# Learning for vision IV training & layers

Karel Zimmermann
http://cmp.felk.cvut.cz/~zimmerk/

Vision for Robotics and Autonomous Systems
https://cyber.felk.cvut.cz/vras/

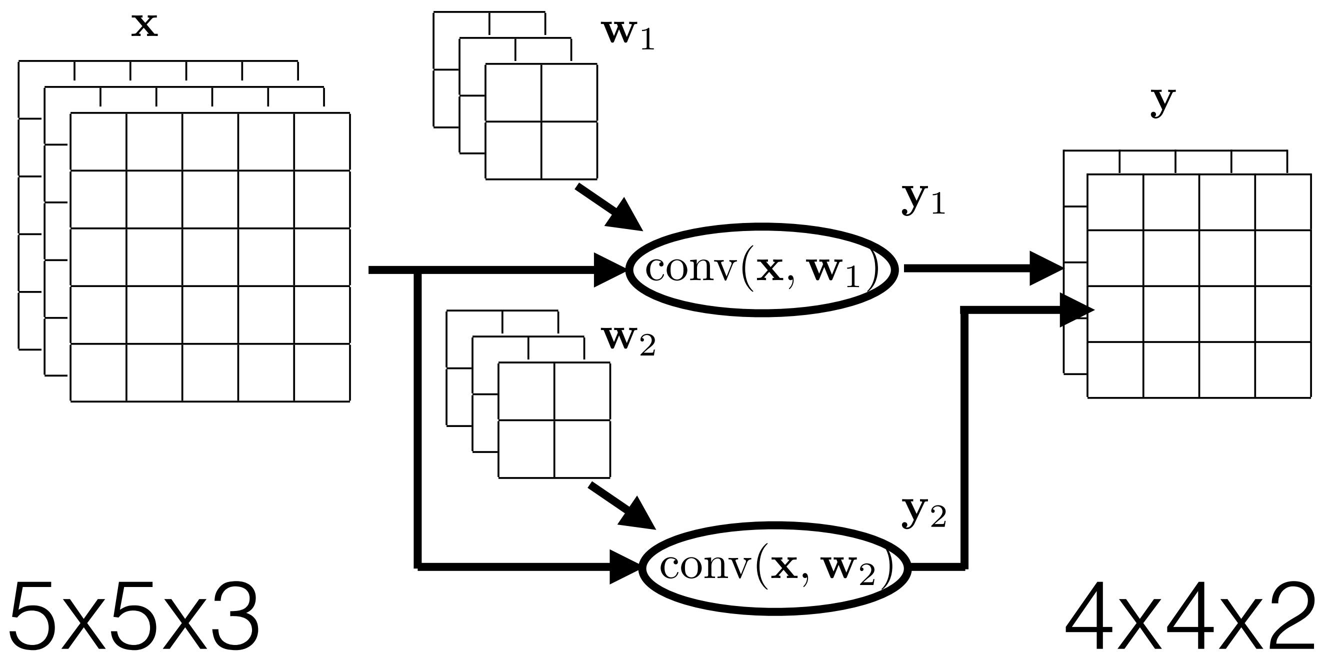
Center for Machine Perception
https://cmp.felk.cvut.cz

Department for Cybernetics
Faculty of Electrical Engineering
Czech Technical University in Prague

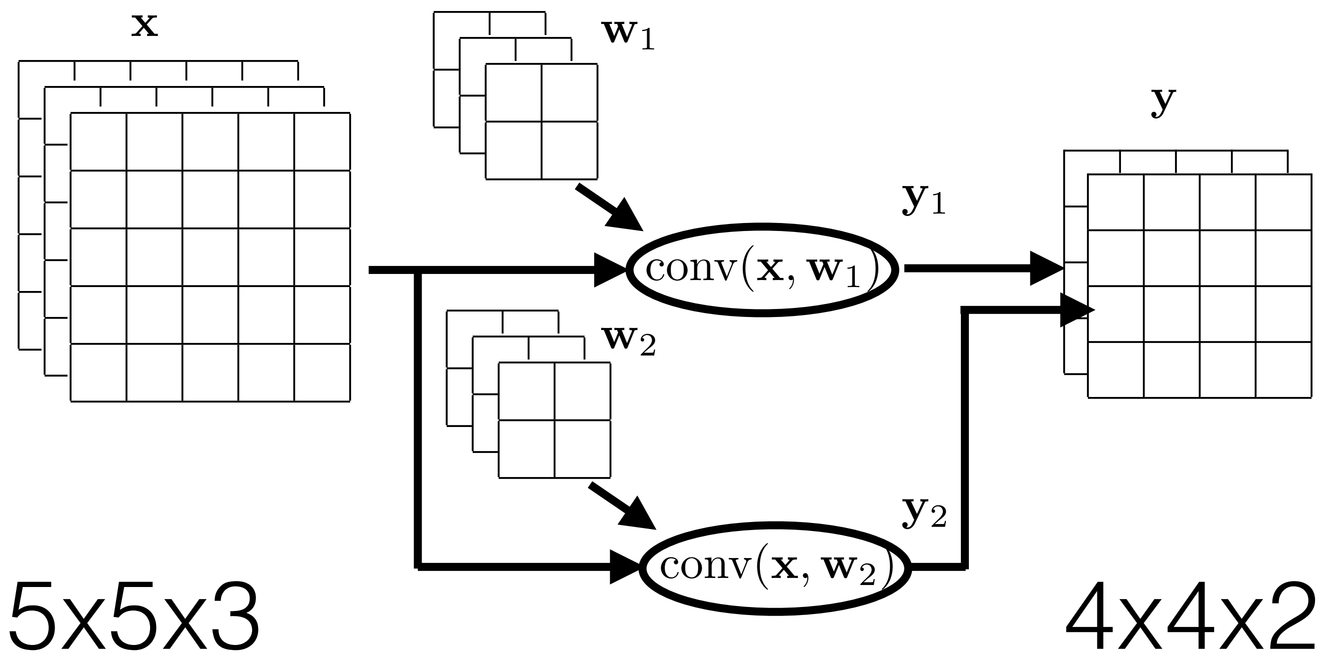
# Outline

- layers:
  - convolutional layer
  - activation function (i.e. non-linearities)
  - batch normalization layer
  - max-pooling layer
  - loss-layers
- summary of the learning procedure
  - train, test, val data,
  - hyper-parameters,
  - regularizations

# 2D convolution forward pass

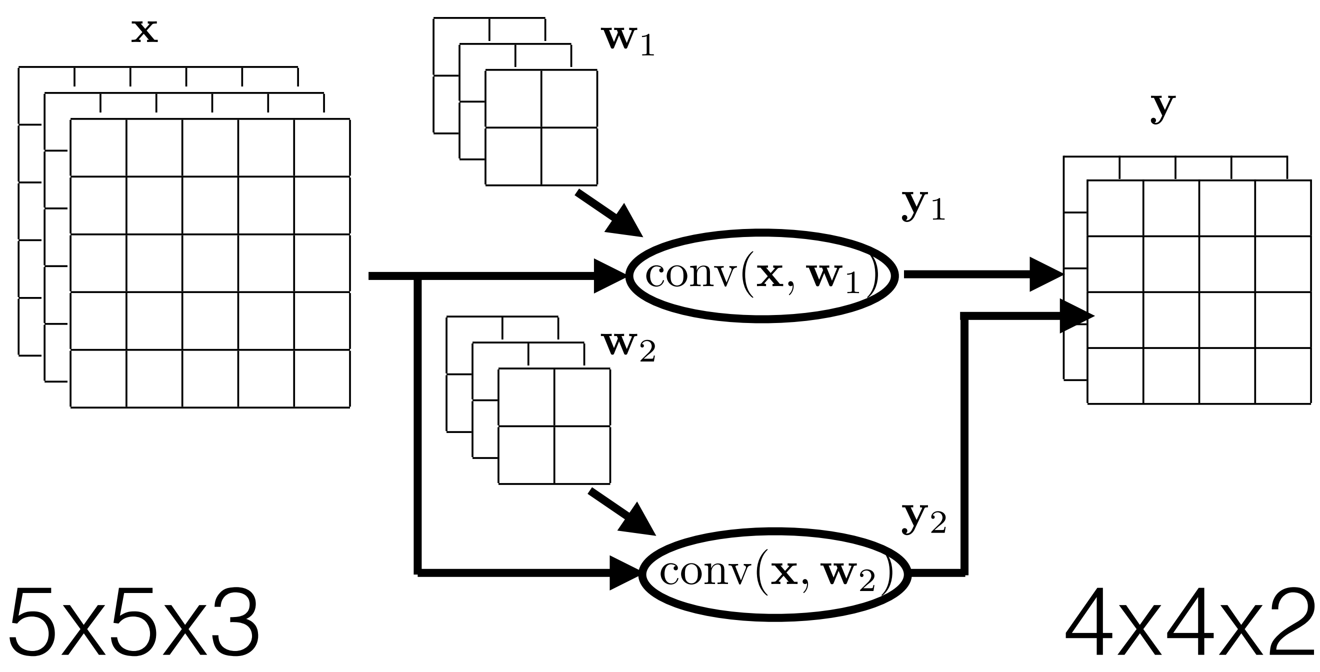


# 2D convolution forward pass




```
# initialise
import torch.nn as nn
# define 2D convolutional layer
first_layer = nn.Conv2d(in_channels=3, out_channels=2,
                        kernel_size=2, stride=1,
                        padding=1)
```

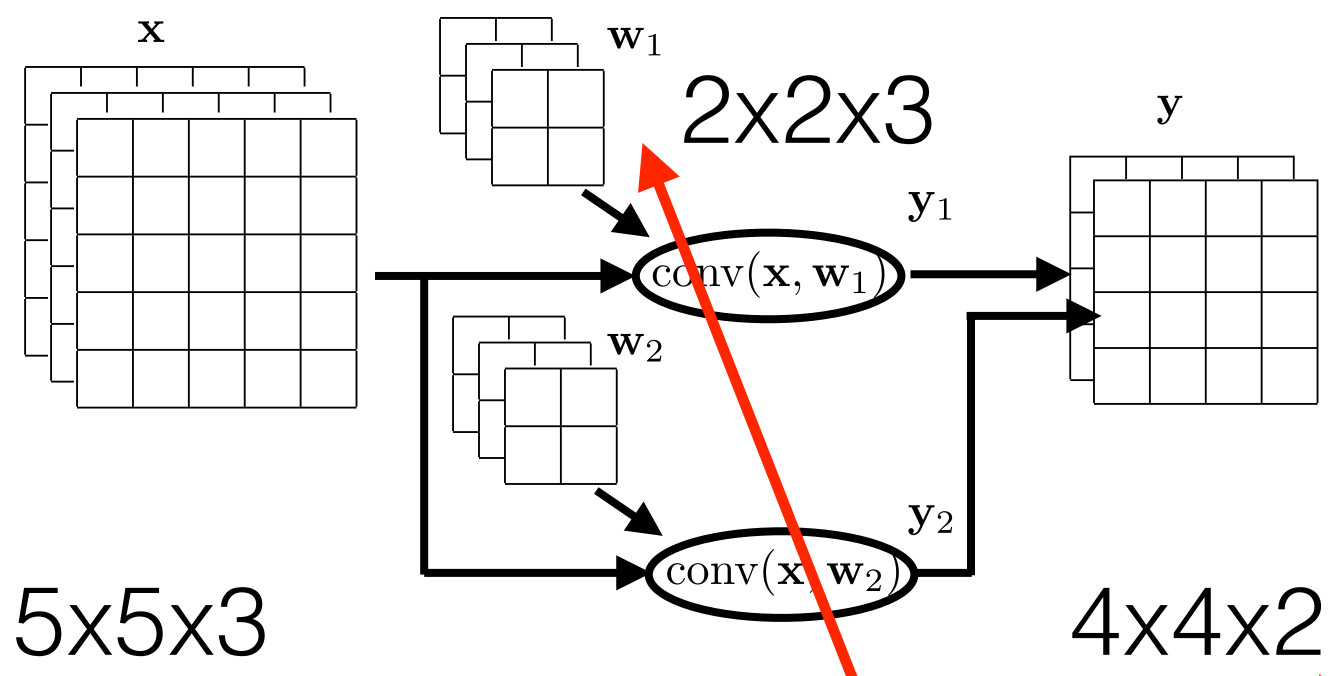
# 2D convolution forward pass




also number of kernels

```
# initialise
import torch.nn as nn
# define 2D convolutional layer
first_layer = nn.Conv2d(in_channels=3, out_channels=3,
                        kernel_size=2, stride=1,
                        padding=1)
```

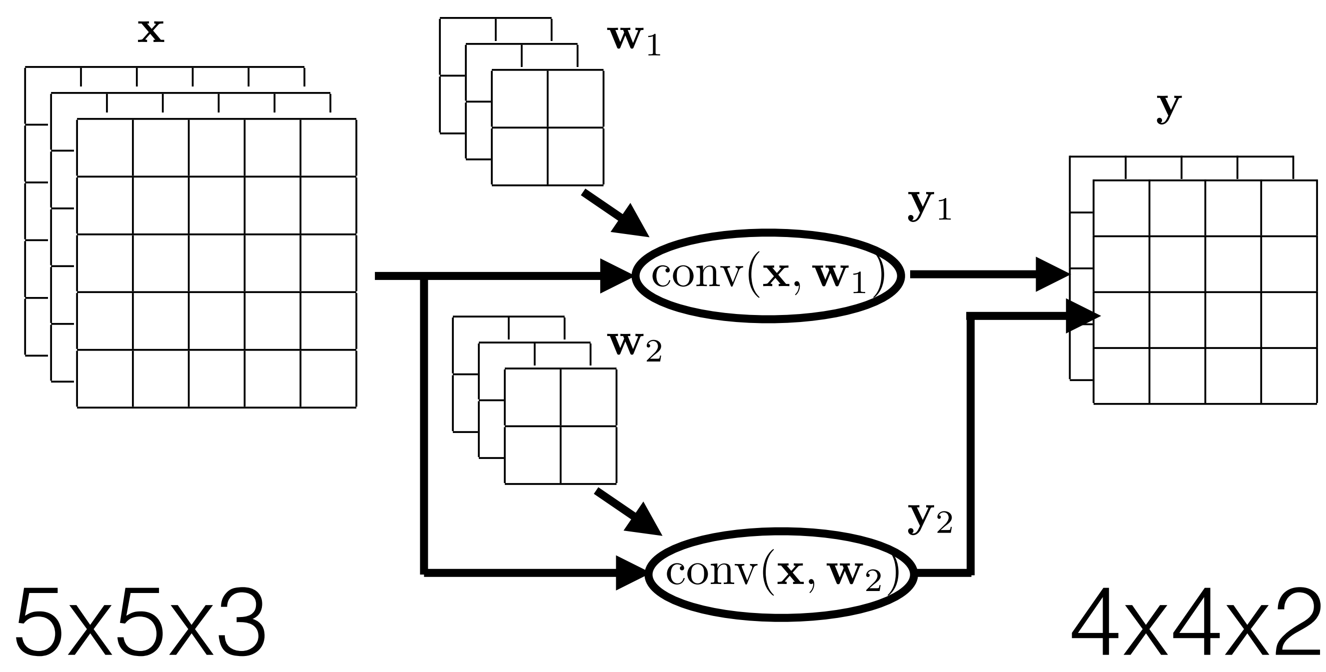
# 2D convolution forward pass




also number of kernels

```
# initialise
import torch.nn as nn
# define 2D convolutional layer
first_layer = nn.Conv2d(in_channels=3, out_channels=2,
                        kernel_size=2, stride=1,
                        padding=1)
```

# 2D convolution forward pass




Very important property of convolutional layer is:

**Local gradient is also convolution !!!**

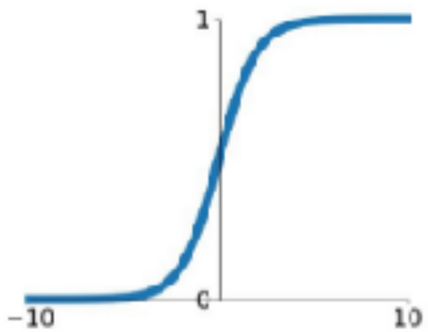
# Outline

- layers:
  - convolutional layer
  - activation function (i.e. non-linearities)
  - batch normalization layer
  - max-pooling layer
  - loss-layers
- summary of the learning procedure
  - train, test, val data,
  - hyper-parameters,
  - regularizations

# Activation functions

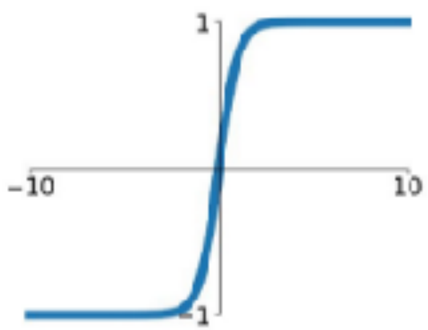
**Sigmoid**

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



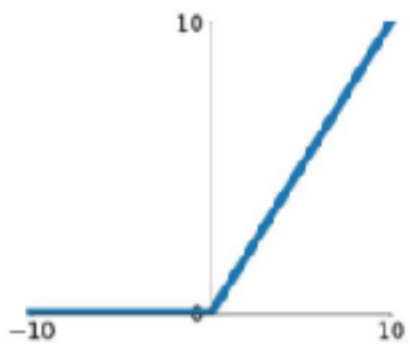

**tanh**

$\tanh(x)$



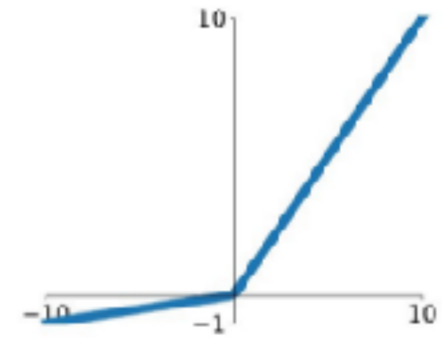

**ReLU**

$\max(0, x)$




**Leaky ReLU**

$\max(0.1x, x)$

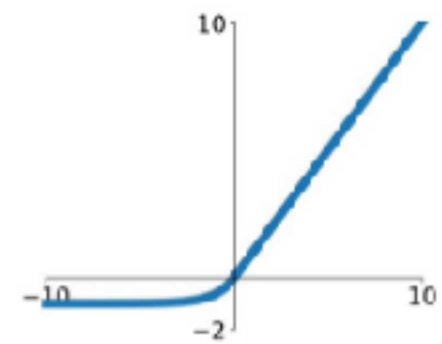



**Maxout**

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

**ELU**

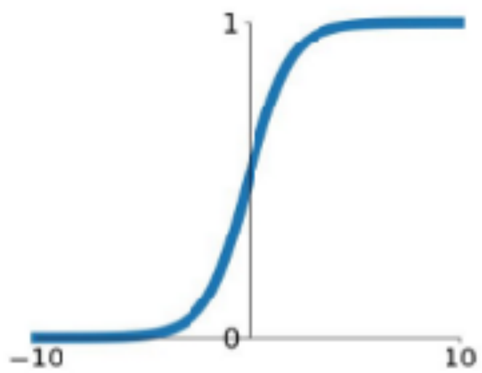
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



- what happen to backprop gradient when weights are **huge**?

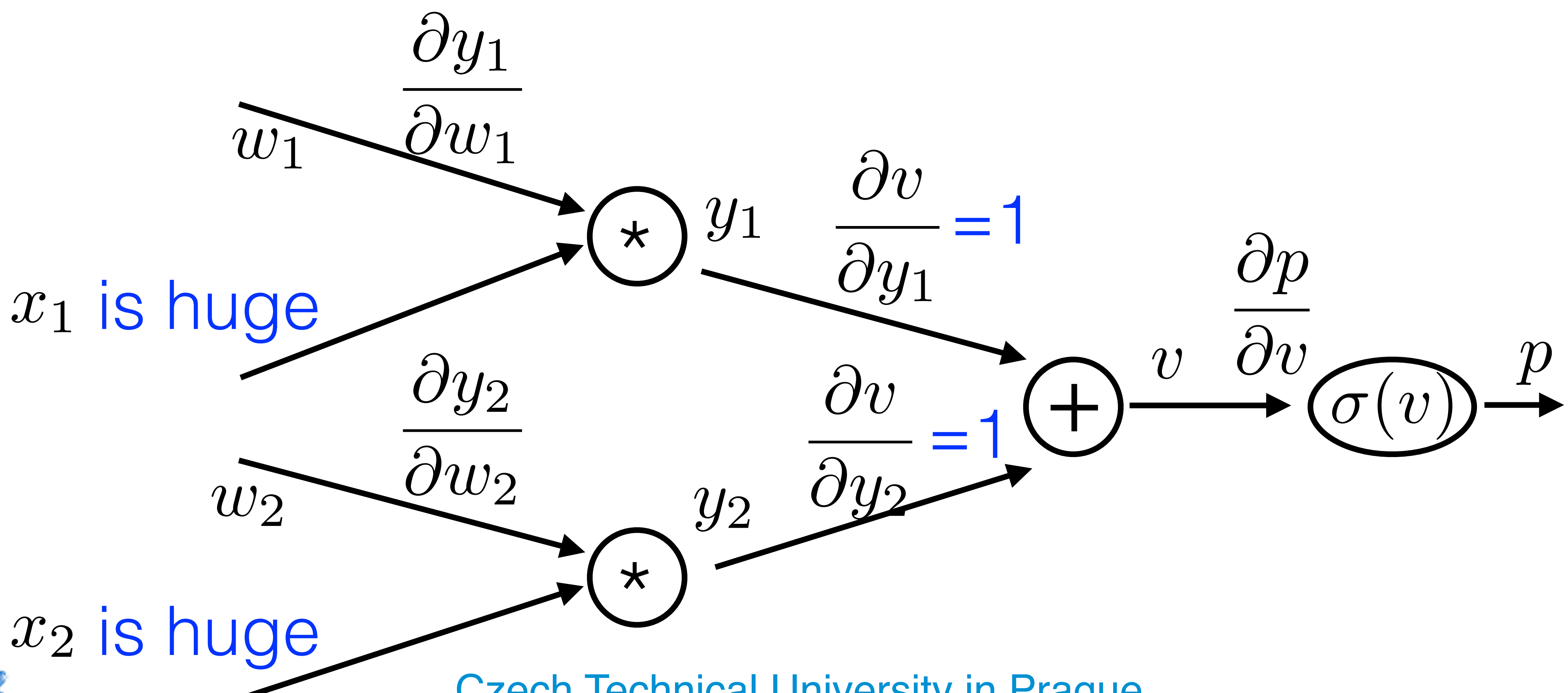
**Sigmoid**

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




- zero gradient when saturated
- not zero-centered (pos. output)
- computationally expensive

