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/

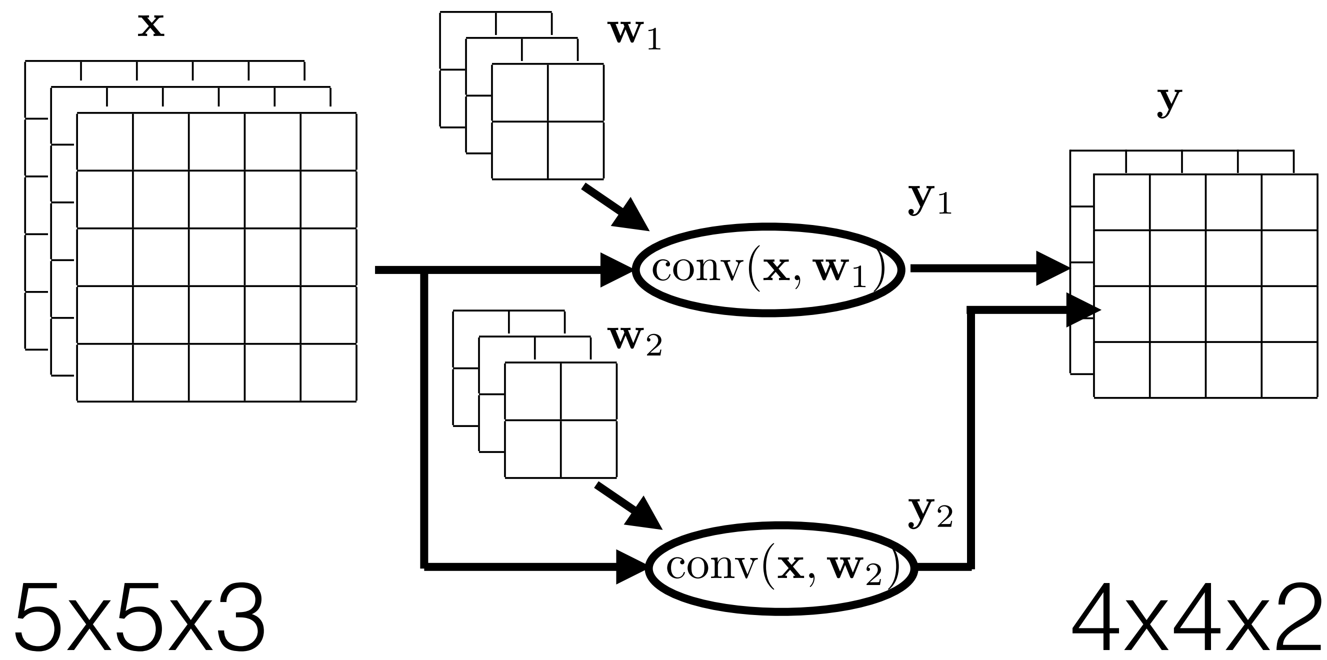
Center for Machine Perception
https://cmp.felk.cvut.cz

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

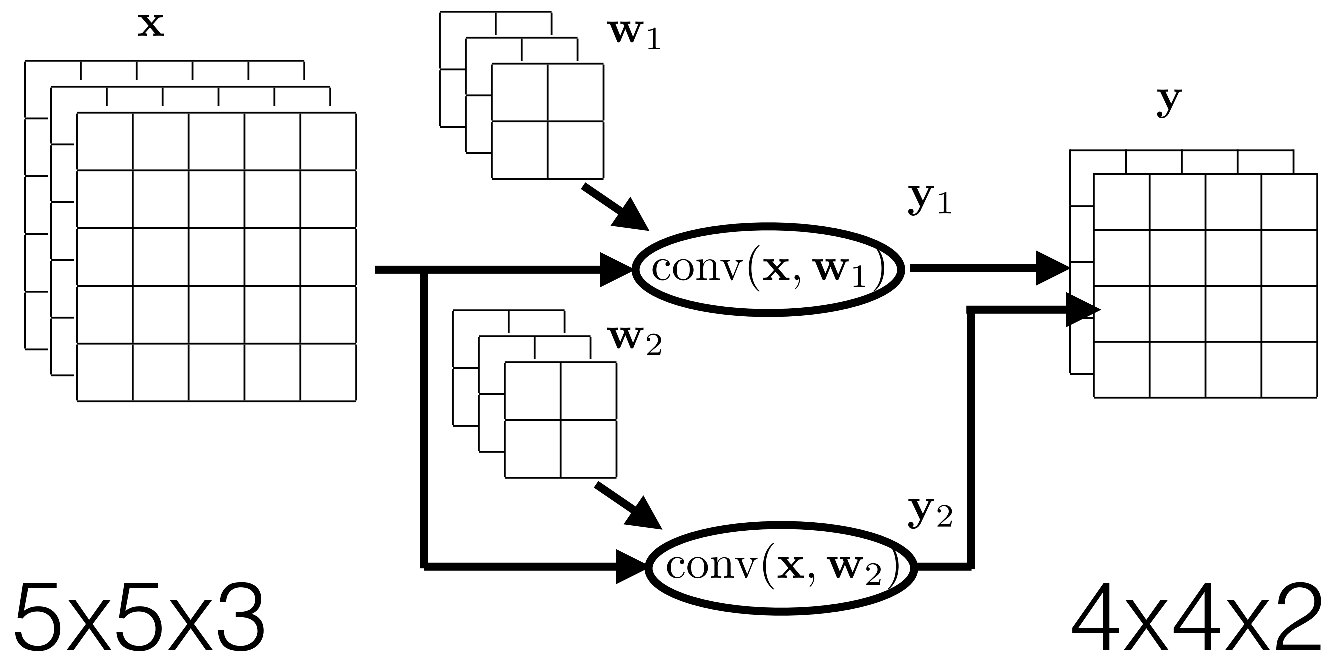
Outline

- 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

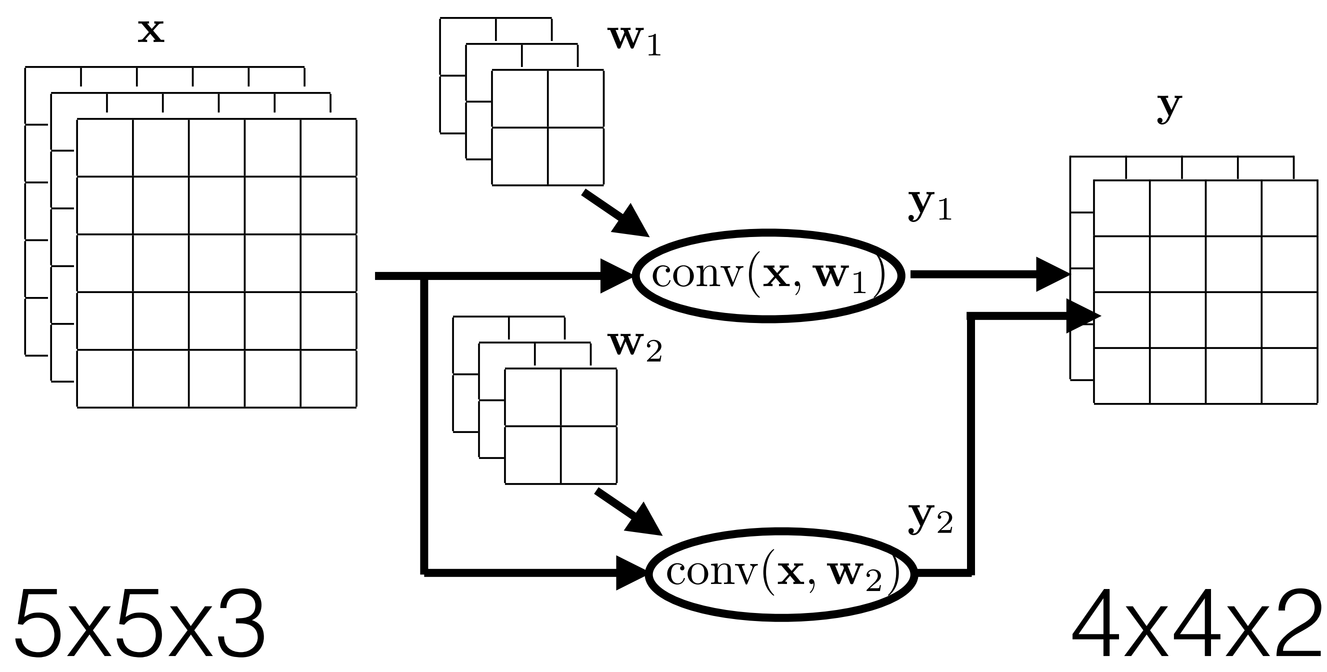


2D convolution forward pass




```python
# 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)
```

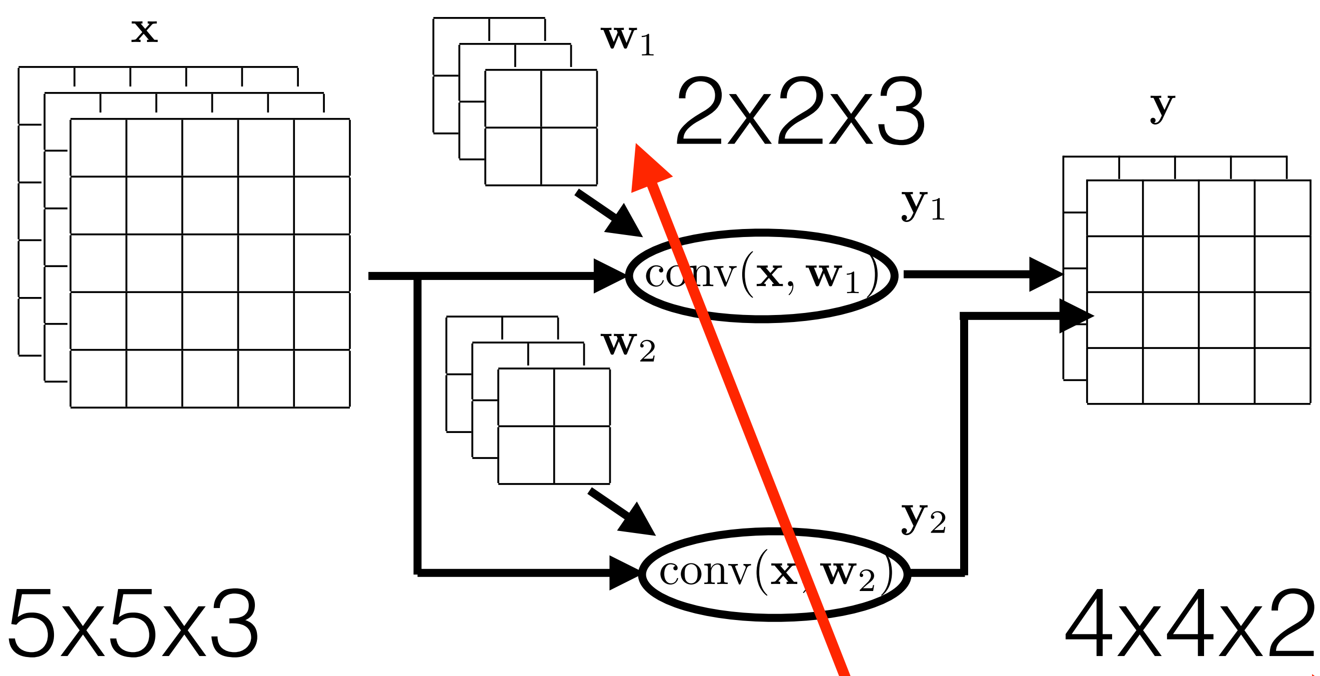
2D convolution forward pass



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)
```

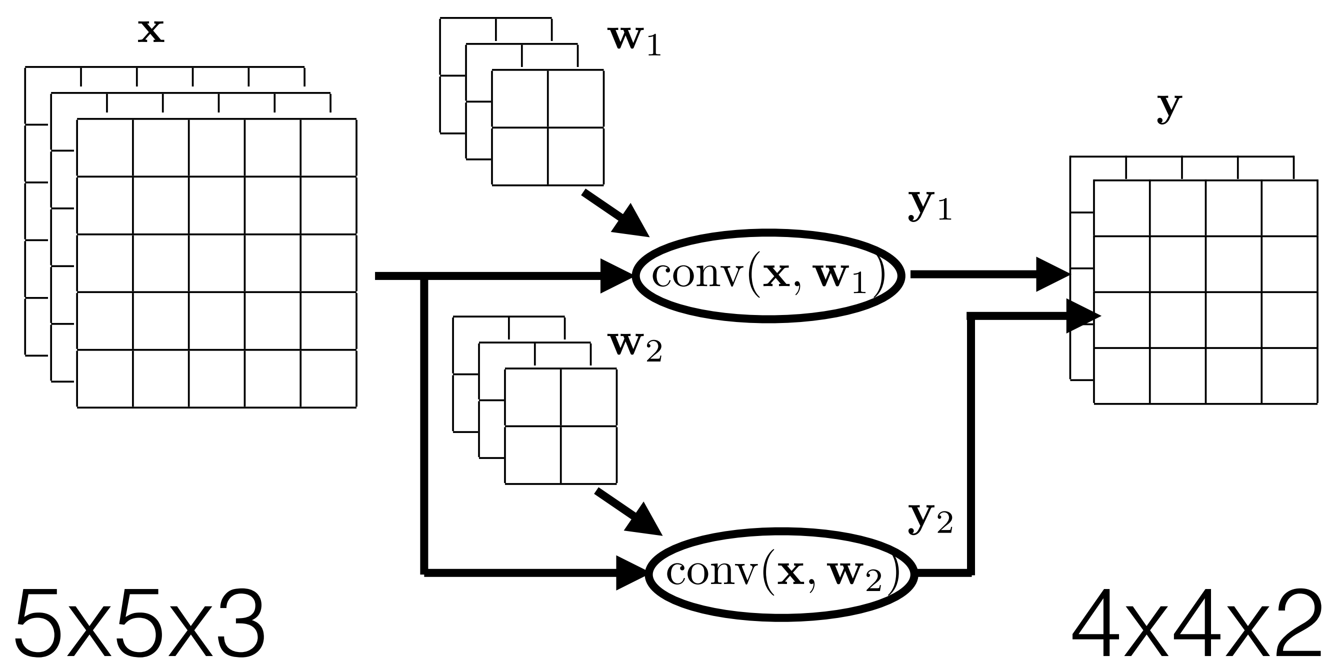
2D convolution forward pass



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



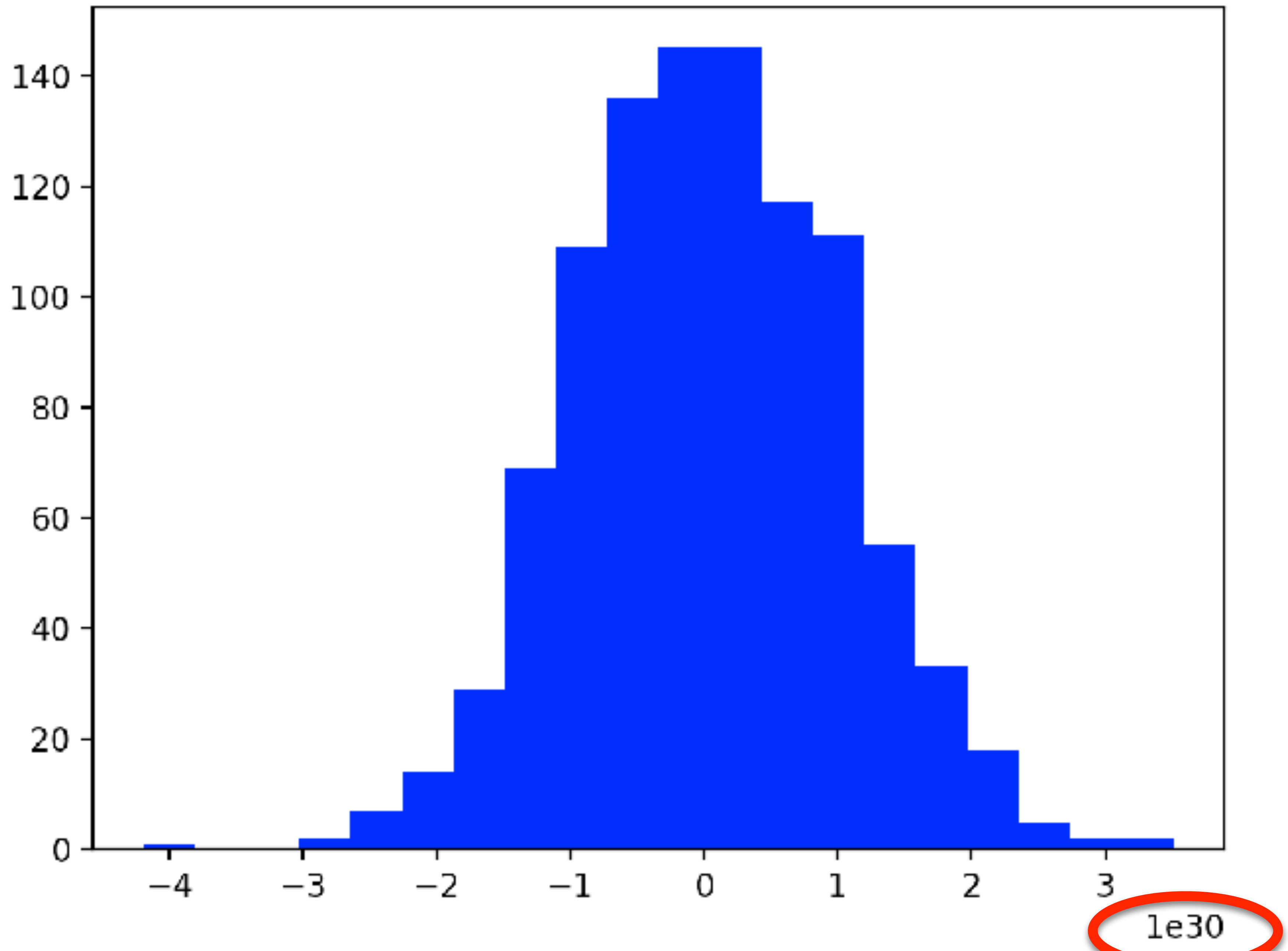
Very important property of convolutional layer is:

Local gradient is also convolution !!!

