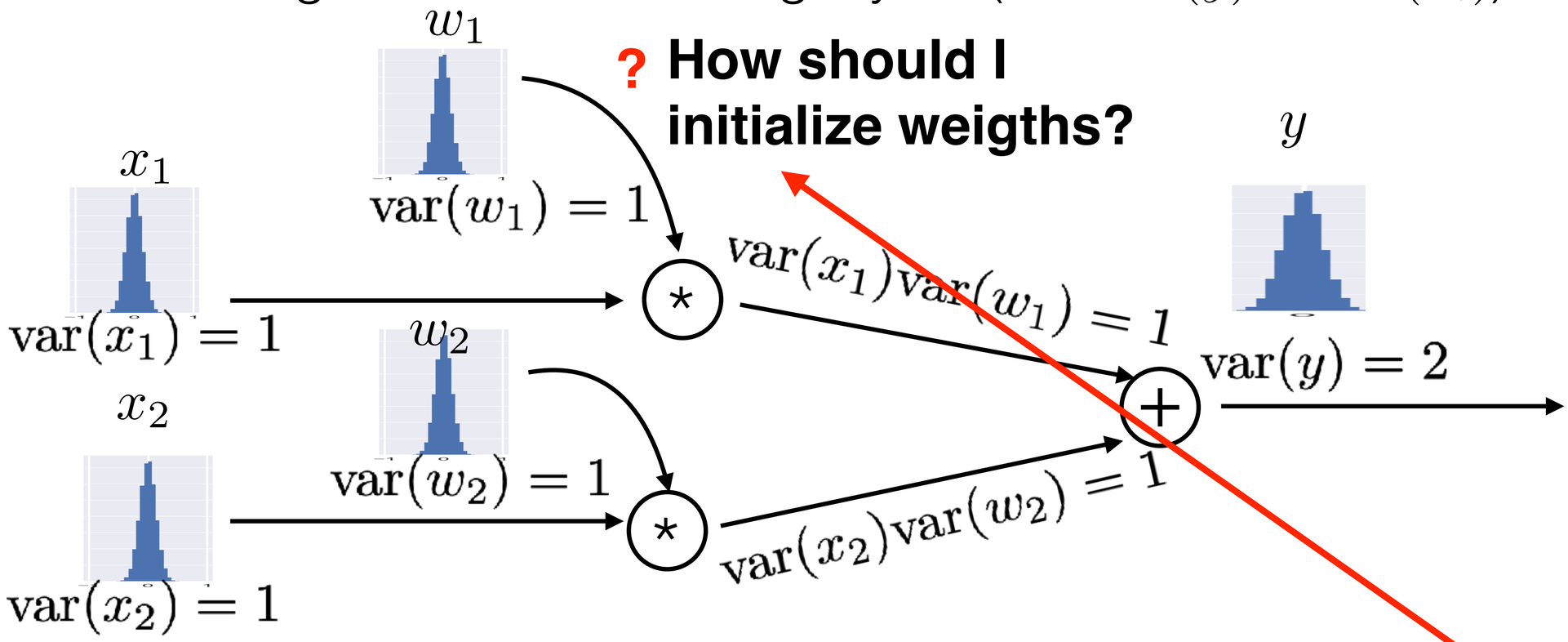
Preserve signal variance among layers (i.e. $\mathrm{var}(y) = \mathrm{var}(x_i)$)



$$\mathrm{var}(x_1 w_1) = (\mathrm{var}(x_1) + \mu_{x_1}^2)(\mathrm{var}(w_1) + \mu_{w_1}^2) - \mu_{x_1}^2 \mu_{w_1}^2 = \mathrm{var}(x_1)\mathrm{var}(w_1) = 1$$

$$\mathrm{var}(y) = \mathrm{var}(x_1 w_1 + x_2 w_2) = \mathrm{var}(x_1 w_1) + \mathrm{var}(x_2 w_2) = 2$$

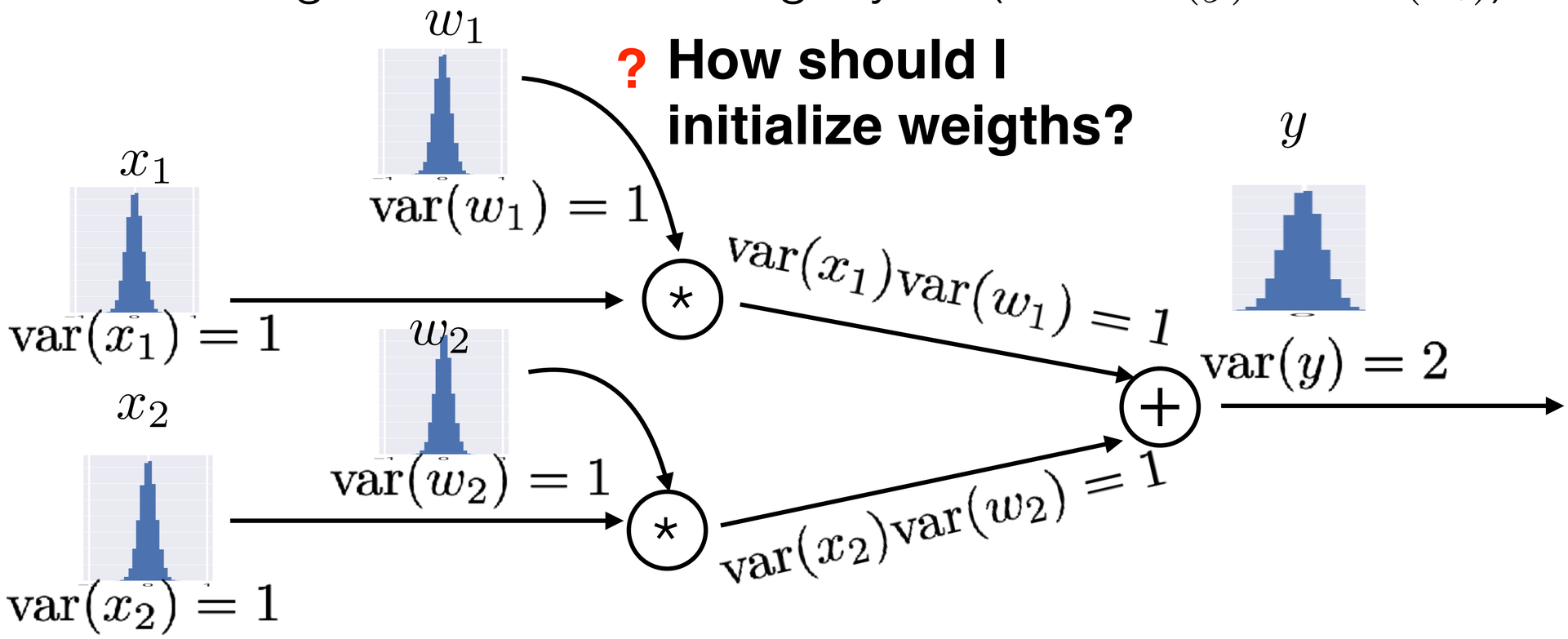
Preserve signal variance among layers (i.e. $\mathrm{var}(y) = \mathrm{var}(x_i)$)



$$\mathrm{var}(x_1 w_1) = (\mathrm{var}(x_1) + \mu_{x_1}^2)(\mathrm{var}(w_1) + \mu_{w_1}^2) - \mu_{x_1}^2 \mu_{w_1}^2 = \mathrm{var}(x_1)\mathrm{var}(w_1) = 1$$

$$\mathrm{var}(y) = \mathrm{var}(x_1 w_1 + x_2 w_2) = \mathrm{var}(x_1 w_1) + \mathrm{var}(x_2 w_2) = 2 \Rightarrow \mathrm{var}(w_i) = \frac{1}{2}$$

Preserve signal variance among layers (i.e. $\mathrm{var}(y) = \mathrm{var}(x_i)$)



$$\mathrm{var}(x_1 w_1) = (\mathrm{var}(x_1) + \mu_{x_1}^2)(\mathrm{var}(w_1) + \mu_{w_1}^2) - \mu_{x_1}^2 \mu_{w_1}^2 = \mathrm{var}(x_1)\mathrm{var}(w_1) = 1$$

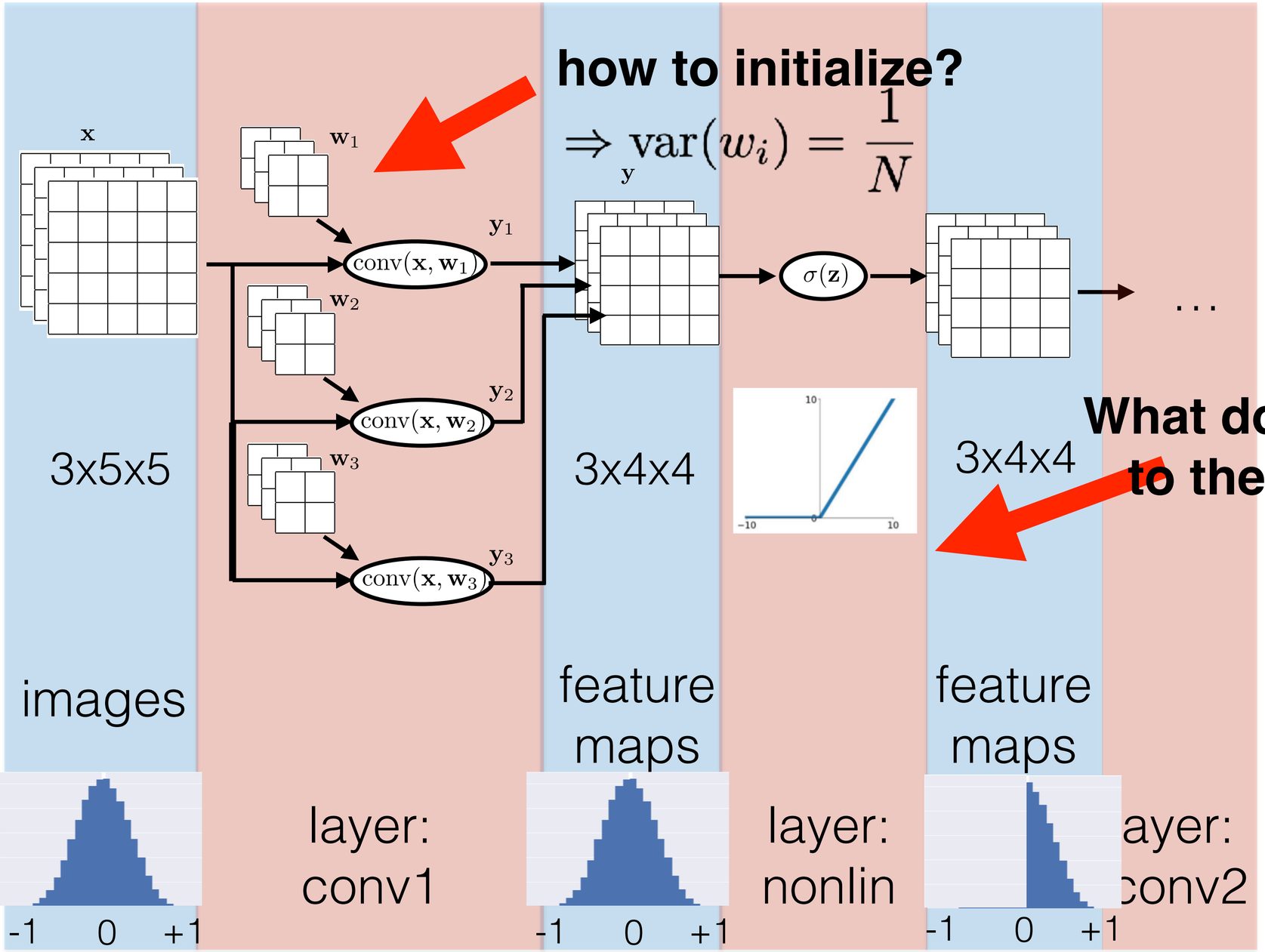
$$\mathrm{var}(y) = \mathrm{var}(x_1 w_1 + x_2 w_2) = \mathrm{var}(x_1 w_1) + \mathrm{var}(x_2 w_2) = 2 \Rightarrow \mathrm{var}(w_i) = \frac{1}{2}$$

$$\mathrm{var}(y) = \mathrm{var}(w_1 x_1 + w_2 x_2 + \cdots + w_N x_N) =$$

$$= \sum_{i=1}^{N} \mathrm{var}(w_i)\mathrm{var}(x_i) \approx N * \mathrm{var}(w_i)\mathrm{var}(x_i) \Rightarrow \mathrm{var}(w_i) = \frac{1}{N}$$

Batch normalization layer [Ioffe and Szegedy 2015]

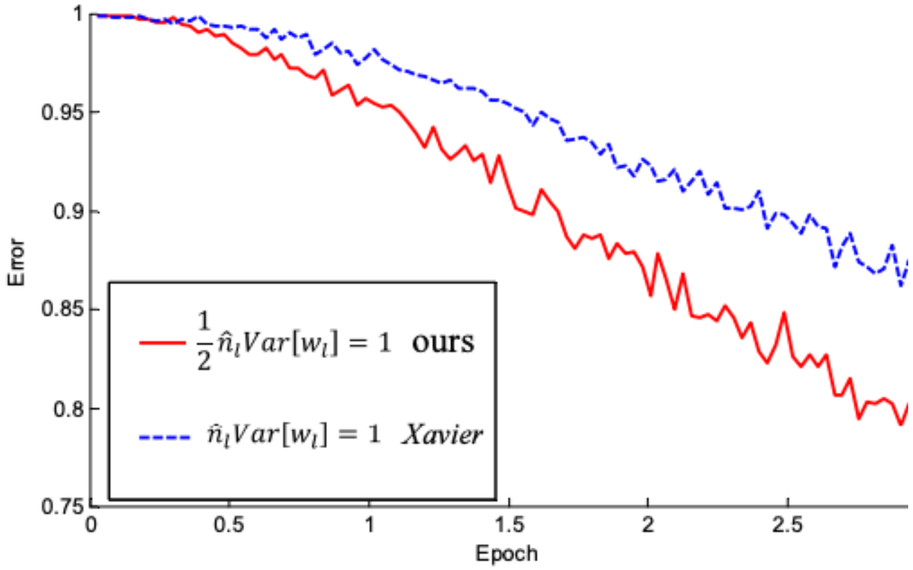
https://arxiv.org/pdf/1502.03167.pdf (over 6k citation)