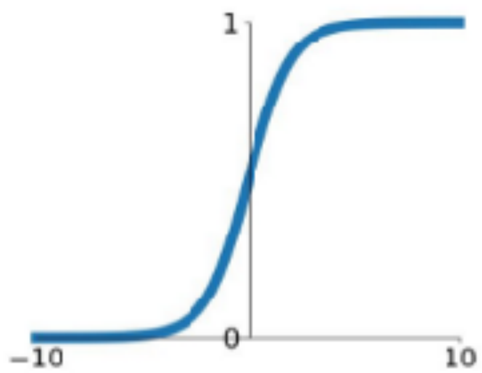
$$\frac{\partial p}{\partial w_1} = \frac{\partial y_1}{\partial w_1}\frac{\partial v}{\partial y_1}\frac{\partial p}{\partial v} = 0 \qquad \frac{\partial p}{\partial w_2} = \frac{\partial y_2}{\partial w_2}\frac{\partial v}{\partial y_1}\frac{\partial p}{\partial v} = 0$$



# Activation functions

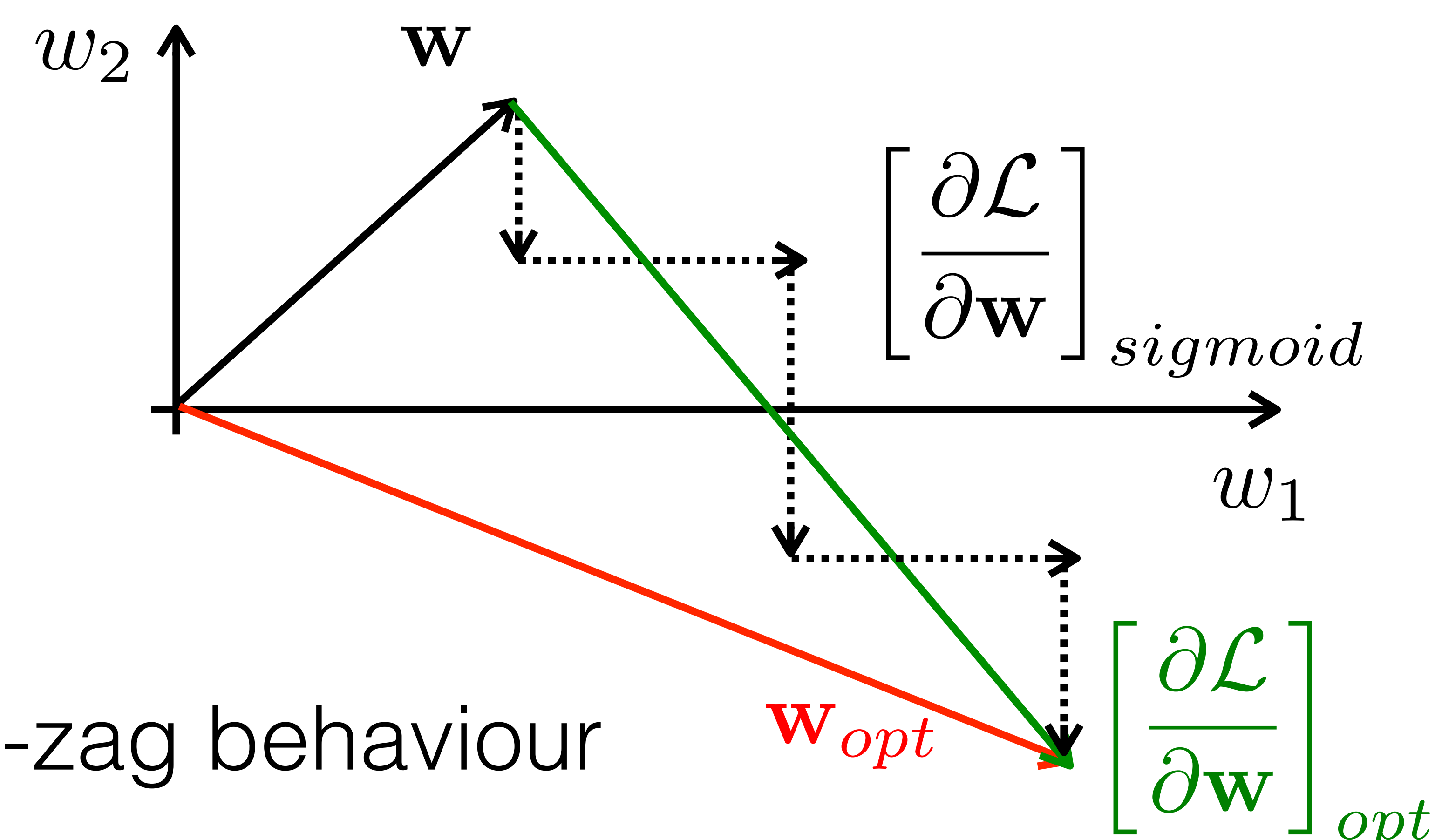
**Sigmoid**

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




- zero gradient when saturated
- not zero-centered (pos. output)
- computationally expensive

$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial p}{\partial \mathbf{w}} \cdot \frac{\partial \mathcal{L}(p)}{\partial p} \quad \begin{matrix} >0 \\ <0 \end{matrix}$$

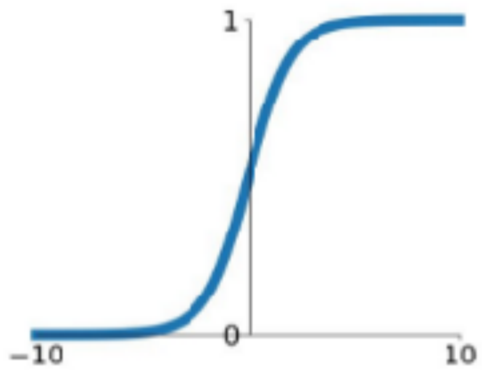



Undesired zig-zag behaviour

# Activation functions

**Sigmoid**

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

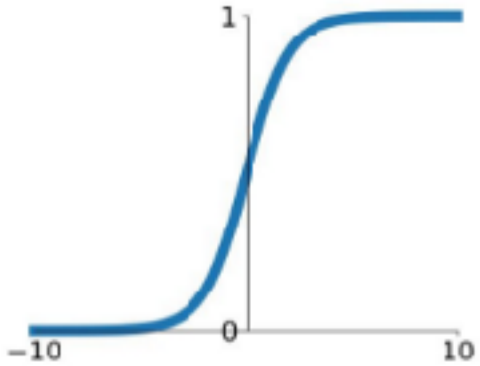



- zero gradient when saturated
- not zero-centered (pos. output)
- computationally expensive

# Activation functions

**Sigmoid**

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



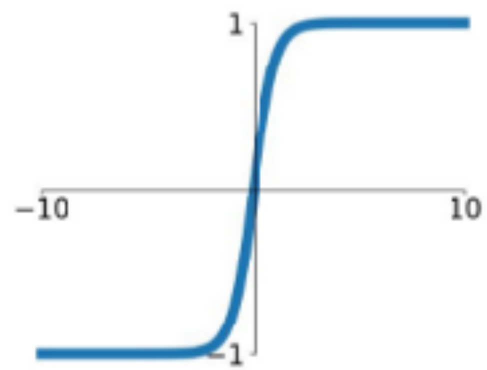

- zero gradient when saturated
- not zero-centered (pos. output)
- computationally expensive

PyTorch: `nn.Sigmoid()`

# Activation functions

**tanh**

$$\tanh(x)$$



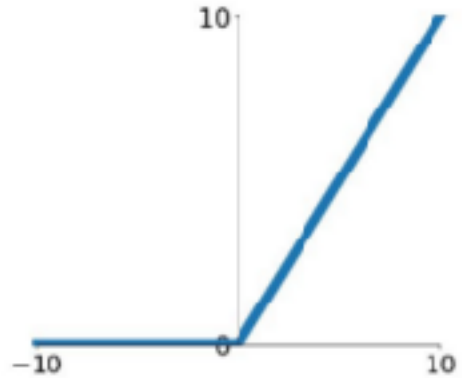

- zero gradient when saturated
- ~~not zero-centered (only positive ouputs)~~
- computationally expensive

PyTorch: `nn.Tanh()`

# Activation functions

**ReLU**

$\max(0, x)$



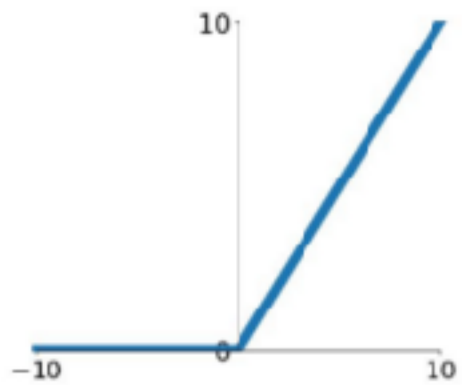

- ~~zero gradient when saturated~~ *(partially => dead ReLU!)*
- not zero-centered (only positive ouputs)
- ~~computationally expensive~~

PyTorch: `nn.ReLu()`

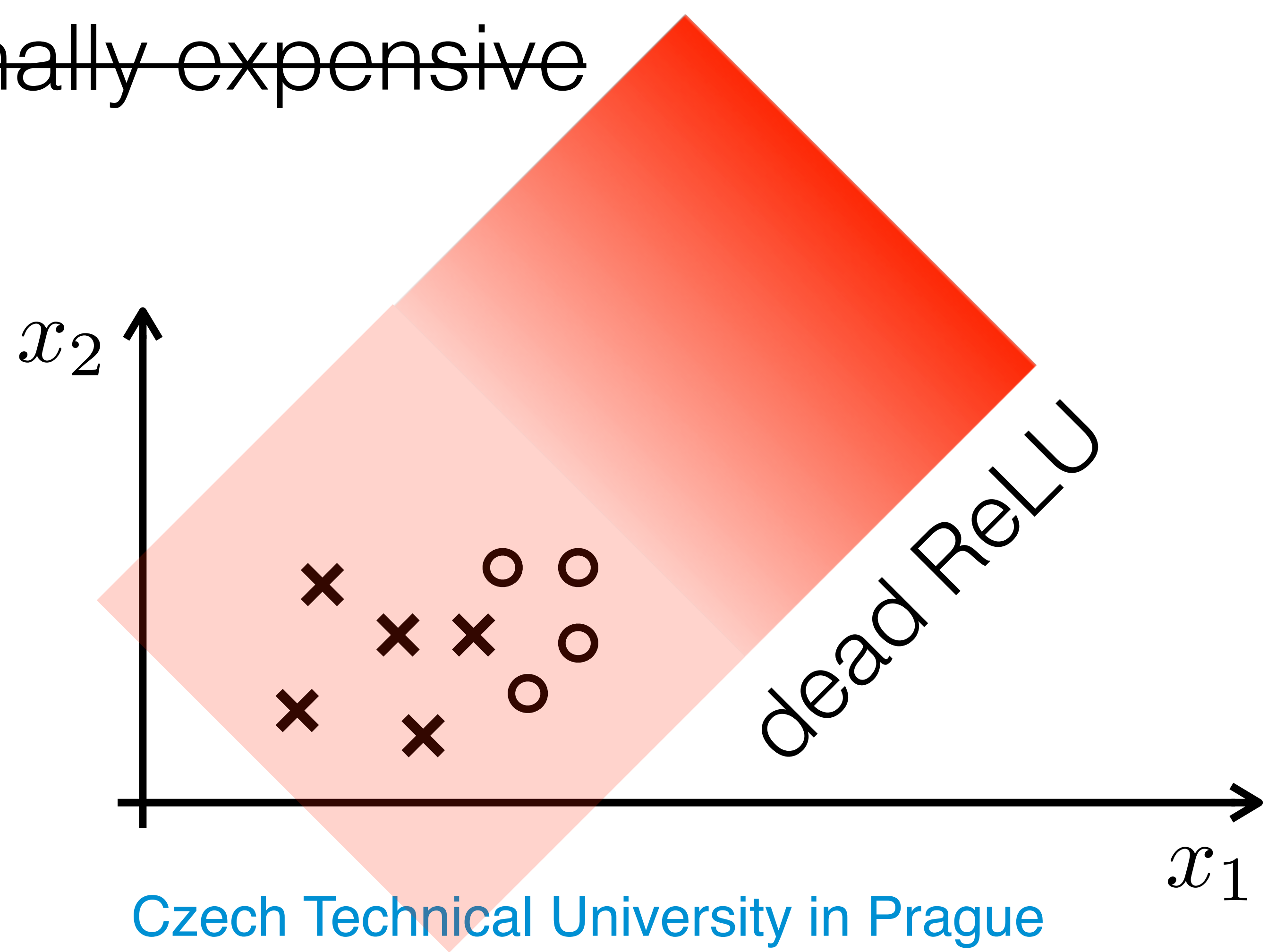
# Activation functions

**ReLU**

$\max(0, x)$



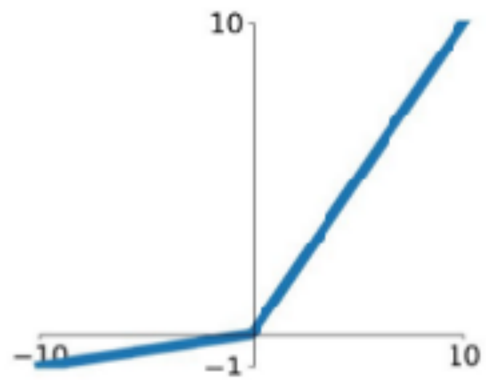

- ~~zero gradient when saturated~~ *(partially => dead ReLU!)*
- not zero-centered (only positive ouputs)
- ~~computationally expensive~~



# Activation functions

**Leaky ReLU**

$\max(0.1x, x)$




- ~~zero gradient when saturated~~
- ~~not zero-centered (only positive ouputs)~~
- ~~computationally expensive~~

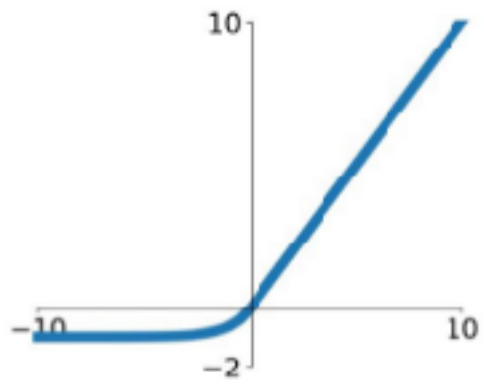
Small gradient for negative values give tiny chance to recover

PyTorch: `nn.LeakyReLU(negative_slope=1e-2)`

# Activation functions

**ELU**

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




- ~~zero gradient when saturated~~ *(partially)*
- ~~not zero-centered (only positive ouputs)~~
- computationally expensive

PyTorch: `nn.LeakyReLU(alpha=1)`

# Outline

- SGD vs deterministic gradient
- what makes learning to fail
- layers:
  - activation function (i.e. non-linearities)
  - initialization
  - batch normalization layer
  - max-pooling layer
  - loss-layers
- summary of the learning procedure
  - train, test, val data,
  - hyper-parameters,
  - regularizations

# Data preprocessing & initializations

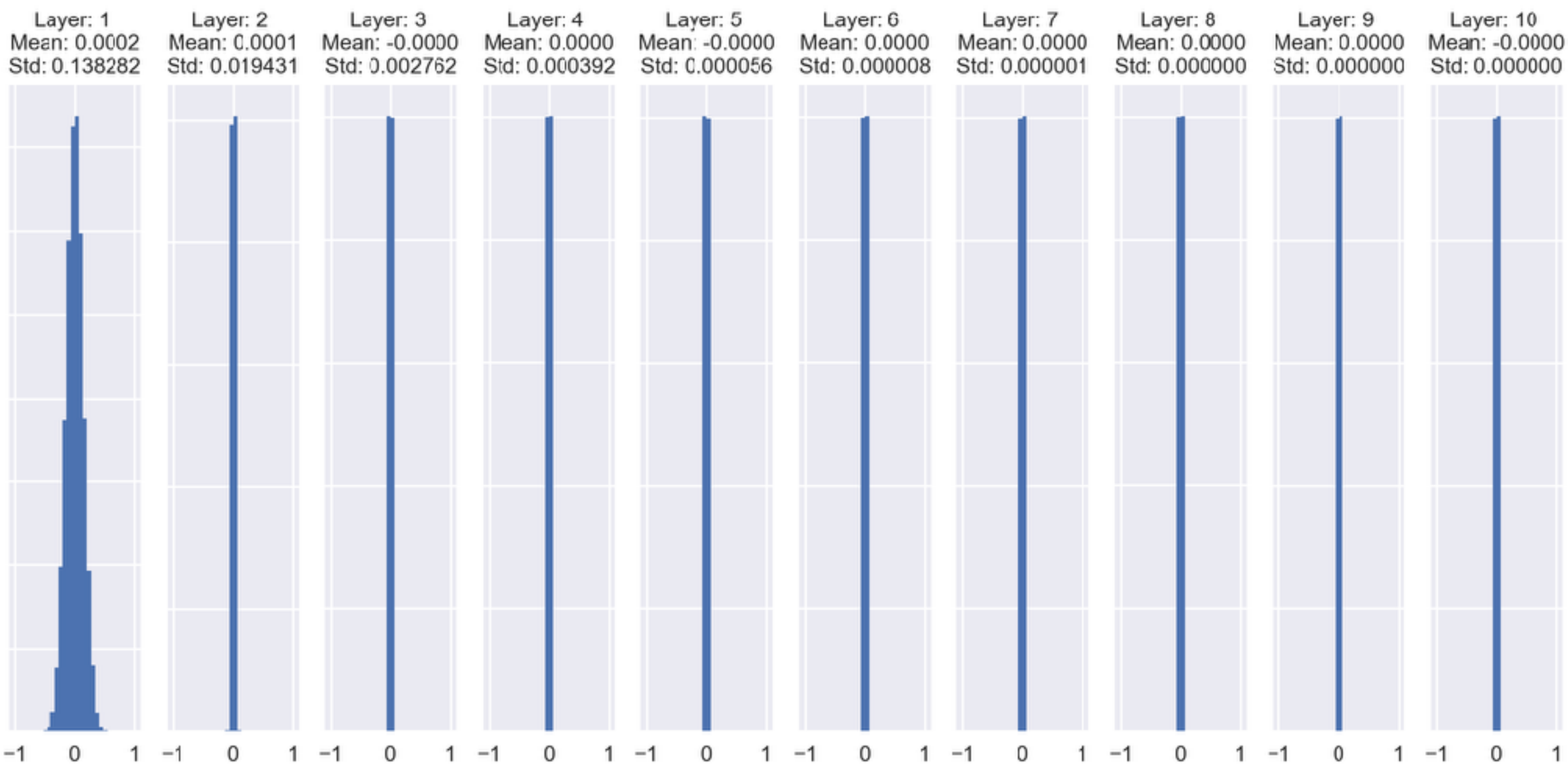
- Pixels values shifted zero mean to avoid only positive inputs and the unwanted “zig-zag” behaviour

# Data preprocessing & initializations

- Pixels values shifted zero mean to avoid only positive inputs and the unwanted “zig-zag” behaviour
- Weight initialization:
  - $\mathbf{w} = 0$ all gradients the same
  - $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \sigma)$ diminishing gradients in backprop
  - $\mathbf{w}^{(i)} \sim \mathcal{N}(\mathbf{0}, \sigma * 1/N^{(i)})$ preserves variance of signal among layers (Xavier init [Glorot 2010])

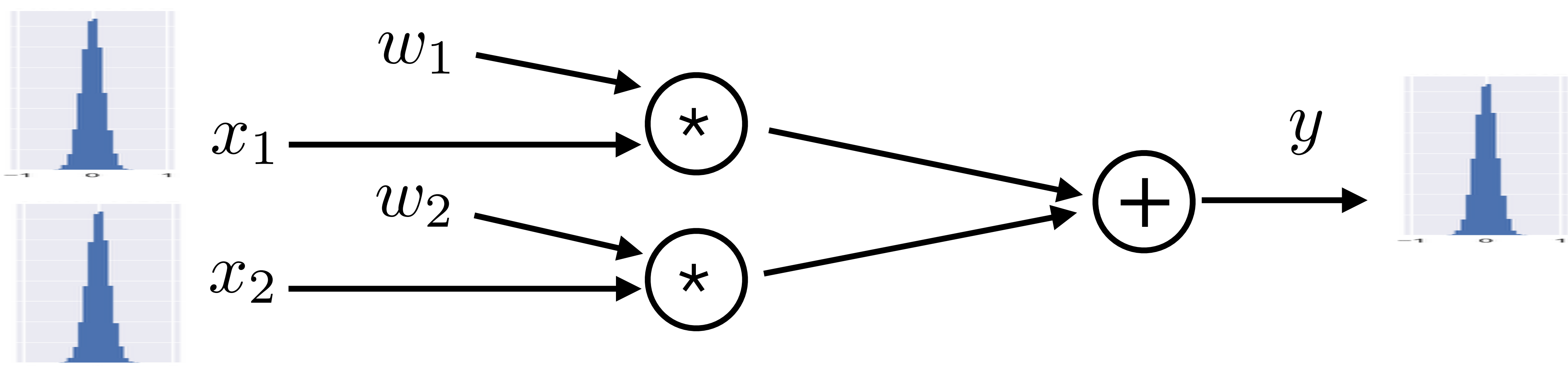
# Xavier initialization [Glorot 2010]

Signal in randomly initialized weights $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \sigma)$ forward (and backward) pass



# Xavier initialization [Glorot 2010]

- We want to preserve variance of signal among layers (i.e. $\mathrm{var}(y) = \mathrm{var}(x_i)$ )