Learning

What happens to deep **conv outputs** when weights are **huge**?

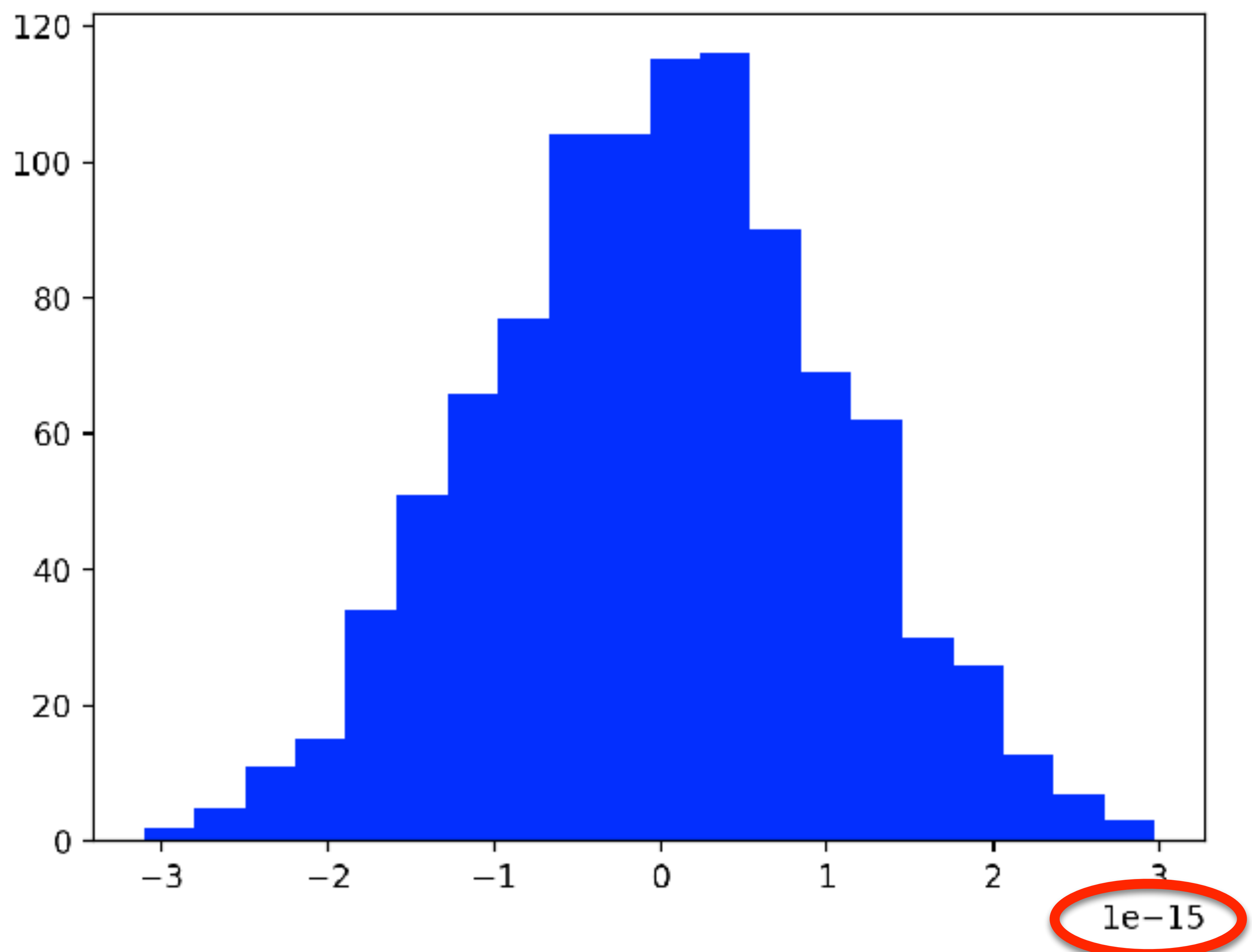
```
y = torch.randn(1000,1)
for i in range(20):
    weights = torch.randn(1000,1000)
    y =  weights @ y
```



Learning

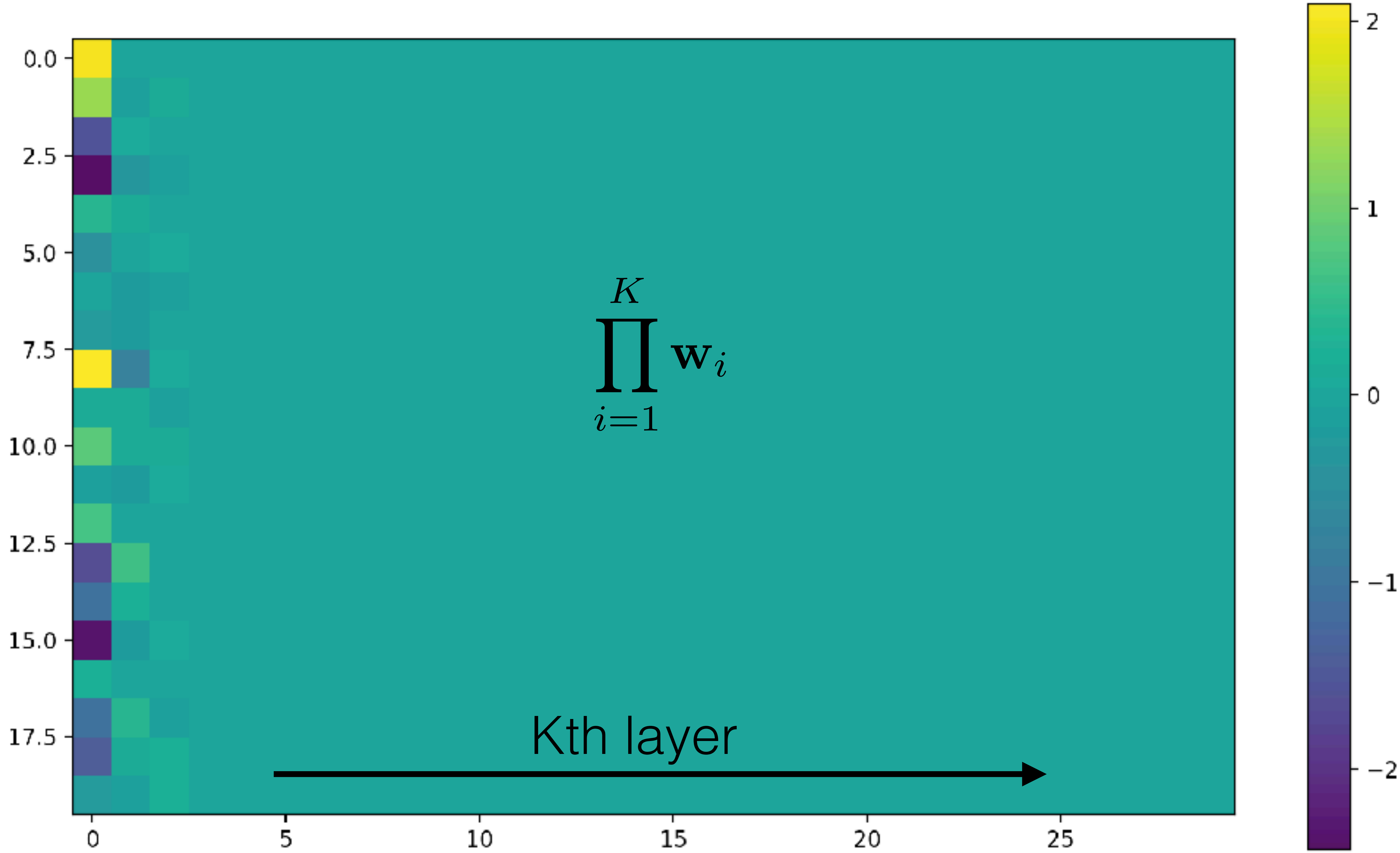
What happens to deep **conv outputs** when weights are **small**?

```
y = torch.randn(1000,1)
for i in range(30):
    weights = torch.randn(1000,1000)/100
    y =  weights @ y
```



Learning

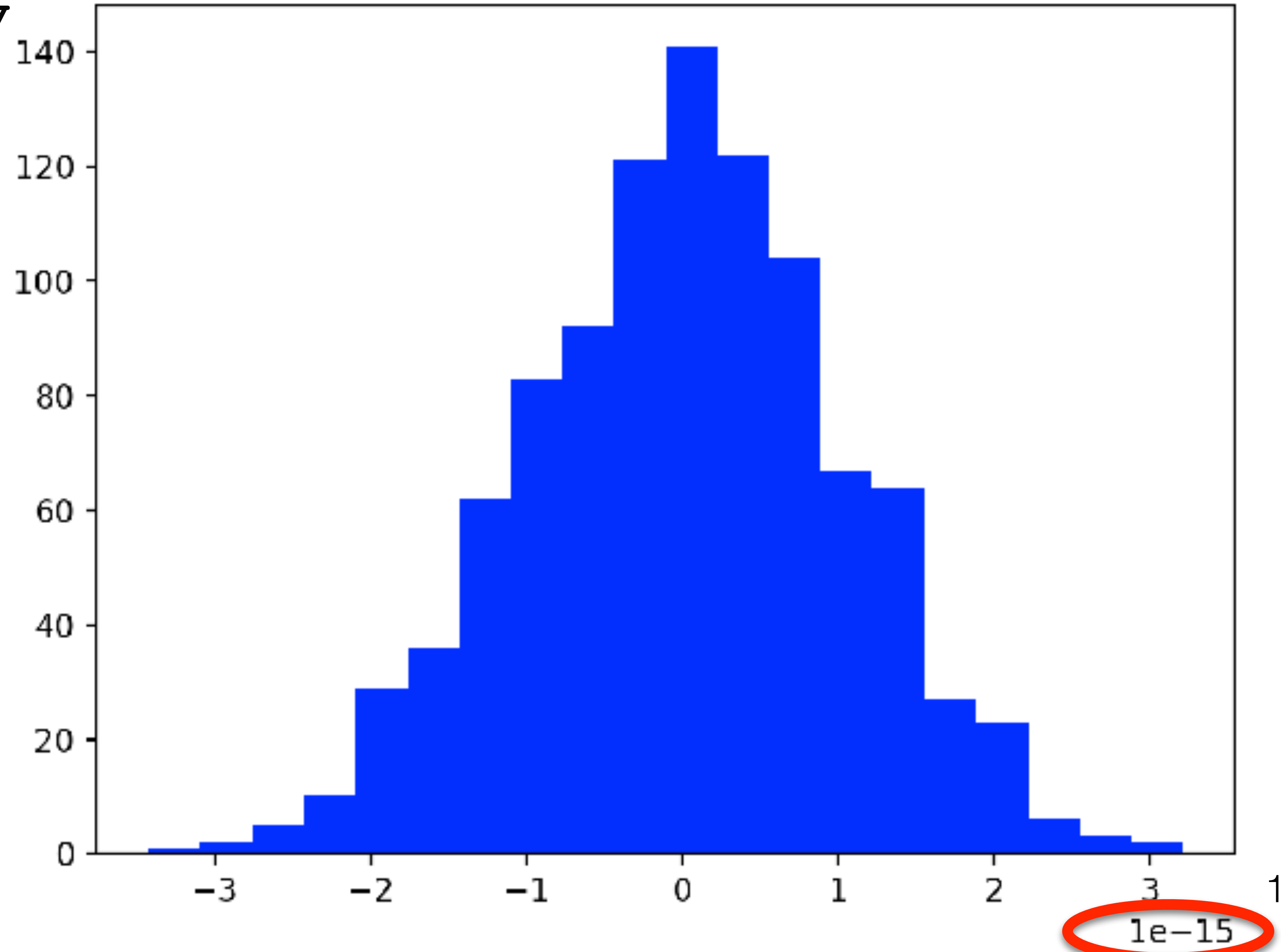
What happens to deep **conv gradient** when weights are **small**?



Learning

What happens to deep **conv gradient** when weights are **small**?

```
x = torch.randn(1000,1)
x.requires_grad_()
y=x
for i in range(30):
    weights = torch.randn(1000,1000)/100
    y =  weights @ y
y.sum().backward()
x.grad
```



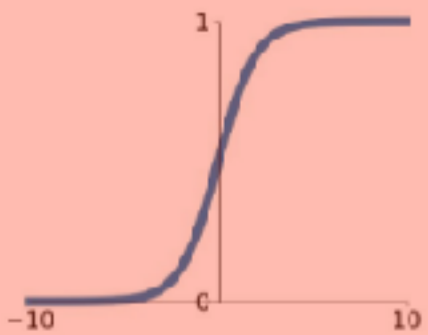
Outline

- 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

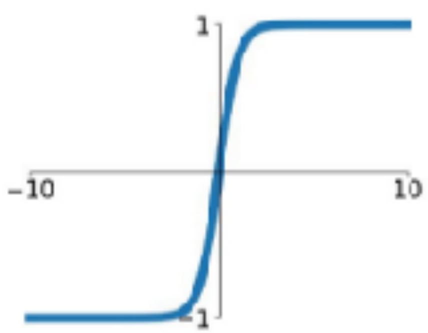
Sigmoid

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



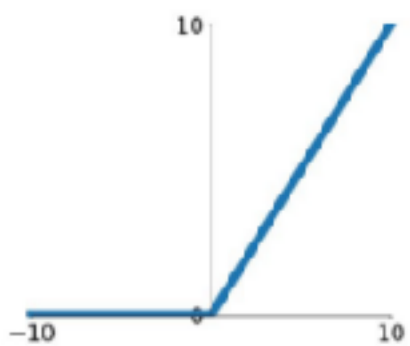
tanh

$\tanh(x)$



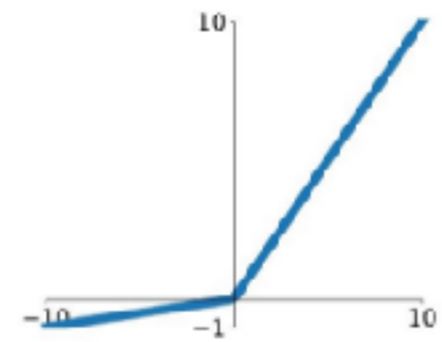
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

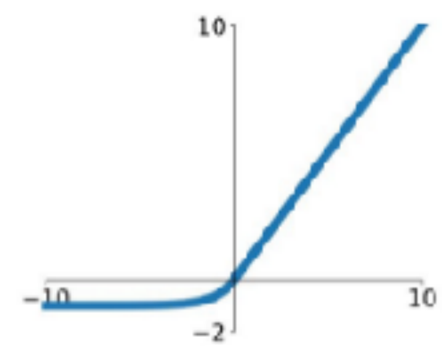


Maxout

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

ELU

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

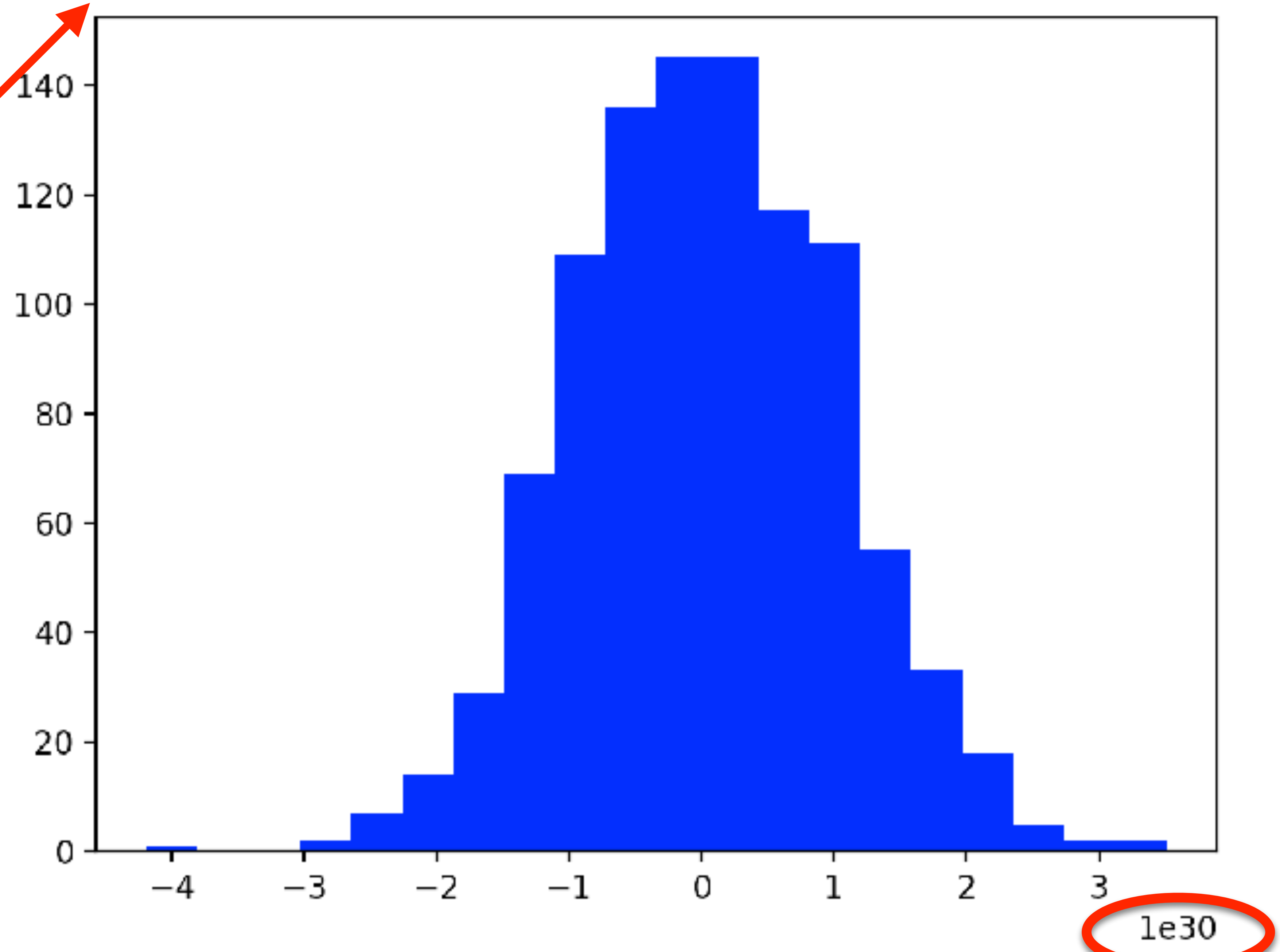


Learning

What happens to deep **conv outputs** when weights are **huge**?