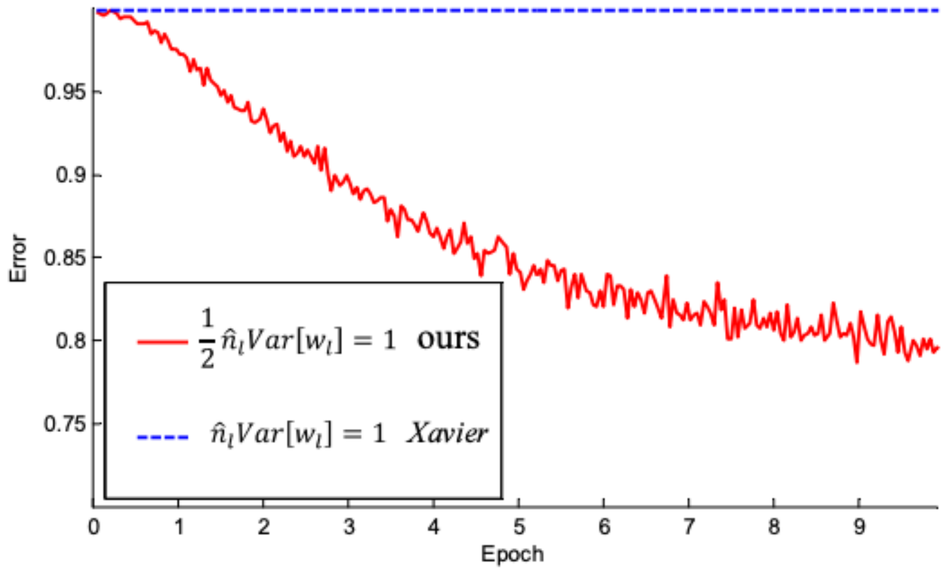
Kaimimg initialization
https://arxiv.org/pdf/1502.01852.pdf

ReLu reduces variance 2x by itself $\Rightarrow \mathrm{var}(w_i) = \frac{2}{N}$

22 layers



30 layers



- PyTorch: `nn.init.xavier_uniform(conv1.weight)`
  `nn.init.calculate_gain('sigmoid')`

Summary so far observed issues

- **Values** in feature maps can easily **explode, diminish** or get **biased**
- **Gradients** can easily **explode** or **diminish** or get **biased**
- We need to keep reasonable values - **what are reasonable values?**

Initialization

Normalization

$$\mu_x = 0 \quad \text{var}(x) = 1$$

Summary so far observed issues

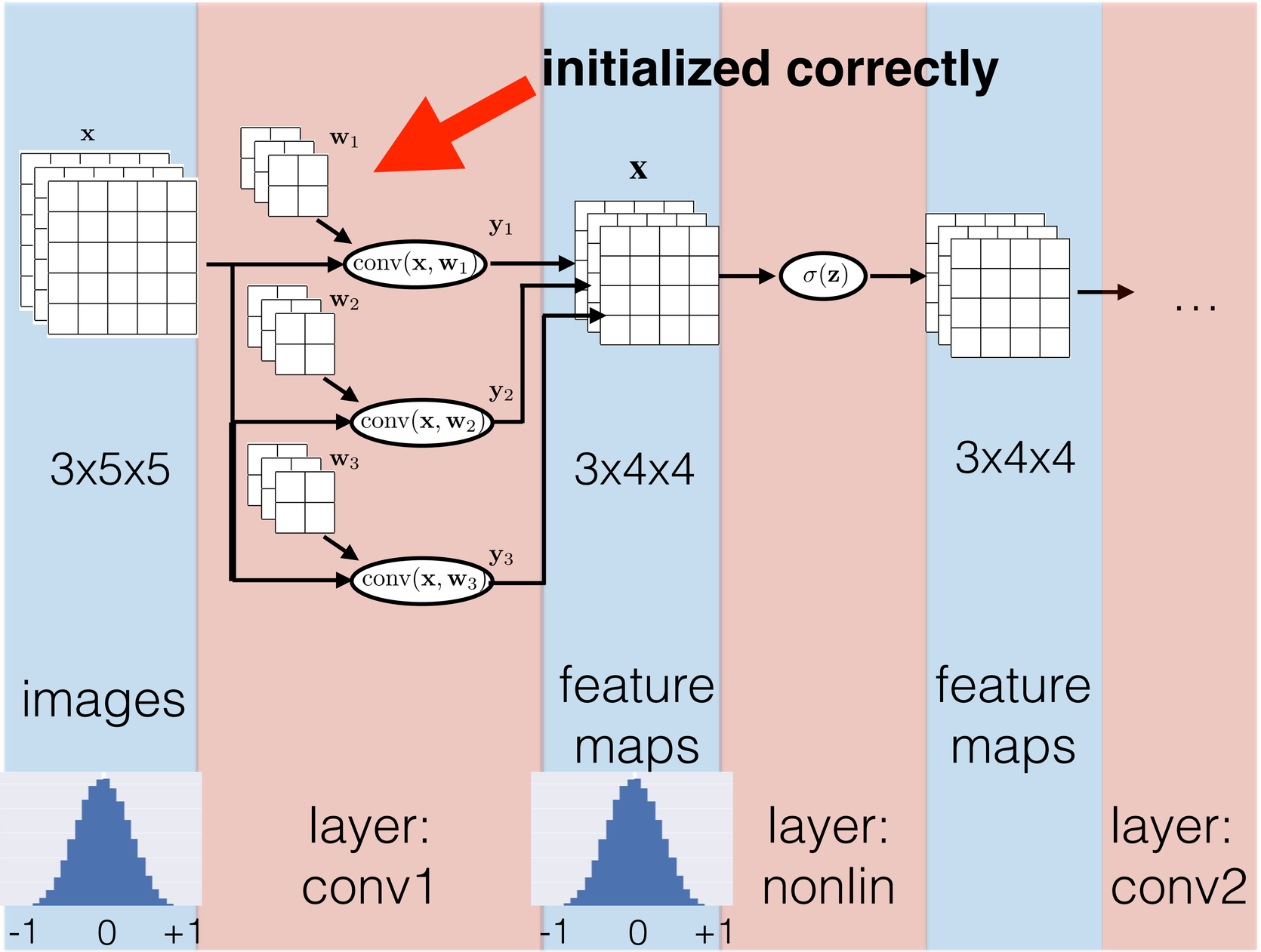
- **Values** in feature maps can easily **explode, diminish** or get **biased**
- **Gradients** can easily **explode** or **diminish** or get **biased**
- We need to keep reasonable values - **what are reasonable values?**

Initialization

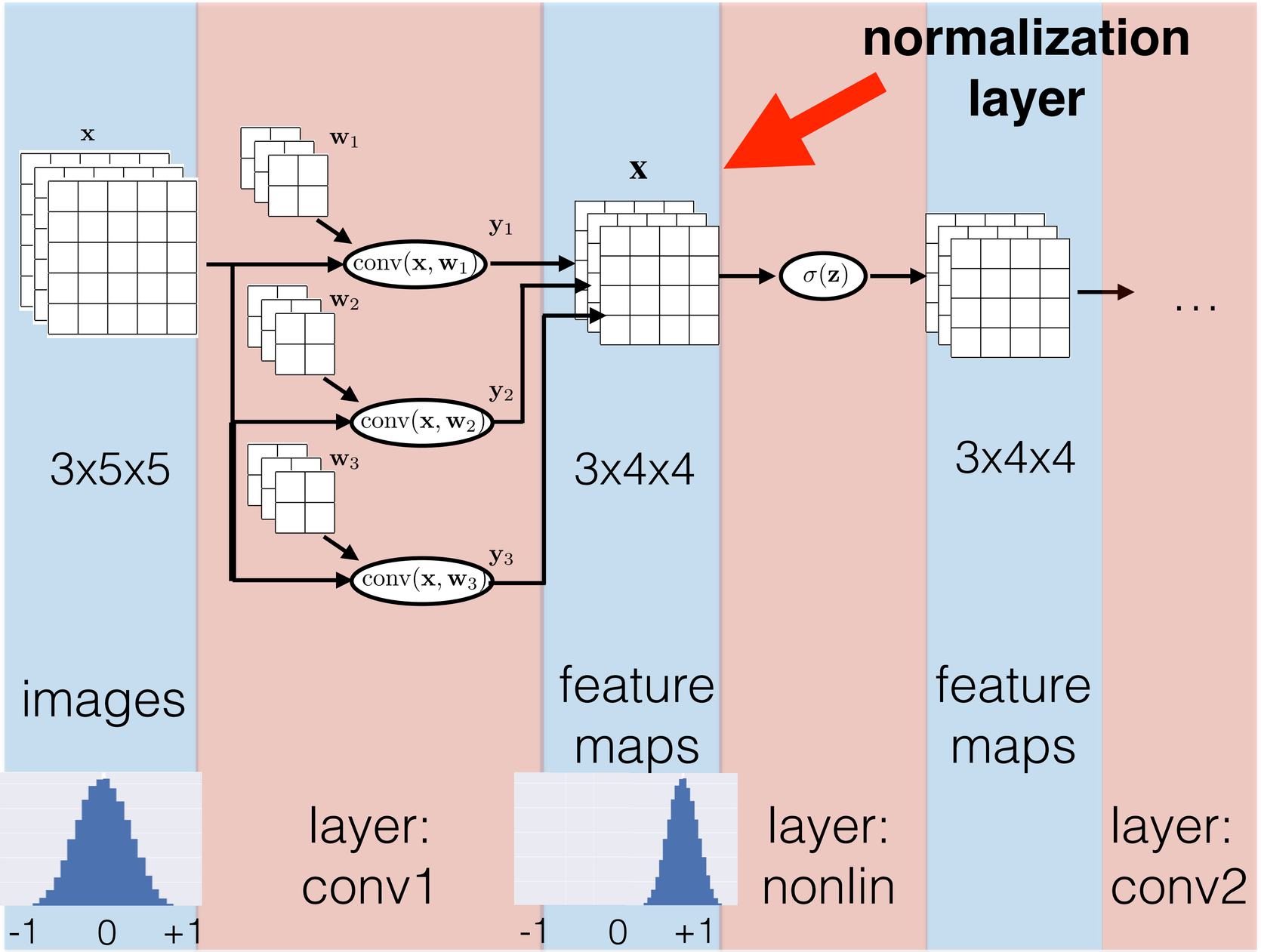
Normalization

$$\mu_x = 0 \quad \text{var}(x) = 1$$

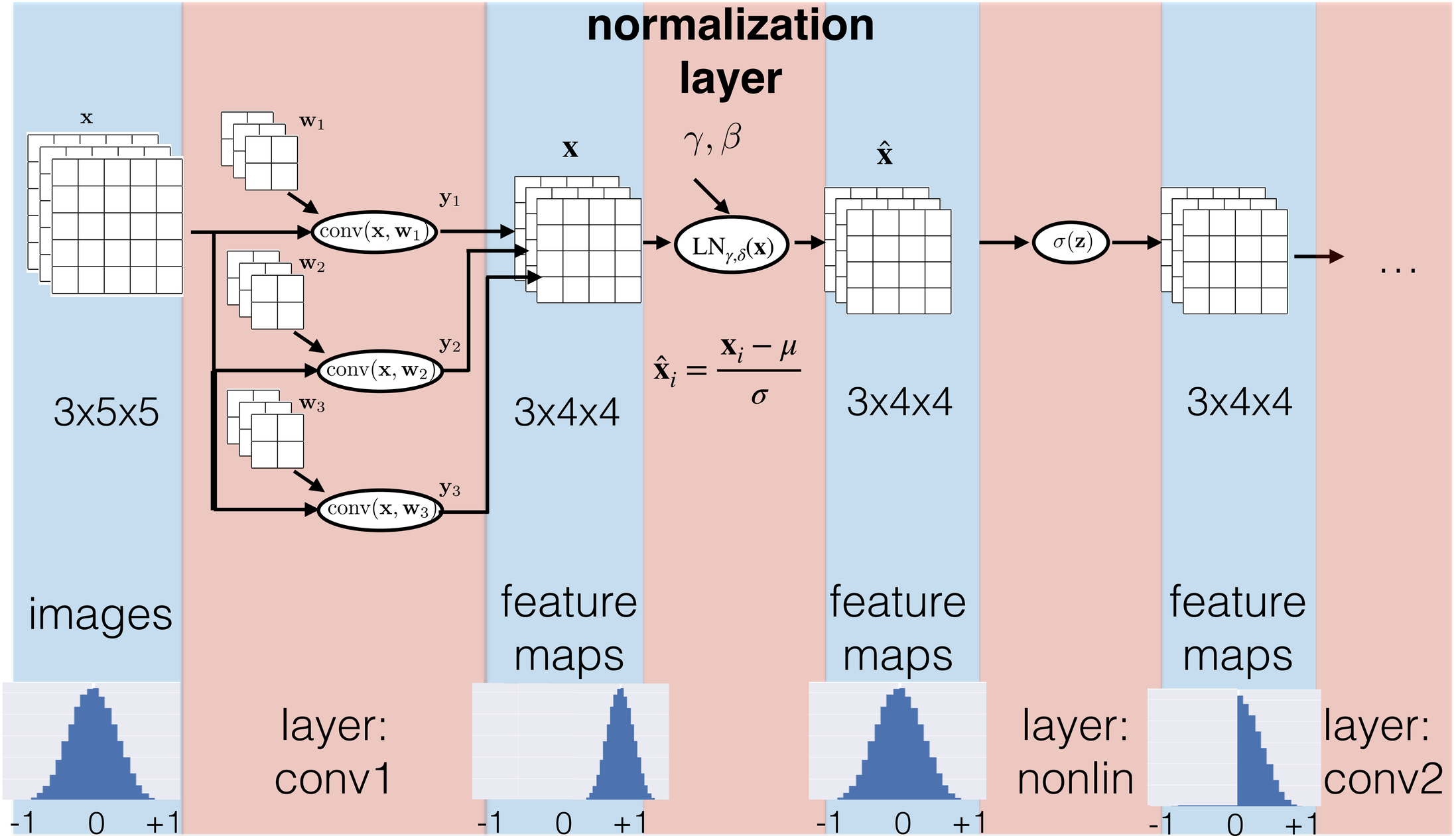
After the gradient update the weights changes …