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

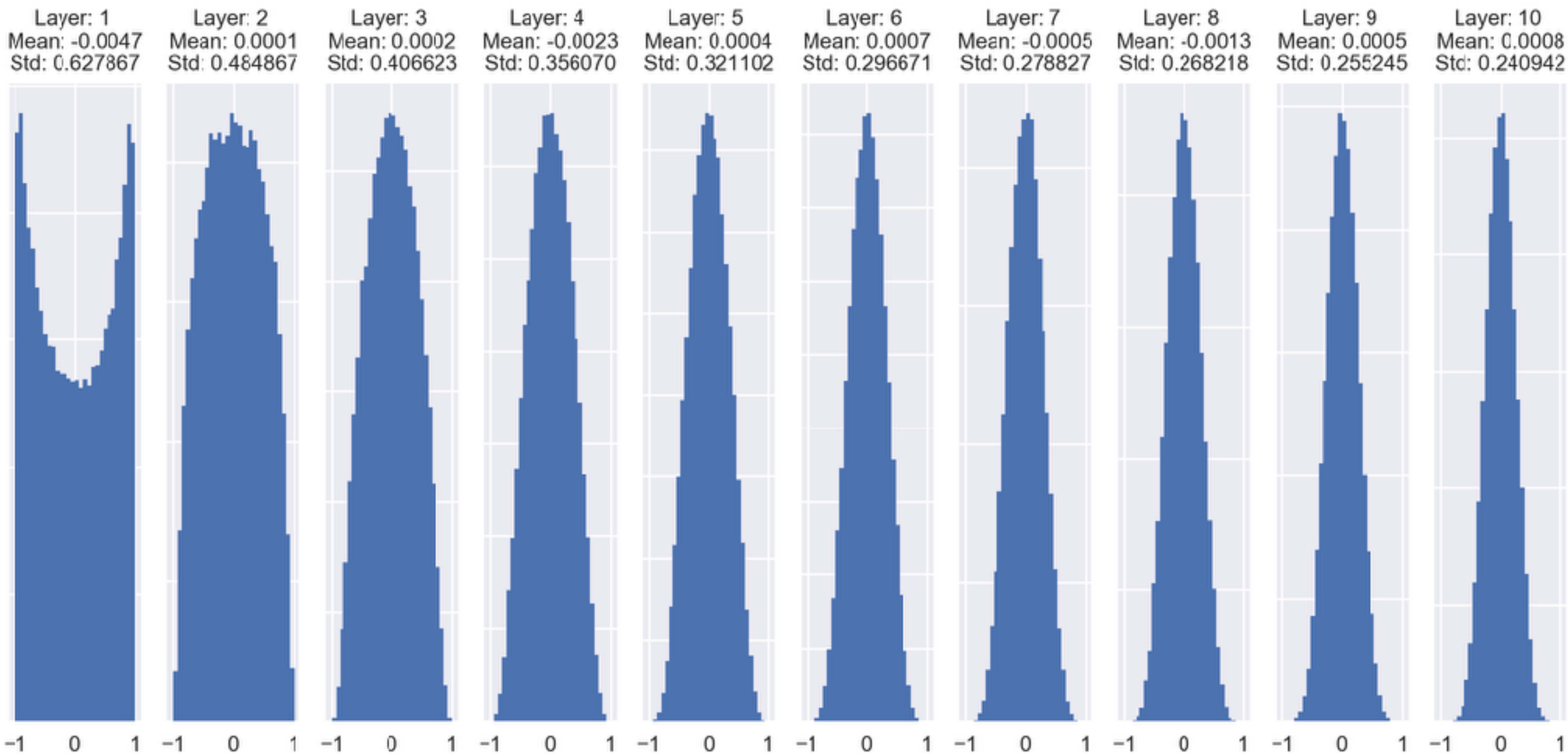
$$= \sum_{i=1}^{N} \mathbb{E}(x_i)^2 \mathrm{var}(w_i) + \mathbb{E}(w_i)^2 \mathrm{var}(x_i) + \mathrm{var}(w_i x_i) =$$

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

$$\Rightarrow N * \mathrm{var}(w_i) = 1$$

# Xavier initialization [Glorot 2010]

Signal in Xavier initialized weights $\mathbf{w}^{(i)} \sim \mathcal{N}(\mathbf{0}, \sigma * 1/N^{(i)})$ forward (and backward) pass (better but not ideal)

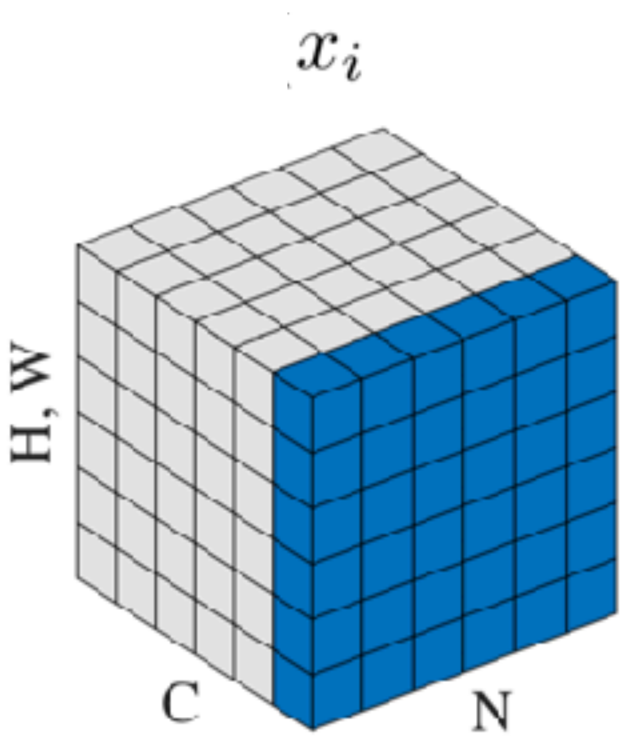


# Outline

- SGD vs deterministic gradient
- what makes learning to fail
- layers:
    - activation function (i.e. non-linearities)
    - initialization
    - batch normalization layer
    - max-pooling layer
    - loss-layers
- summary of the learning procedure
    - train, test, val data,
    - hyper-parameters,
    - regularizations

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

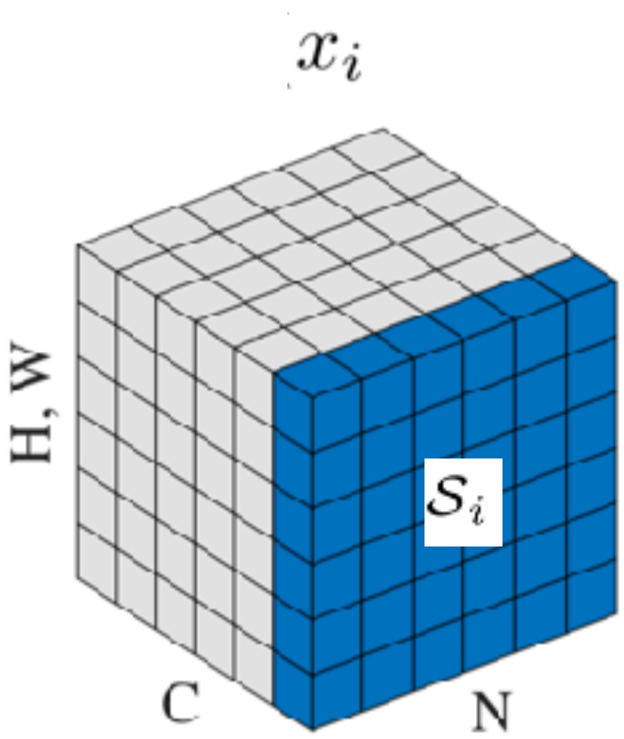
Batch is 4D tensor (visualization in 3D) of values $x_i$ (cubes)




$$i = (i_N, i_C, i_H, i_W)$$
is 4D index

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

Batch is 4D tensor (visualization in 3D) of values $x_i$ (cubes)




$$i = (i_N, i_C, i_H, i_W)$$
is 4D index

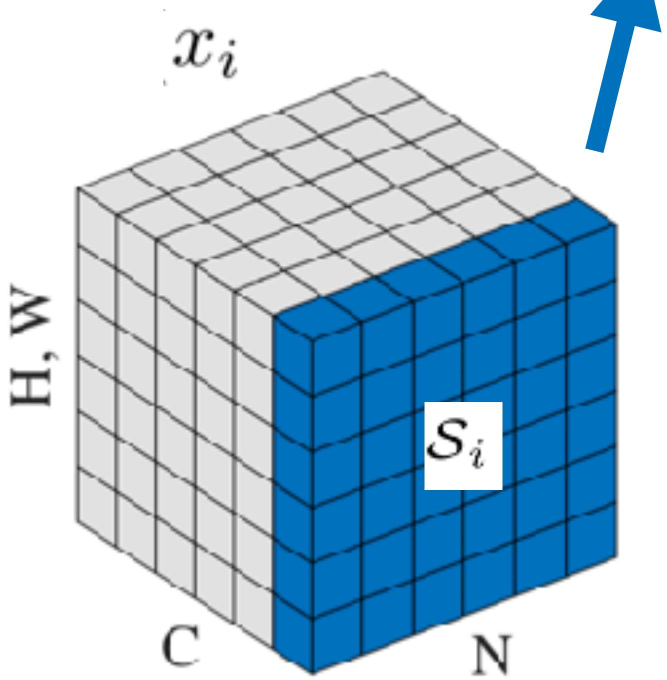
Set of cubes determined by indices
$$\mathcal{S}_i = \{k \mid k_C = i_C\}$$

$$\mathcal{S}_{1,1,1,1} = \{(1,1,1,1),(2,1,1,1),\ldots(N,1,H,W)\}$$
$$\vdots \qquad \vdots \qquad \vdots \qquad \vdots$$
$$\mathcal{S}_{N,1,H,W} = \{(1,1,1,1),(2,1,1,1),\ldots(N,1,H,W)\}$$

# Batch normalization layer [Ioffe and Szegedy 2015]

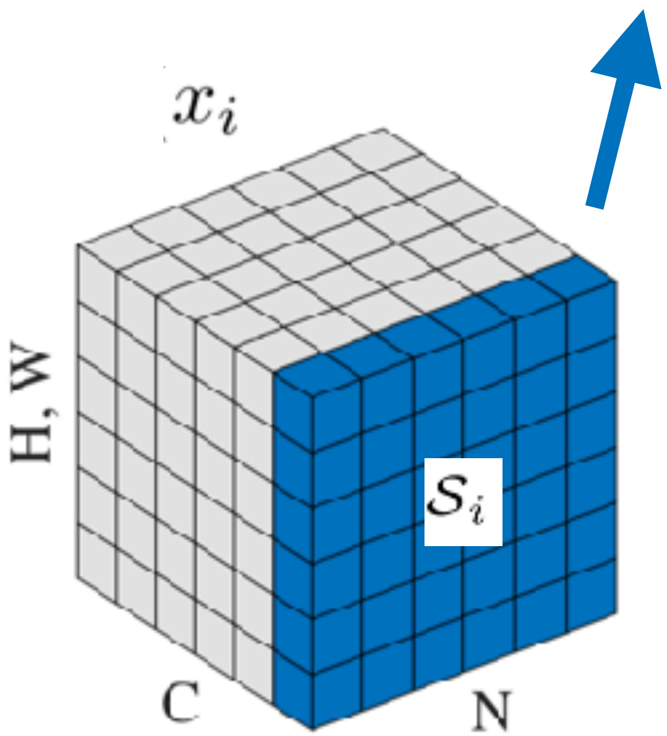
https://arxiv.org/pdf/1502.03167.pdf (over 6k citation)

$$\mu_i = \frac{1}{m} \sum_{k \in \mathcal{S}_i} x_k, \quad \sigma_i = \sqrt{\frac{1}{m} \sum_{k \in \mathcal{S}_i} (x_k - \mu_i)^2 + \epsilon},$$




For each channel i compute mean a std

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



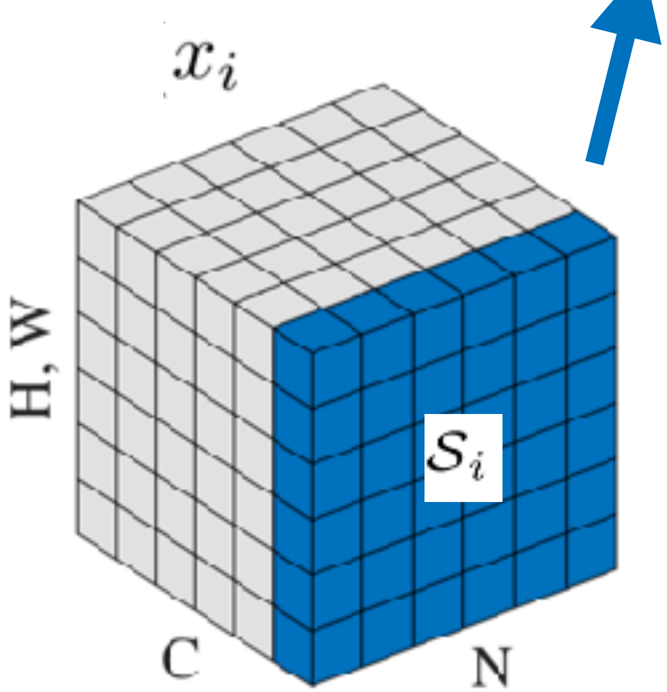

$$\mu_i = \frac{1}{m}\sum_{k\in\mathcal{S}_i} x_k, \quad \sigma_i = \sqrt{\frac{1}{m}\sum_{k\in\mathcal{S}_i}(x_k-\mu_i)^2+\epsilon},$$

$$\hat{x}_i = \frac{1}{\sigma_i}(x_i-\mu_i)$$

Normalize all values in channel i by estimated mu and std

# Batch normalization layer [Ioffe and Szegedy 2015]

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



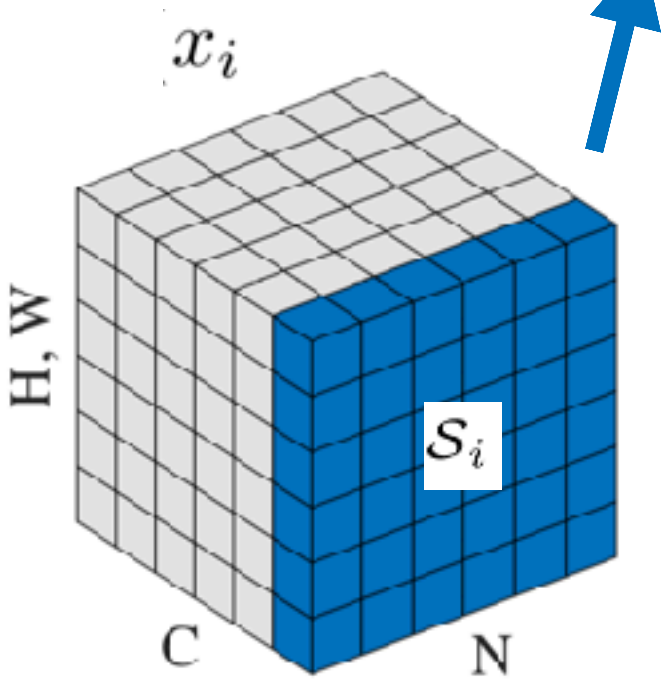

$$\mu_i = \frac{1}{m}\sum_{k\in\mathcal{S}_i} x_k, \quad \sigma_i = \sqrt{\frac{1}{m}\sum_{k\in\mathcal{S}_i}(x_k-\mu_i)^2+\epsilon},$$

$$\hat{x}_i = \frac{1}{\sigma_i}(x_i-\mu_i)$$

$$y_i = \gamma\hat{x}_i + \beta,$$

In some cases biased values are needed => introduce trainable affine transformation initialized in gamma=1, beta =0

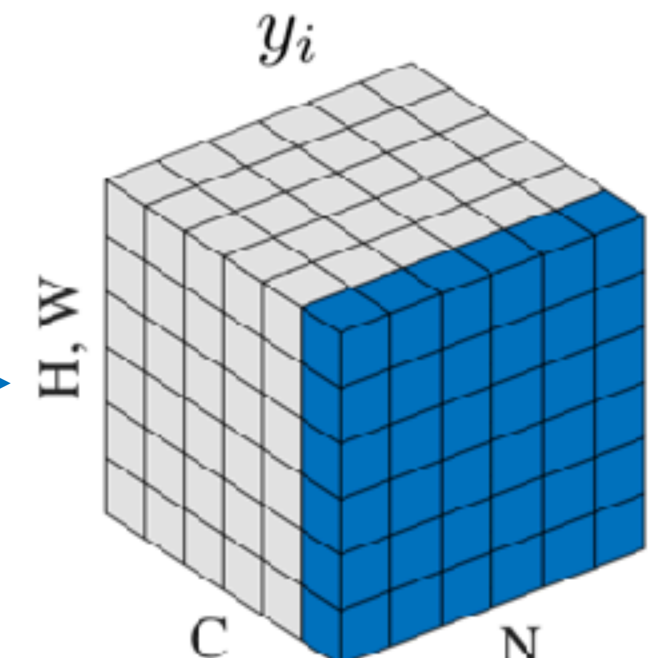
# Batch normalization layer [Ioffe and Szegedy 2015]
https://arxiv.org/pdf/1502.03167.pdf (over 6k citation)




$$\mu_i = \frac{1}{m} \sum_{k \in \mathcal{S}_i} x_k, \quad \sigma_i = \sqrt{\frac{1}{m} \sum_{k \in \mathcal{S}_i} (x_k - \mu_i)^2 + \epsilon},$$

$$\hat{x}_i = \frac{1}{\sigma_i}(x_i - \mu_i)$$

$$y_i = \gamma \hat{x}_i + \beta,$$

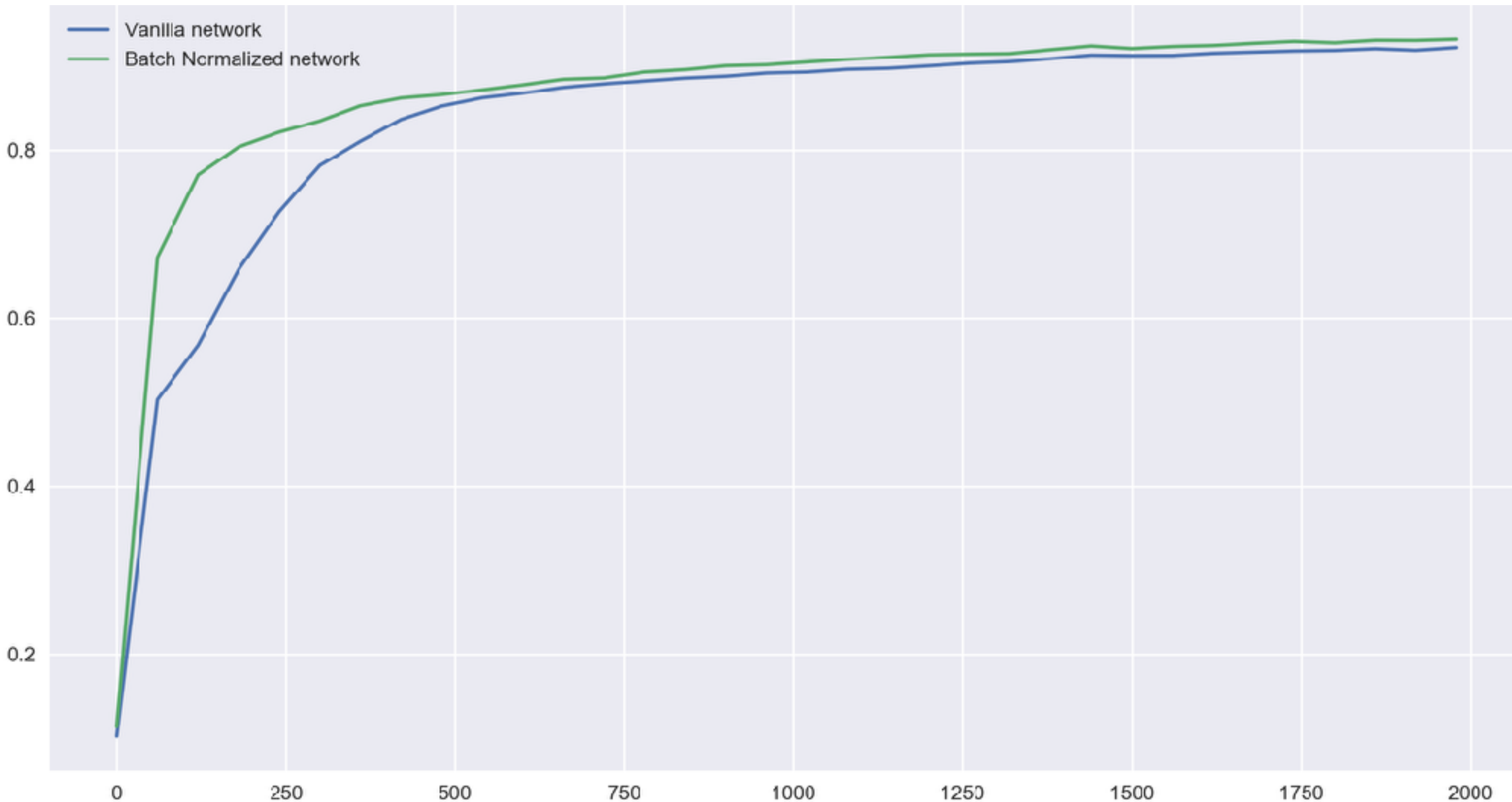



- Testing phase: $\mu_i = \mathbb{E}[x_i]$ and $\sigma_i = \mathbb{E}[(x_i - \mathbb{E}[x_i])^2]$ estimated over the whole training set.