```
y = torch.randn(1000,1)
for i in range(20):
    weights = torch.randn(1000,1000)
    y =  weights @ y
```

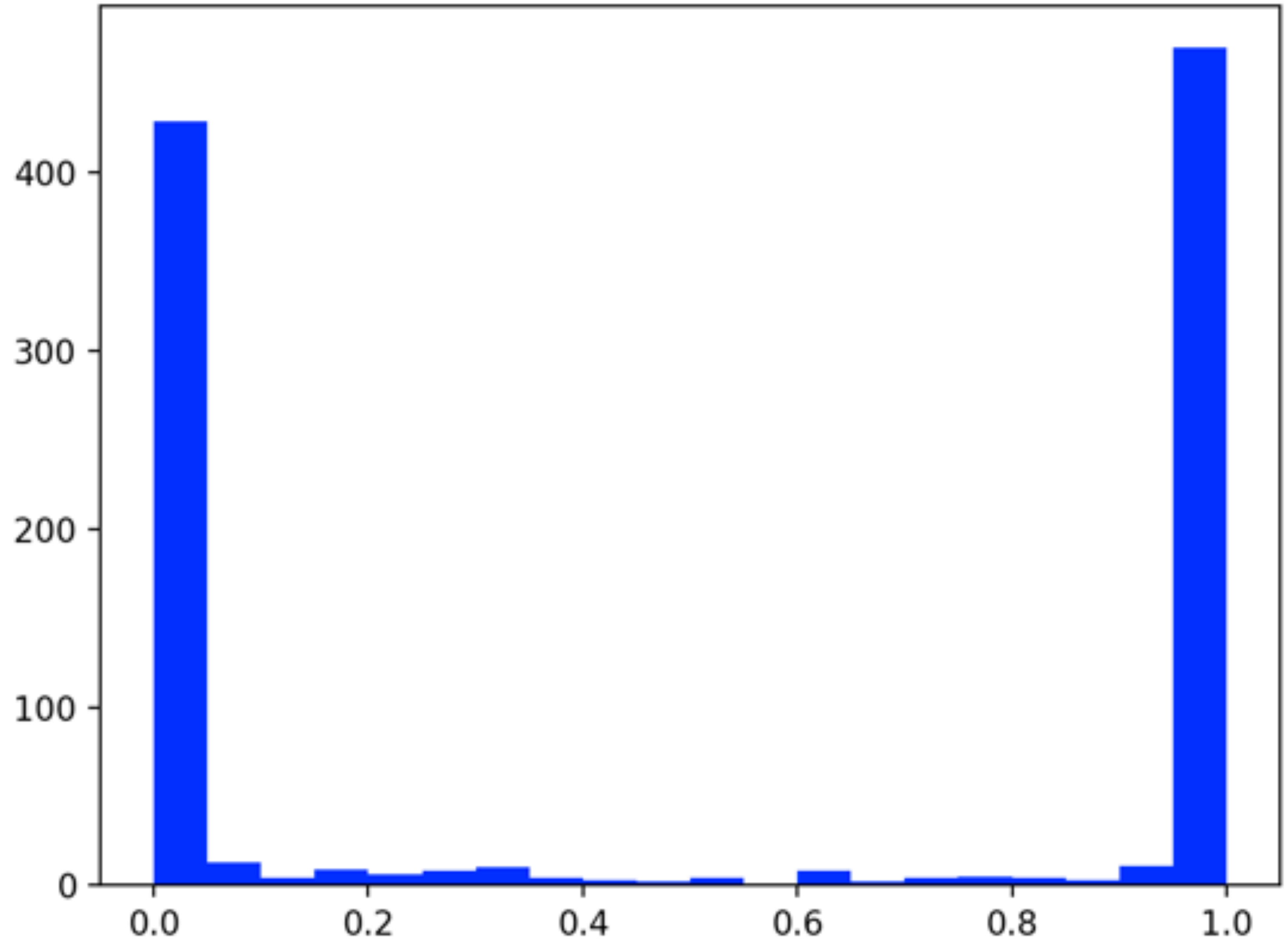
sigmoid



Learning

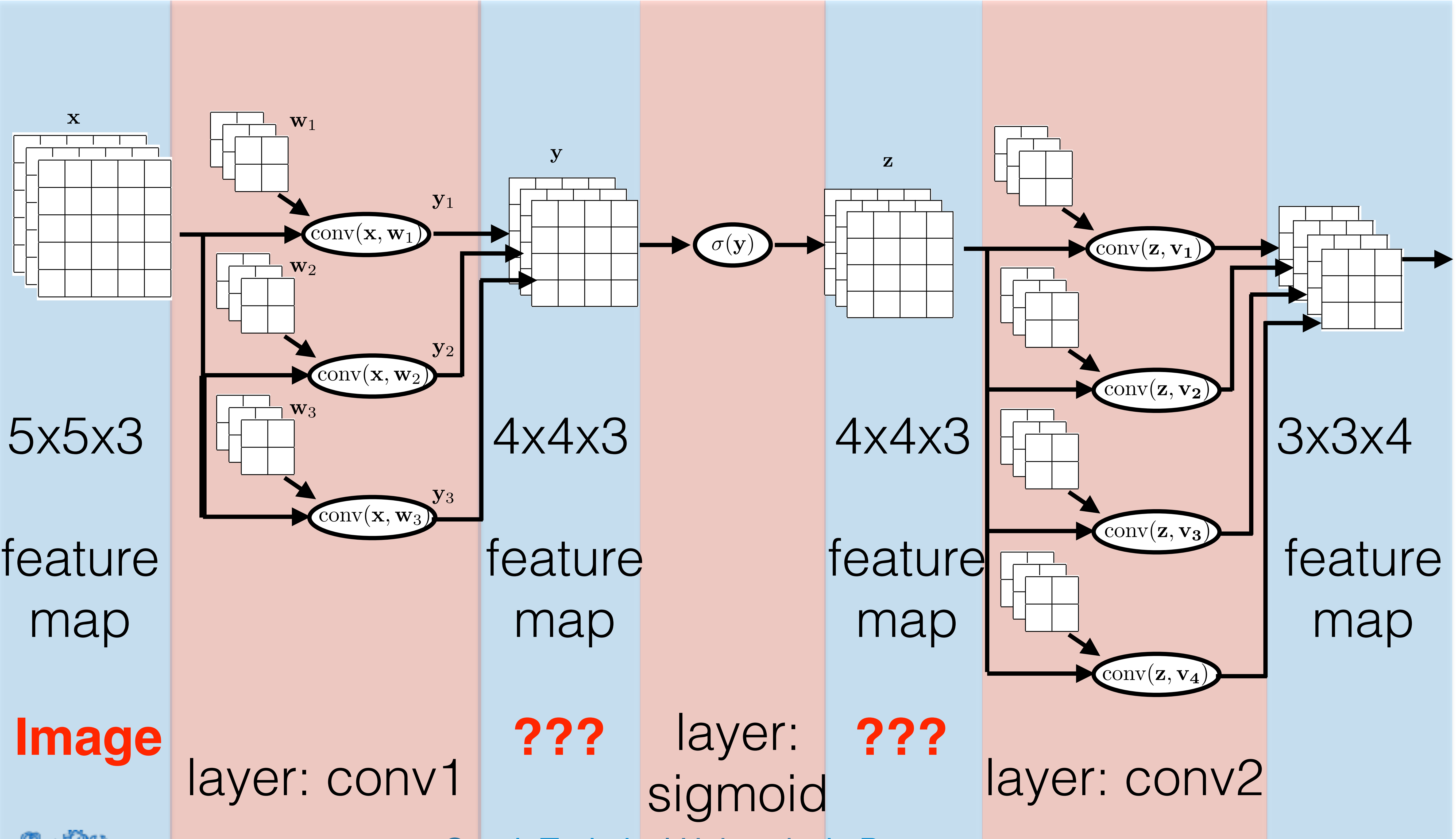
What happens to deep **sigm outputs** when weights are **huge**?

```
y = torch.randn(1000,1)
for i in range(30):
    weights = torch.randn(1000,1000)
    y =  torch.sigmoid(weights @ y)
```



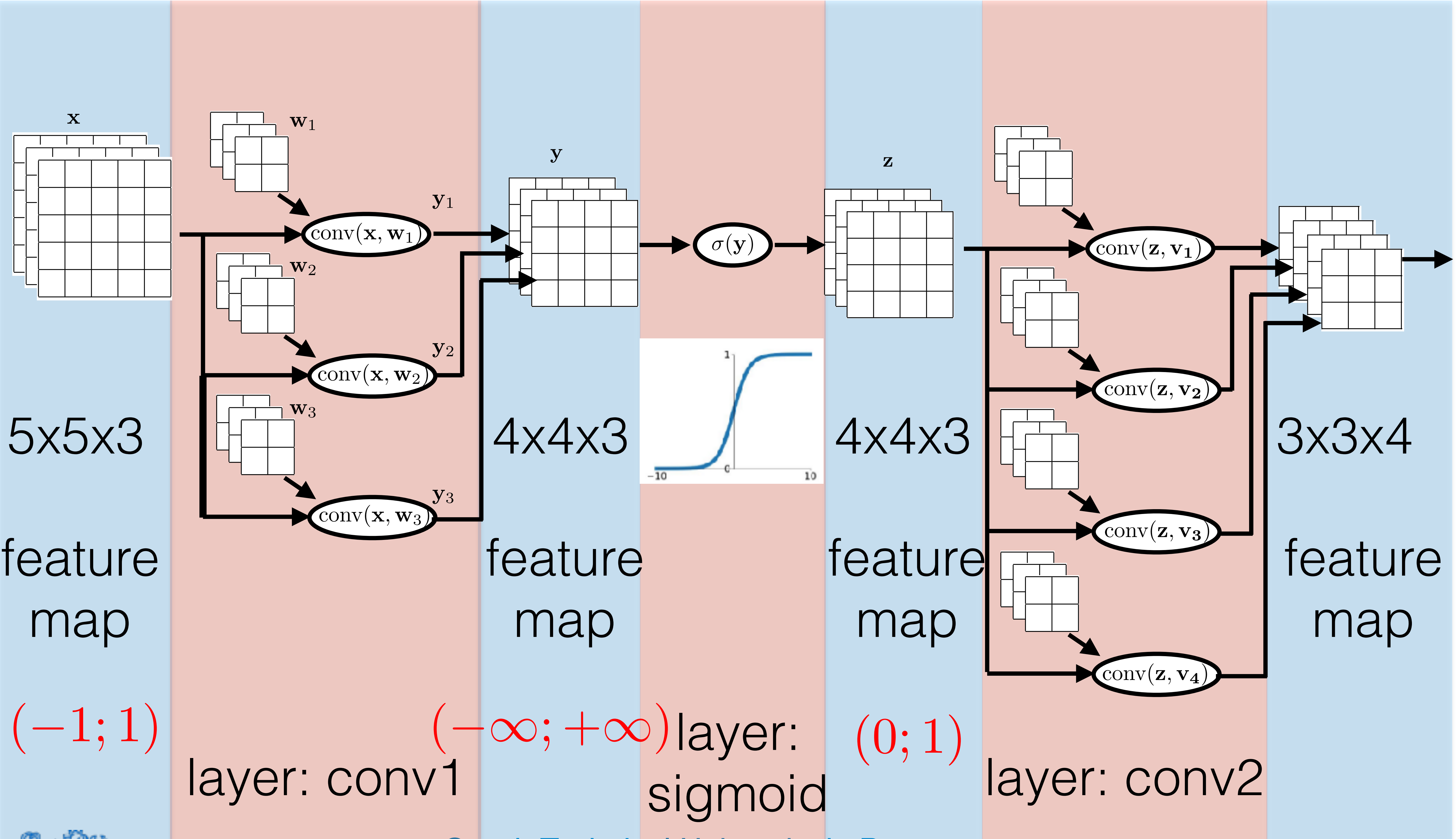
Learning

- let us plug image as input, what **values** are propagated?