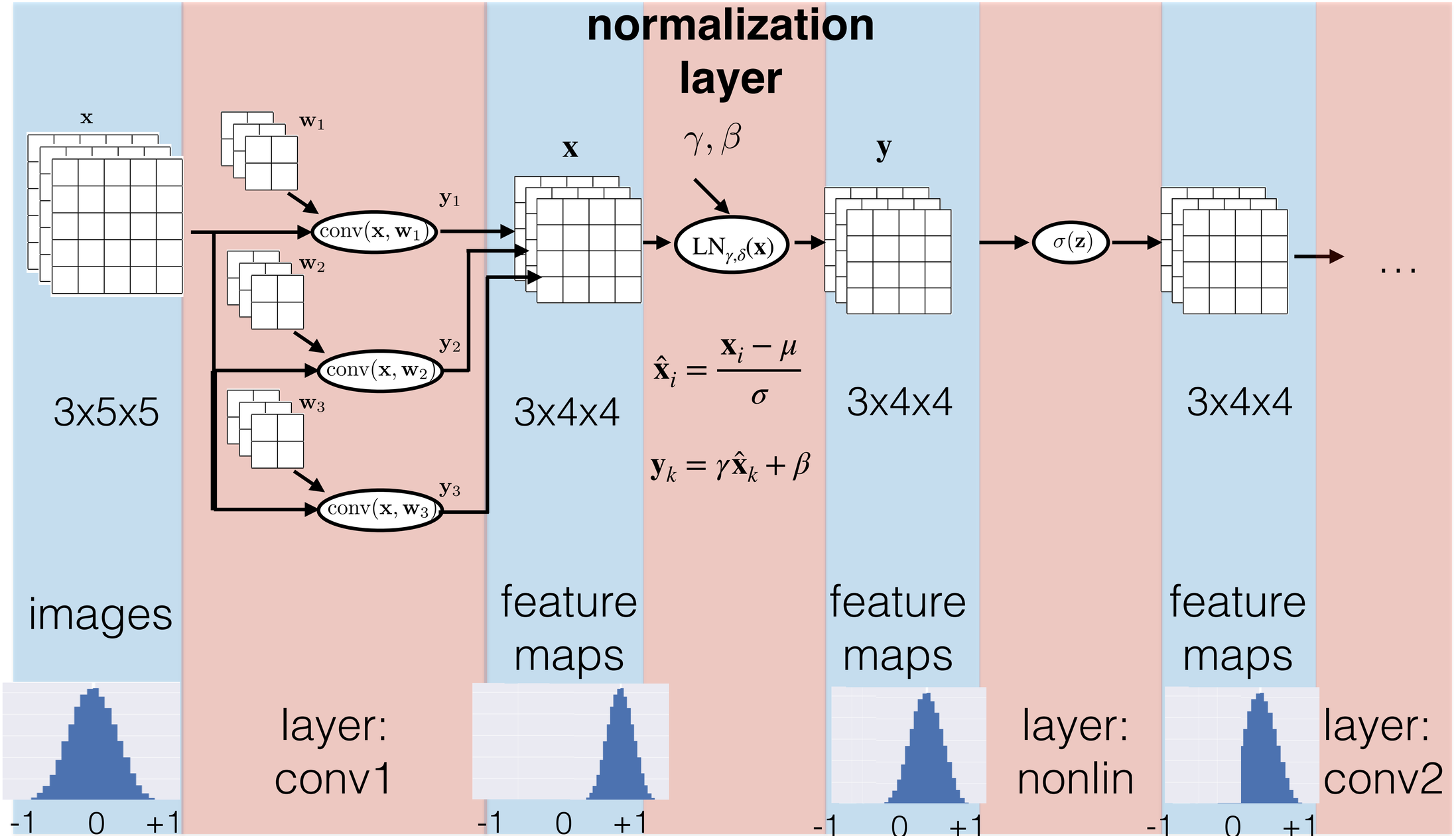
After the gradient update the weights changes …



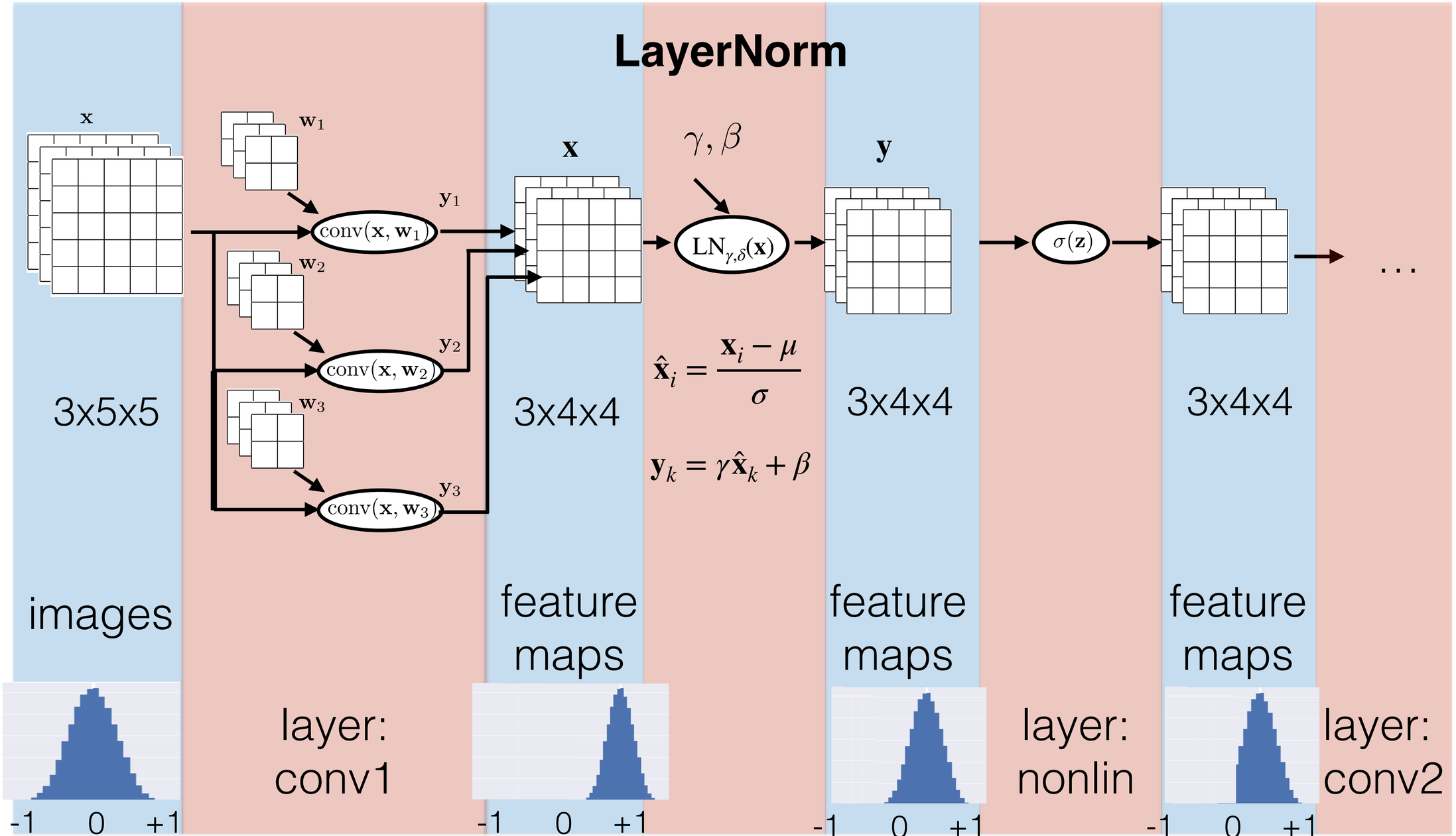
After the gradient update the weights changes …



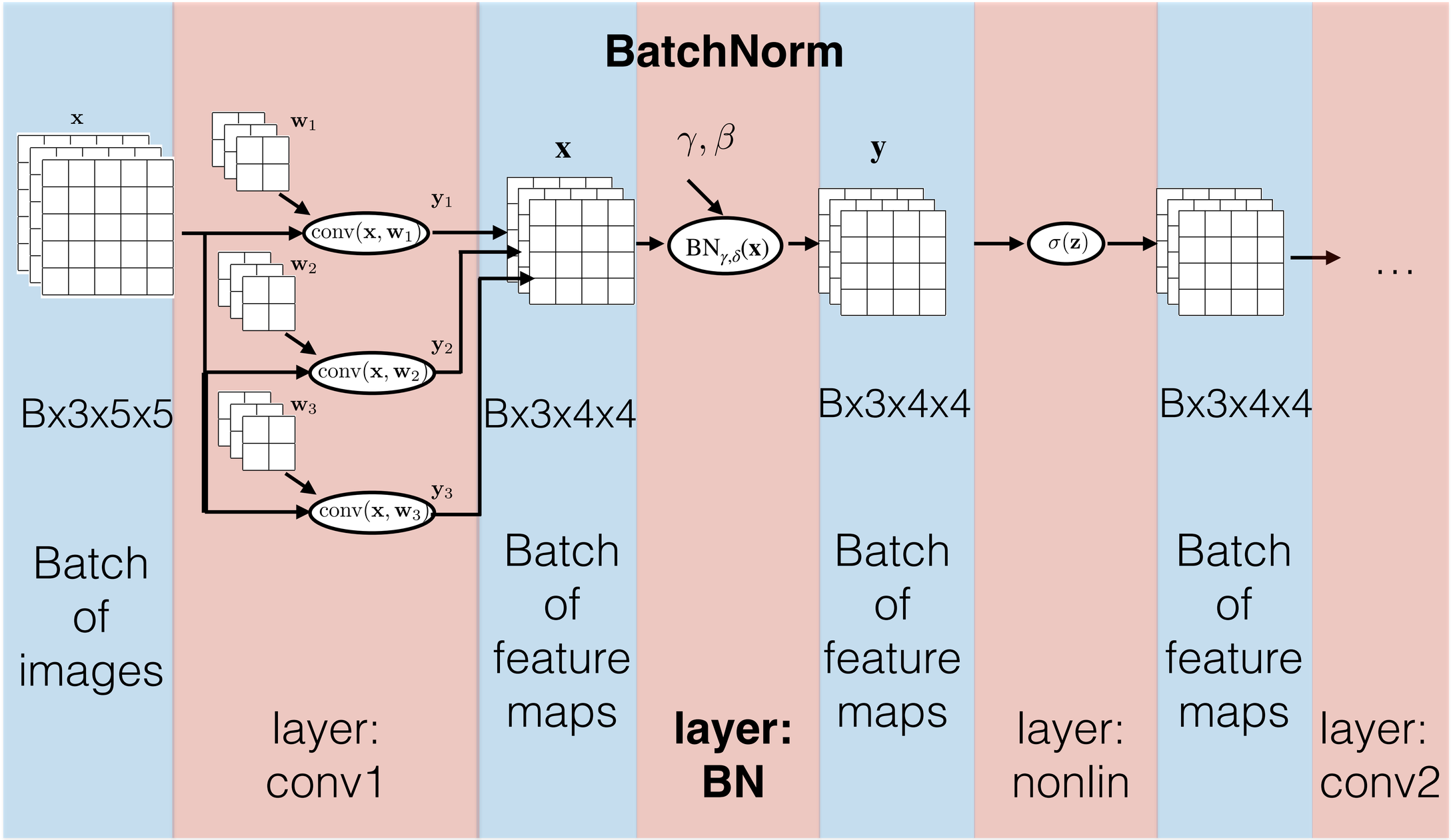
After the gradient update the weights changes …
Using the same mean all channels too restrictive



Using the **same mean/cov** for all channels **too restrictive**
Using the **different mean/cov** for each channel **is ill-conditioned**



Batch normalization layer [Ioffe and Szegedy 2015]
https://arxiv.org/pdf/1502.03167.pdf (over 6k citation)



Batch normalization layer [Ioffe and Szegedy 2015]
https://arxiv.org/pdf/1502.03167.pdf (over 6k citation)

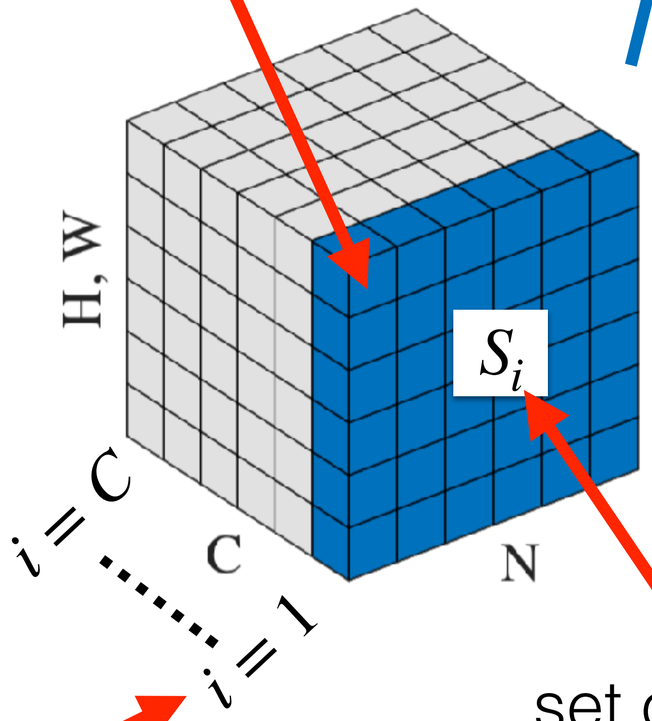
internal index within channel i

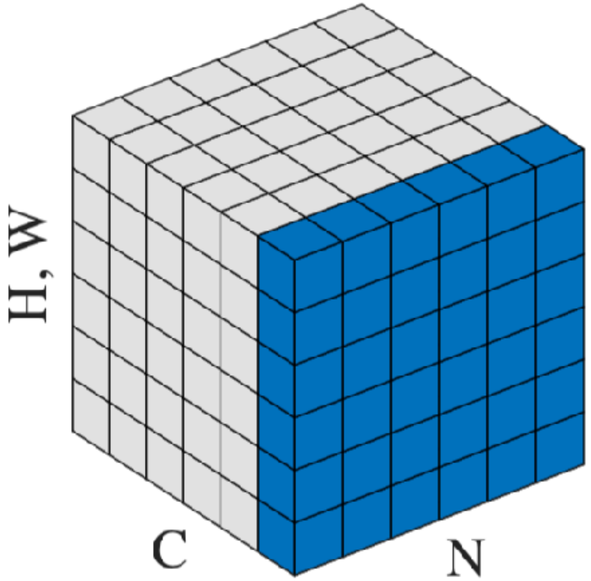
k

$$\mu_i = \frac{1}{m}\sum_{k \in S_i} \mathbf{x}_k \qquad \sigma_i = \sqrt{\frac{1}{m}\sum_{k \in S_i} (\mathbf{x}_k - \mu_i)^2 + \epsilon}$$