# Batch normalization layer [Ioffe and Szegedy 2015]

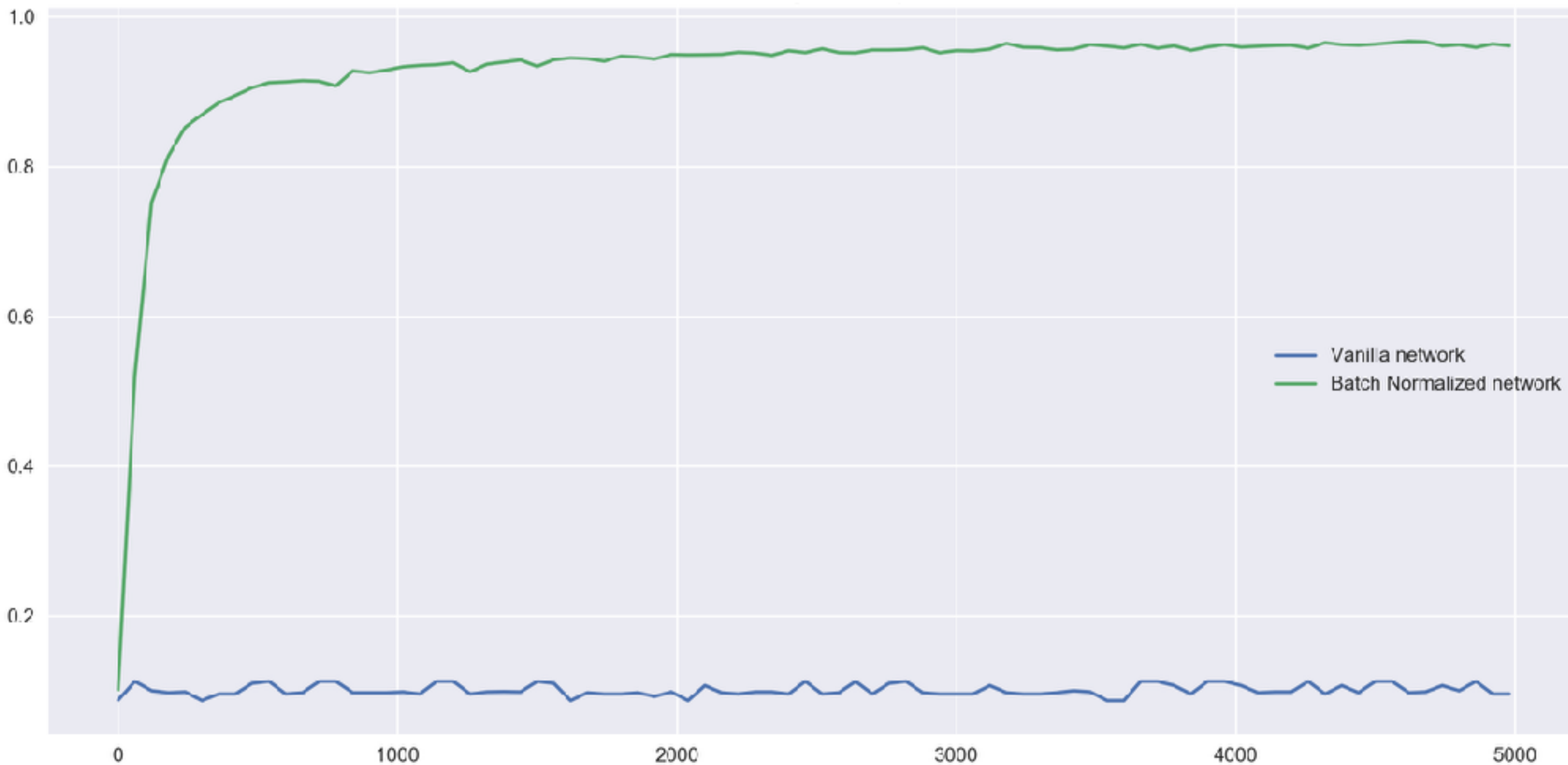
https://arxiv.org/pdf/1502.03167.pdf (over 6k citation)

## Good weight initialization

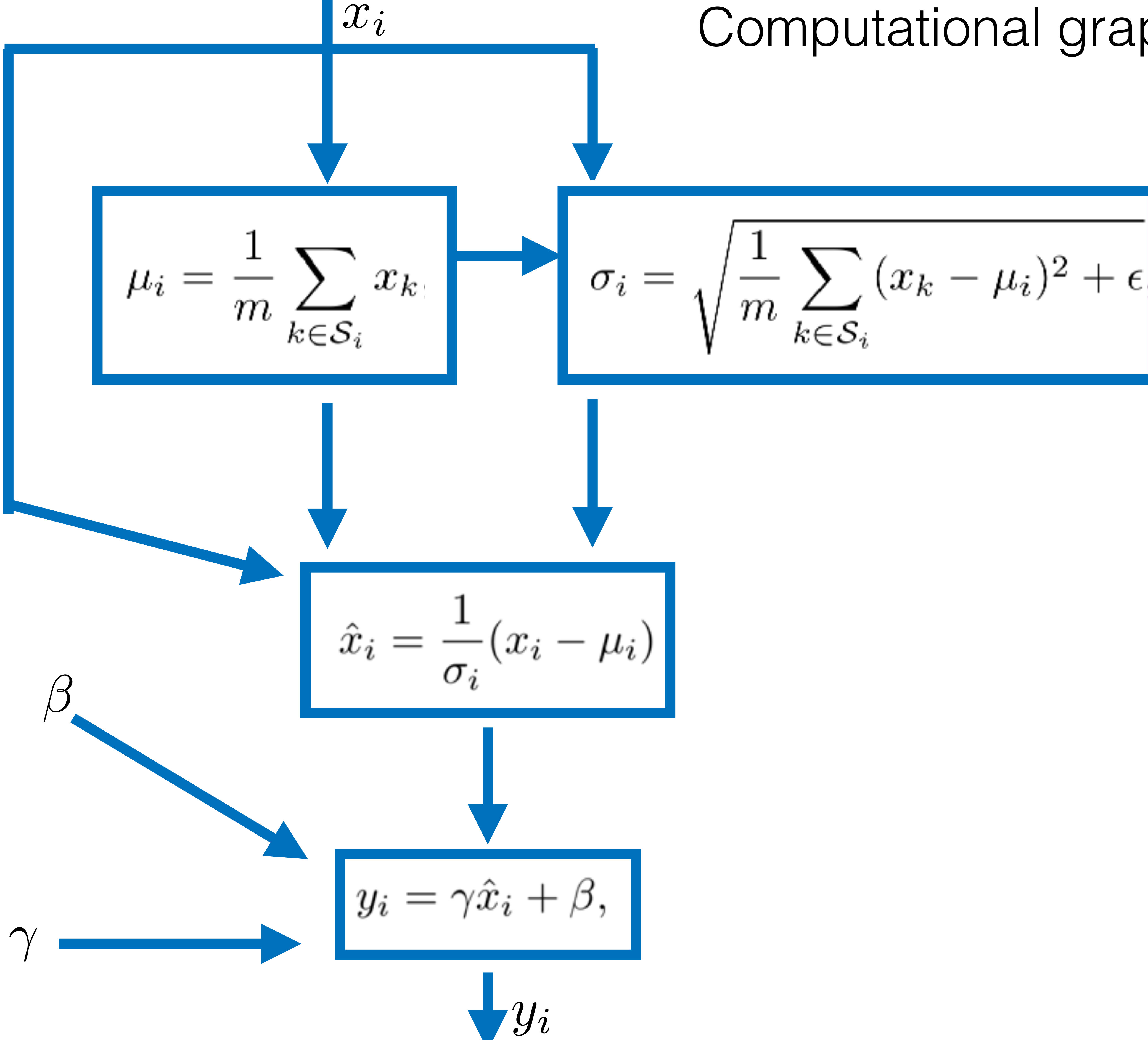


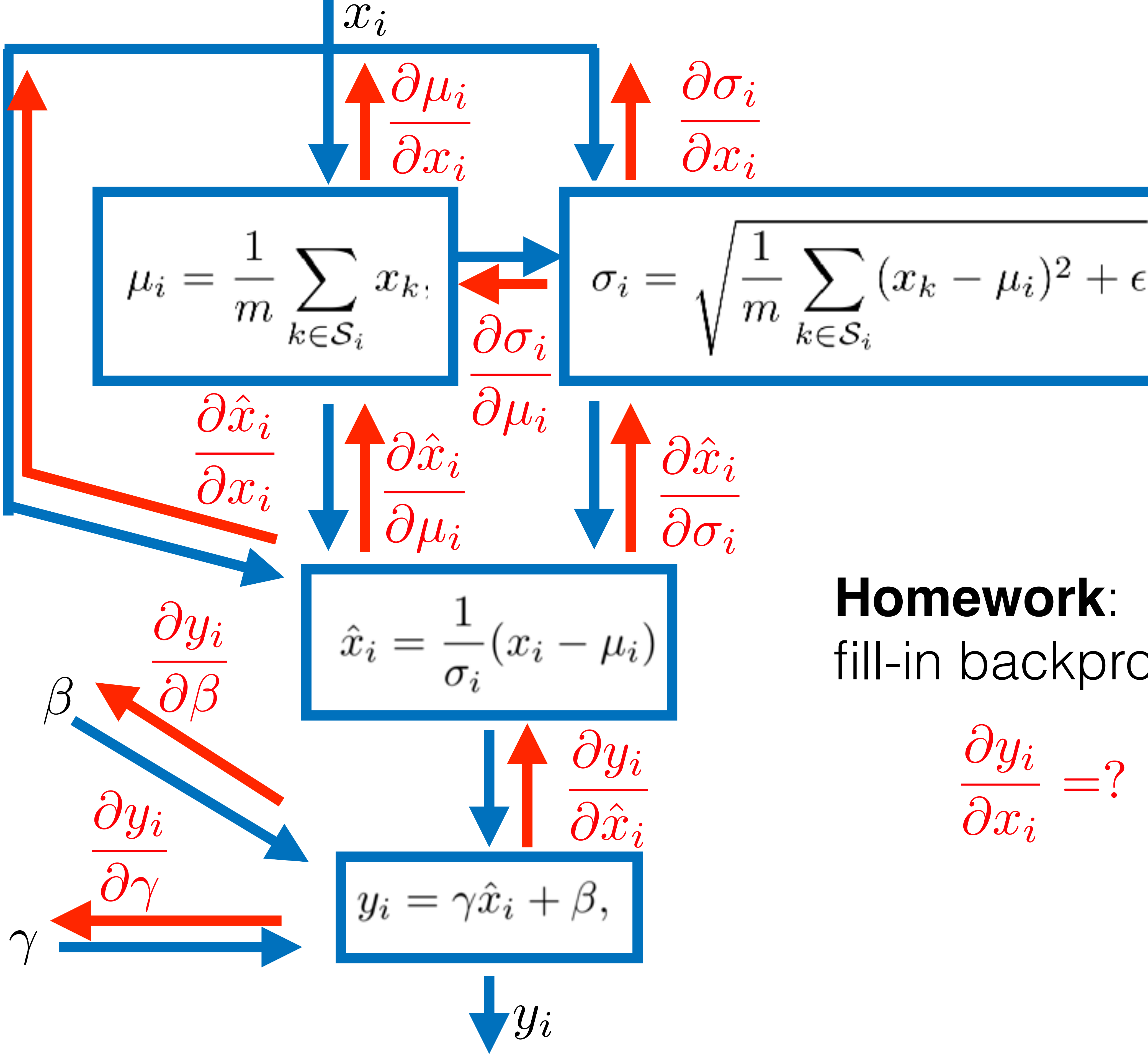
# Batch normalization layer [Ioffe and Szegedy 2015]
https://arxiv.org/pdf/1502.03167.pdf (over 6k citation)

## Bad weight initialization



# Computational graph of BN



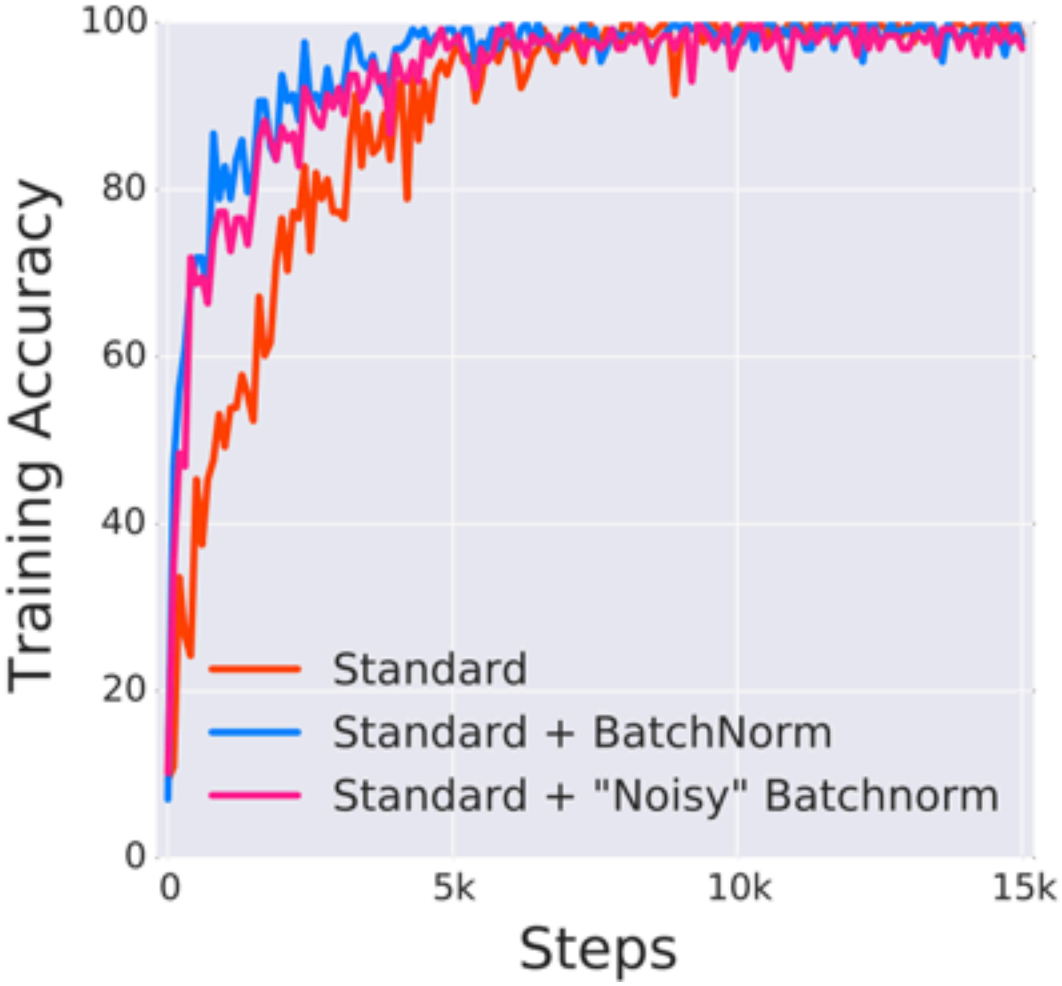



**Homework**:
fill-in backprop of BN

$$\frac{\partial y_i}{\partial x_i} = ?$$

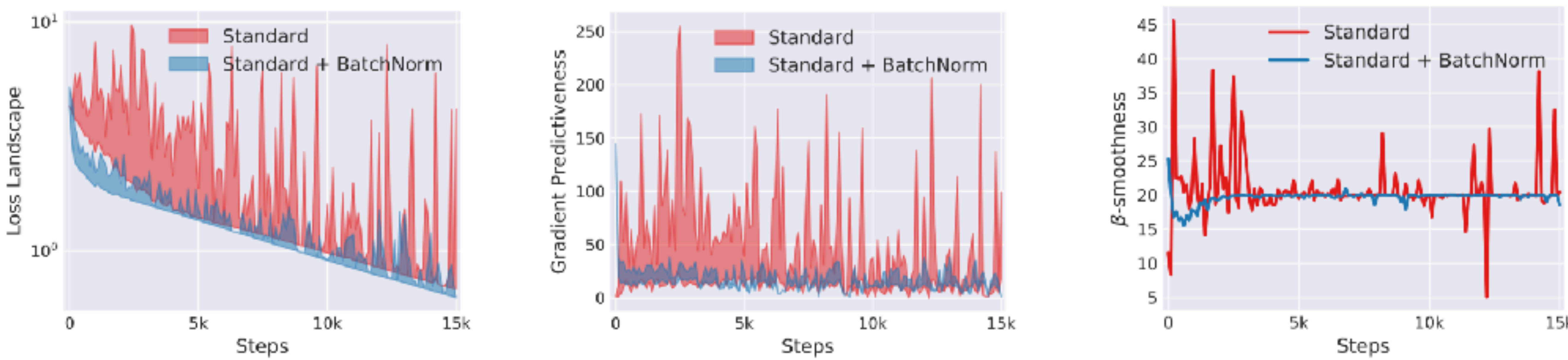
# Why batch normalization helps?? https://arxiv.org/pdf/1805.11604.pdf

- **Covariate shift:** change in the distribution the input values during testing
- Original explanation: BN reduces covariance shift
- Experiment with injected noisy covariance shift reveals, that this is not the issue.



# Why batch normalization helps??
https://arxiv.org/pdf/1805.11604.pdf
[Santurkar, NIPS, 2019]

- They show that BN improves beta-smoothness (i.e. Lipschitzness in loss and gradient) and predictivness.




(a) loss landscape (b) gradient predictiveness (c) "effective" $\beta$-smoothness

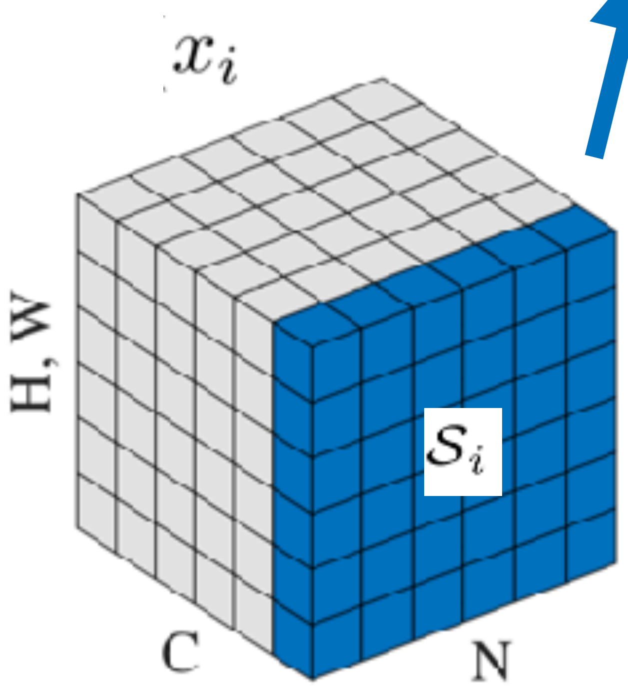
# Batch Normalization - conclusions

- **Forward pass** (no mini-batch available):
- The same, but $\mu_i = \mathbb{E}[x_i]$ and $\sigma_i = \mathbb{E}[(x_i - \mathbb{E}[x_i])^2]$ estimated over the whole training set.
- **suffers from training/testing discrepancy.**
- **BN is reparametrization** of the original NN with the same expressive power.
- **BN is model regularizer:** one training example always normalized differently => small jittering
- **Works well on classification** problems, the reason is partially unclear (beta-smoothness or covariate shift).
- **Not suitable for recurrent networks.** Different BN for each time-stamp => need to store statistics for each time-stamp.
- **Does not work on generative netoworks.** The reason is unclear.

## **Batch normalization layer** [Ioffe and Szegedy 2015]

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

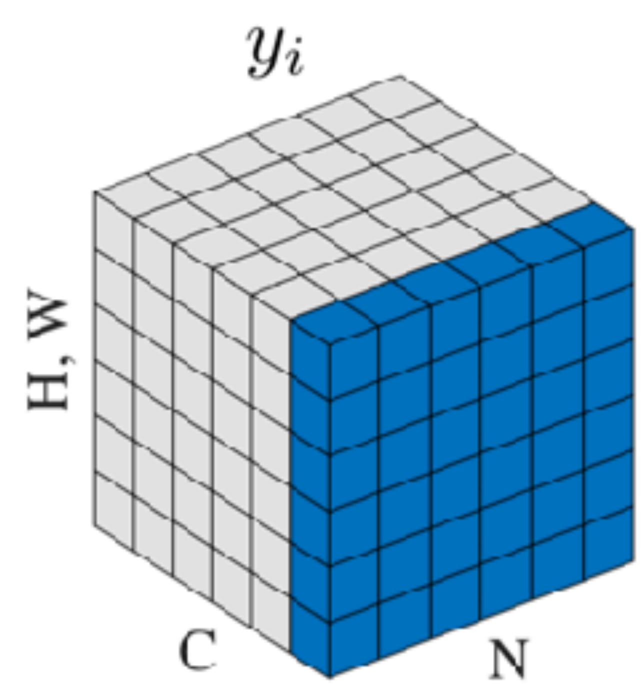
$$\mu_i = \frac{1}{m} \sum_{k \in \mathcal{S}_i} x_k, \quad \sigma_i = \sqrt{\frac{1}{m} \sum_{k \in \mathcal{S}_i} (x_k - \mu_i)^2 + \epsilon},$$




$$\hat{x}_i = \frac{1}{\sigma_i}(x_i - \mu_i)$$

$$y_i = \gamma \hat{x}_i + \beta,$$

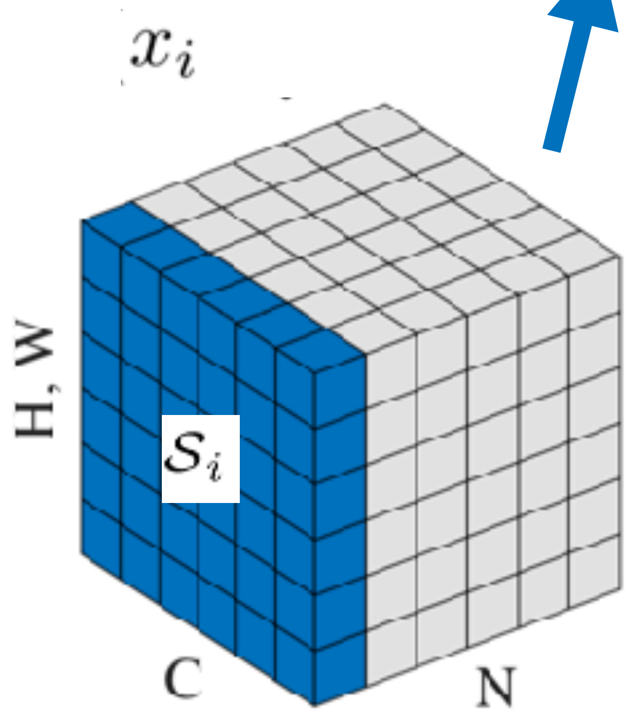
$$\mathcal{S}_i = \{k \mid k_C = i_C\}$$