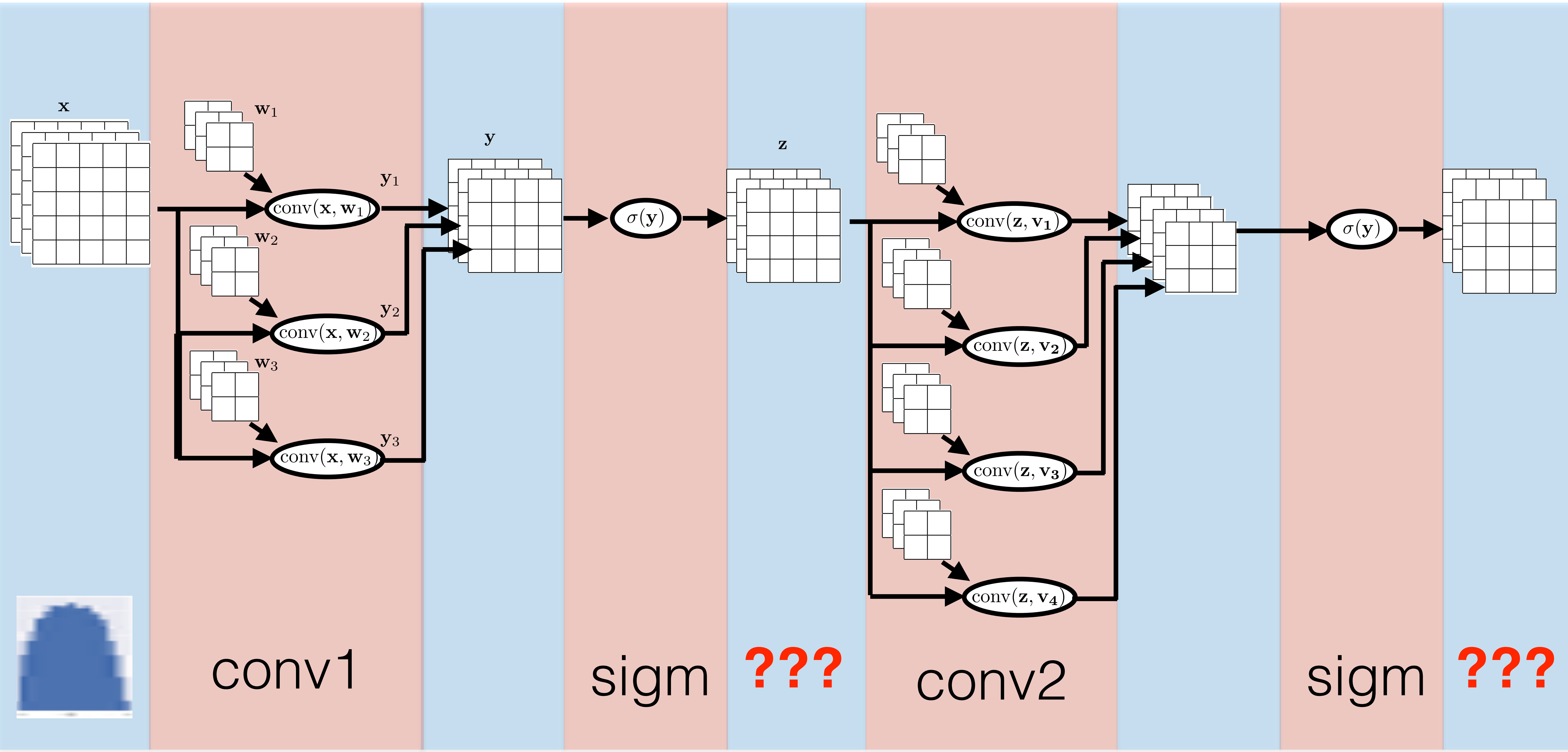


Learning

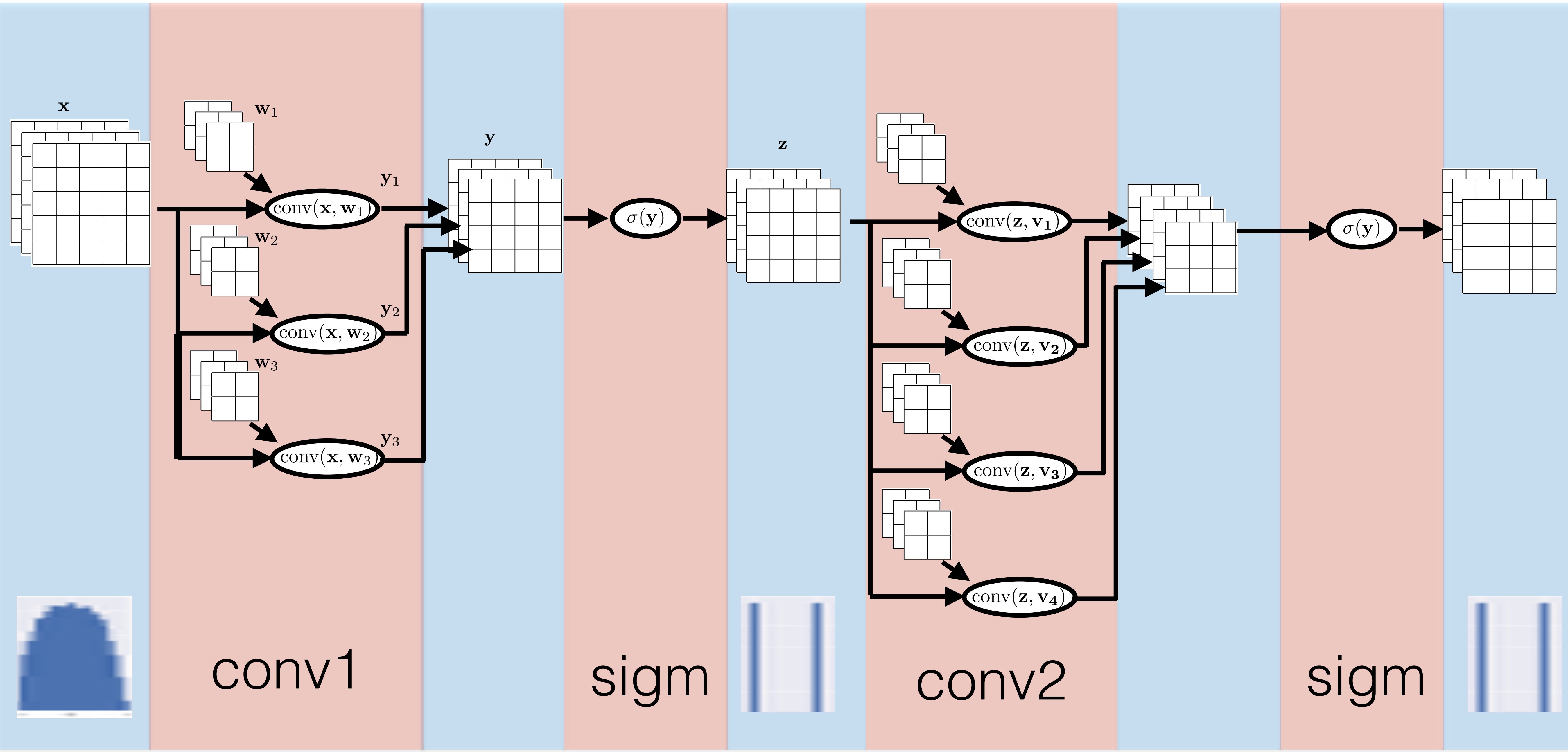
- let us plug image as input, what **values** are propagated?



Learning
What happens to deep **sigm outputs** when weights are **huge**?



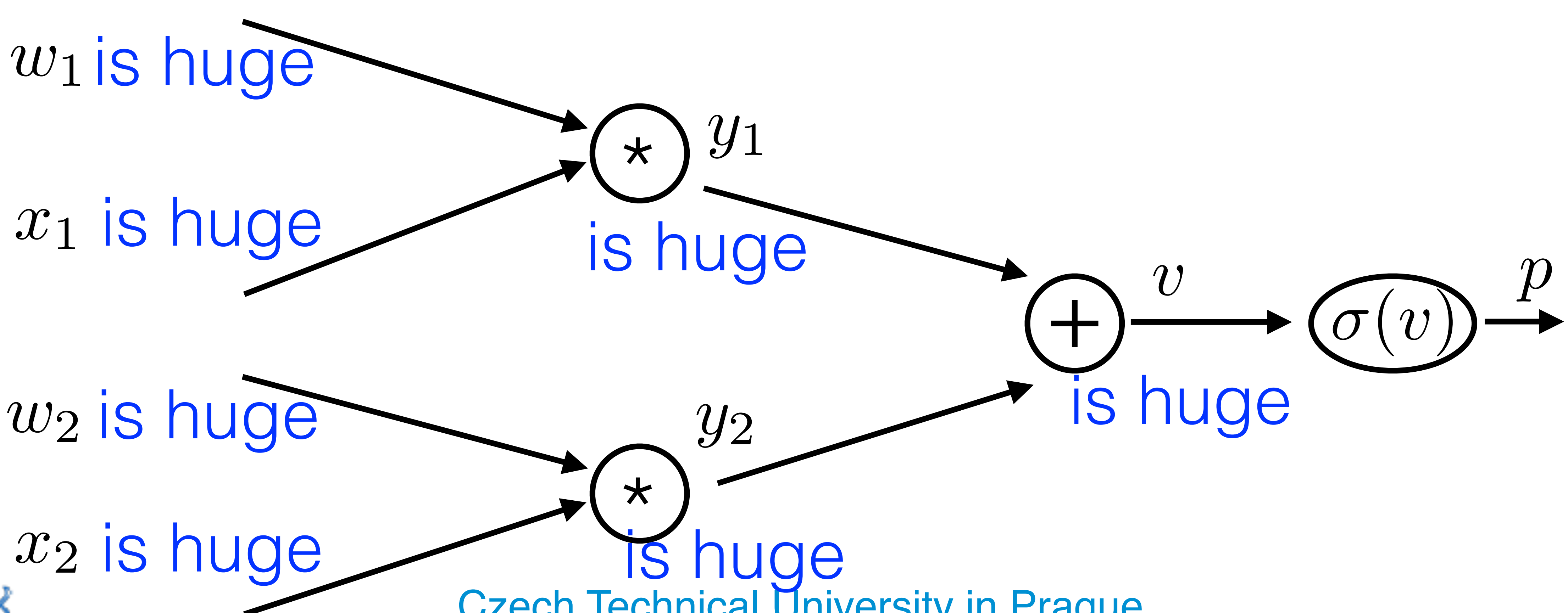
Learning
What happens to deep **sigm outputs** when weights are **huge**?



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

$$\frac{\partial p}{\partial w_1} = ?$$

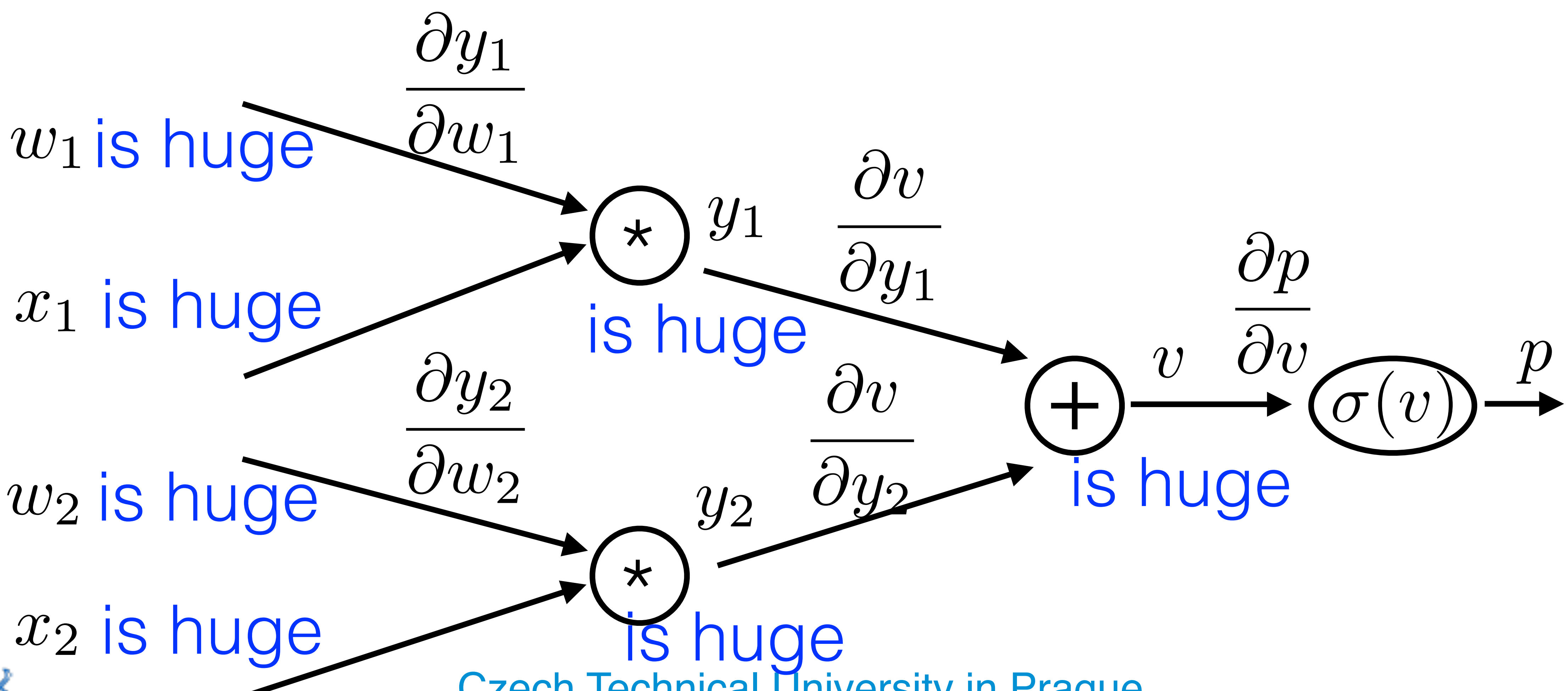
$$\frac{\partial p}{\partial w_2} = ?$$



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

$$\frac{\partial p}{\partial w_1} = ?$$

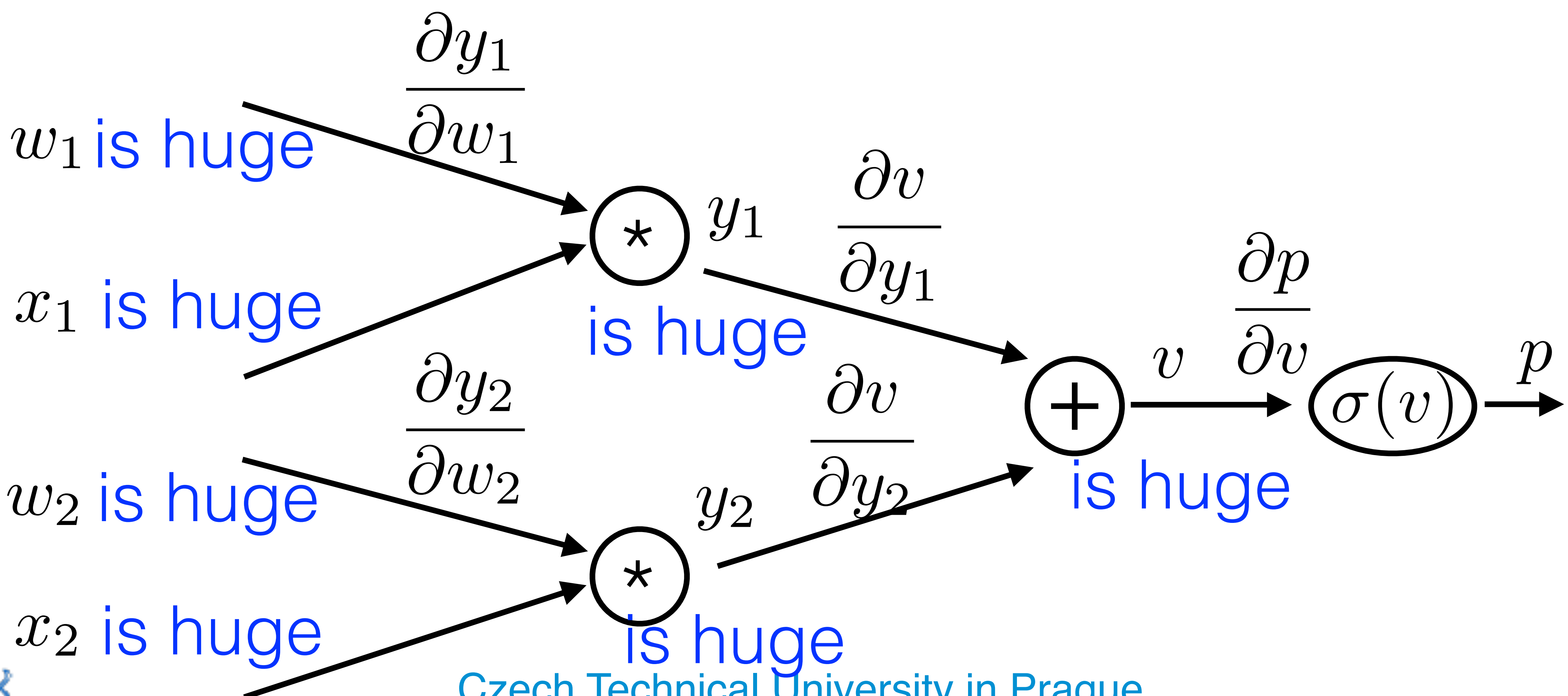
$$\frac{\partial p}{\partial w_2} = ?$$



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

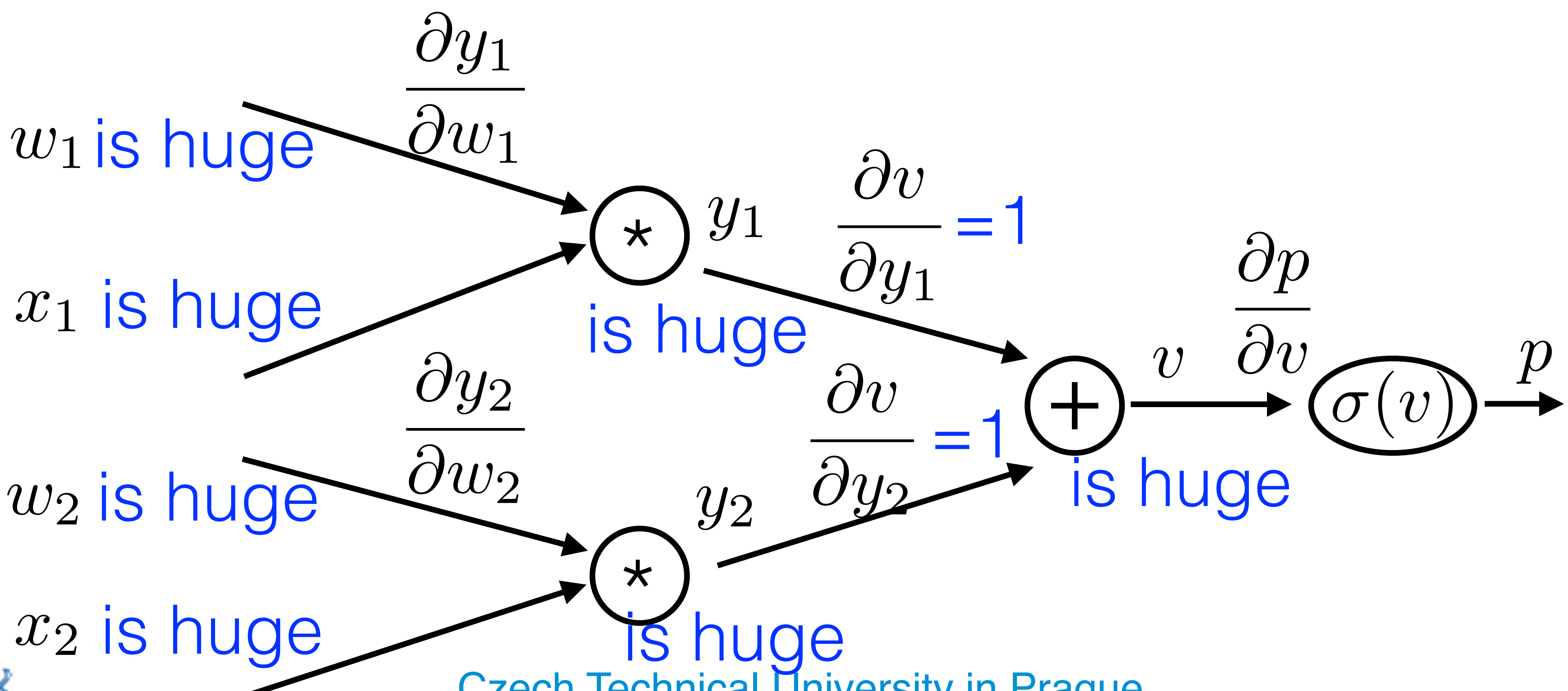
$$\frac{\partial p}{\partial w_1} = \frac{\partial y_1}{\partial w_1}\frac{\partial v}{\partial y_1}\frac{\partial p}{\partial v} = ?$$

$$\frac{\partial p}{\partial w_2} = \frac{\partial y_2}{\partial w_2}\frac{\partial v}{\partial y_1}\frac{\partial p}{\partial v} = ?$$



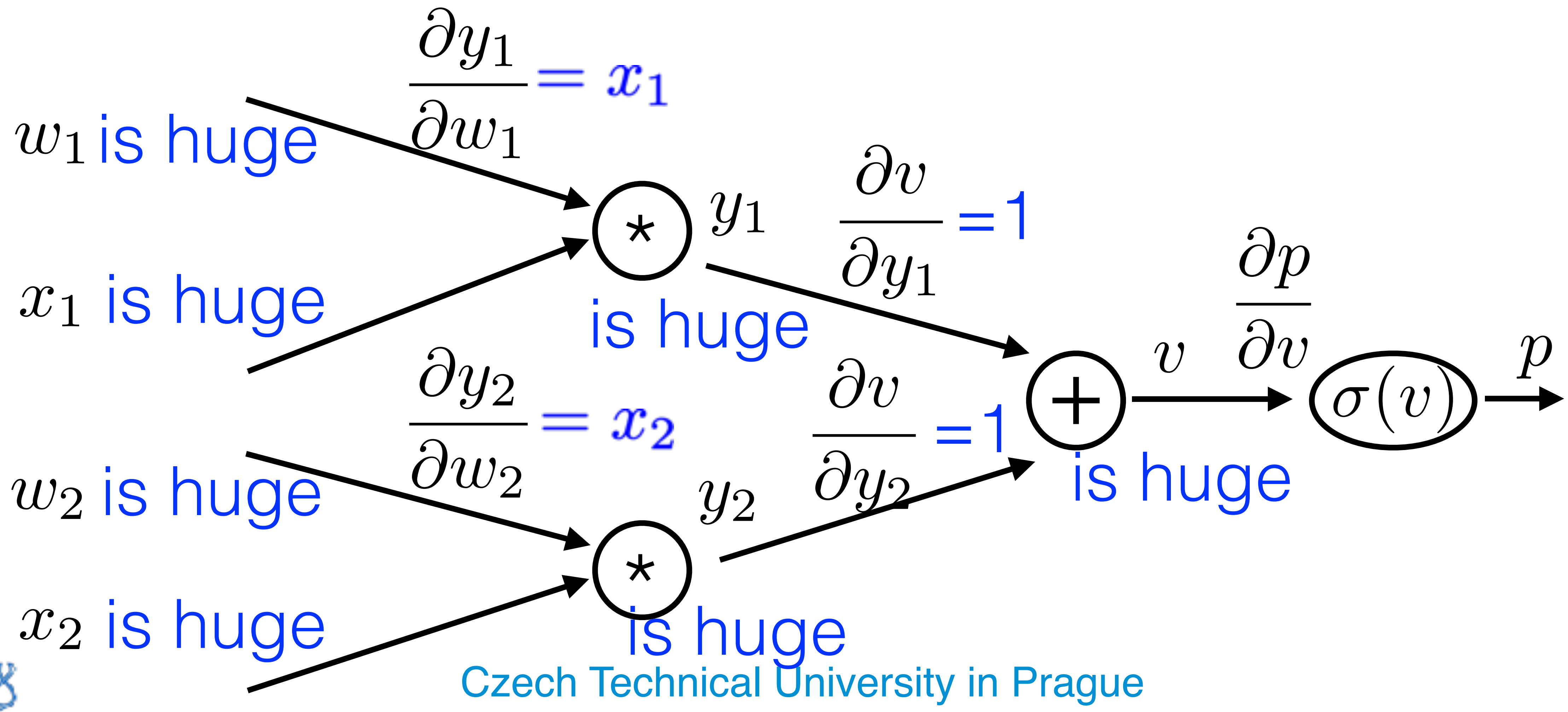
- what happen to **backprop gradient** when weights are **huge**?