$$\hat{\mathbf{x}}_{ki} = \frac{\mathbf{x}_{ki} - \mu_i}{\sigma_i}$$

$$\mathbf{y}_{ki} = \gamma_i \hat{\mathbf{x}}_{ki} + \beta_i$$

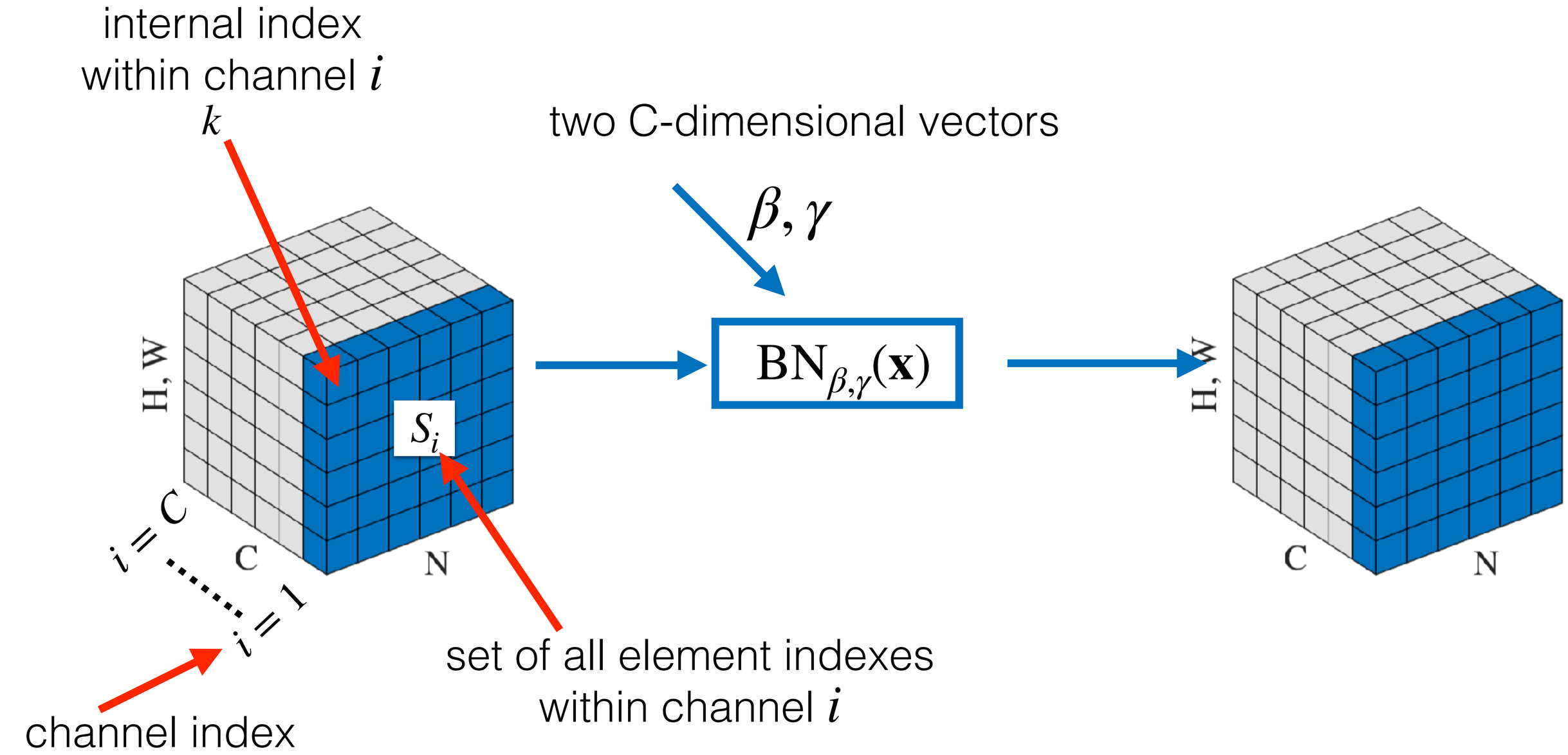




set of all element indexes within channel i

channel index

Batch normalization layer [Ioffe and Szegedy 2015]
https://arxiv.org/pdf/1502.03167.pdf (over 6k citation)



batch size channels width height

```
>>> input = torch.randn(20, 100, 35, 45)
>>> m = nn.BatchNorm2d(100)
>>> output = m(input)
```

What is dimensionality of the output?

the same: 20x100x35x45

What is dimensionality of mean μ ?

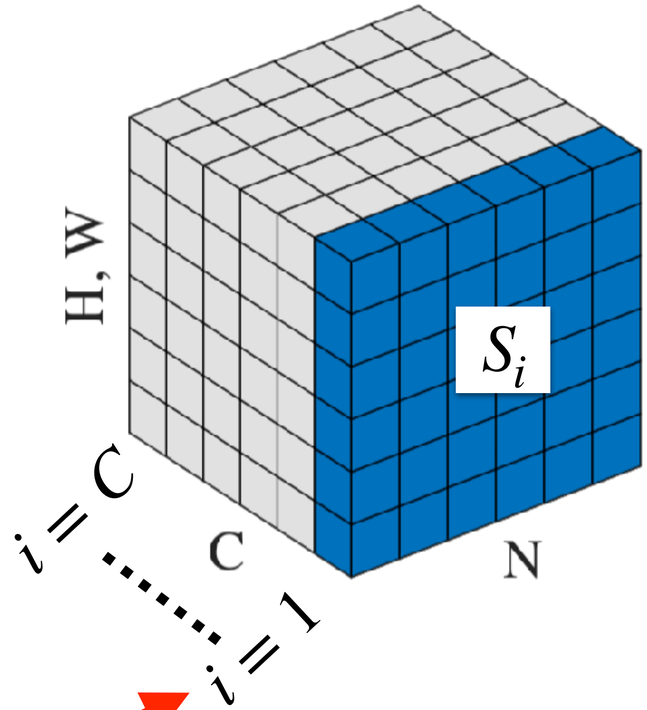
100 dimensional vector

What is dimensionality of mean γ ?

100 dimensional vector

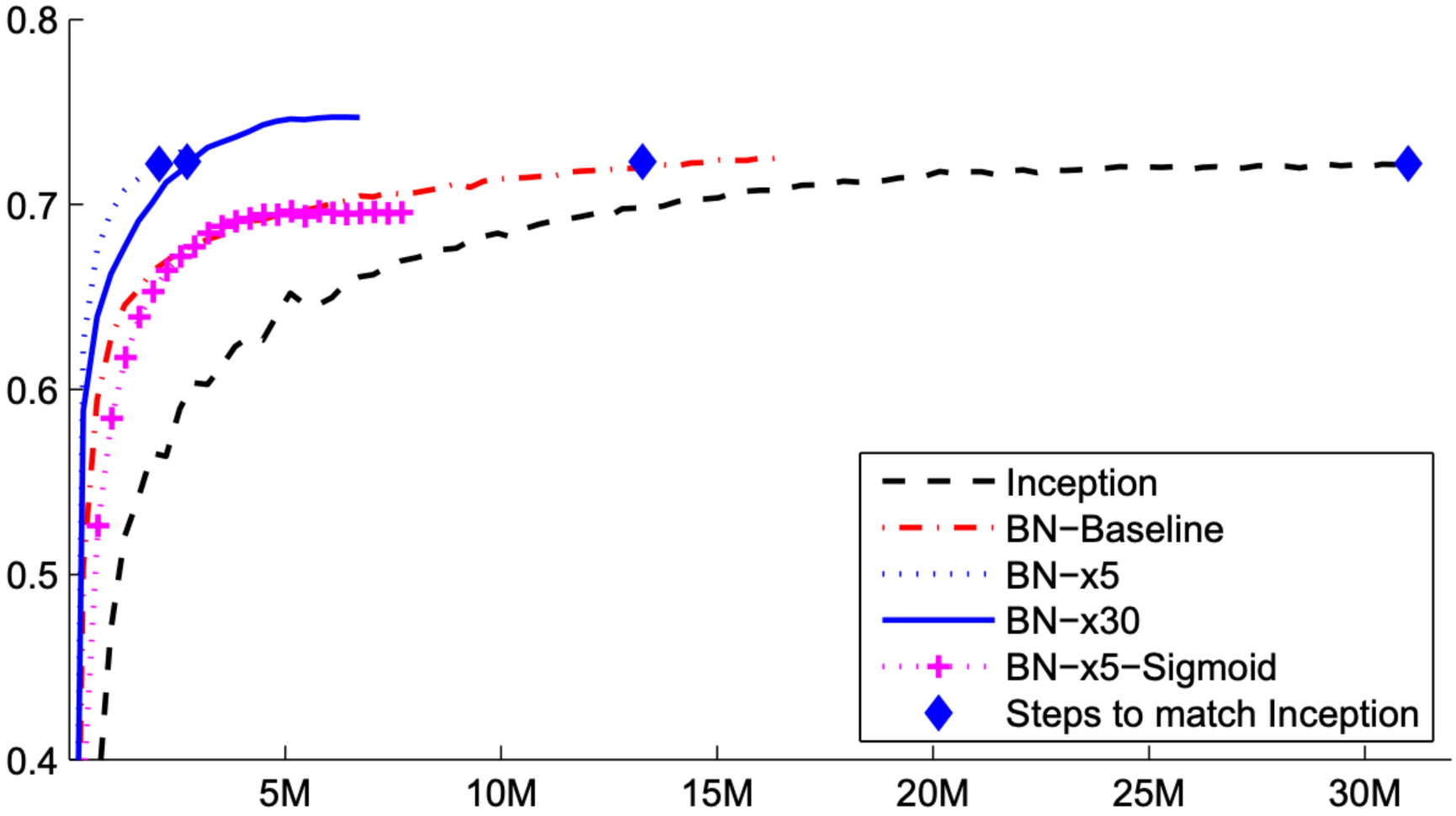
Which μ, σ are used during inference (testing)?

μ, σ estimated either by exponential smoothing or over the whole training set.



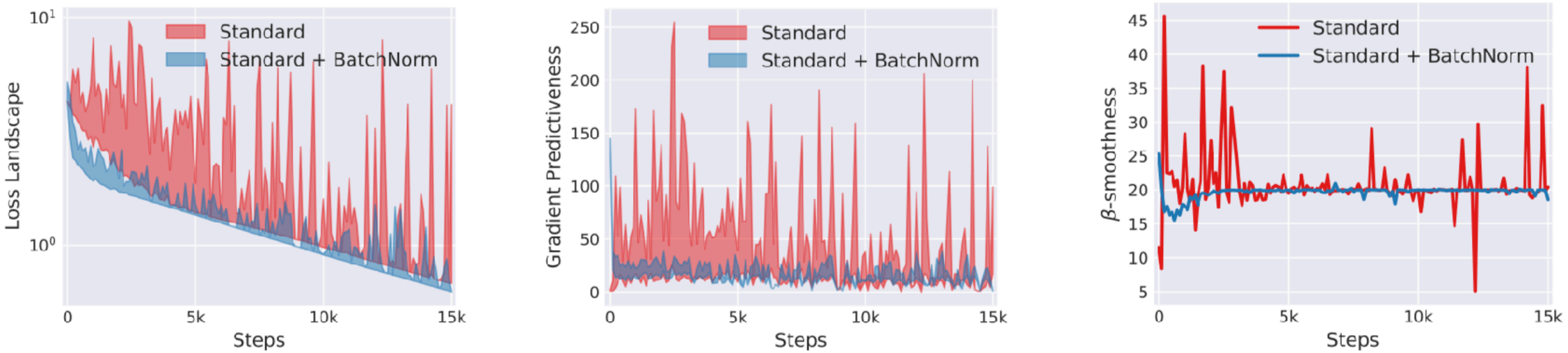
It provably improves learning

Batch normalization layer [Ioffe and Szegedy 2015]
https://arxiv.org/pdf/1502.03167.pdf (over 6k citation)



Why batch normalization helps??