# **Layer normalization** [Ba, Kiros, Hinton 2016]

https://arxiv.org/pdf/1607.06450.pdf

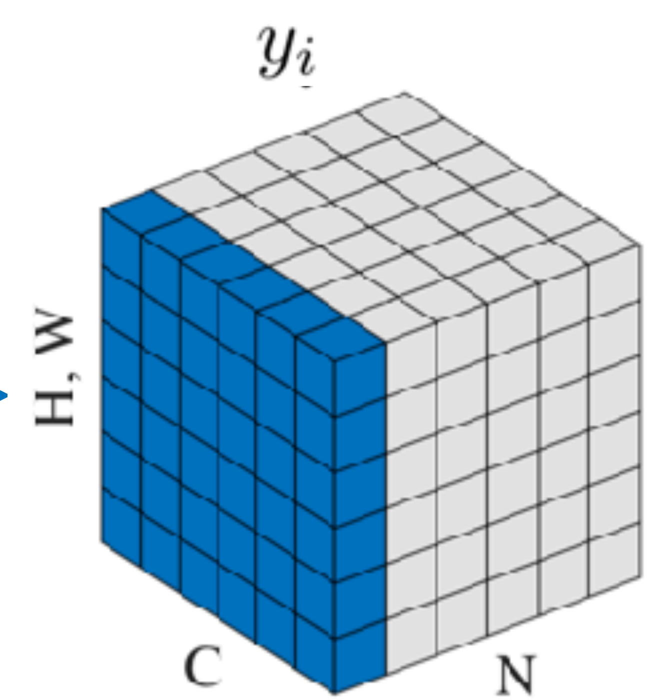
$$\mu_i = \frac{1}{m} \sum_{k \in \mathcal{S}_i} x_k, \quad \sigma_i = \sqrt{\frac{1}{m} \sum_{k \in \mathcal{S}_i} (x_k - \mu_i)^2 + \epsilon},$$




$$\hat{x}_i = \frac{1}{\sigma_i}(x_i - \mu_i)$$

$$\mathcal{S}_i = \{k \mid k_N = i_N\}$$

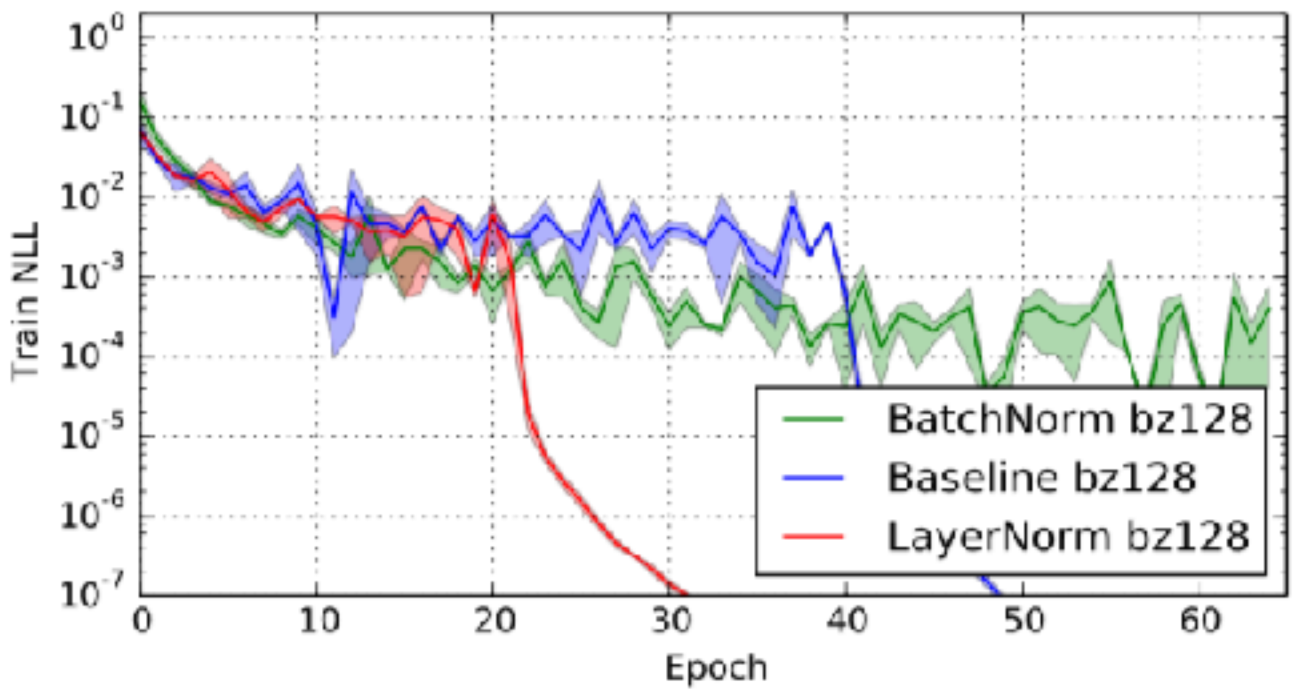
$$y_i = \gamma \hat{x}_i + \beta,$$

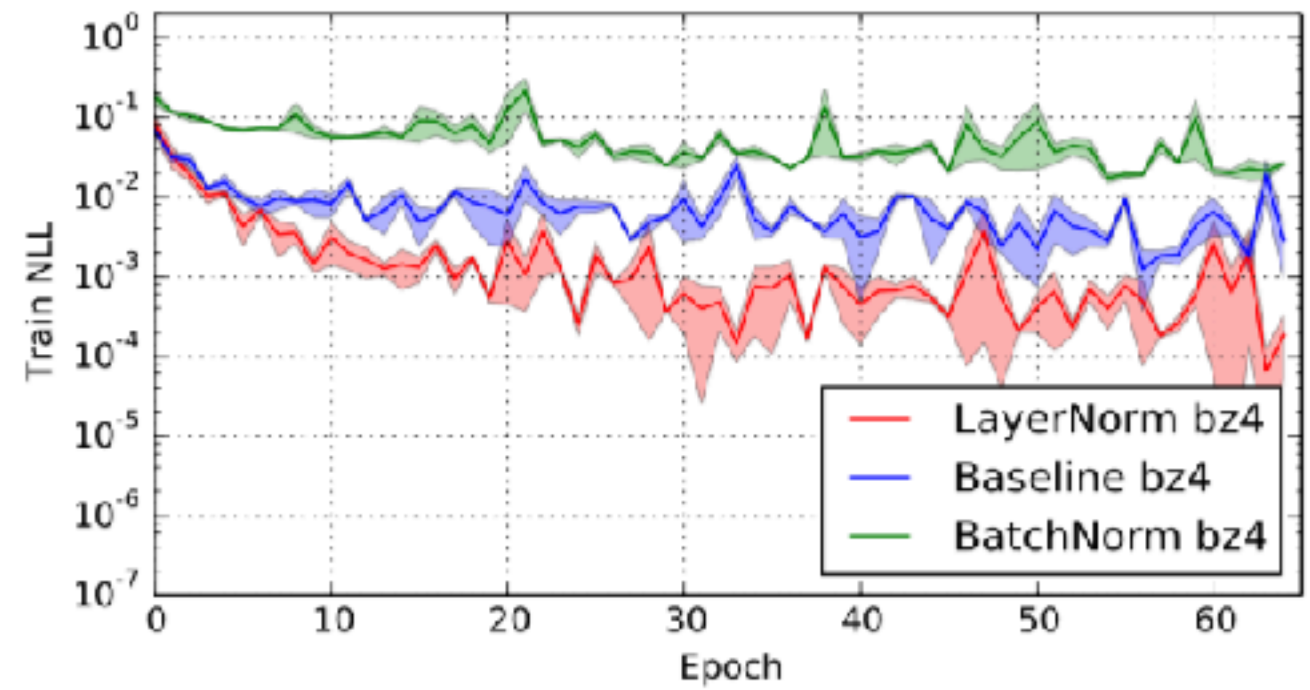


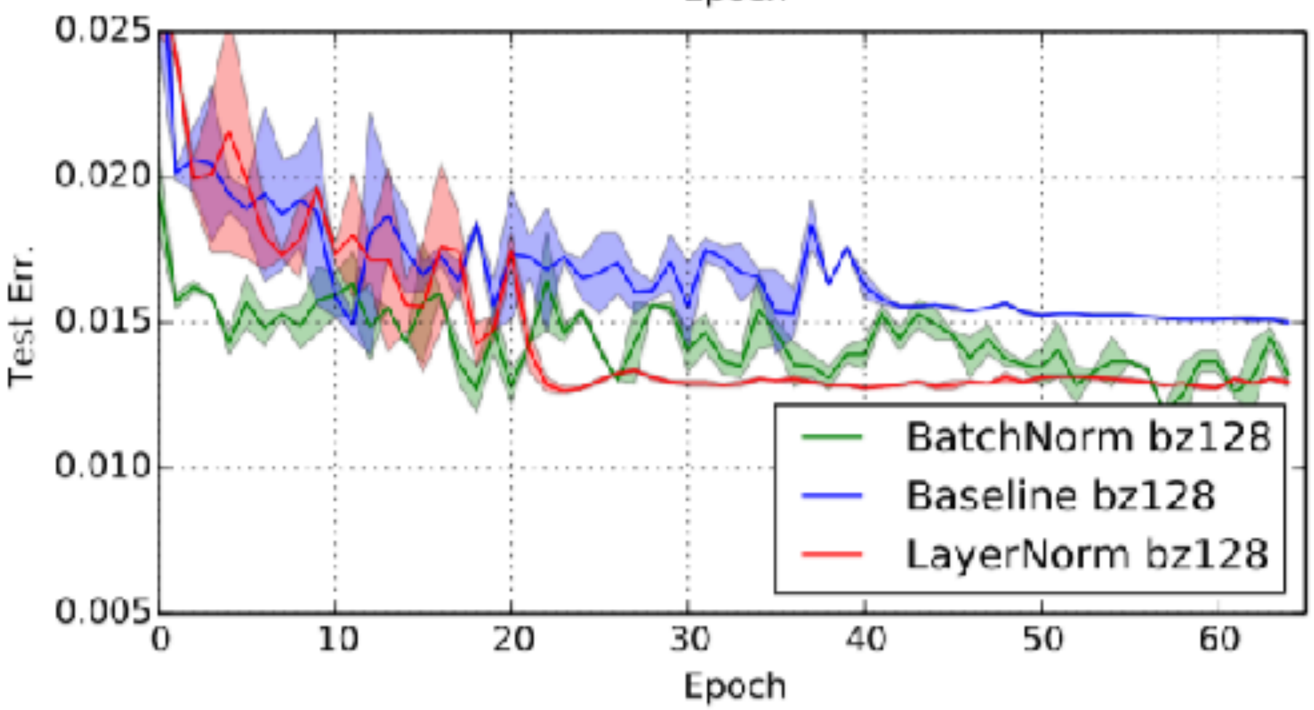

Layer normalization performs well on RNN

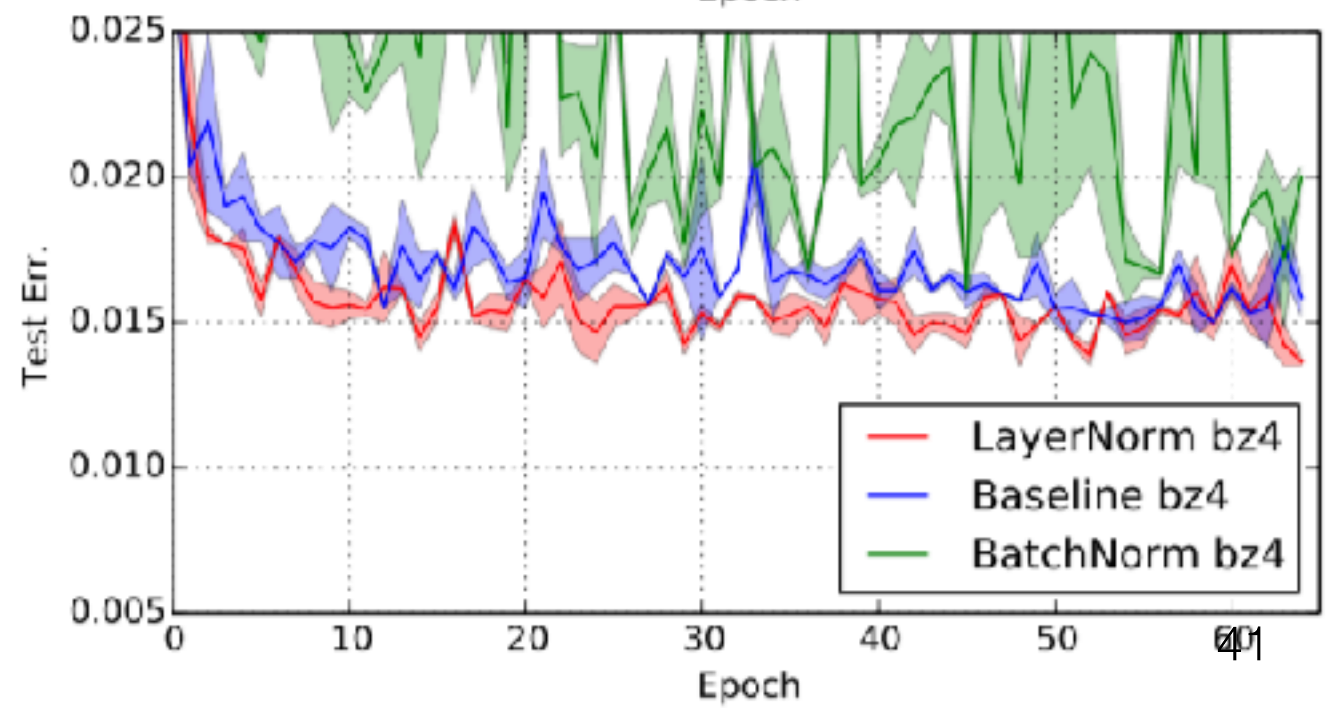
# Layer Normalization - conclusions

- **Forward pass** (no mini-batch needed):
- => no trainin./testing dicrepancy as with BN.
- **Work well on recurrent networks.**
- **Work well for small mini-batches**



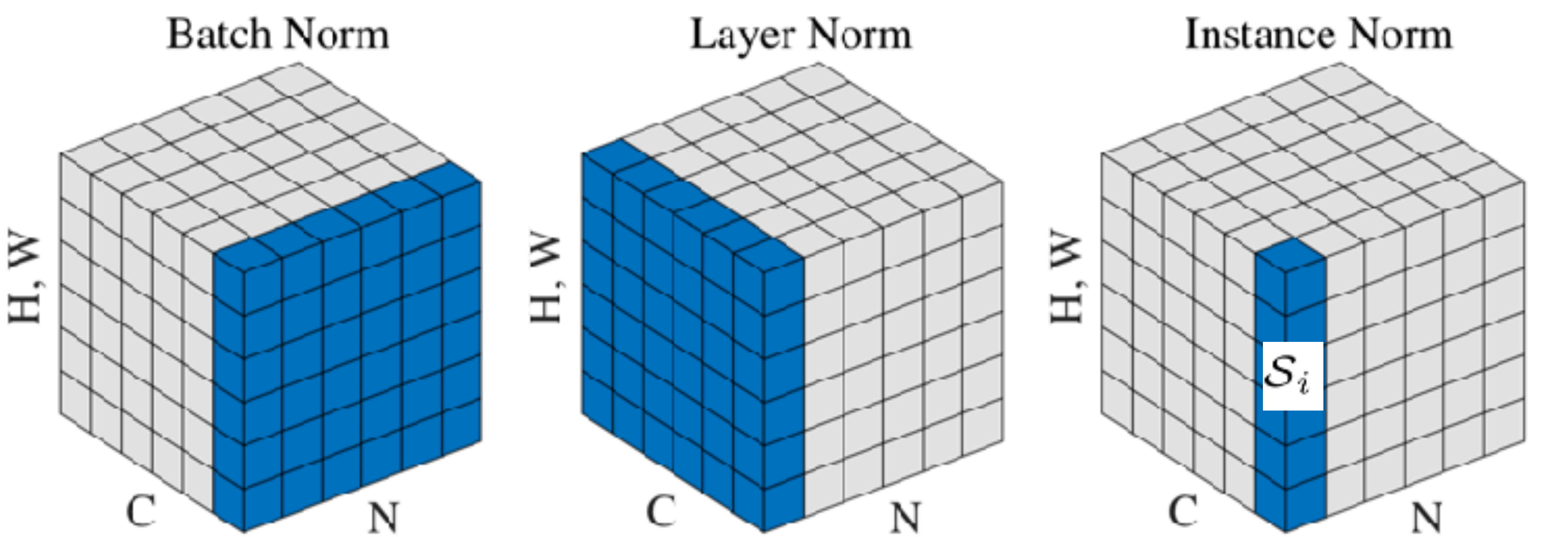



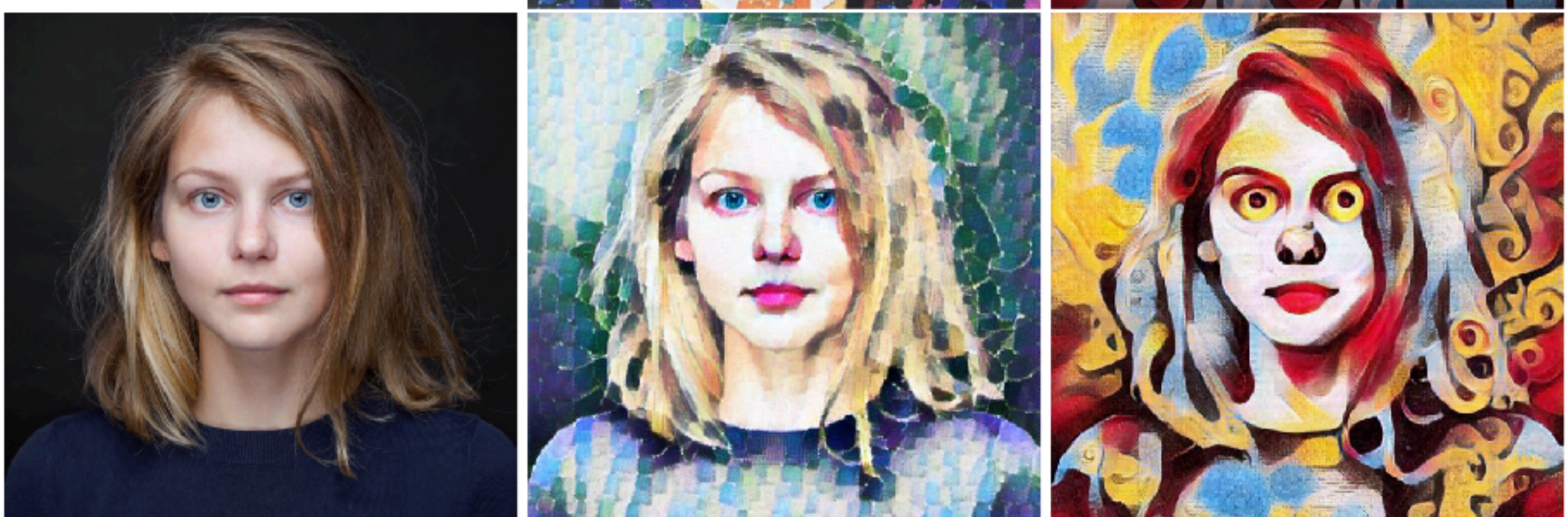
# Instance normalization [Ulyanov, Vidaldi, Lempitsky 2017]
https://arxiv.org/pdf/1607.08022.pdf

$$\mathcal{S}_i = \{k \mid k_C = i_C, k_N = i_N\}$$



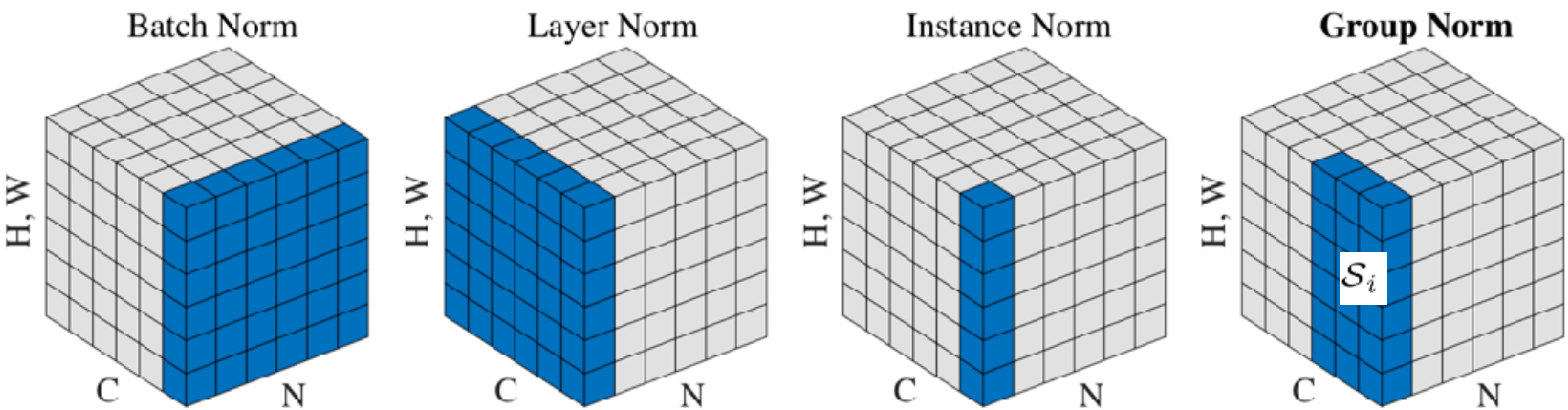
# Instance Normalization - conclusions

- Idea: network should be insensitive to constrast
- Works well on style transfer and GAN networks
- It does not outperform BN on image classification tasks



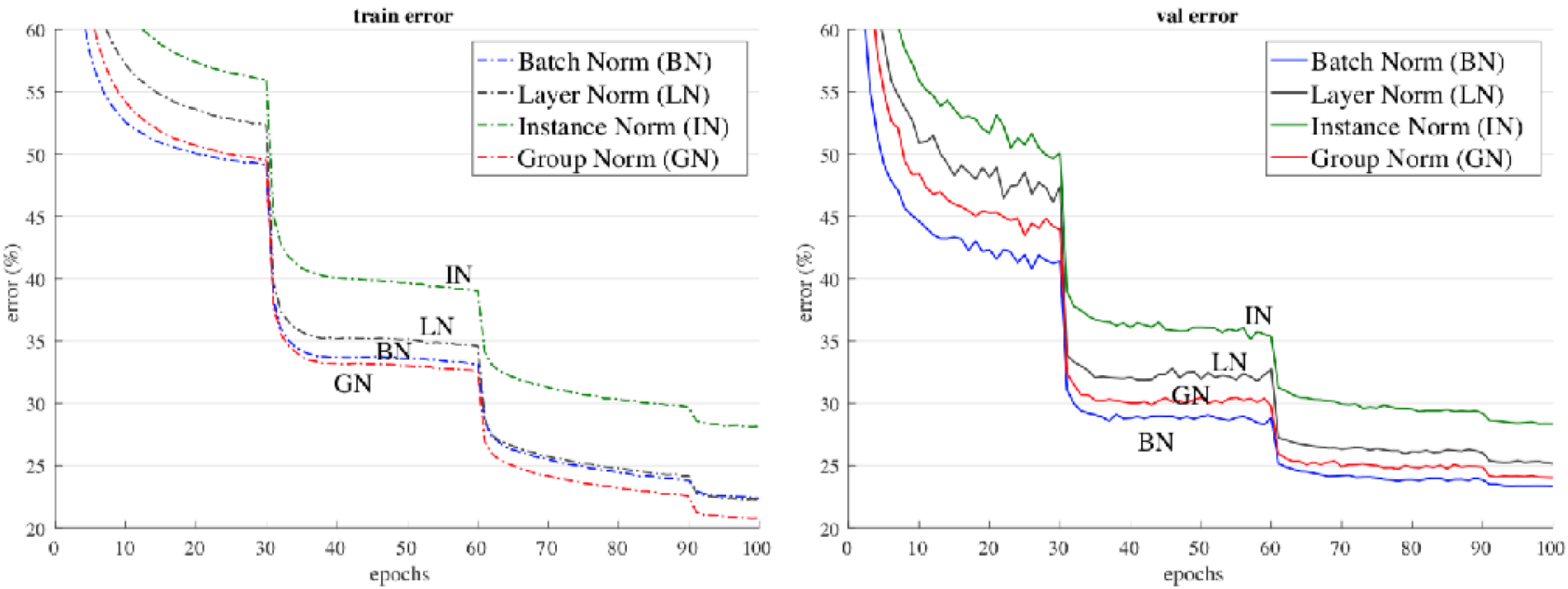
# Group normalization [Wu, He, 2018]
https://arxiv.org/pdf/1803.08494.pdf

Group normalization performs well for style transfer (GANs) and RNN but does not outperform BN for image classification



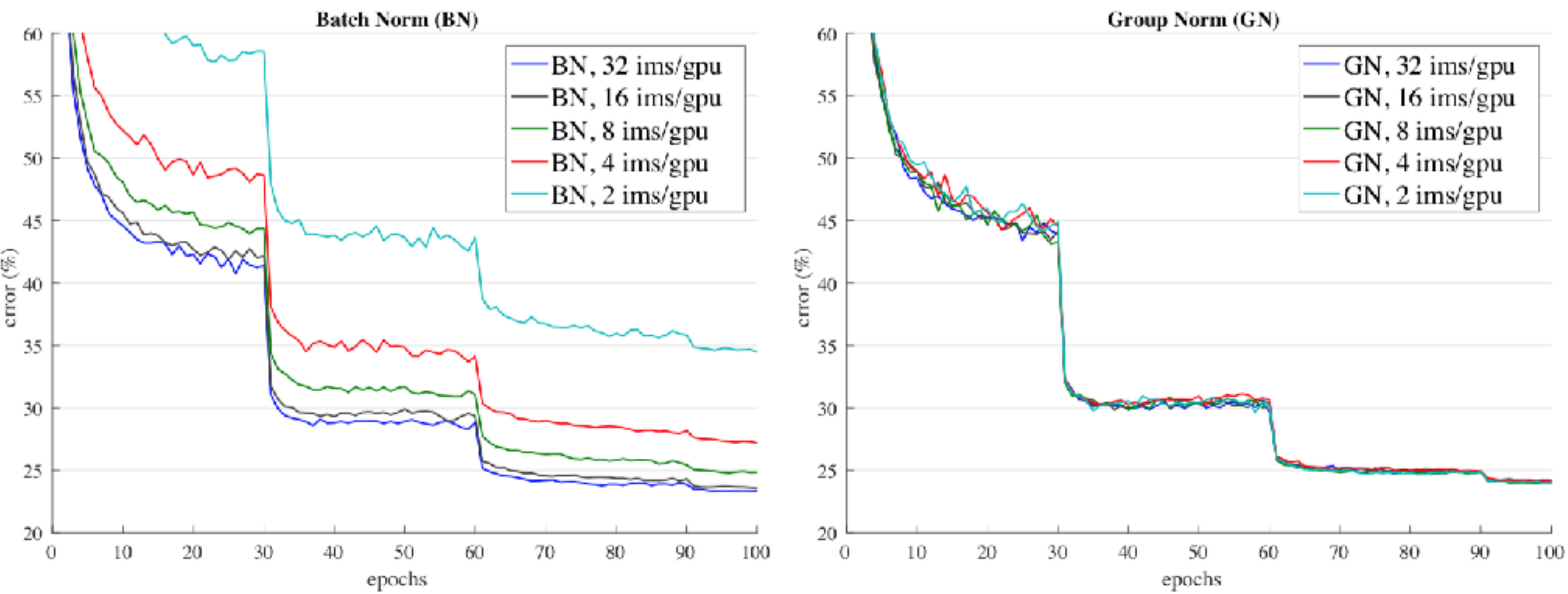
# Group Normalization - conclusions

- It achieves performance comparable with BN on image classification tasks (for mini-batch 32).