$$\frac{\partial p}{\partial w_1} = \frac{\partial y_1}{\partial w_1} \cdot 1 \quad \frac{\partial p}{\partial v} = ? \qquad \frac{\partial p}{\partial w_2} = \frac{\partial y_2}{\partial w_2} \cdot 1 \quad \frac{\partial p}{\partial v} = ?$$



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

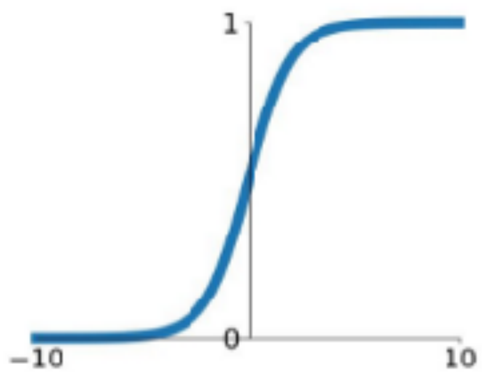
$$\frac{\partial p}{\partial w_1} = x_1 \cdot 1 \quad \frac{\partial p}{\partial v} = ? \qquad \frac{\partial p}{\partial w_2} = x_2 \cdot 1 \quad \frac{\partial p}{\partial v} = ?$$



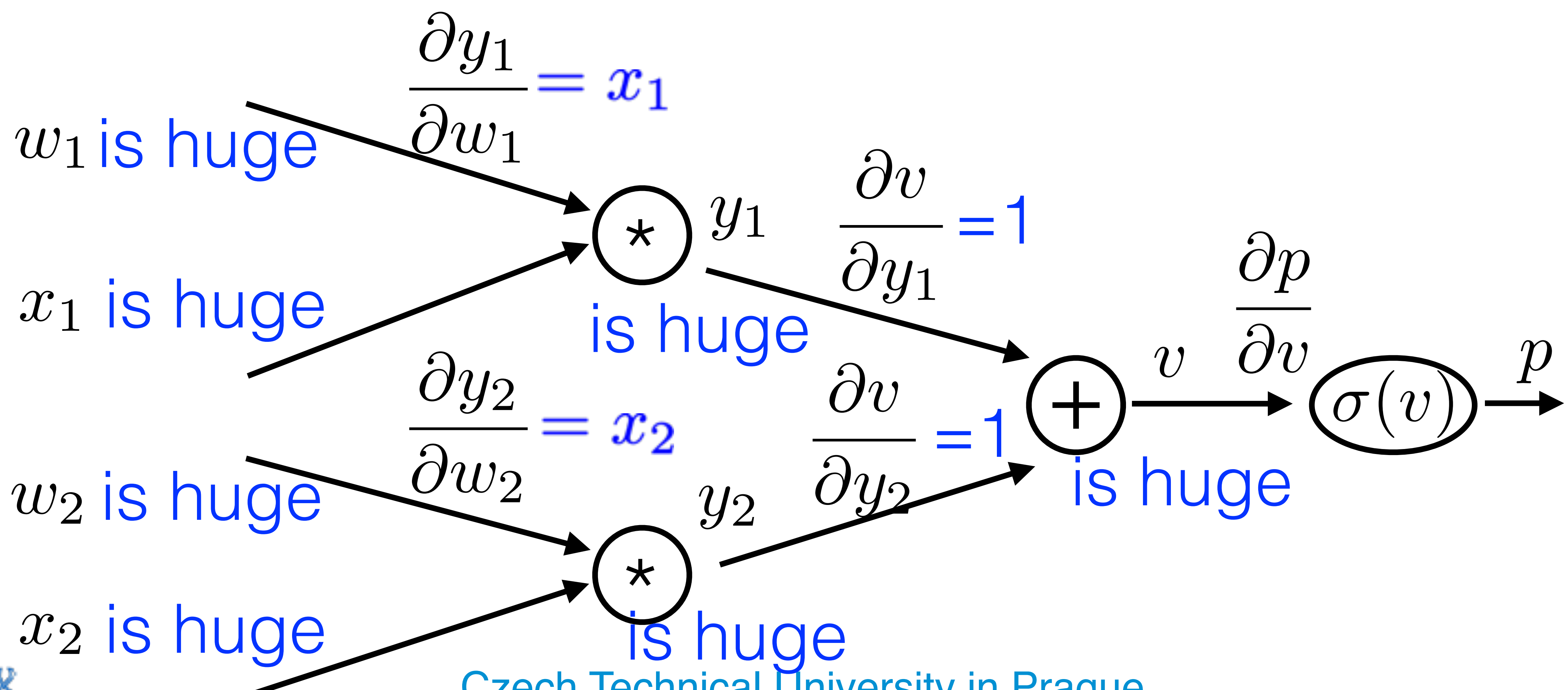
- what happen to **backprop gradient** when weights are **huge**?

Sigmoid

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



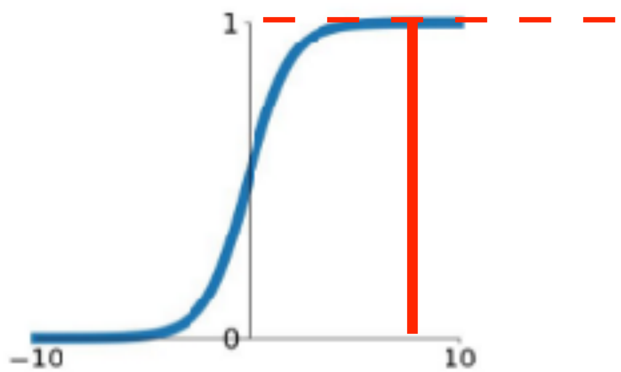
$$\frac{\partial p}{\partial w_1} = x_1 \cdot 1 \quad \frac{\partial p}{\partial v} = ? \qquad \frac{\partial p}{\partial w_2} = x_2 \cdot 1 \quad \frac{\partial p}{\partial v} = ?$$



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

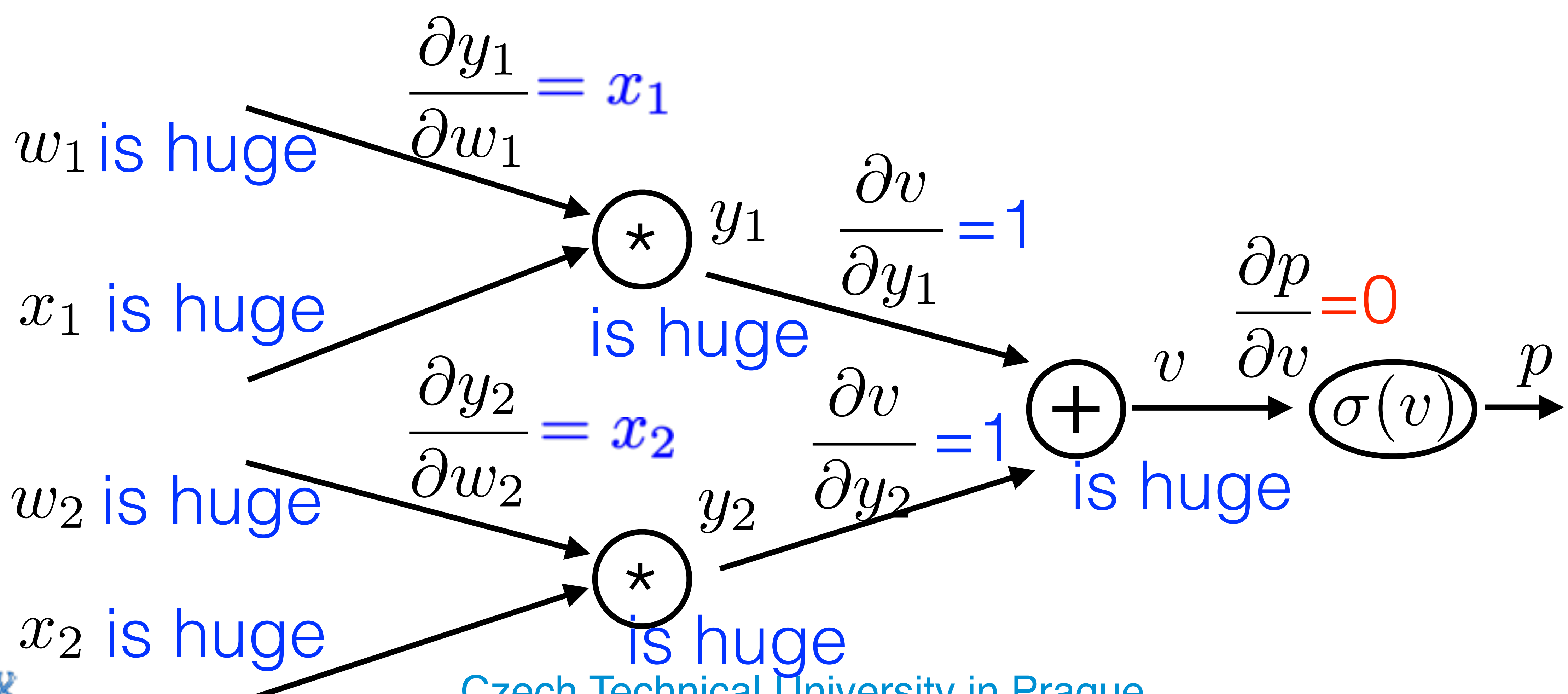
Sigmoid

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



$$\frac{\partial p}{\partial v} = 0$$

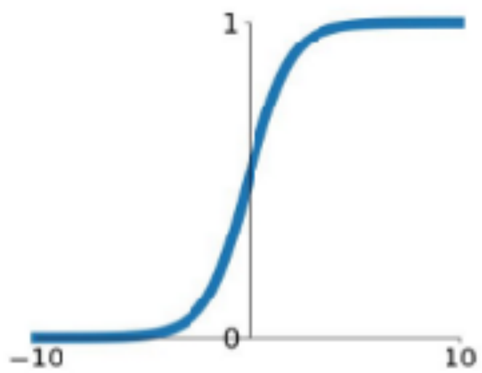
$$\frac{\partial p}{\partial w_1} = x_1 \cdot 1 \quad \frac{\partial p}{\partial v} = 0 \qquad\qquad \frac{\partial p}{\partial w_2} = x_2 \cdot 1 \quad \frac{\partial p}{\partial v} = 0$$



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

Sigmoid

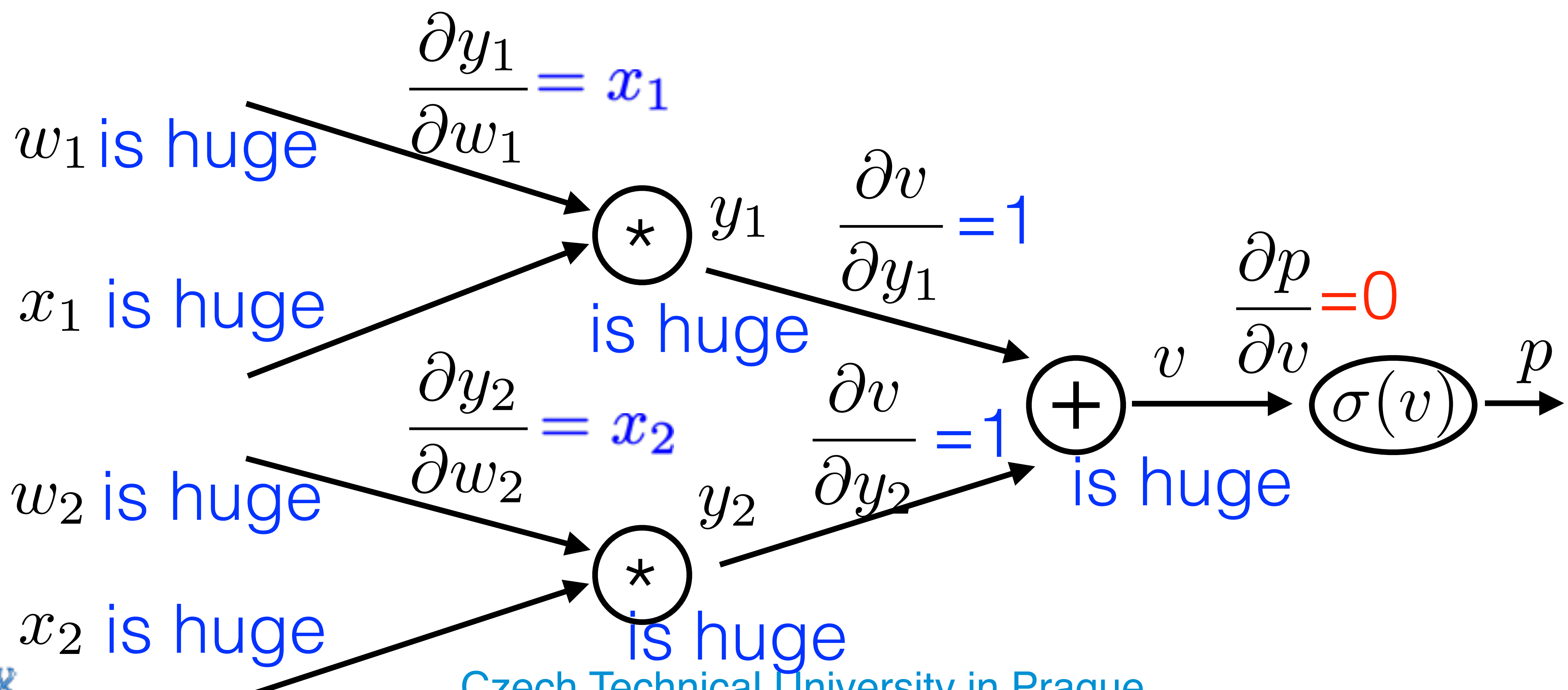
$\sigma(x) = \frac{1}{1+e^{-x}}$



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

$\frac{\partial p}{\partial w_1} = x_1 \cdot 1 \quad \frac{\partial p}{\partial v} = 0$

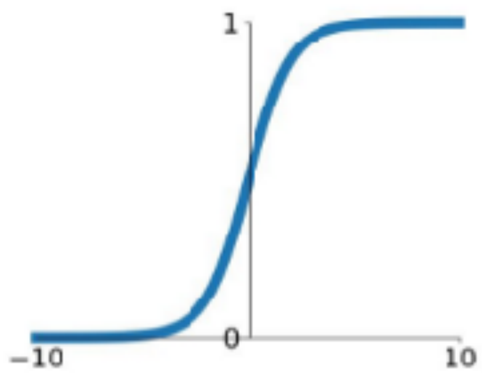
$\frac{\partial p}{\partial w_2} = x_2 \cdot 1 \quad \frac{\partial p}{\partial v} = 0$



- what happens when sigmoid input is only positive?