- It is still **unclear** remain an active research topic
- **Reduces internal covariate shift**, i.e. changes in distribution of layer inputs
- **Improves gradient flow and smoother loss** [Santurkar, NIPS, 2019]

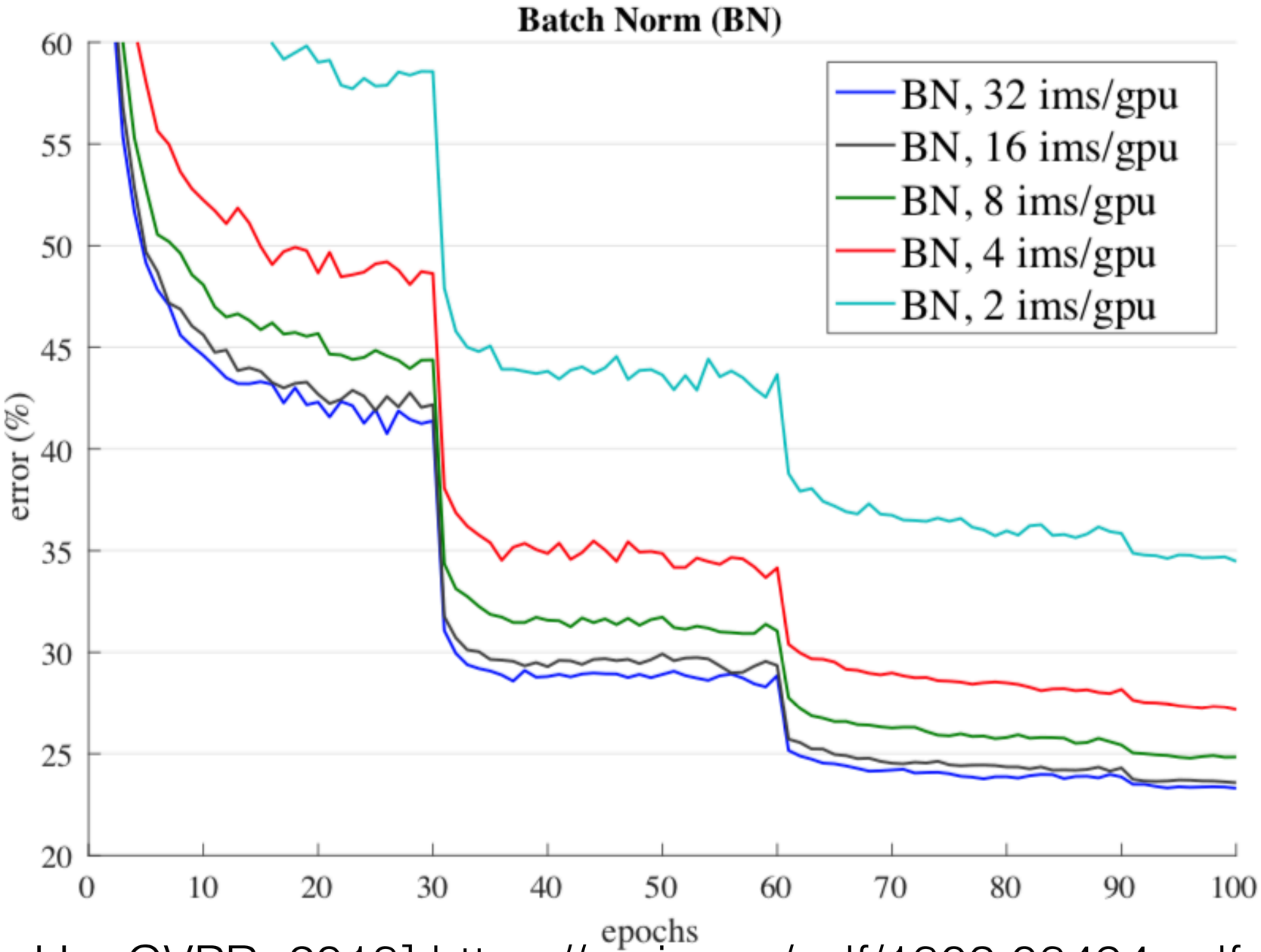


(a) loss landscape (b) gradient predictiveness (c) "effective" β-smoothness

https://arxiv.org/pdf/1805.11604.pdf

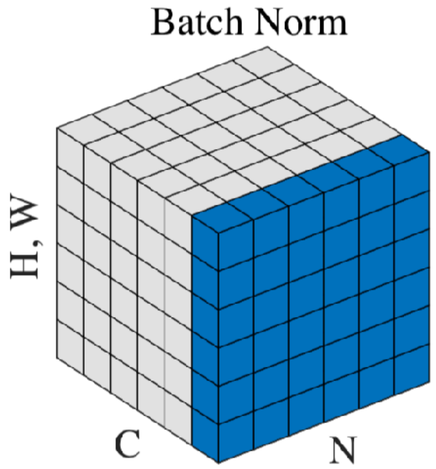
- **BN is model regularizer:** one training example always normalized differently => small feature map jittering (dataset augmentation) => better generalization

Batchnorm drawback: sensitivity to batch size

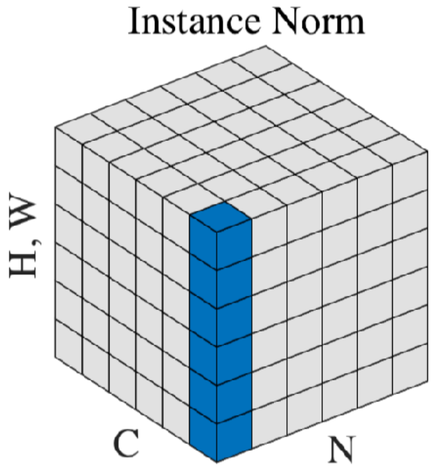


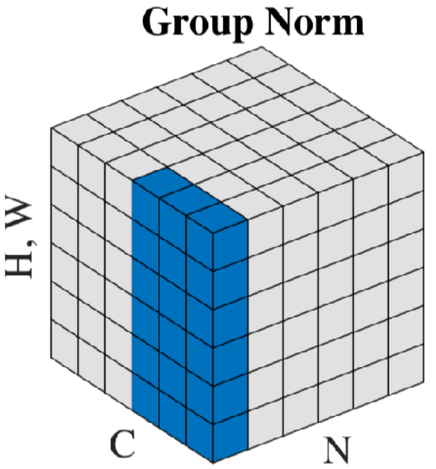
[Wu, He, CVPR, 2018] https://arxiv.org/pdf/1803.08494.pdf

There are other normalizations

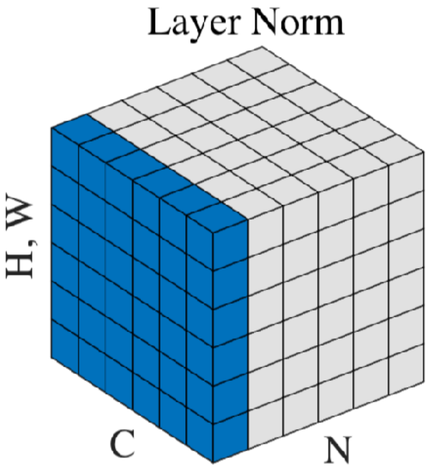


- Channel-wise normalization
- Good performance
- Sensitive to batch size





Group normalization [Wu, He, 2018]
https://arxiv.org/pdf/1803.08494.pdf



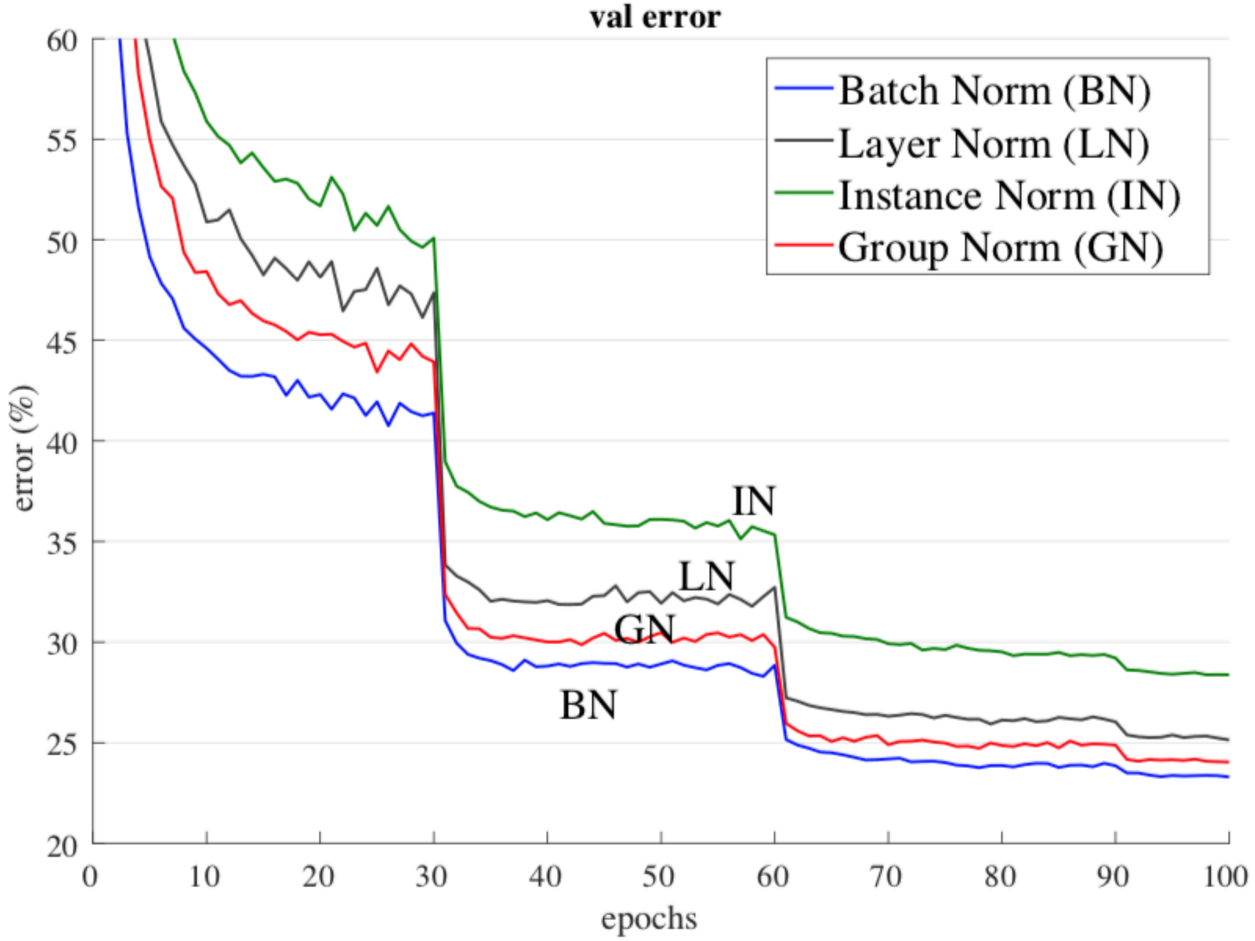
- Instance-wise normalization
- Worse performance
- Insensitive to batch size

- Why do not take best of both worlds?

Classification task

- BN vs GN performance comparable for sufficiently large batch sizes

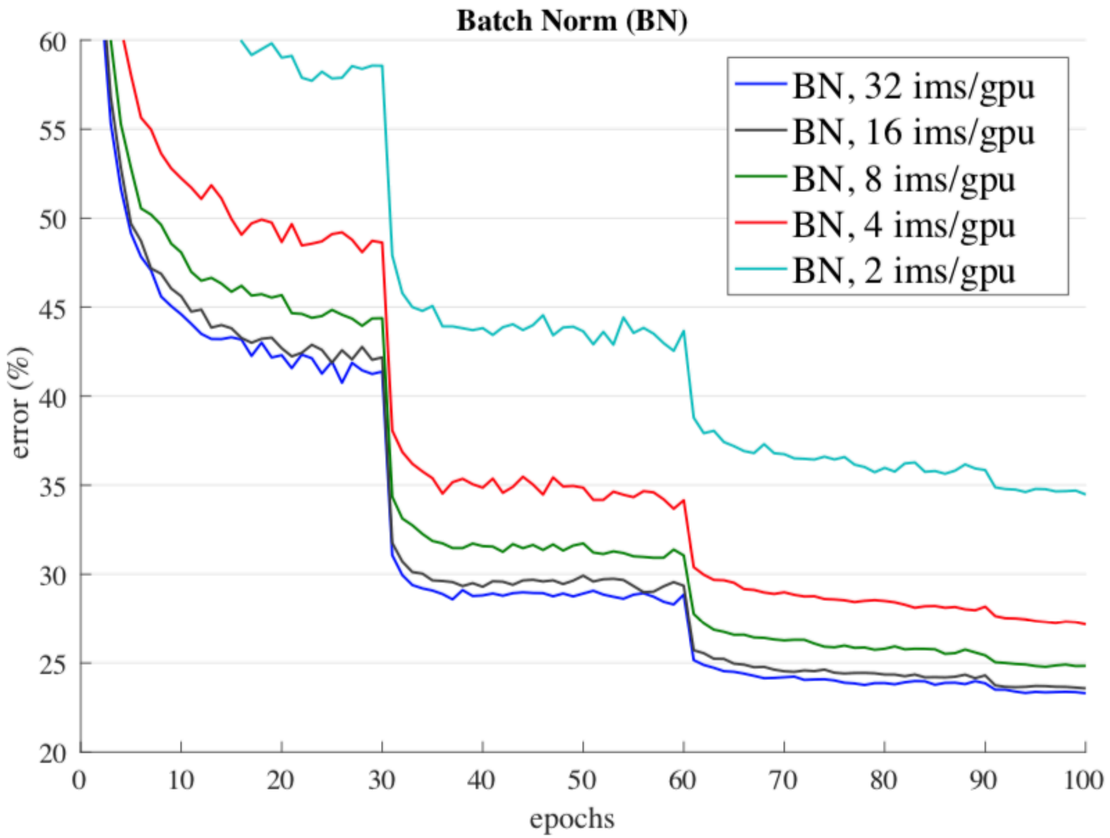
batch size = 32



[Wu, He, CVPR, 2018] https://arxiv.org/pdf/1803.08494.pdf

Group Normalization - conclusions

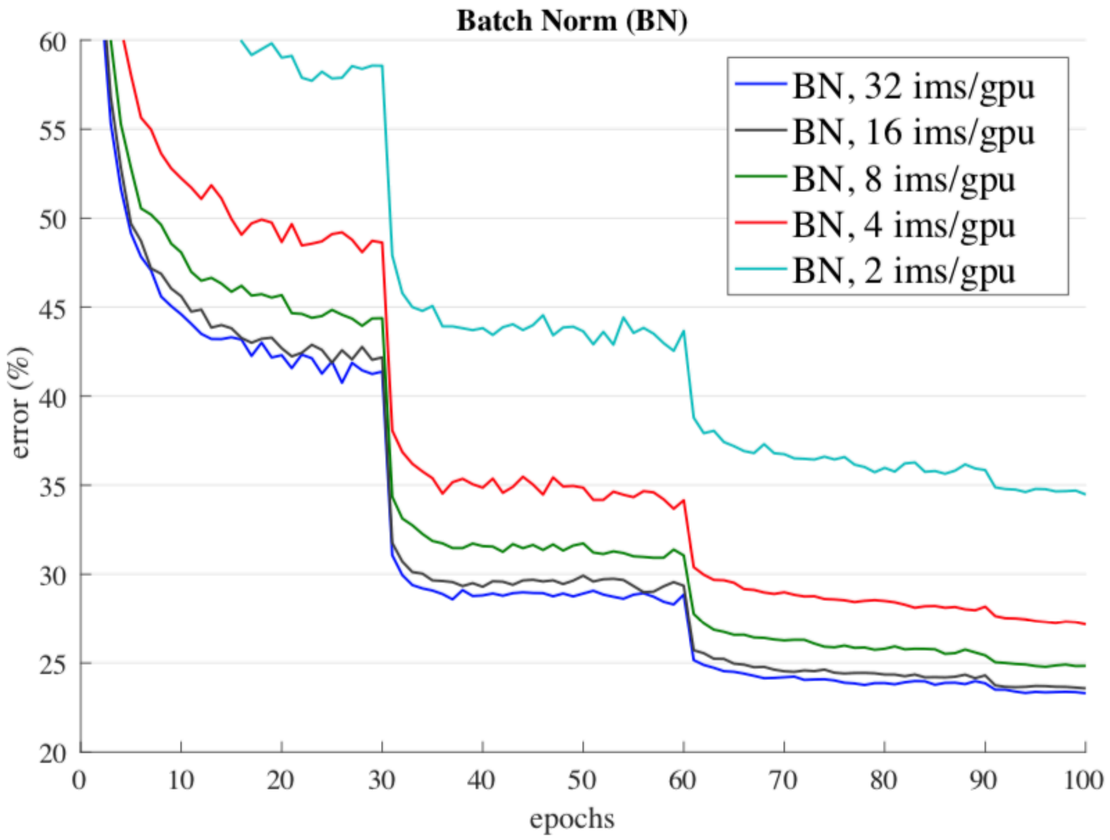
- BN is **sensitive** to mini-batch size.