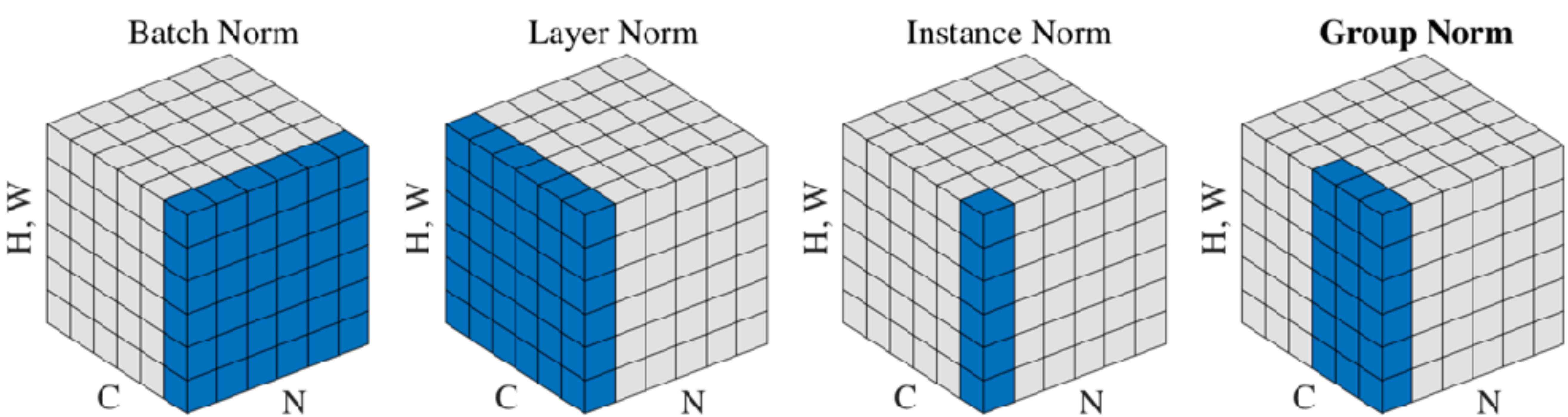
# Group Normalization - conclusions

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



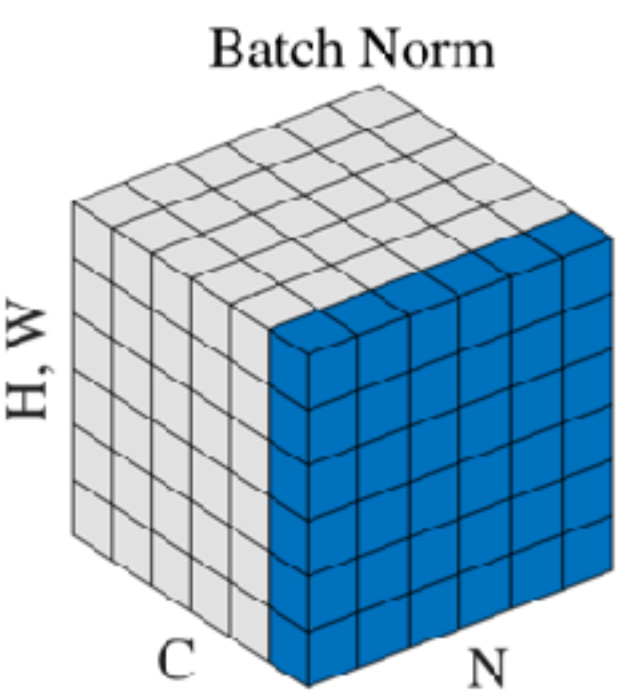
# Group Normalization - conclusions

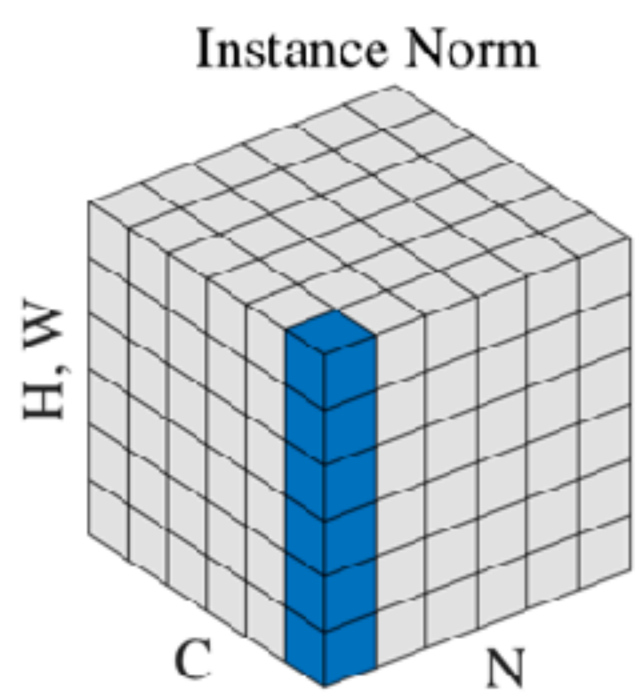
- Why GN works better?
- LN makes implicit assumption that all channels are of the same importance when computing the mean.
- This does not have to be right => GN allows to compute different statistics for different groups of channels => larger flexibility.



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

- BN good for classification, IN good for style transfer
- Idea is to combine both.



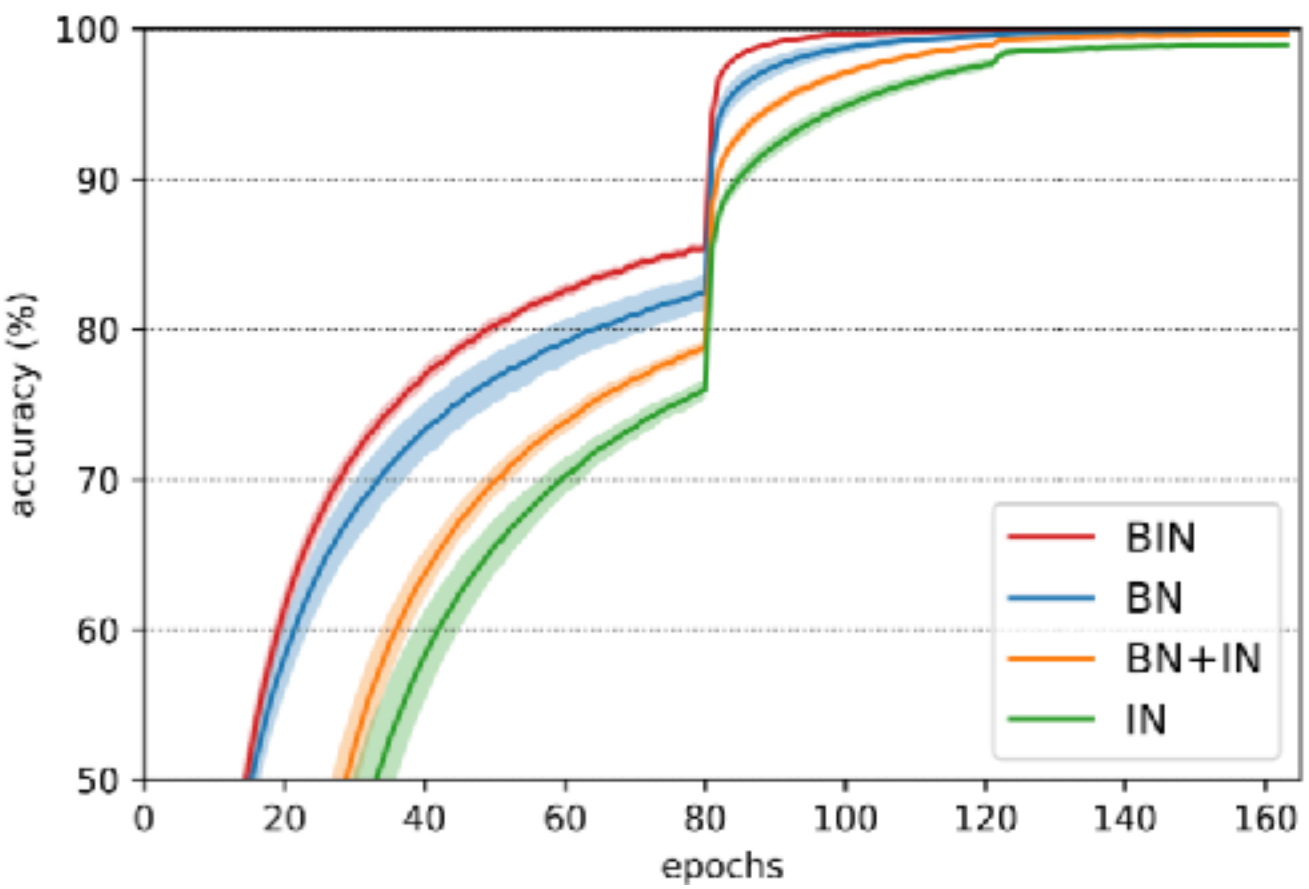
# Batch-Instance normalization

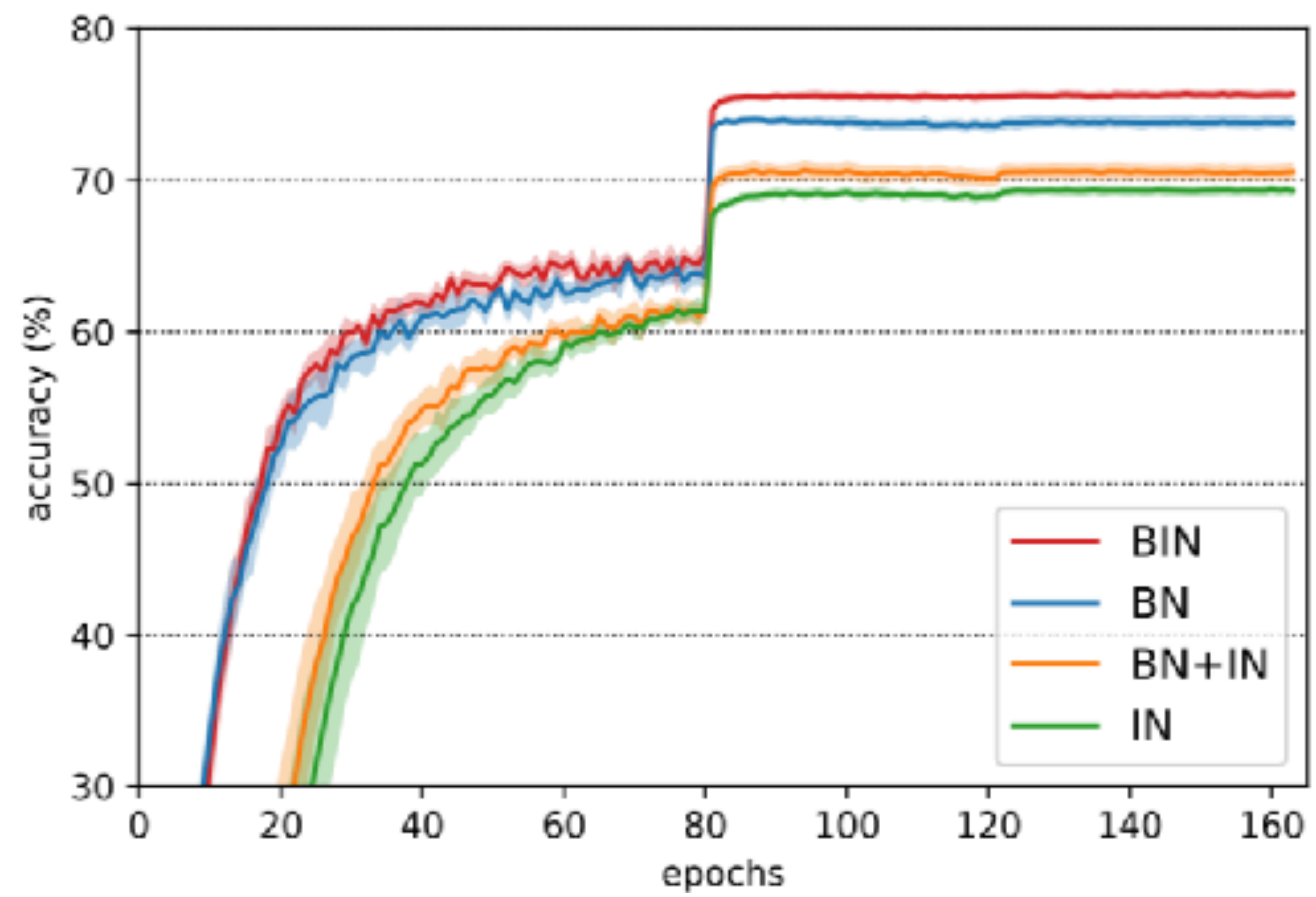
https://arxiv.org/pdf/1805.07925.pdf

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

- BIN combines BN and IN
- Three trainable parameters
- Suitable for both style transfer and classification

**Classification results**: ResNet-101 on CIFAR-100



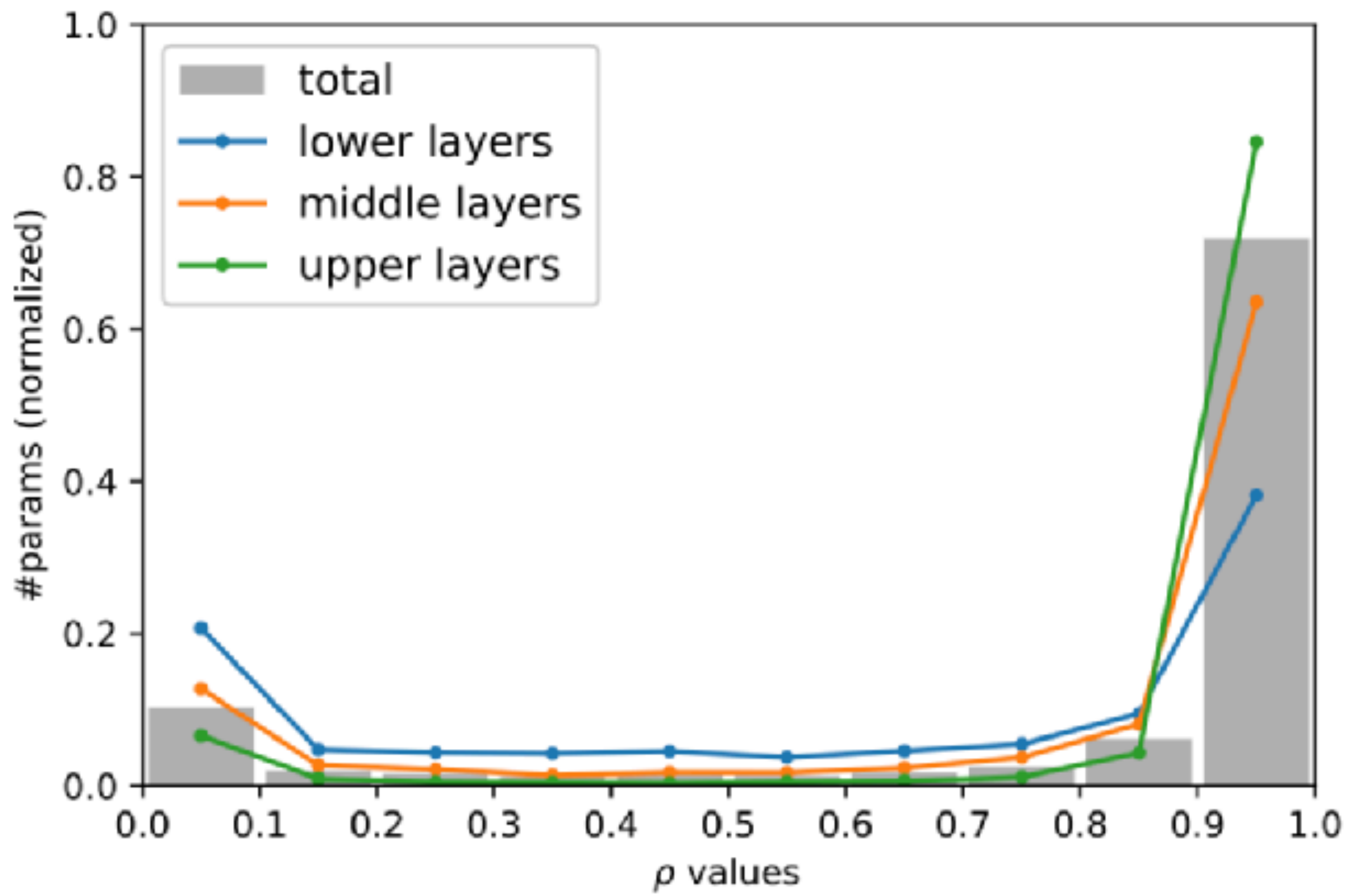
# Batch-Instance normalization

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 combines BN a IN
- Three trainable parameters
- Suitable for both style transfer and classification

**Classification results**: ResNet-101 on CIFAR-100

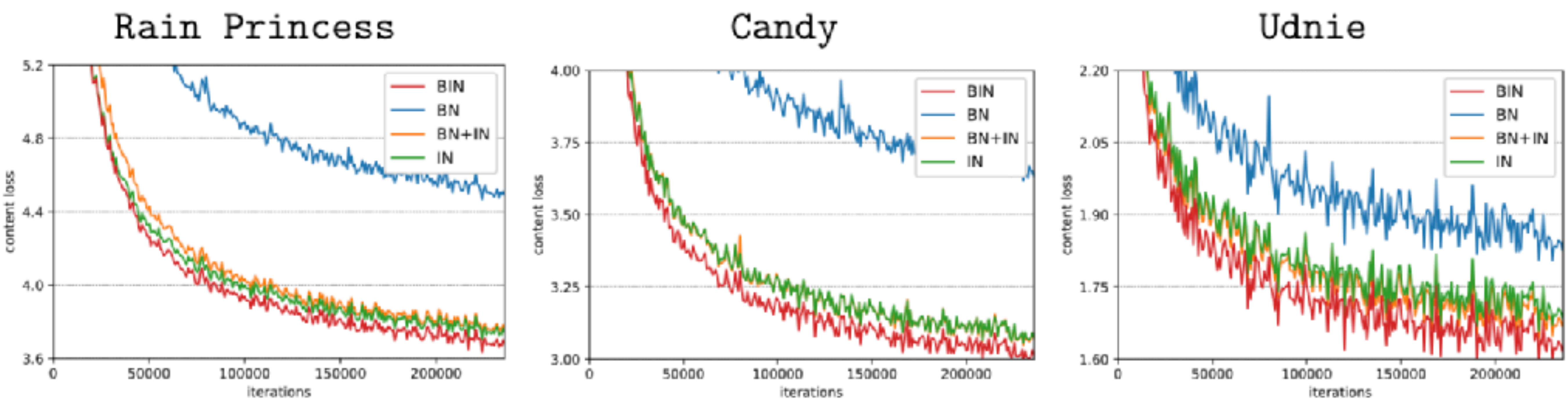


# Batch-Instance normalization
https://arxiv.org/pdf/1805.07925.pdf

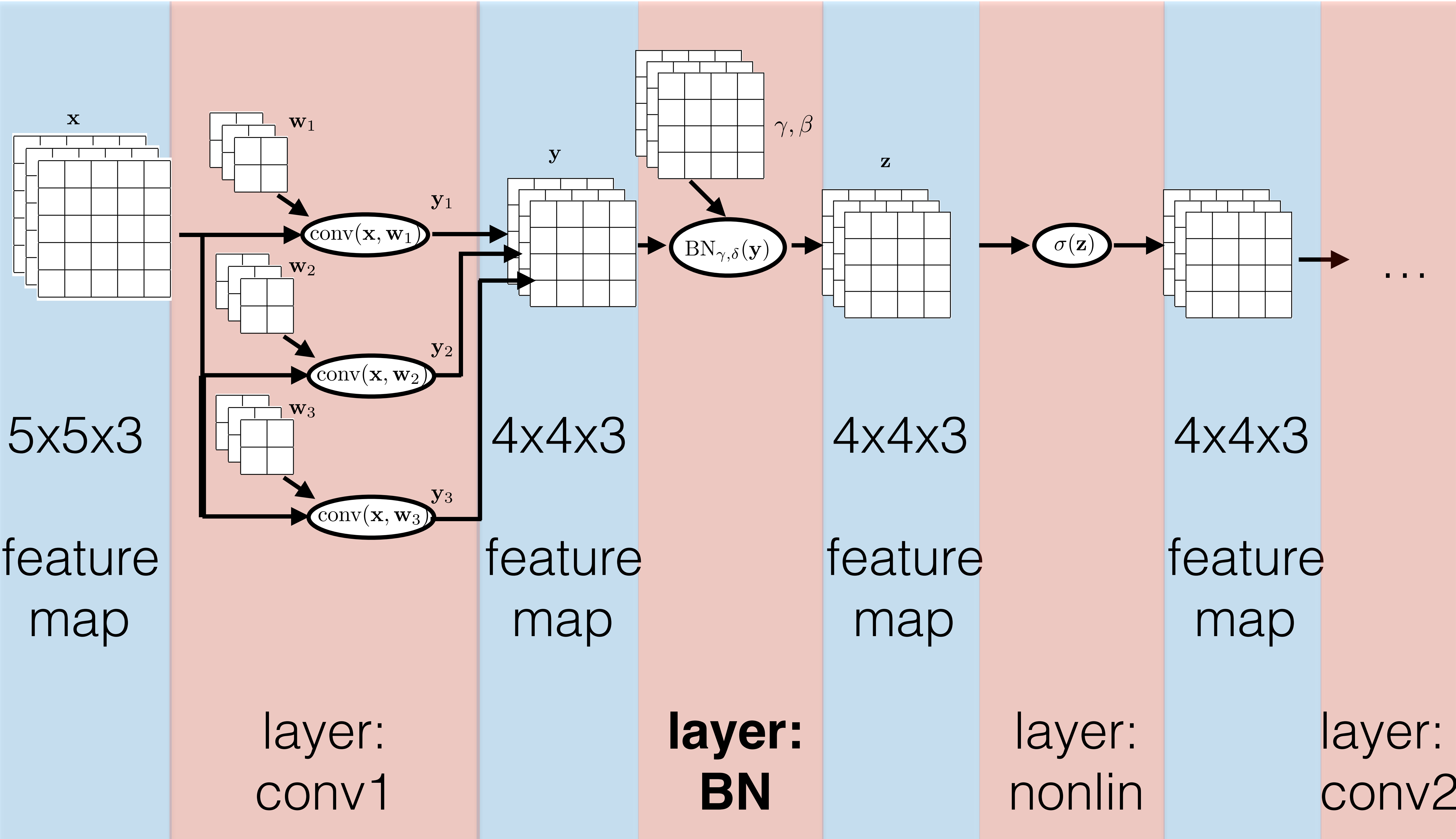
$$y = \left(\rho \cdot \hat{x}^{(BN)} + (1-\rho) \cdot \hat{x}^{(IN)}\right) \cdot \gamma + \beta$$