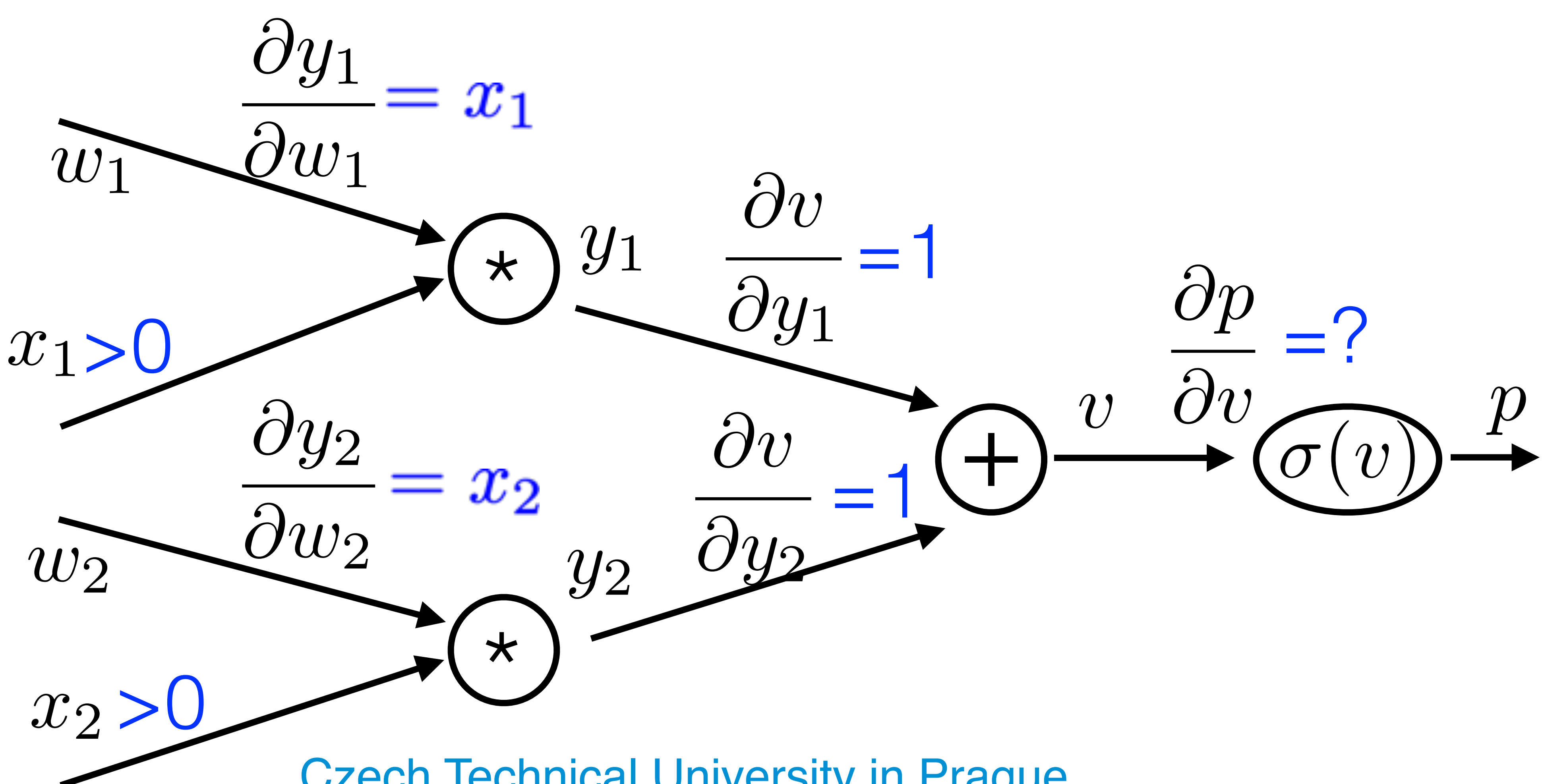
Sigmoid

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



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

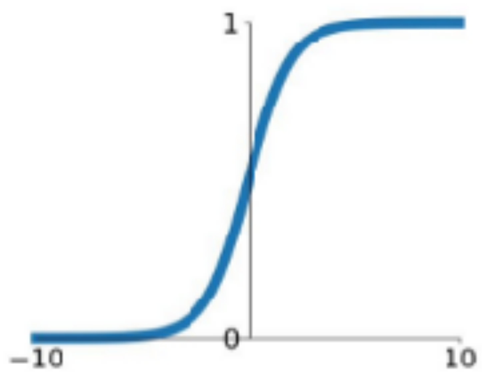
$\frac{\partial p}{\partial w_1} = x_1 \cdot 1 \quad \frac{\partial p}{\partial v} = ? \qquad \frac{\partial p}{\partial w_2} = x_2 \cdot 1 \quad \frac{\partial p}{\partial v} = ?$



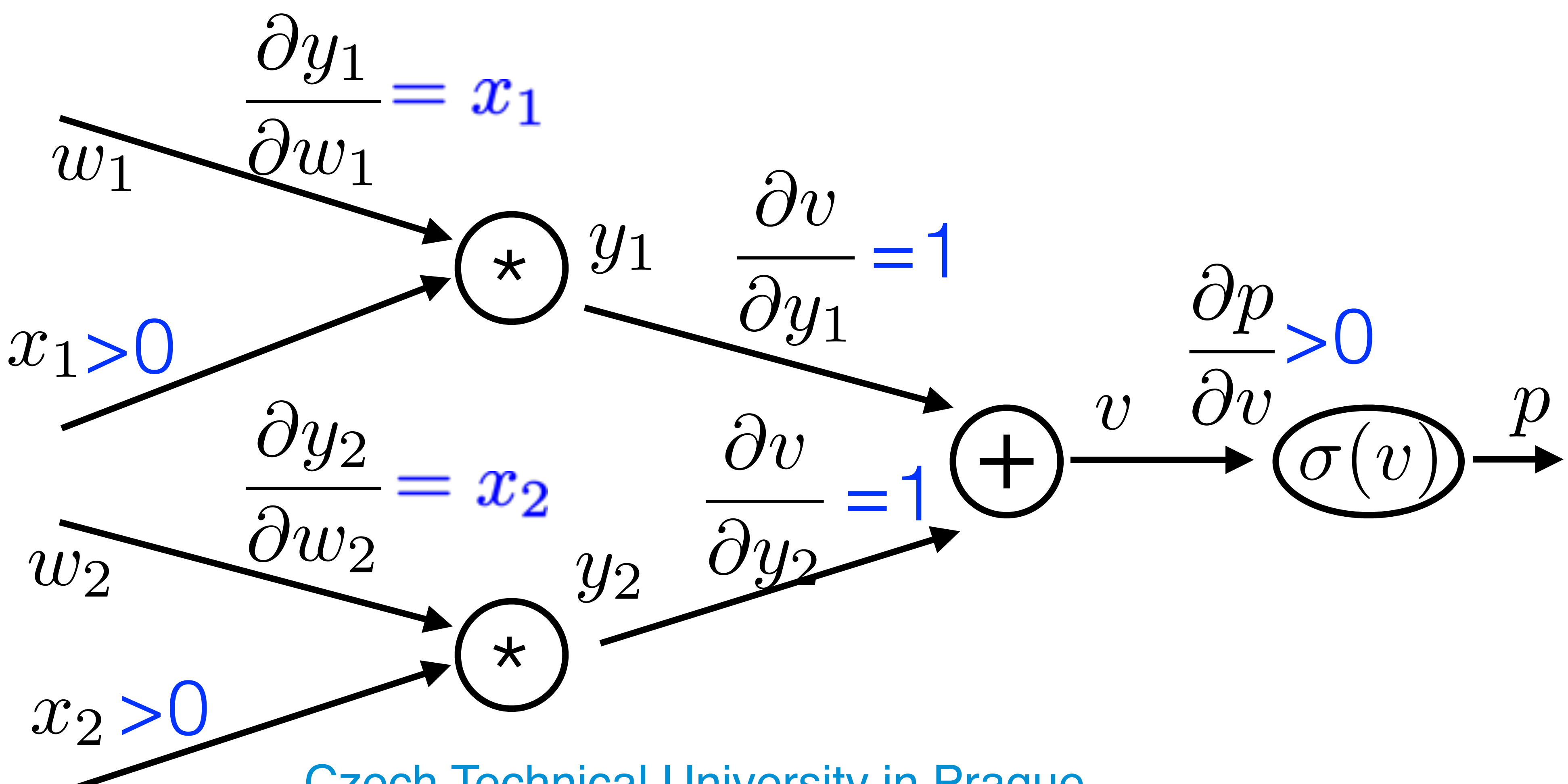
- what happens when sigmoid input is only positive?

Sigmoid

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



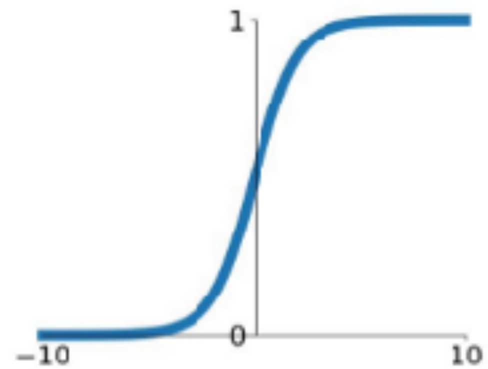
$$\frac{\partial p}{\partial w_1} = x_1 \cdot 1 \cdot \frac{\partial p}{\partial v} = ? \qquad \frac{\partial p}{\partial w_2} = x_2 \cdot 1 \cdot \frac{\partial p}{\partial v} = ?$$



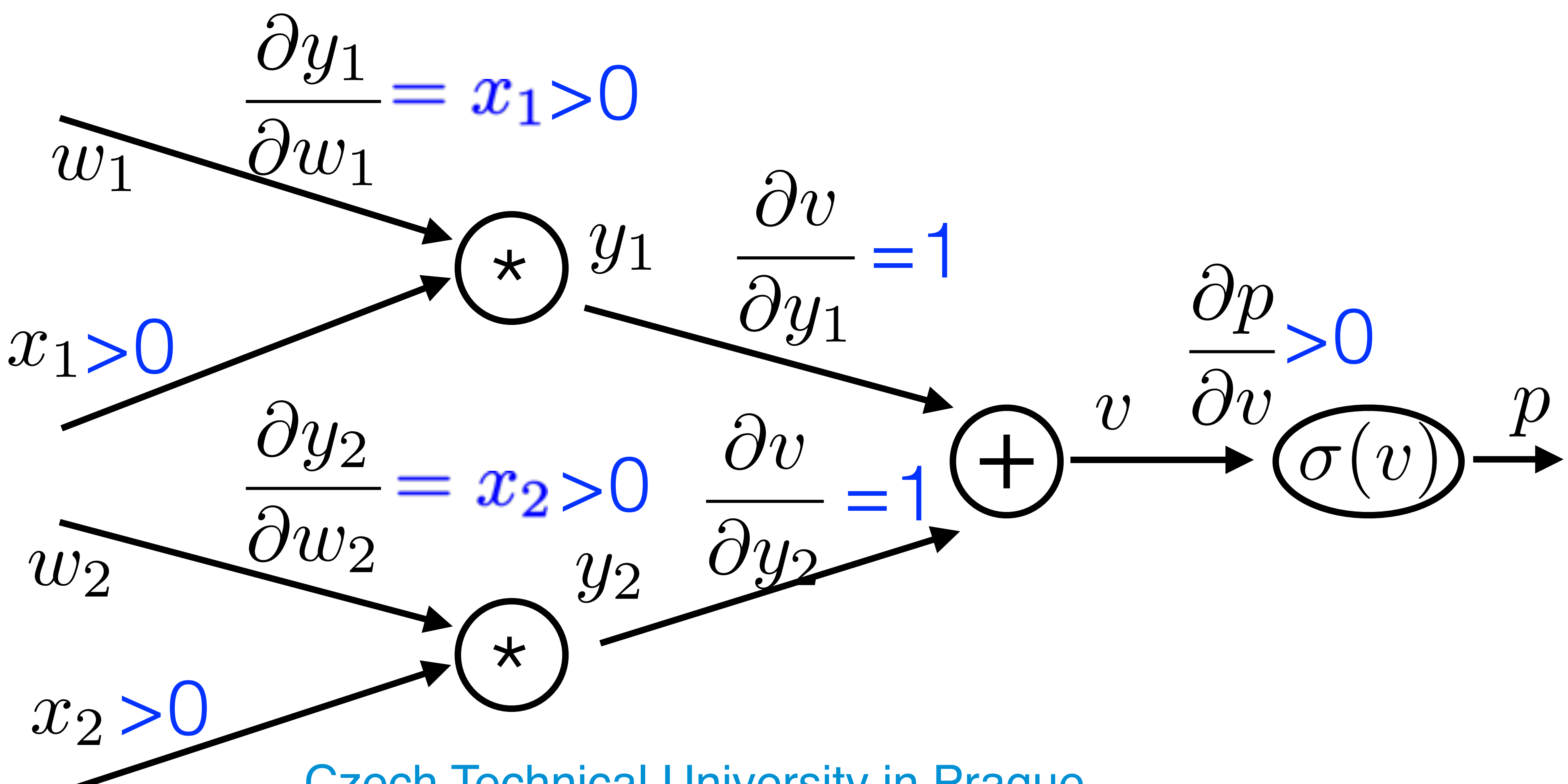
- what happens when sigmoid input is only positive?