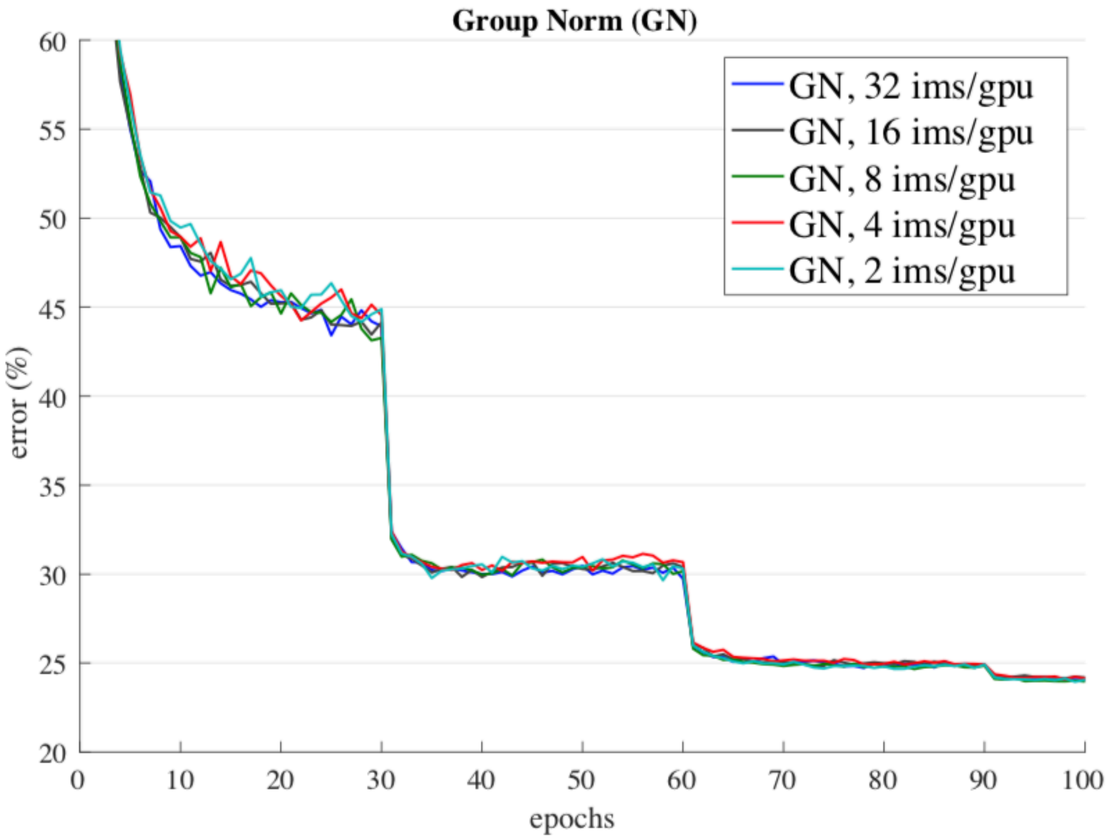


[Wu, He, CVPR, 2018] https://arxiv.org/pdf/1803.08494.pdf

Group Normalization - conclusions

- BN is **sensitive** to mini-batch size.
- GN is **insensitive** to mini-batch size.
- For smaller mini-batches GN outperforms BN significantly

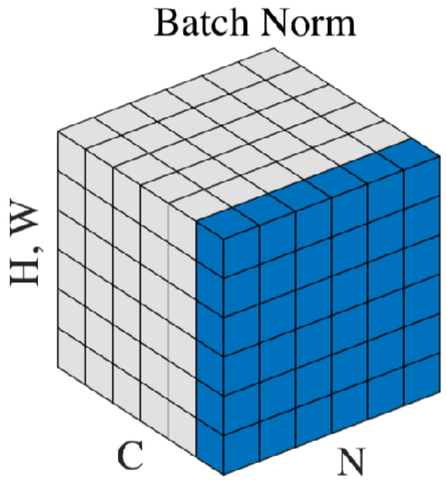




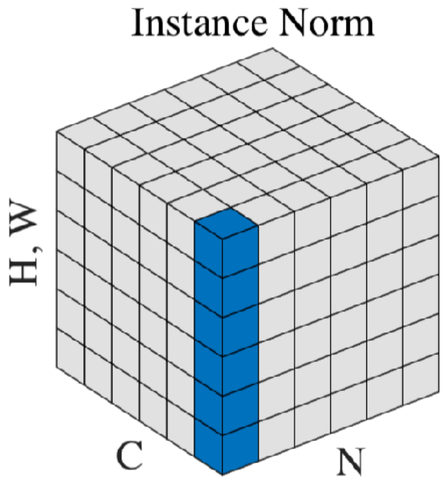
[Wu, He, CVPR, 2018] https://arxiv.org/pdf/1803.08494.pdf

Batch-Instance normalization
https://arxiv.org/pdf/1805.07925.pdf

What if I even learn to combine normalizations?



$\hat{x}^{(BN)}$



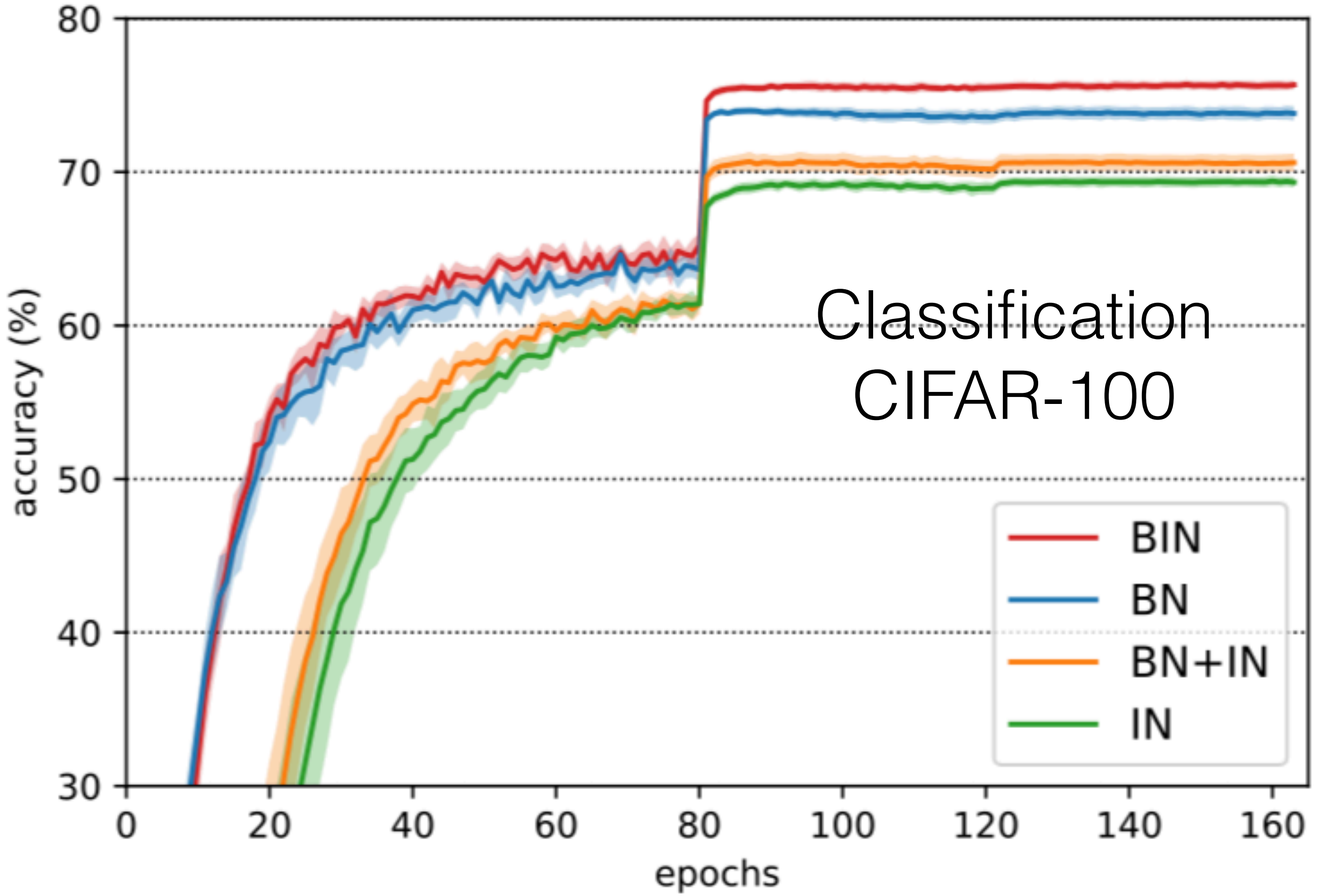
$\hat{x}^{(IN)}$

Batch-Instance normalization

https://arxiv.org/pdf/1805.07925.pdf

$$y = \left(\rho \cdot \hat{x}^{(BN)} + (1-\rho) \cdot \hat{x}^{(IN)}\right) \cdot \gamma + \beta$$

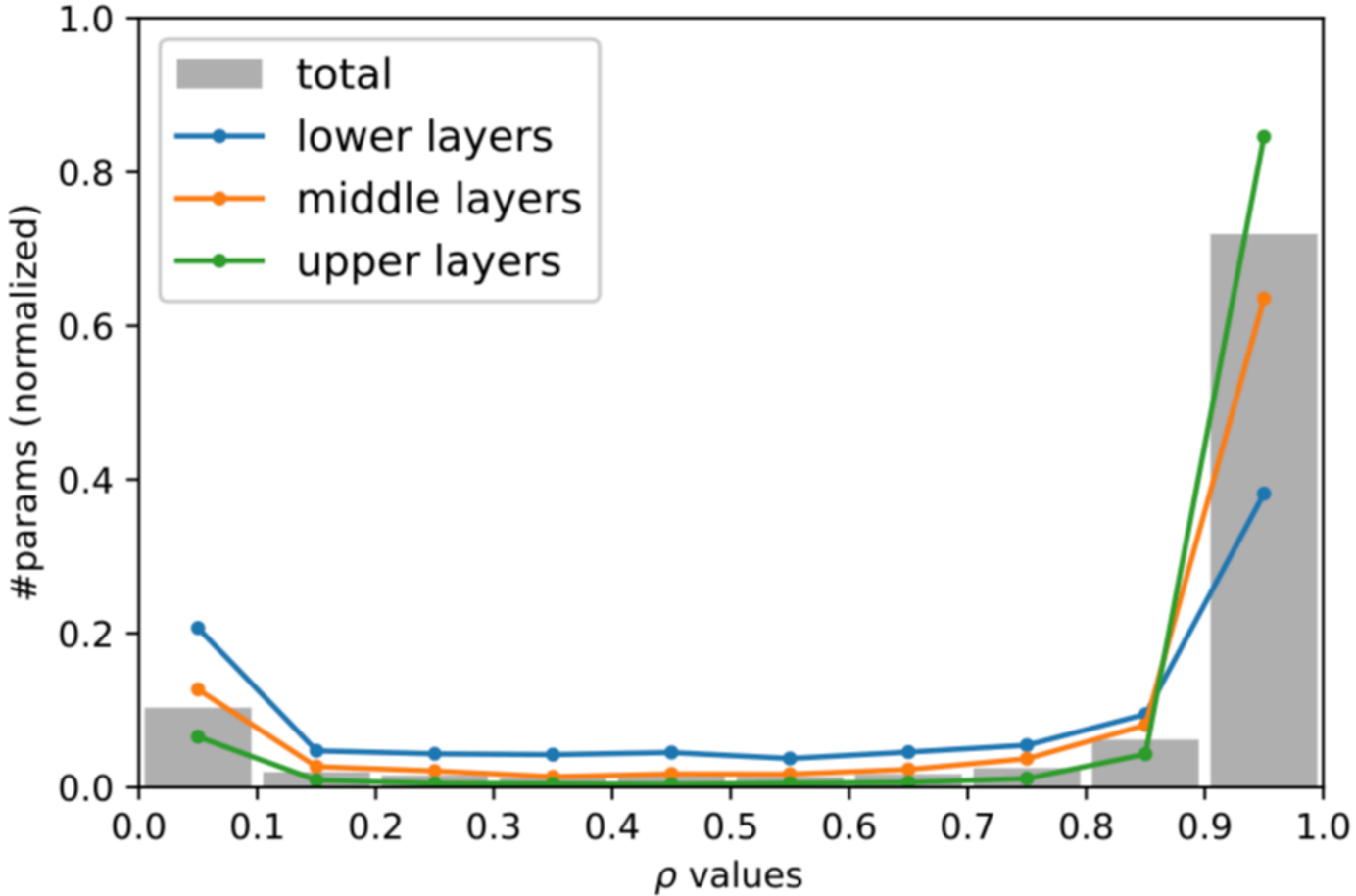
- BIN is learnable combination of BN a IN
- Suitable for both style transfer and classification