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

**Style trasfer results**: ResNet-101 on CIFAR-100



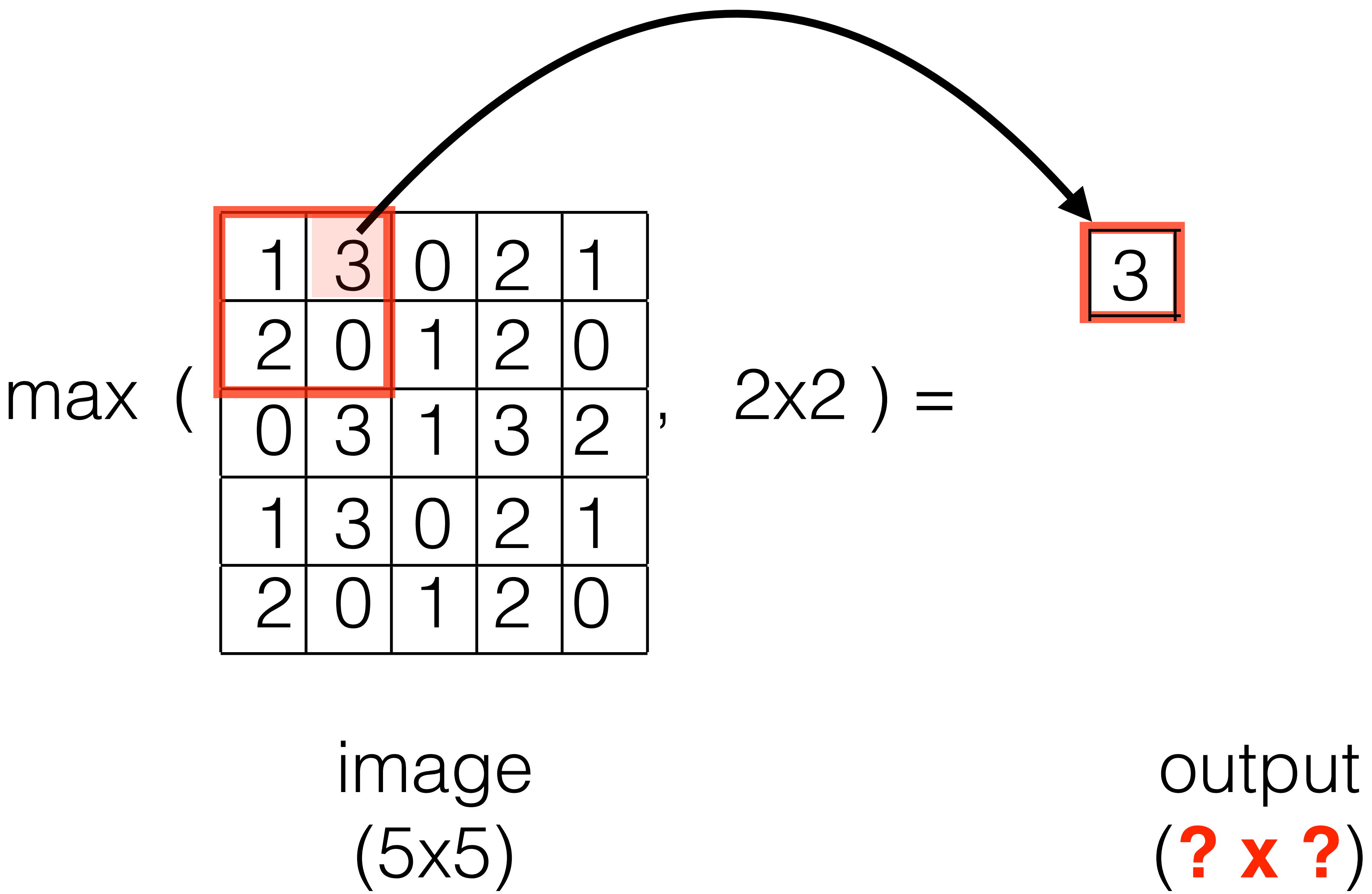
# Batch normalization layer [Ioffe and Szegedy 2015]
https://arxiv.org/pdf/1502.03167.pdf (over 6k citation)



# Outline

- SGD vs deterministic gradient
- what makes learning to fail
- layers:
  - activation function (i.e. non-linearities)
  - batch normalization layer
  - max-pooling layer
  - loss-layers
- summary of the learning procedure
  - train, test, val data,
  - hyper-parameters,
  - regularizations

# Max-pooling



# Max-pooling

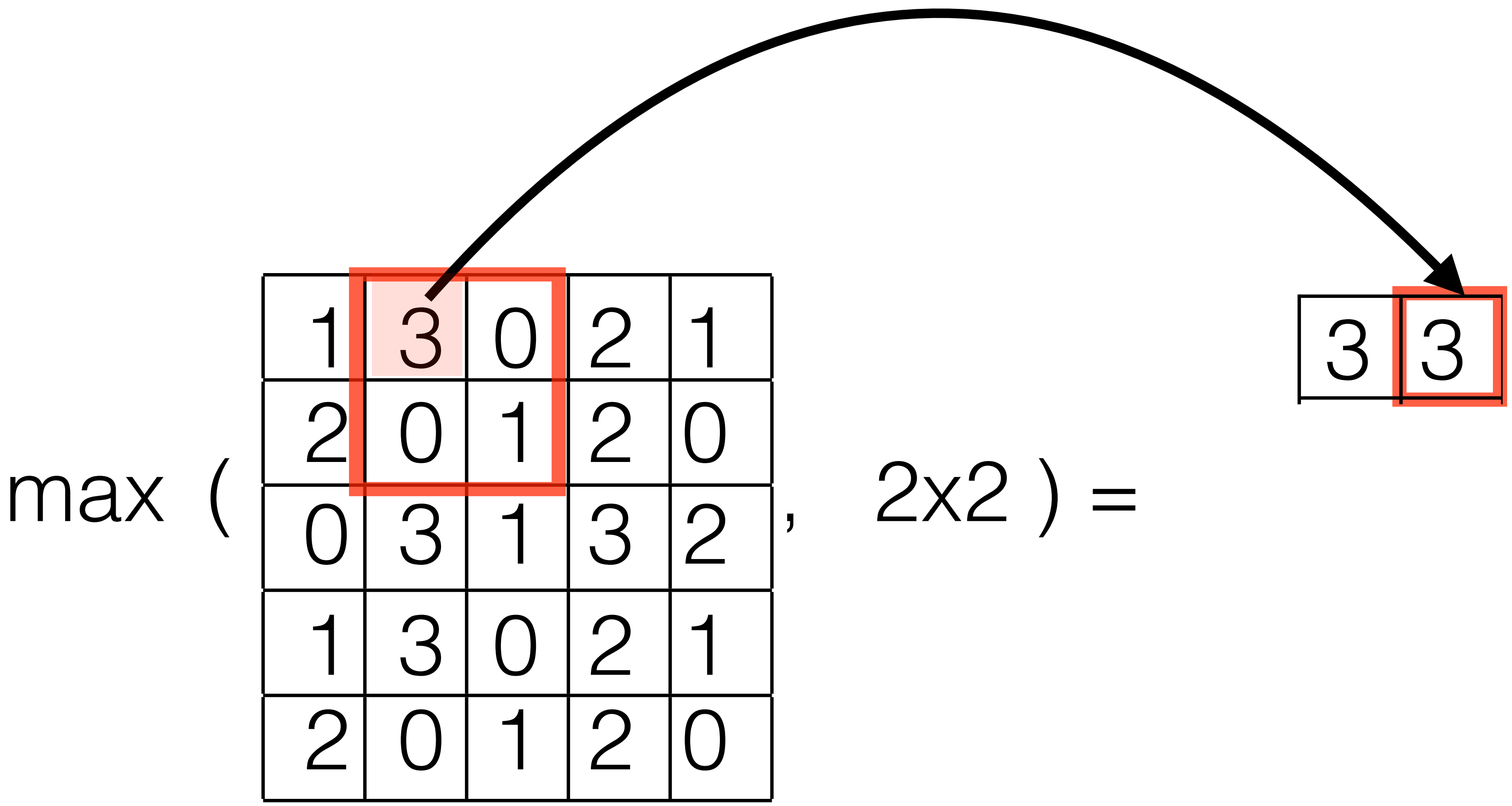



image
(5x5)

output
(**? x ?**)

# Max-pooling

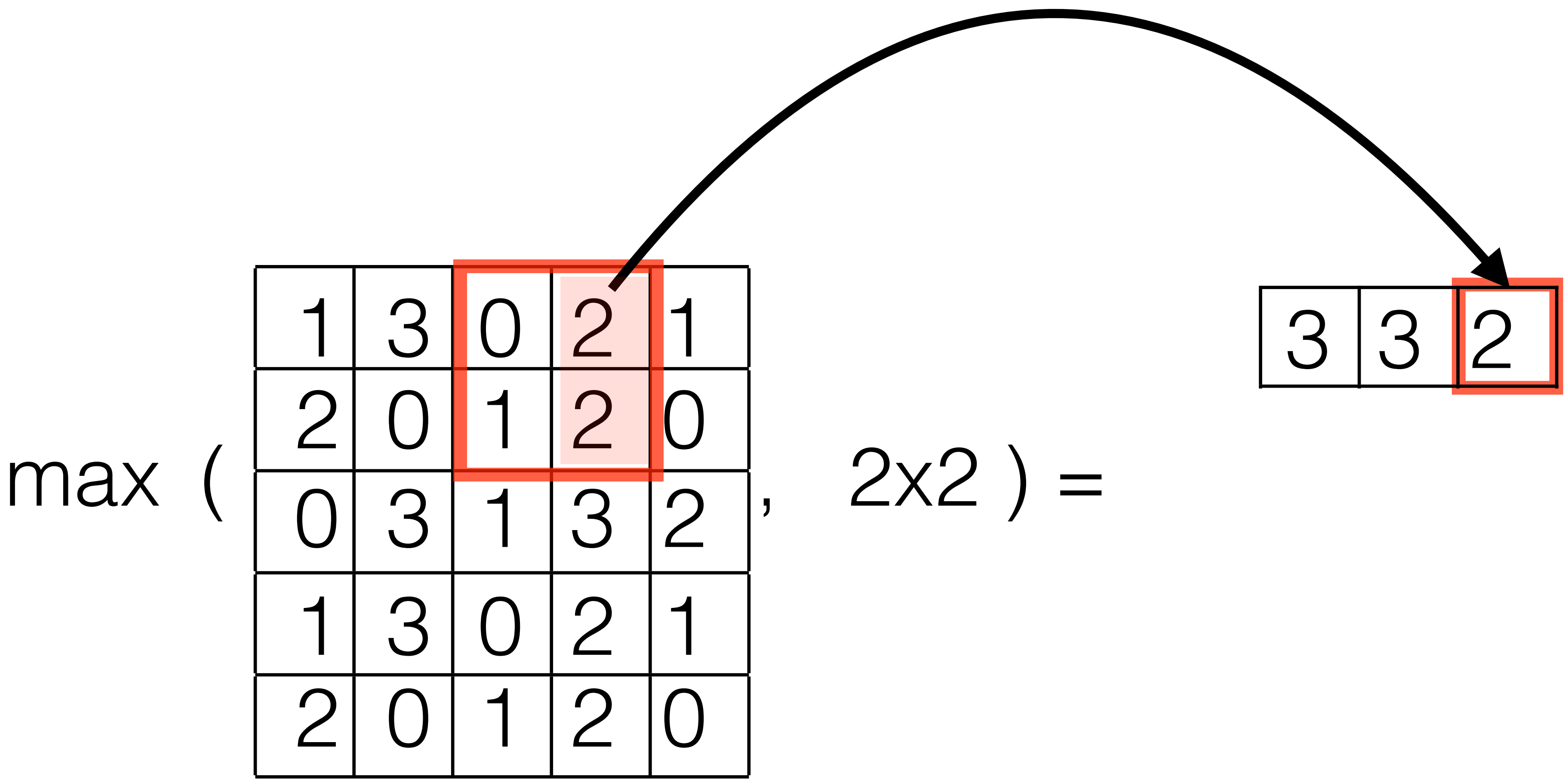



image
(5x5)

output
(**? x ?**)

# Max-pooling

max ( image, 2x2 ) = output

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 0 | 2 | 1 |
| 2 | 0 | 1 | 2 | 0 |
| 0 | 3 | 1 | 3 | 2 |
| 1 | 3 | 0 | 2 | 1 |
| 2 | 0 | 1 | 2 | 0 |

image
(5x5)

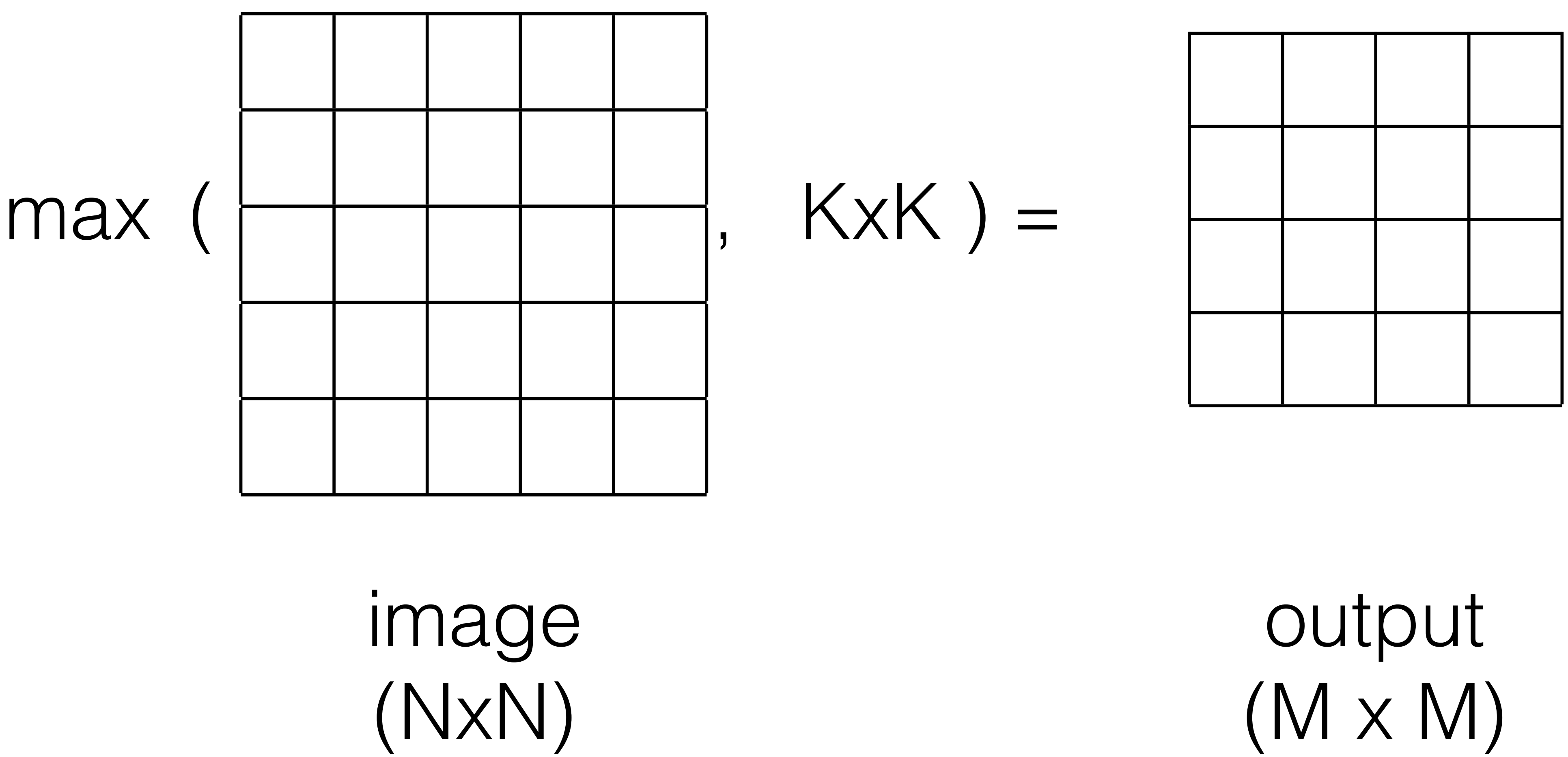
| | | | |
|---|---|---|---|
| 3 | 3 | 2 | 2 |
| 3 | 3 | 3 | 3 |
| 3 | 3 | 3 | 3 |
| 3 | 3 | 2 | 2 |

output
(**4 x 4**)

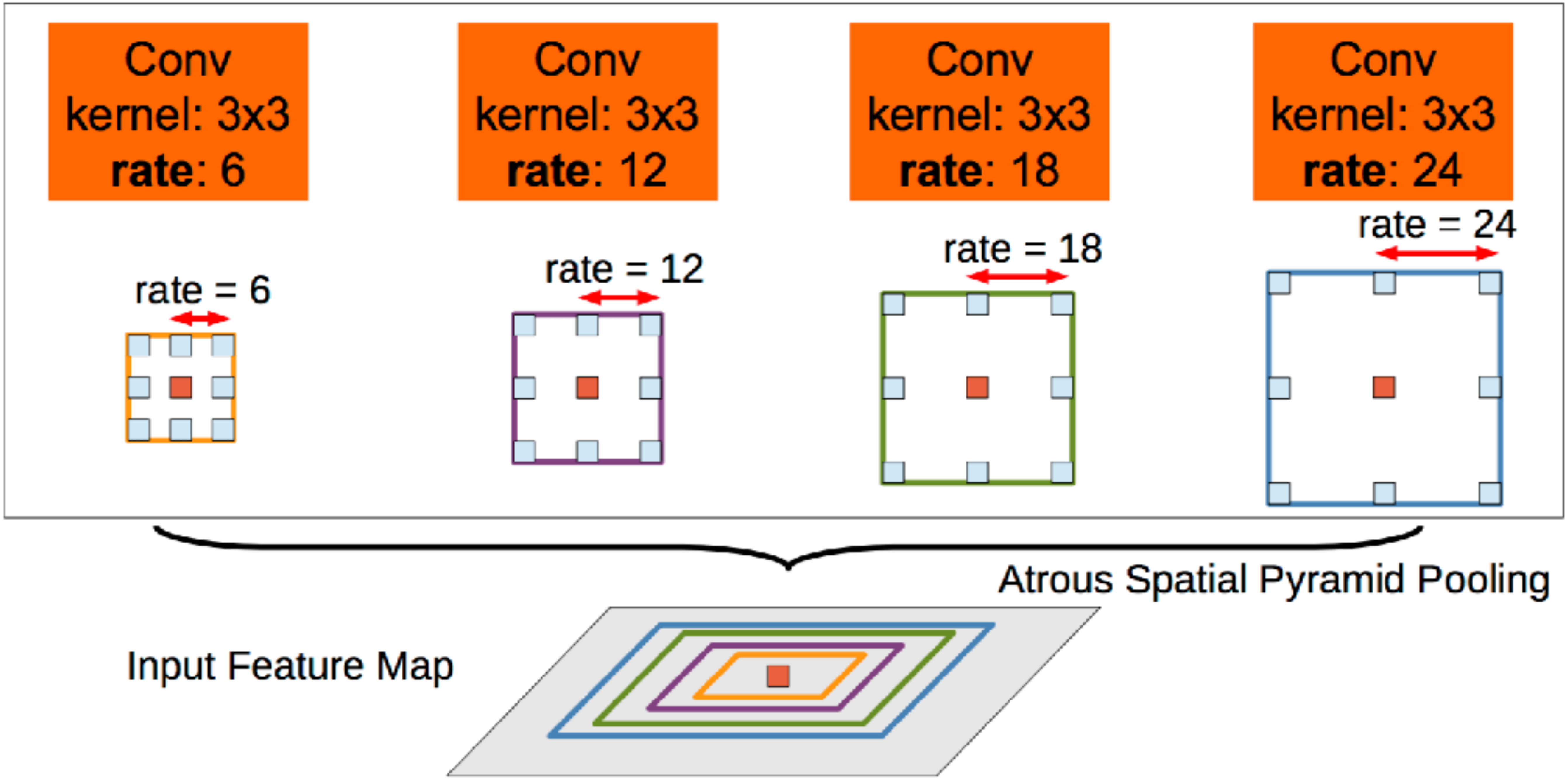
# Max-pooling

M = (N+2*pad-K) / stride + 1

The same as for convolution



# Atrous Spatial Pyramid Pooling (ASPP)




[Chen et al. TPAMI 2018] https://arxiv.org/pdf/1606.00915.pdf

# Max-pooling feed-forward

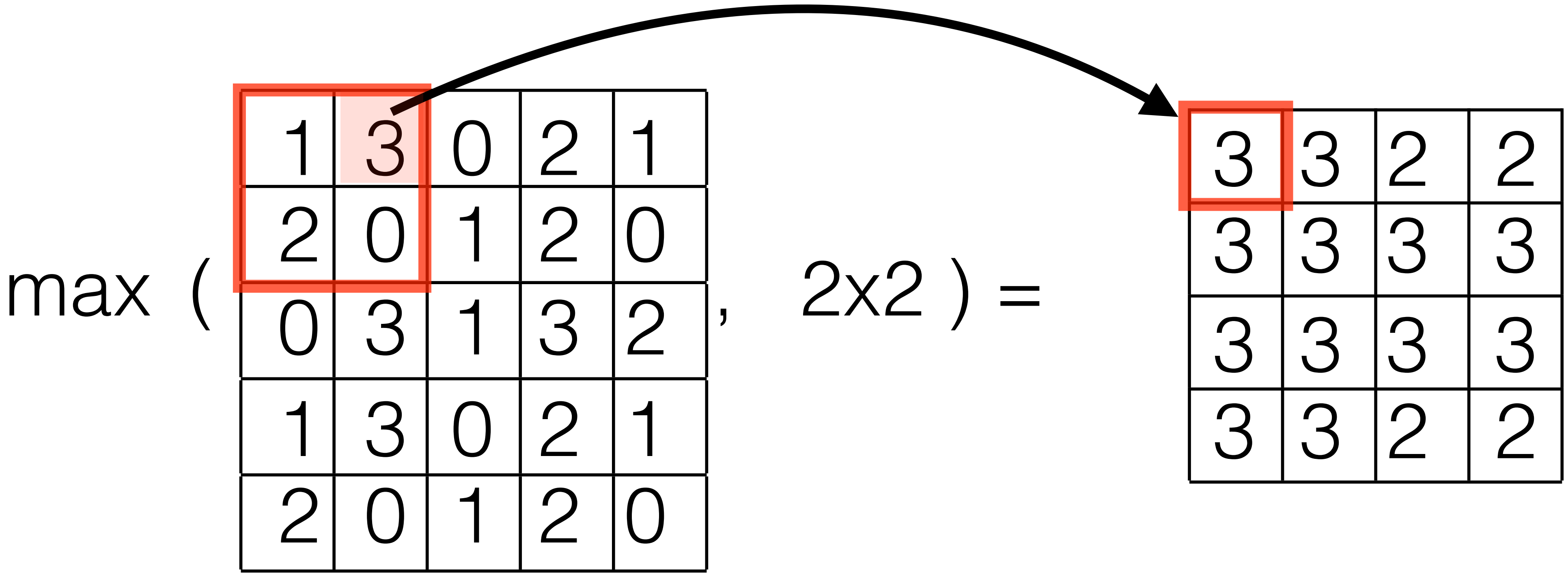
$$\max\left(\begin{bmatrix}1&3&0&2&1\\2&0&1&2&0\\0&3&1&3&2\\1&3&0&2&1\\2&0&1&2&0\end{bmatrix},\ 2x2\right)=\begin{bmatrix}3&3&2&2\\3&3&3&3\\3&3&3&3\\3&3&2&2\end{bmatrix}$$