Sigmoid

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



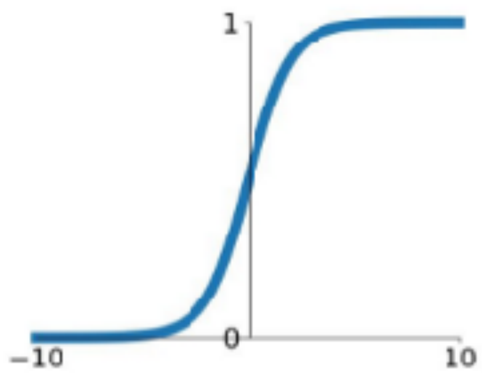
$$\frac{\partial p}{\partial w_1} = x_1 \cdot 1 \cdot \frac{\partial p}{\partial v} > 0 \qquad \frac{\partial p}{\partial w_2} = x_2 \cdot 1 \cdot \frac{\partial p}{\partial v} > 0$$



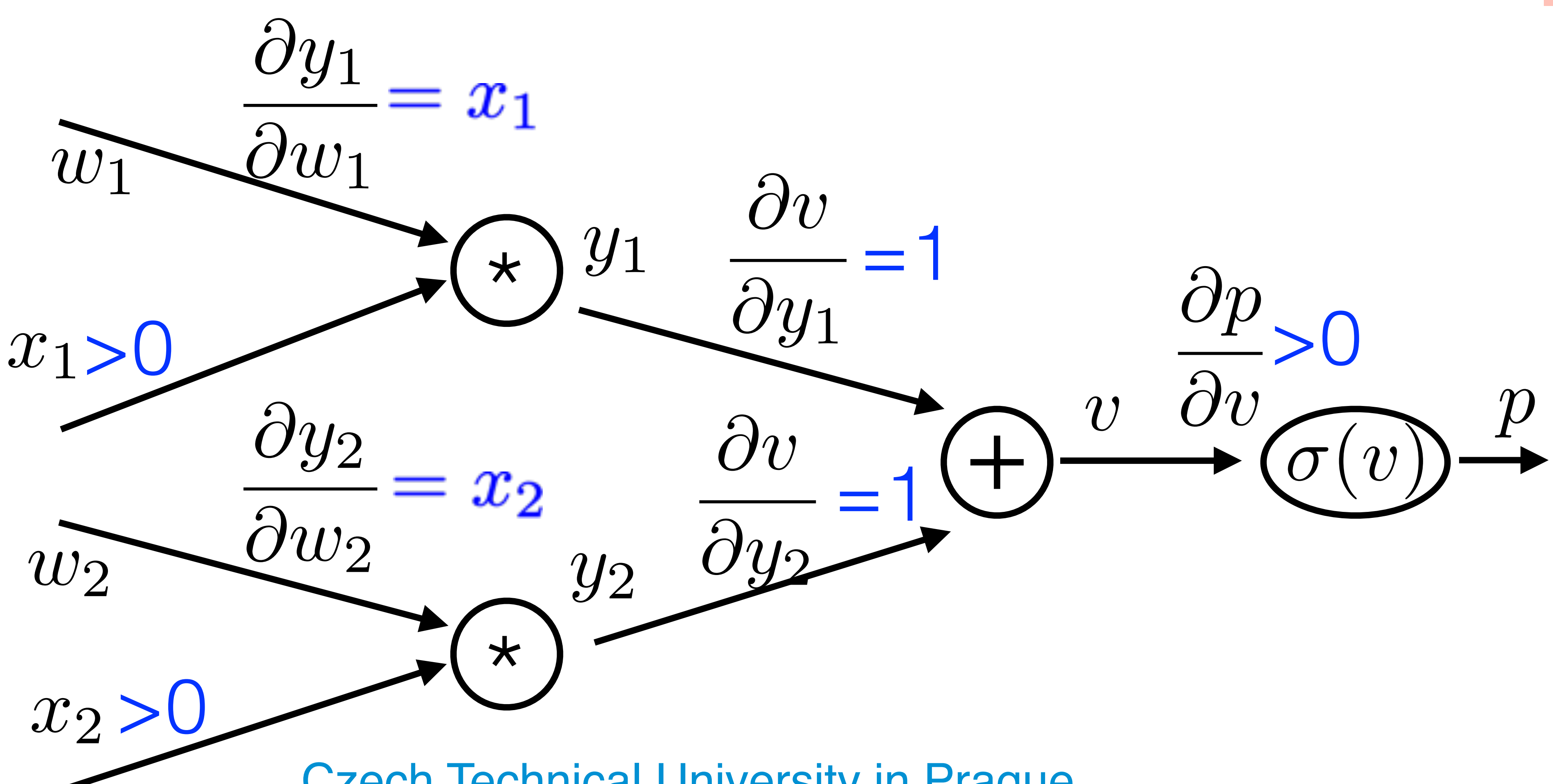
- what happens when sigmoid input is only positive?

Sigmoid

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



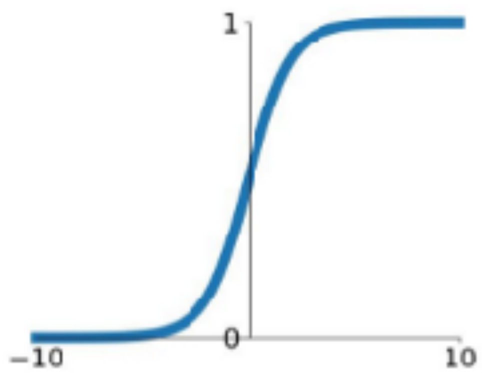
$$\frac{\partial p}{\partial w_1} = x_1 \cdot 1 \cdot \frac{\partial p}{\partial v} > 0 \qquad \frac{\partial p}{\partial w_2} = x_2 \cdot 1 \cdot \frac{\partial p}{\partial v} > 0 \Rightarrow \frac{\partial p}{\partial \mathbf{w}} > 0$$



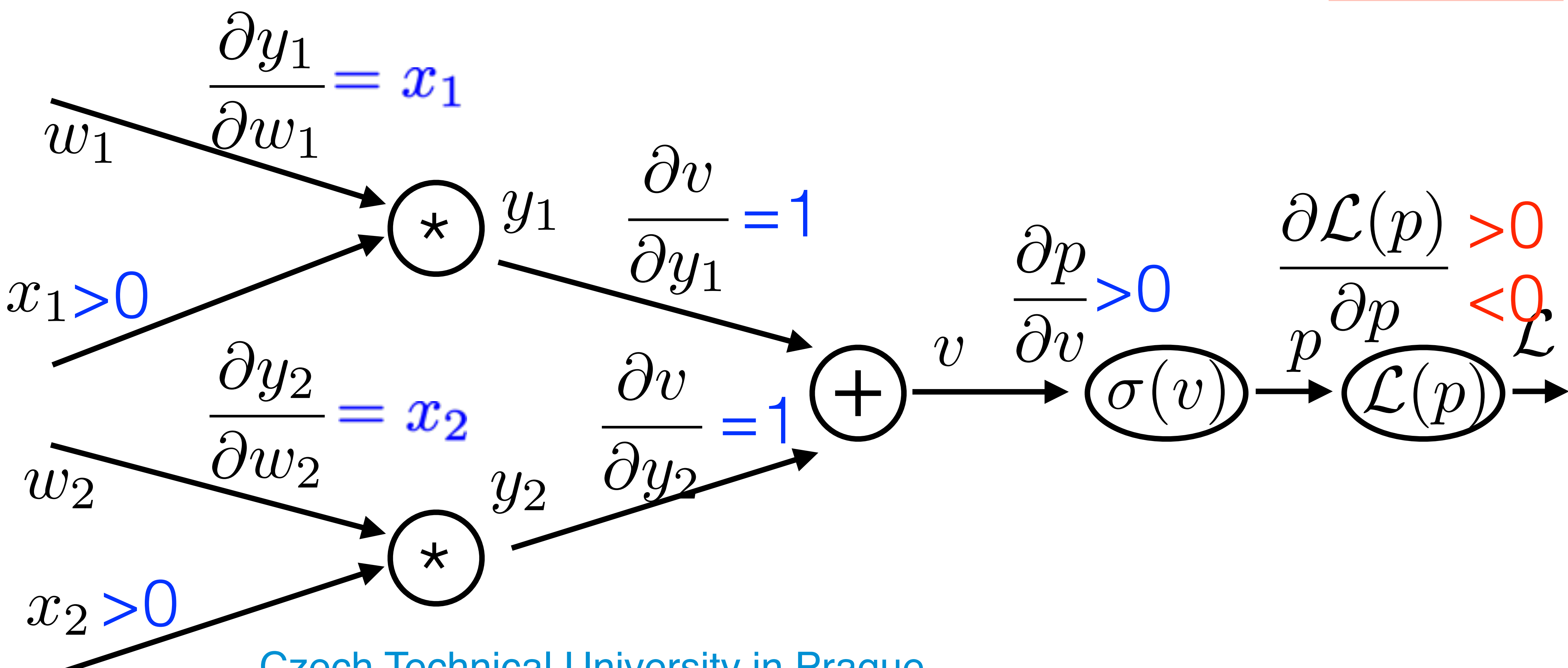
- what happens when sigmoid input is only positive?

Sigmoid

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



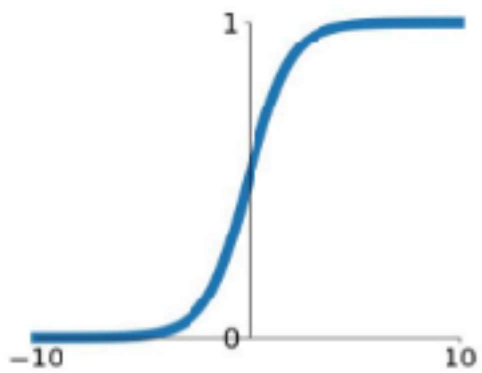
$$\frac{\partial p}{\partial w_1} = x_1 \cdot 1 \cdot \frac{\partial p}{\partial v} > 0 \qquad \frac{\partial p}{\partial w_2} = x_2 \cdot 1 \cdot \frac{\partial p}{\partial v} > 0 \Rightarrow \frac{\partial p}{\partial \mathbf{w}} > 0$$



- what happens when sigmoid input is only positive?

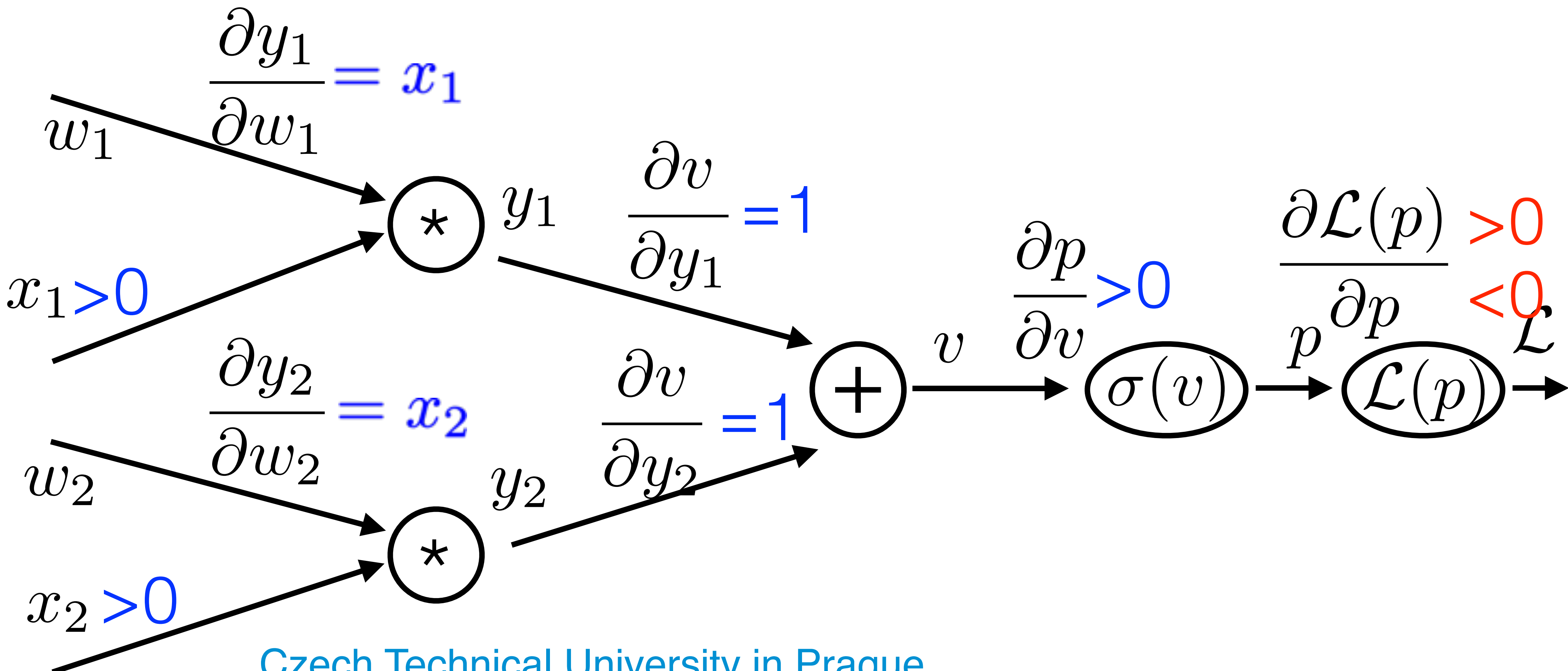
Sigmoid

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



$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}(p)}{\partial p}.$$

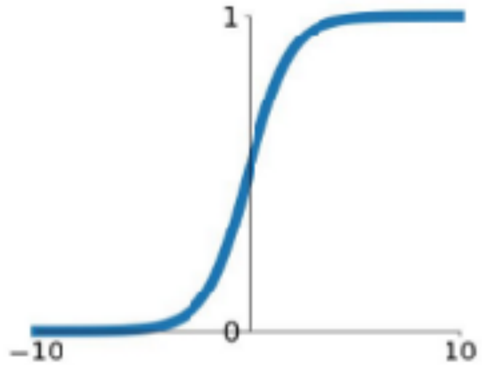
$$\frac{\partial p}{\partial w_1} = x_1 \cdot 1 \cdot \frac{\partial p}{\partial v} > 0 \qquad \frac{\partial p}{\partial w_2} = x_2 \cdot 1 \cdot \frac{\partial p}{\partial v} > 0 \Rightarrow \frac{\partial p}{\partial \mathbf{w}} > 0$$



- what happens when sigmoid input is only positive?

Sigmoid

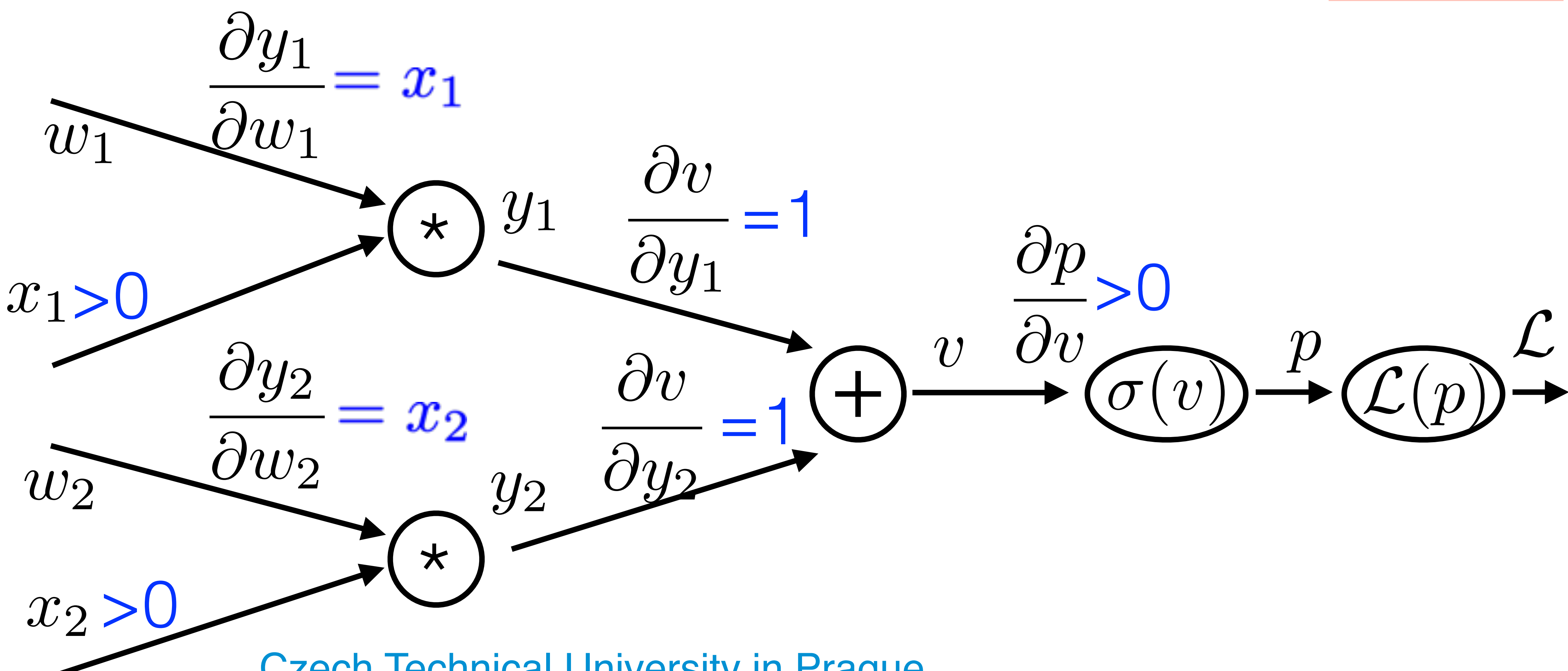
$\sigma(x) = \frac{1}{1+e^{-x}}$



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

$$\frac{\partial p}{\partial w_1} = x_1 \cdot 1 \cdot \frac{\partial p}{\partial v} \quad >0$$

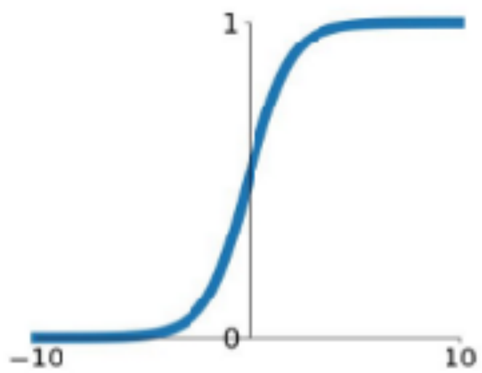
$$\frac{\partial p}{\partial w_2} = x_2 \cdot 1 \cdot \frac{\partial p}{\partial v} \quad >0 \quad => \quad \frac{\partial p}{\partial \mathbf{w}} > 0$$



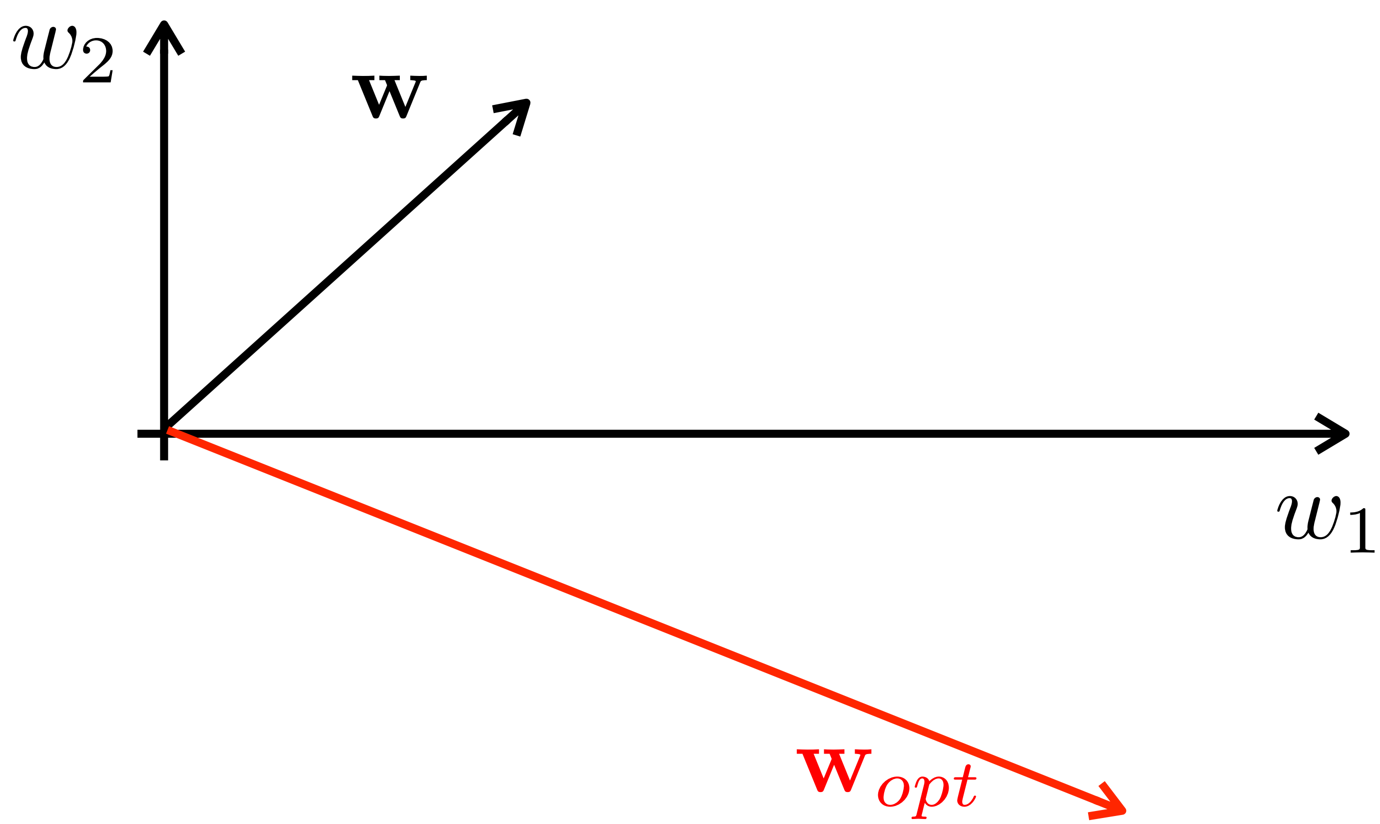
- what happens when sigmoid input is only positive?