Batch-Instance normalization
https://arxiv.org/pdf/1805.07925.pdf

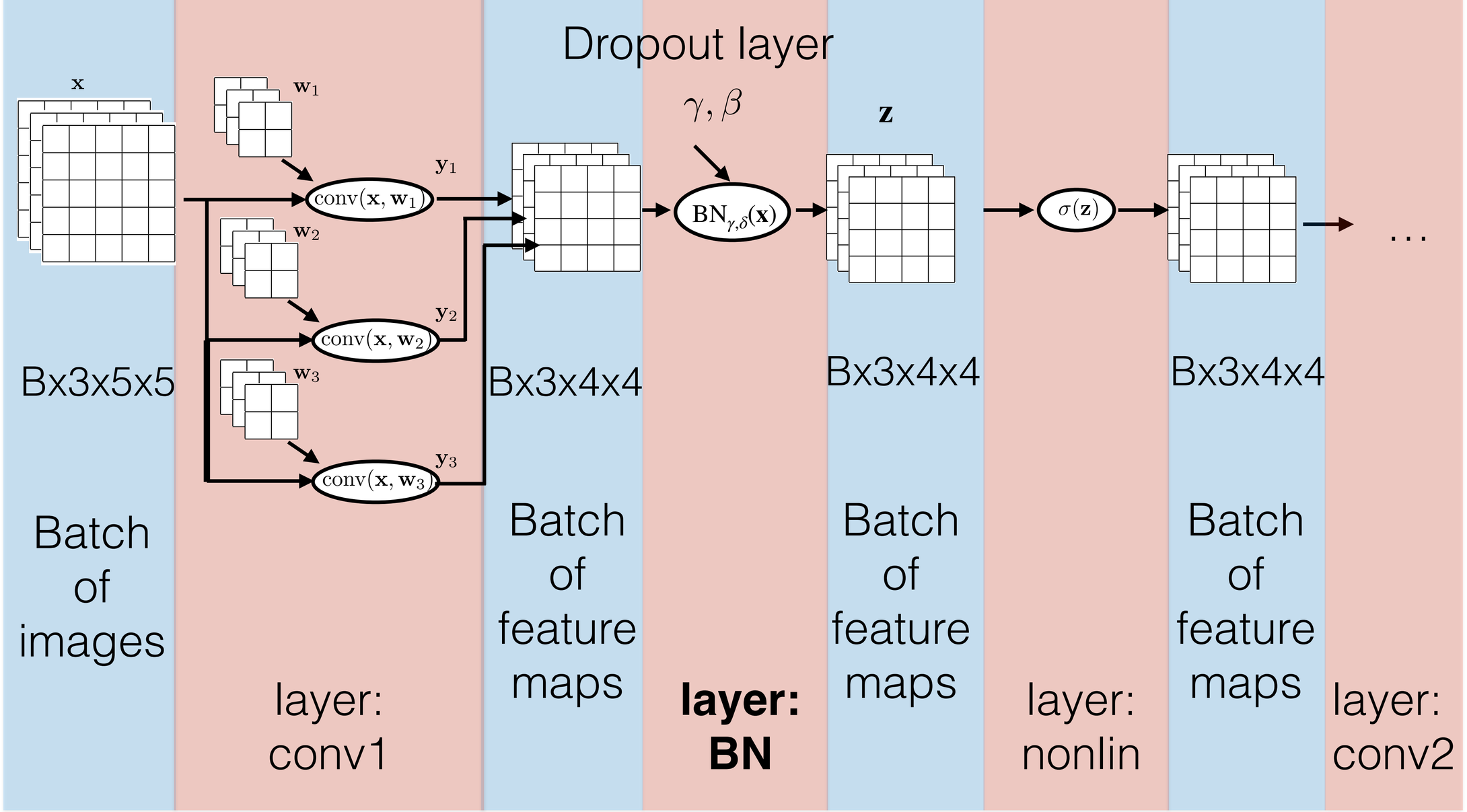
$$y = \left(\rho \cdot \hat{x}^{(BN)} + (1-\rho) \cdot \hat{x}^{(IN)}\right) \cdot \gamma + \beta$$

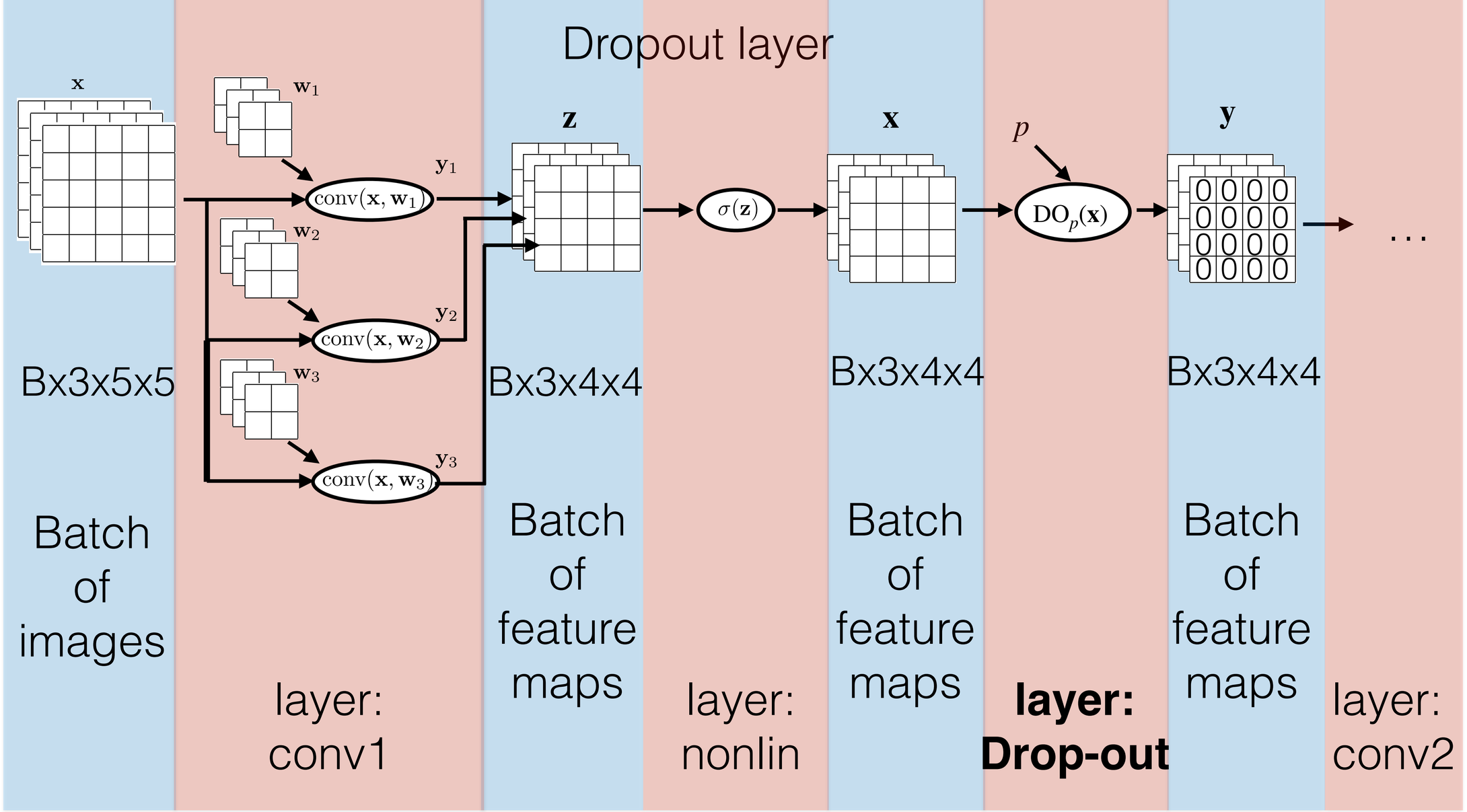
- BIN is learnable combination of BN a IN
- Suitable for both style transfer and classification



Three more ways to suppress overfitting

- Dropout
- Filter pruning
- Weight decay

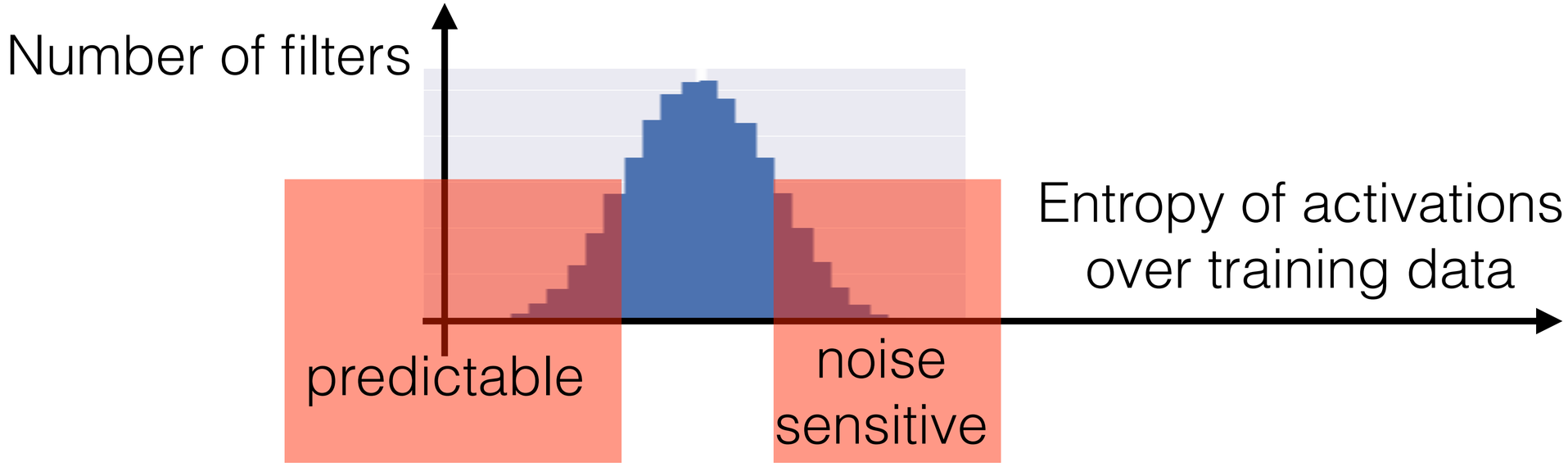

Dropout layer
γ, β
$\mathbf{x}$
$\mathbf{w}_1$
$\mathbf{y}_1$
$\mathrm{conv}(\mathbf{x}, \mathbf{w}_1)$
$\mathbf{w}_2$
$\mathbf{y}_2$
$\mathrm{conv}(\mathbf{x}, \mathbf{w}_2)$
$\mathbf{w}_3$
$\mathbf{y}_3$
$\mathrm{conv}(\mathbf{x}, \mathbf{w}_3)$
$\mathrm{BN}_{\gamma,\delta}(\mathbf{x})$
$\mathbf{z}$
$\sigma(\mathbf{z})$
...
Bx3x5x5
Bx3x4x4
Bx3x4x4
Bx3x4x4
Batch of images
layer: conv1
Batch of feature maps
layer: BN
Batch of feature maps
layer: nonlin
Batch of feature maps
layer: conv2



- FCN layer: drop-out, ConvNet: Spatial drop-out, Transformers: Attention drop-out
- Drop-out is removed during testing (inference)

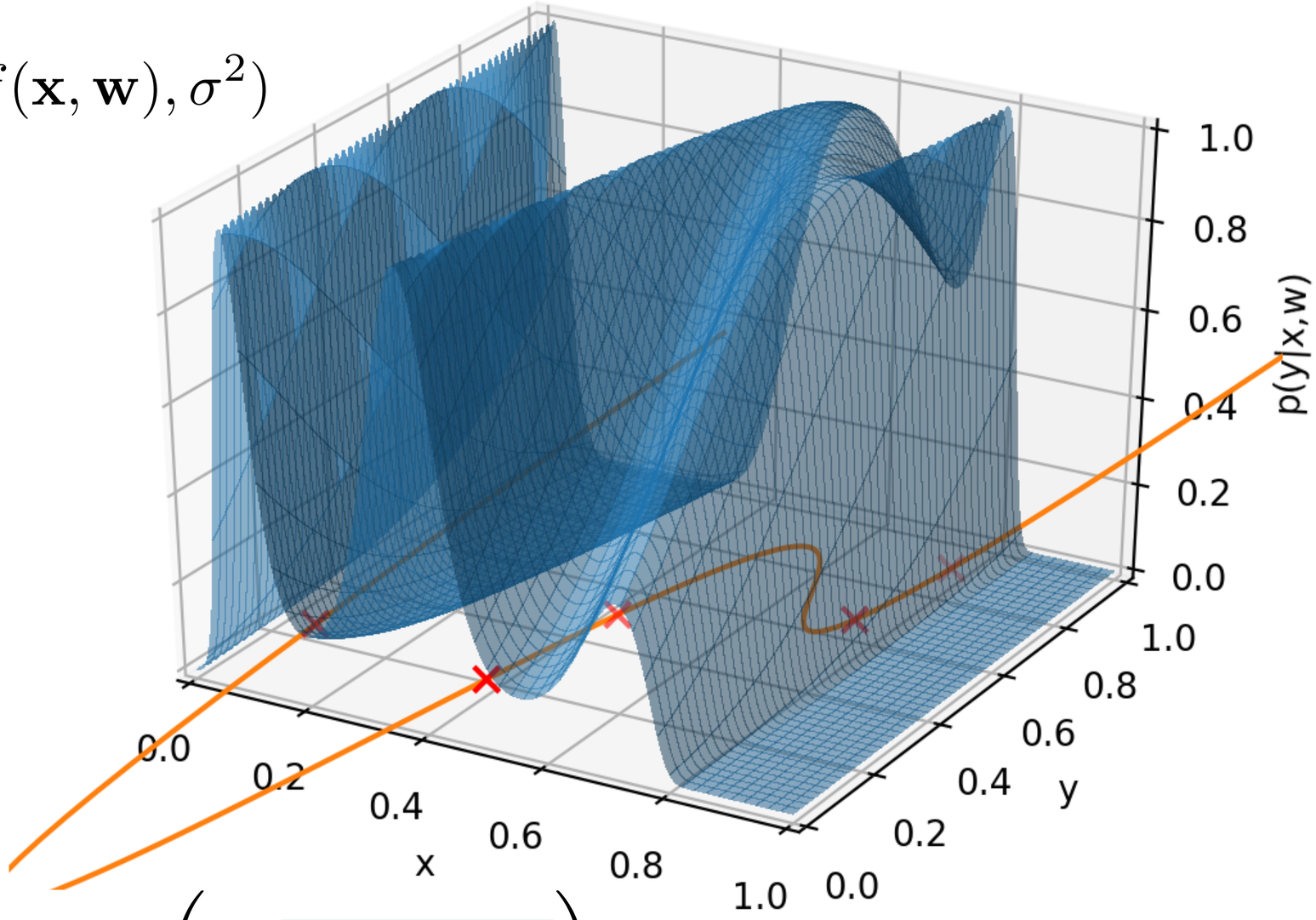
Filter pruning

- Filters with negative response for all training data are dead.
- Filters with positive response for all training data are suspicious.
- Entropy-based filter pruning:
    - High-entropy filters usually react on noise
    - Low-entropy filters usually react on almost nothing or almost everything