# Max-pooling Backprop

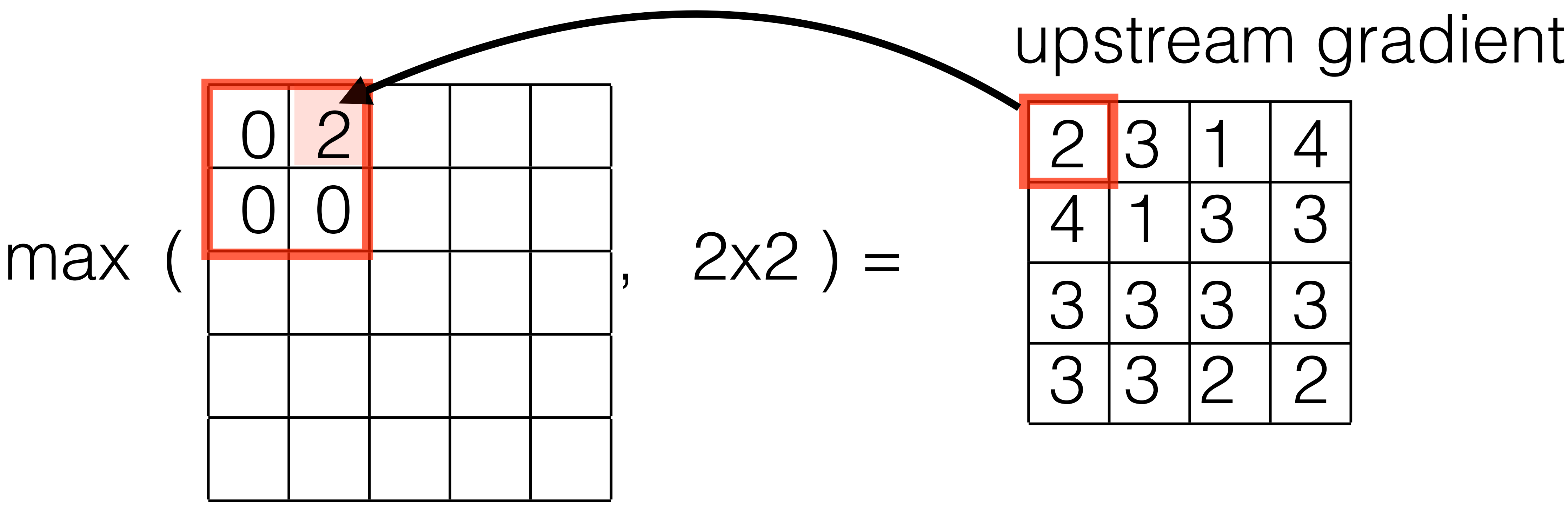
upstream gradient

$$\max\left(\begin{bmatrix}?&?&&&\\?&?&&&\\&&&&\\&&&&\\&&&&\end{bmatrix},\ 2x2\right)=\begin{bmatrix}2&3&1&4\\4&1&3&3\\3&3&3&3\\3&3&2&2\end{bmatrix}$$

## Max-pooling feed-forward




## Max-pooling Backprop



## Max-pooling feed-forward

max (

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 0 | 2 | 1 |
| 2 | 0 | 1 | 2 | 0 |
| 0 | 3 | 1 | 3 | 2 |
| 1 | 3 | 0 | 2 | 1 |
| 2 | 0 | 1 | 2 | 0 |

, 2x2 ) =

| | | | |
|---|---|---|---|
| 3 | 3 | 2 | 2 |
| 3 | 3 | 3 | 3 |
| 3 | 3 | 3 | 3 |
| 3 | 3 | 2 | 2 |

## Max-pooling Backprop

max (

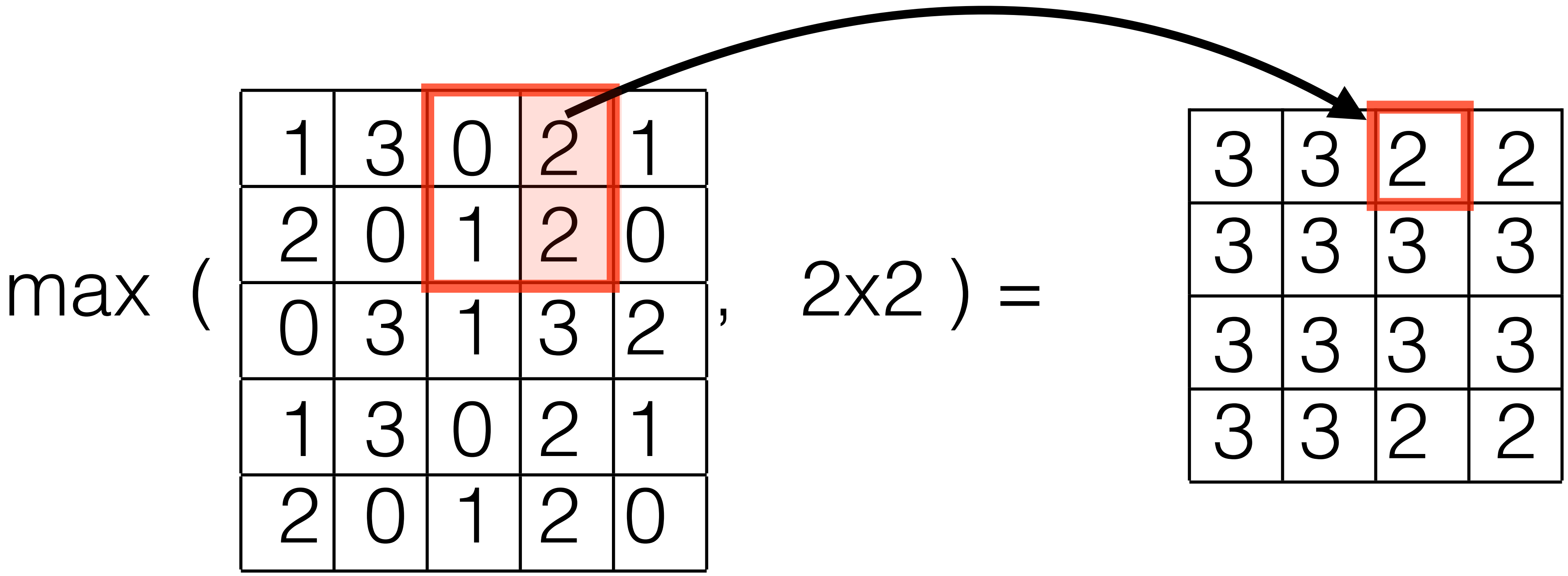
| | | | | |
|---|---|---|---|---|
| 0 | 5 | 0 | | |
| 0 | 0 | 0 | | |
| | | | | |
| | | | | |
| | | | | |

, 2x2 ) =

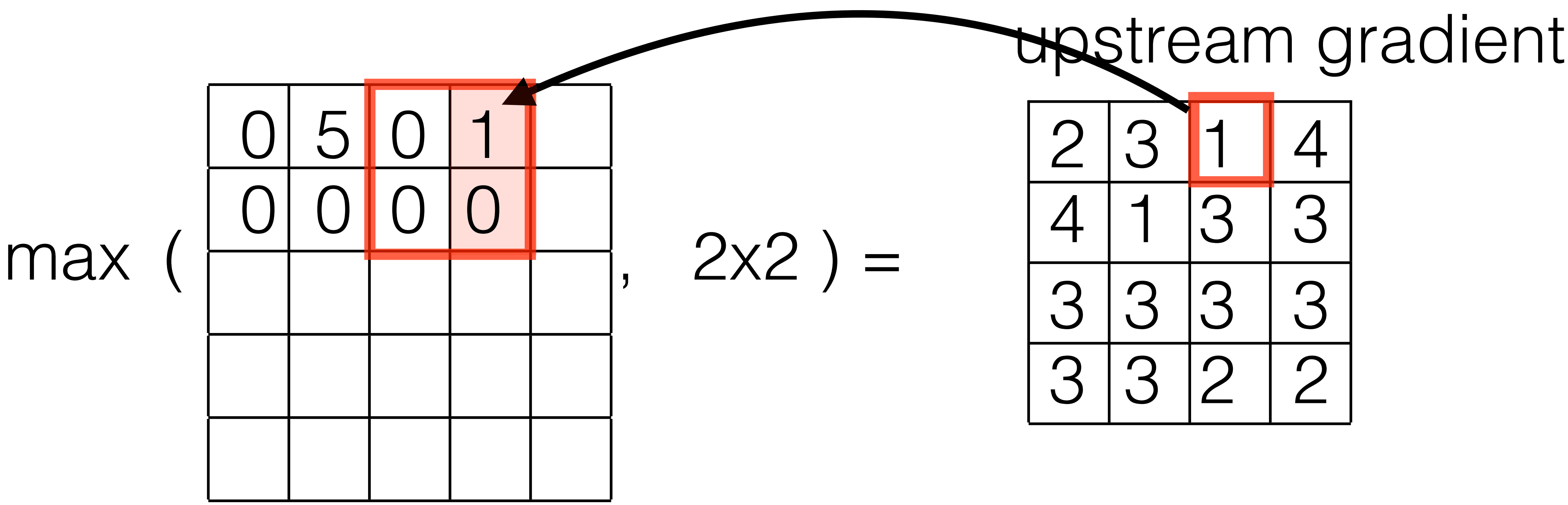
upstream gradient

| | | | |
|---|---|---|---|
| 2 | 3 | 1 | 4 |
| 4 | 1 | 3 | 3 |
| 3 | 3 | 3 | 3 |
| 3 | 3 | 2 | 2 |

# Max-pooling feed-forward




# Max-pooling Backprop



# Max-pooling summary

- Forward pass
  - similar to convolution but takes maximum over kernel
  - it has no parameters to be learnt!
- Backprop
  - propagate gradient only to active connections
- Main purpose is to reduce dimensionality and overfitting
- It seems that max pooling layers will disappear in future
  - should be avoided in generative models (GAN, VAE)
  - they can be replaced by conv-layers with larger stride in discriminative models https://arxiv.org/abs/1412.6806
  - Geoffrey Hinton: "*The pooling operation used in convolutional neural networks is a big mistake and the fact that it works so well is a disaster.*" (Reddit AMA)

# Outline

- SGD vs deterministic gradient
- what makes learning to fail
- layers:
  - activation function (i.e. non-linearities)
  - batch normalization layer
  - max-pooling layer
  - loss-layers
- regularizations
- summary of the learning procedure
  - train, test, val data,
  - hyper-parameters,

# Loss functions

- Regression:
  - L2 loss
  - L1 loss
- Classification:
  - cross entropy loss (N-output classifier $\mathbf{f}(\mathbf{x}, \mathbf{w})$ )
  - logistic loss (single output dichotomy classifier $f(\mathbf{x}, \mathbf{w})$ )

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

PyTorch: `nn.MSELoss()`

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

PyTorch: `nn.L1Loss()`

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

PyTorch: `nn.SmoothL1Loss()`

# Loss functions

- Regression:
  - L2 loss
  - L1 loss
- Classification:
  - cross entropy loss (N-output classifier $\mathbf{f}(\mathbf{x}, \mathbf{w})$ )
  - logistic loss (single output dichotomy classifier $f(\mathbf{x}, \mathbf{w})$ )

(1) convert output to probability (softmax function)

$$\mathbf{s}(\mathbf{f}(\mathbf{x}, \mathbf{w})) = \begin{bmatrix} \exp(f_1(\mathbf{x}, \mathbf{w})) \\ \exp(f_2(\mathbf{x}, \mathbf{w})) \\ \vdots \\ \exp(f_N(\mathbf{x}, \mathbf{w})) \end{bmatrix} / \sum_{k=1}^{N} \exp(f_k(\mathbf{x}, \mathbf{w}))$$

(2) compute cross entropy `torch.nn.CrossEntropyLoss`

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

# Loss functions

- Regression:
  - L2 loss
  - L1 loss
- Classification:
  - cross entropy loss (N-output classifier $\mathbf{f}(\mathbf{x}, \mathbf{w})$ )
  - logistic loss (single output dichotomy classifier $f(\mathbf{x}, \mathbf{w})$ )

$$L(\mathbf{w}) = \sum_i \log\left[1 + \exp(-y_i\, f(\mathbf{x}_i, \mathbf{w}))\right]$$

PyTorch: `nn.BCEWithLogitsLoss()`

Derivative can be found here:
https://deepnotes.io/softmax-crossentropy

# Loss functions

- Regression:
  - L2 loss
  - L1 loss
- Classification:
  - cross entropy loss (N-output classifier $\mathbf{f}(\mathbf{x}, \mathbf{w})$ )
  - logistic loss (single output dichotomy classifier $f(\mathbf{x}, \mathbf{w})$ )
  - Kulback-Leibler loss

$$L_{KL}(\mathbf{w}) = \sum_i y_i \cdot \log\left(y_i - f(\mathbf{x}_i, \mathbf{w})\right)$$

PyTorch: `torch.nn.NLLLoss()`

# Loss functions

- Regression:
  - L2 loss
  - L1 loss
- Classification:
  - cross entropy loss (N-output classifier $\mathbf{f}(\mathbf{x}, \mathbf{w})$ )
  - logistic loss (single output dichotomy classifier $f(\mathbf{x}, \mathbf{w})$ )
  - Kulback-Leibler loss
- Ranking:
  - Ranking loss

$$L_{rank}(\mathbf{w}) = \sum_{(i,j)\in\mathcal{T}} \max\{0, \quad -y_{ij} \cdot (f(\mathbf{x}_i, \mathbf{w}) - f(\mathbf{x}_j, \mathbf{w})) + \epsilon\}$$

PyTorch: `torch.nn.Margin RankingLoss()`

# Outline

- SGD vs deterministic gradient
- what makes learning to fail
- layers:
  - activation function (i.e. non-linearities)
  - batch normalization layer
  - max-pooling layer
  - loss-layers
- regularizations
- summary of the learning procedure
  - train, test, val data,
  - hyper-parameters,

# Regularization

- L2, L1 norms on weights (weight decay param. in SGD)
- Batch norm is regularization
- Drop out is regularization (it trains committee of experts)
- Jittering of training data is regularization

# Outline

- SGD vs deterministic gradient
- what makes learning to fail
- layers:
  - activation function (i.e. non-linearities)
  - batch normalization layer
  - max-pooling layer
  - loss-layers
- regularizations
- summary of the learning procedure
  - train, test, val data,
  - hyper-parameters,

# Training procedure

- Choose:
  - Weight initialization
  - Network architecture (ideally re-use pre-trained net)
  - Learning rate and other hyper-parameters.
  - Loss + regularization
- Divide data on three represenatative subsets:
  - Training data (the set on which the backprop is used to estimate weights)
  - Validation data (the set on which hyper-param are tuned)
  - Testing data (the set on which the error is only observed)

# Hyper parameters tuning

- Weight initialization (Xavier)

# Hyper parameters tuning

- Weight initialization (Xavier)
- Trn error is huge =>underfitting
  - decrease regularization strength
  - increase model capacity

# Hyper parameters tuning

- Weight initialization (Xavier)
- Trn error is huge =>underfitting
    - decrease regularization strength
    - increase model capacity
- Trn error explodes to infinity=> huge learning rate
  - decrease the learning rate

# Hyper parameters tuning

- Weight initialization (Xavier)
- Trn error is huge =>underfitting
    - decrease regularization strength
    - increase model capacity
- Trn error explodes to infinity=> huge learning rate
  - decrease the learning rate
- Trn error is decreasing very slowly => small learning rate
  - increase learning rate

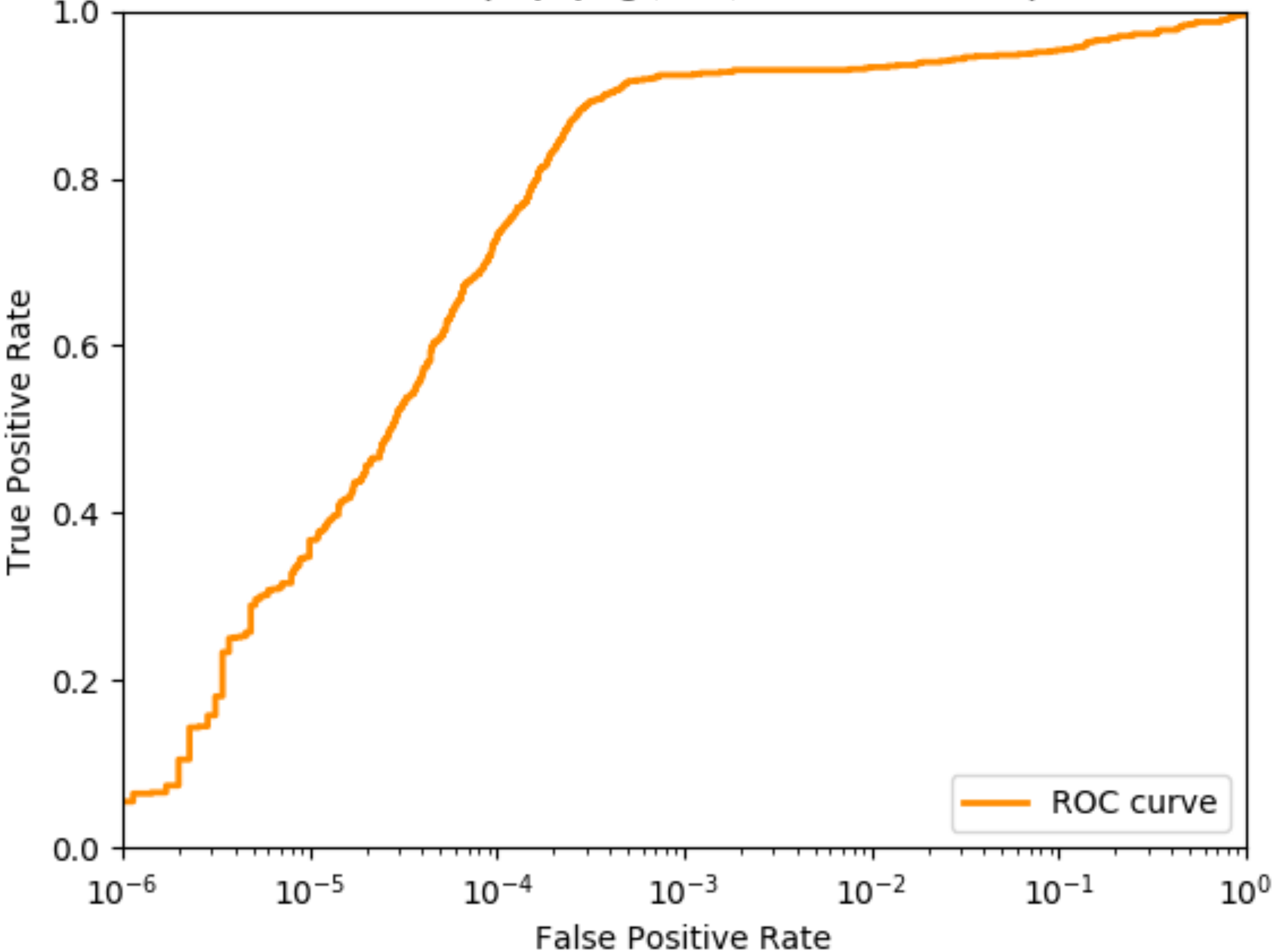
# Hyper parameters tuning

- Weight initialization (Xavier)
- Trn error is huge =>underfitting
    - decrease regularization strength
    - increase model capacity
- Trn error explodes to infinity=> huge learning rate
    - decrease the learning rate
- Trn error is decreasing very slowly => small learning rate
    - increase learning rate
- Tst error>>Trn error => overfitting
    - increase strength of regularization
    - decrease model capacity
    - Tst data are too far from Trn data
      (should come from the same distribution)

# Hyper parameters tuning

- Weight initialization (Xavier)
- Trn error is huge =>underfitting
    - decrease regularization strength
    - increase model capacity
- Trn error explodes to infinity=> huge learning rate
    - decrease the learning rate
- Trn error is decreasing very slowly => small learning rate
    - increase learning rate
- Tst error>>Trn error => overfitting
    - increase strength of regularization
    - decrease model capacity
    - Tst data are too far from Trn data (should come from the same distribution)
- Trn error>>Tst error =>bad division on training/testing data

# ROC curve



1.0
0.8
0.6
0.4
0.2
0.0
True Positive Rate
$10^{-6}$
$10^{-5}$
$10^{-4}$
$10^{-3}$
$10^{-2}$
$10^{-1}$
$10^{0}$
False Positive Rate
ROC curve