Weight decay

$$p(y|\mathbf{x}, \mathbf{w}) \sim \mathcal{N}_y(f(\mathbf{x}, \mathbf{w}), \sigma^2)$$



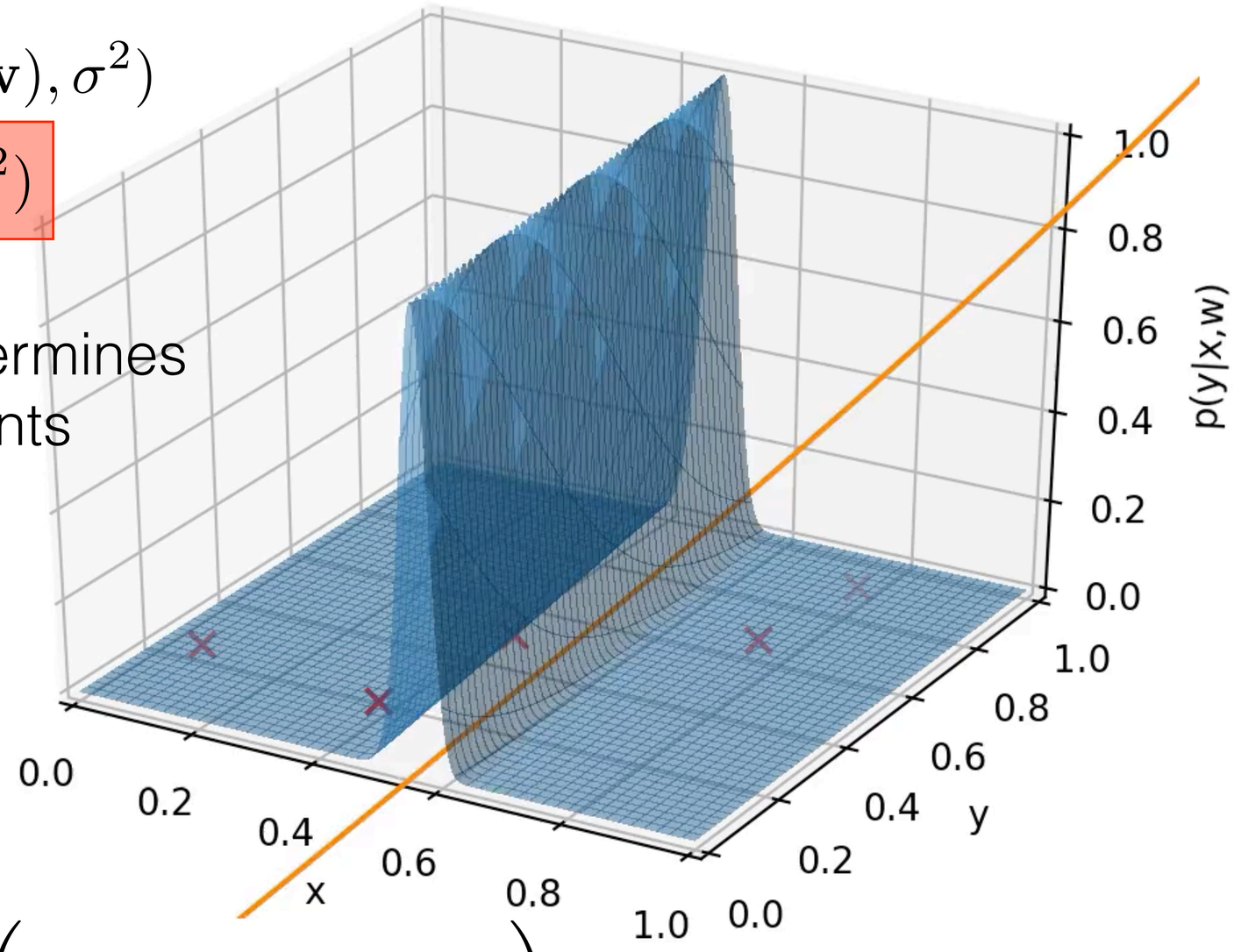
$$\mathbf{w}^* = \arg\max_{\mathbf{w}} \left(\prod_i p(y_i|\mathbf{x}_i, \mathbf{w}) \right) = \arg\min_{\mathbf{w}} \sum_i (f(\mathbf{x}_i, \mathbf{w}) - y_i)^2$$

Weight decay

$$p(y|\mathbf{x}, \mathbf{w}) \sim \mathcal{N}_y(f(\mathbf{x}, \mathbf{w}), \sigma^2)$$

$$p(\mathbf{w}) \sim \mathcal{N}_w(\mathbf{0},\ \sigma^2)$$

Norm of weights determines Lipchitz constraints on the $\mathbf{f(x, w)}$



$$\mathbf{w}^* = \arg\max_{\mathbf{w}} \left(\prod_i p(y_i|\mathbf{x}_i, \mathbf{w}) p(\mathbf{w}) \right) = \arg\min_{\mathbf{w}} \sum_i (f(\mathbf{x}_i, \mathbf{w}) - y_i)^2 + \|\mathbf{w}\|$$

How is this penalty connect to decaying weights?

$$\mathbf{w}^* = \arg\max_{\mathbf{w}} \left(\prod_i p(y_i|\mathbf{x}_i, \mathbf{w}) \right) = \arg\min_{\mathbf{w}} \mathscr{L}\big(f(\mathbf{x}, \mathbf{w}), y\big)$$

$$\mathbf{w} := \mathbf{w} - \alpha \frac{\partial \mathscr{L}\big(f(\mathbf{x}, \mathbf{w}), y\big)}{\partial \mathbf{w}}$$

$$\mathbf{w}^* = \arg\max_{\mathbf{w}} \left(\prod_i p(y_i|\mathbf{x}_i, \mathbf{w}) p(\mathbf{w}) \right) = \arg\min_{\mathbf{w}} \mathscr{L}\big(f(\mathbf{x}, \mathbf{w}), y\big) + \|\mathbf{w}\|$$

$$\mathbf{w} := \mathbf{w} - \alpha \left[\frac{\partial \mathscr{L}\big(f(\mathbf{x}, \mathbf{w}), y\big)}{\partial \mathbf{w}} + 2\mathbf{w} \right] = (1 - 2\alpha)\mathbf{w} - \alpha \frac{\partial \mathscr{L}\big(f(\mathbf{x}, \mathbf{w}), y\big)}{\partial \mathbf{w}}$$

Weight decay implemented as a part of optimizer

Conclusions

- **Robust initialization:** set weights to have “normal” values in all layers
- **Normalization:**
  - **BN is reparametrization** of the original NN which has the same expressive power.
  - **BN yields** less sensitivity to vanishing or exploding gradient (improves beta smoothness => faster learning).
  - **BN is model regularizer:** one training example always normalized differently => small feature map jittering (dataset augmentation) => better generalization
  - **BN works well on classification** problems with **larger batches** (>4).
  - **BN not suitable for transformers and recurrent networks.** Different length of training samples within a single batch => LayerNorm used instead.
  - Alternatives **LN, GN, IN, BIN** are typically used for **smaller batches, recurrent, generative** and **transformers** networks**.**
- **Regularization:**
  - **Drop-out**: reduces overfitting by randomly deactivating neurons/channels
  - **Weight decay**: reduces overfitting by reducing reliance on specific kernels

Loss functions

PyTorch:

Regression:

$$L_2(\mathbf{w}) = \sum_i \|\mathbf{f}(\mathbf{x}_i, \mathbf{w}) - \mathbf{y}_i\|_2^2$$

`nn.MSELoss()`

$$L_1(\mathbf{w}) = \sum_i |\mathbf{f}(\mathbf{x}_i, \mathbf{w}) - \mathbf{y}_i|$$

`nn.L1Loss()`

$$L_{1_{\text{smooth}}}(\mathbf{w}) = \begin{cases} \sum_i 0.5\|\mathbf{f}(\mathbf{x}_i, \mathbf{w}) - \mathbf{y}_i\|_2^2, & \text{if } |\mathbf{f}(\mathbf{x}_i, \mathbf{w}) - \mathbf{y}_i| < 1. \\ \sum_i |\mathbf{f}(\mathbf{x}_i, \mathbf{w}) - \mathbf{y}_i| + 0.5, & \text{otherwise.} \end{cases}$$

`nn.SmoothL1Loss()`

Classification (cross entropy):

$$\mathscr{L}(\mathbf{w}) = \sum_i \log(\mathbf{s}_{y_i}(\mathbf{f}(\mathbf{x}_i, \mathbf{w})))$$

input logits: `torch.nn.NLLLoss`

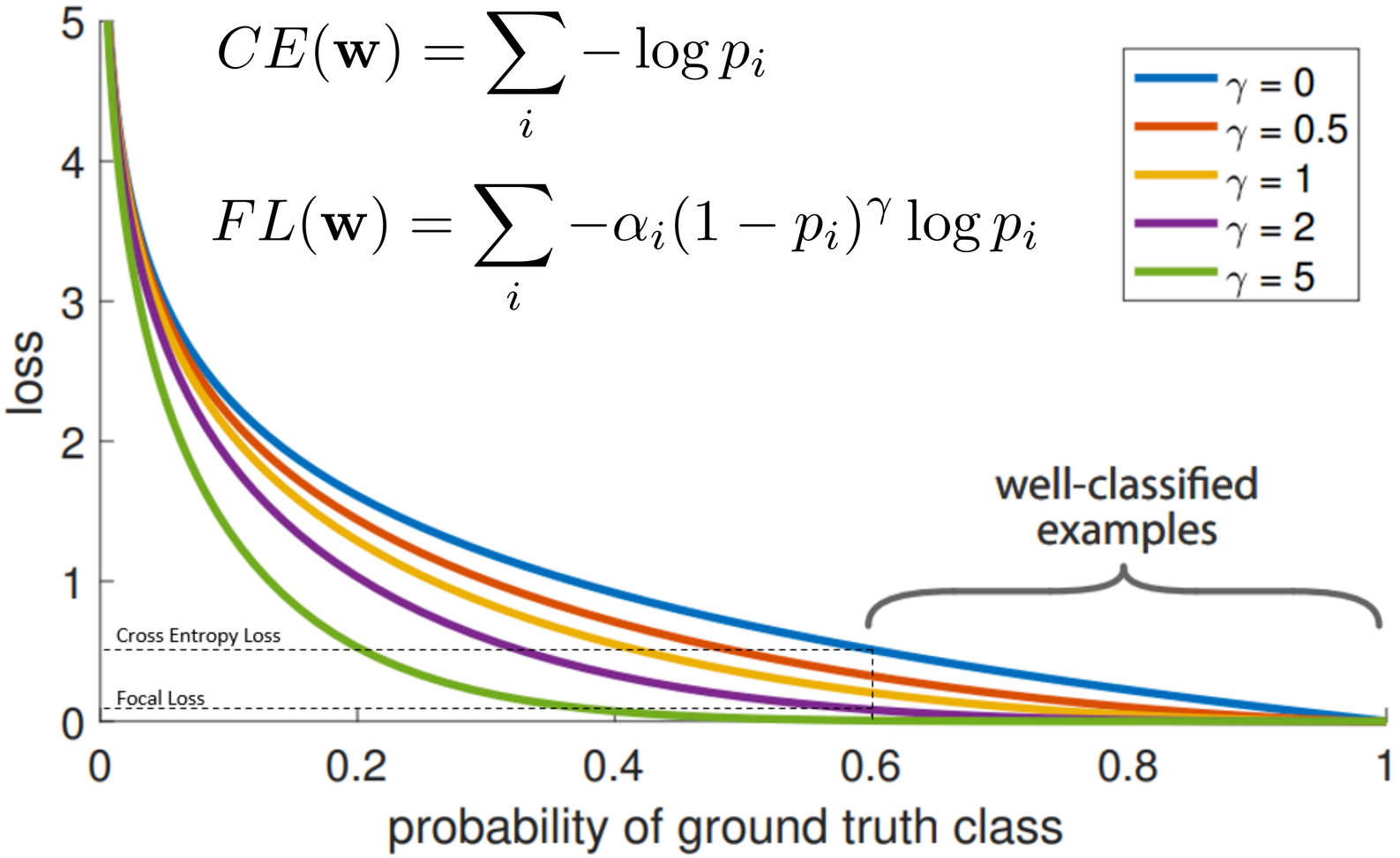
input probs: `torch.nn.CrossEntropyLoss`

Loss functions

focal loss = less aggressive cross-entropy (γ) + unbalanced classes (α)

$$CE(\mathbf{w}) = \sum_i -\log p_i$$

$$FL(\mathbf{w}) = \sum_i -\alpha_i (1-p_i)^\gamma \log p_i$$



Loss functions: Ranking loss

PyTorch: `torch.nn.MarginRankingLoss()`

Trn data triplets:

$$(\mathbf{x}_i, \mathbf{x}_j, y_{ij})$$

Interpretation:

$$\text{if } \; y_{ij} = +1, \quad \Rightarrow \quad f(\mathbf{x}_i, \mathbf{w}) > f(\mathbf{x}_j, \mathbf{w})$$
$$\text{if } \; y_{ij} = -1, \quad \Rightarrow \quad f(\mathbf{x}_i, \mathbf{w}) < f(\mathbf{x}_j, \mathbf{w})$$

Loss construction:

$$y_i(f(\mathbf{x}_i, \mathbf{w}) - f(\mathbf{x}_j, \mathbf{w})) > 0$$

$$\mathscr{L}_{\text{rank}}(\mathbf{w}) = \sum_{ij} \text{ReLU}\Big(-y_i\big(f(\mathbf{x}_i, \mathbf{w}) - f(\mathbf{x}_j, \mathbf{w})\big)\Big)$$