Sigmoid

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



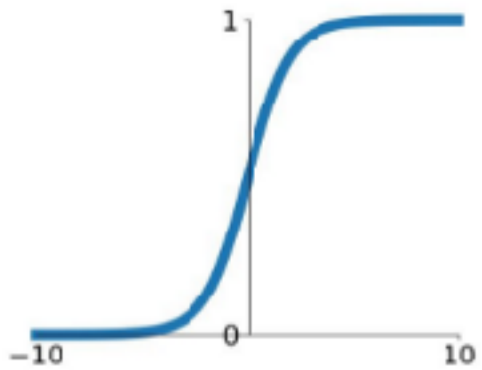
$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}(p)}{\partial p} \cdot \frac{\partial p}{\partial \mathbf{w}} \quad \begin{matrix} >0 \\ <0 \end{matrix}$$



- what happens when sigmoid input is only positive?

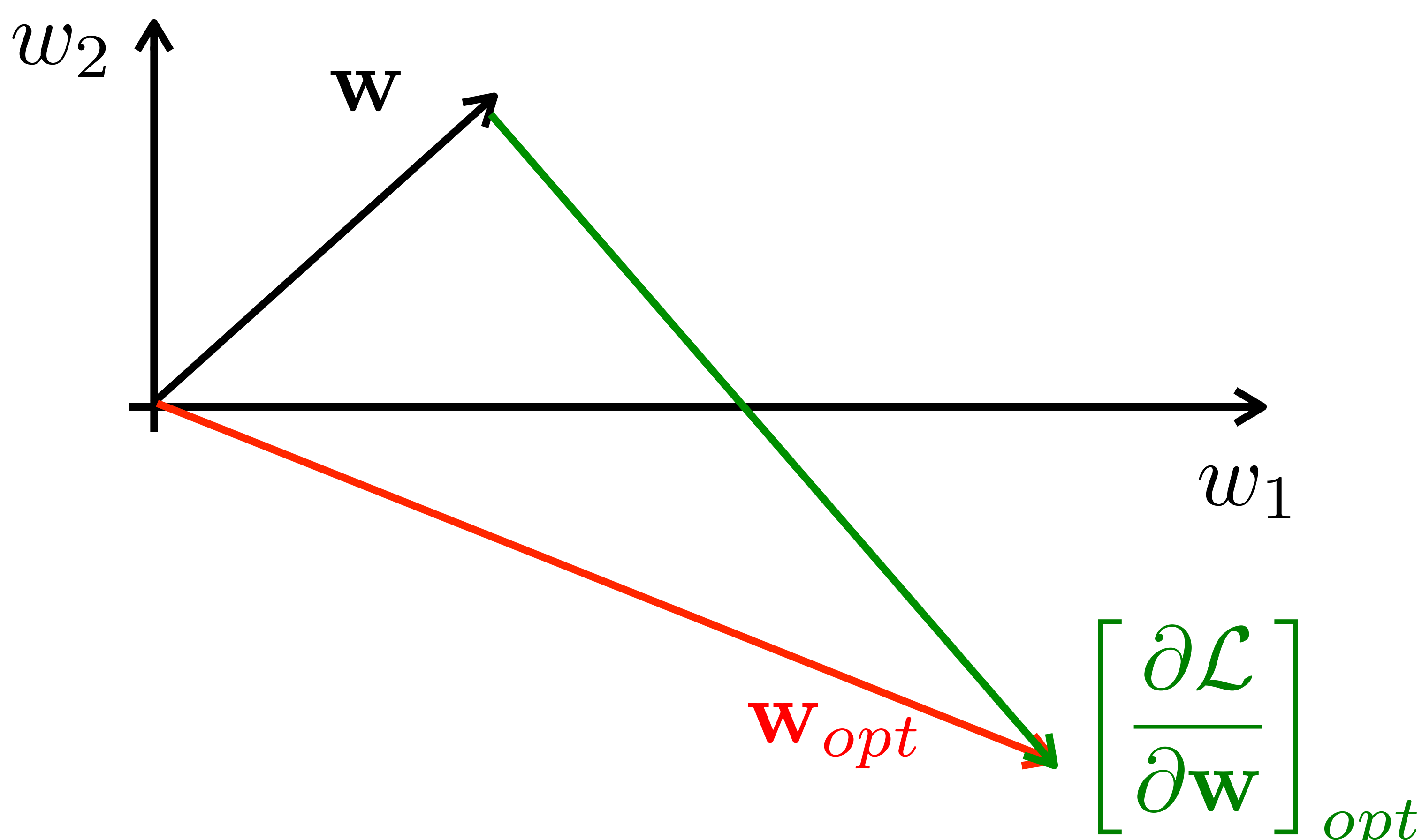
Sigmoid

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



$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}(p)}{\partial p} \cdot \frac{\partial p}{\partial \mathbf{w}}$$

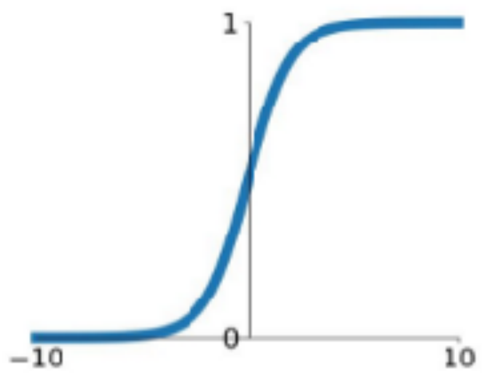
>0
<0



- what happens when sigmoid input is only positive?

Sigmoid

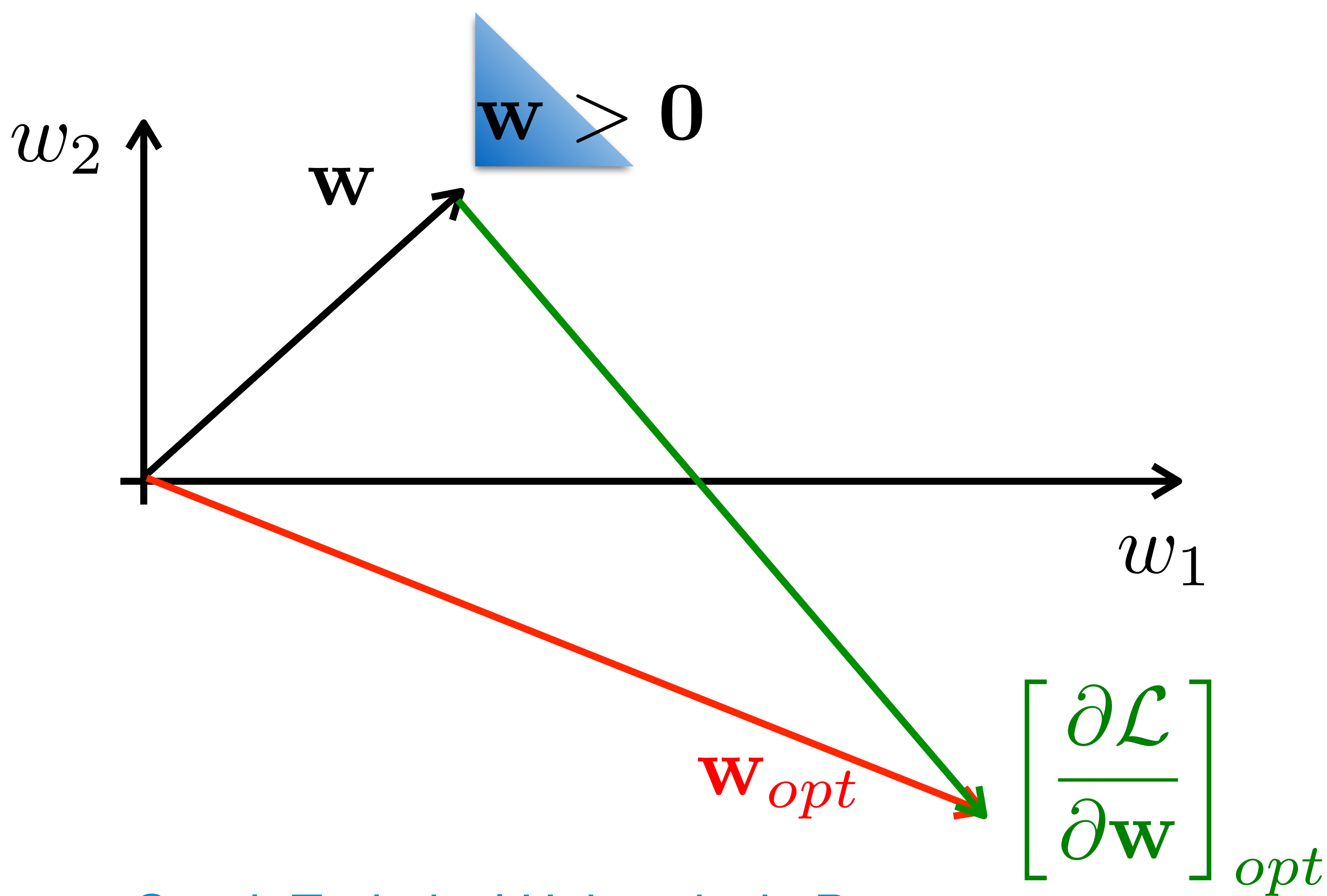
$$\sigma(x) = \frac{1}{1+e^{-x}}$$



$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}(p)}{\partial p} \cdot \frac{\partial p}{\partial \mathbf{w}}$$

>0

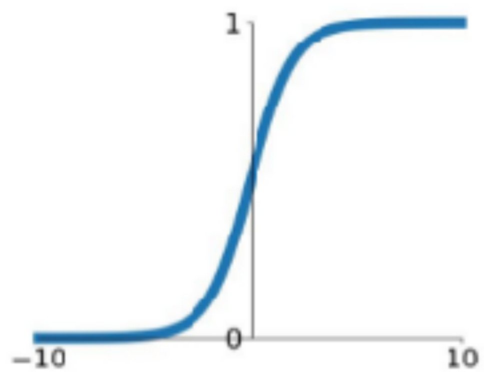
<0



- what happens when sigmoid input is only positive?

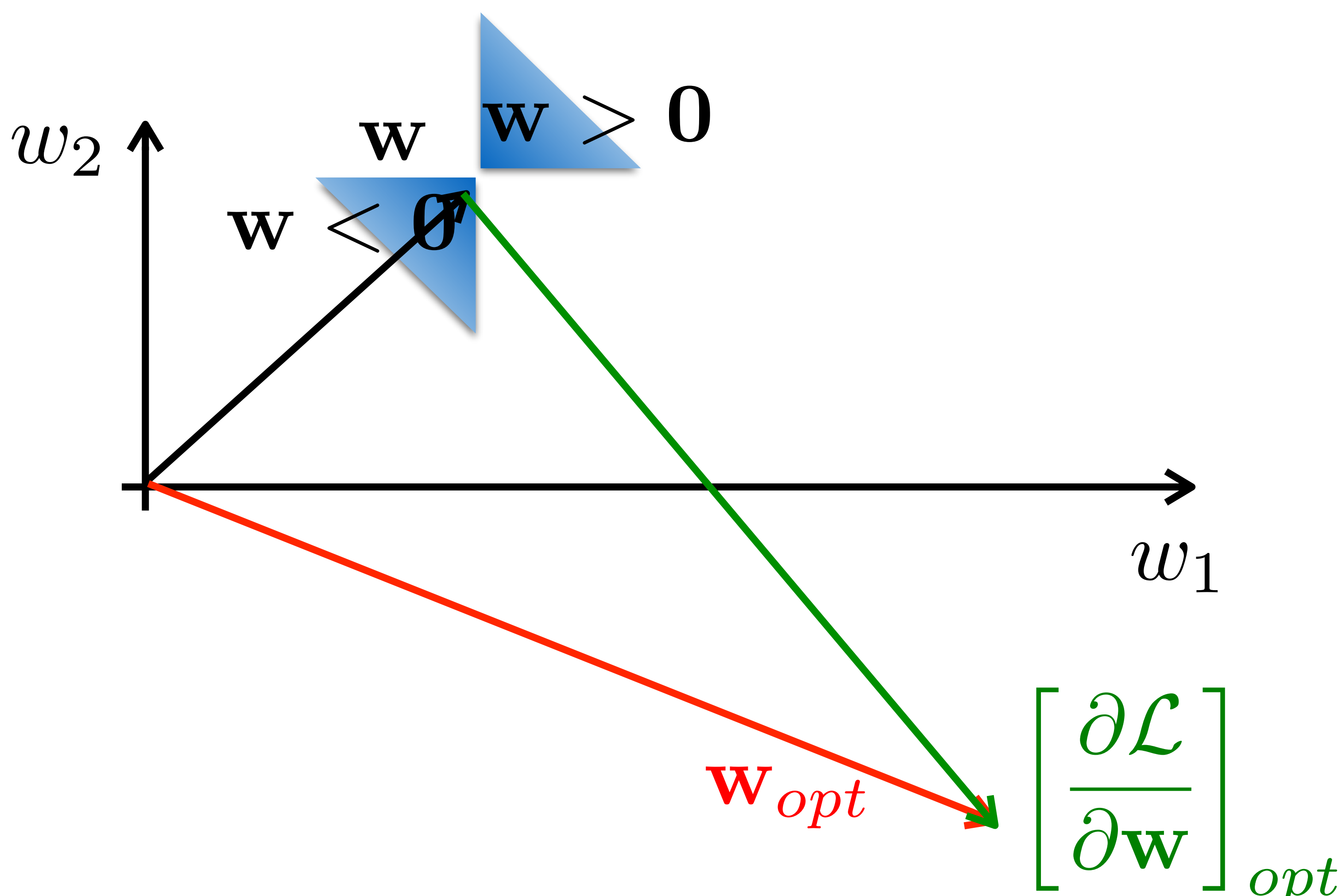
Sigmoid

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



$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}(p)}{\partial p} \cdot \frac{\partial p}{\partial \mathbf{w}}$$

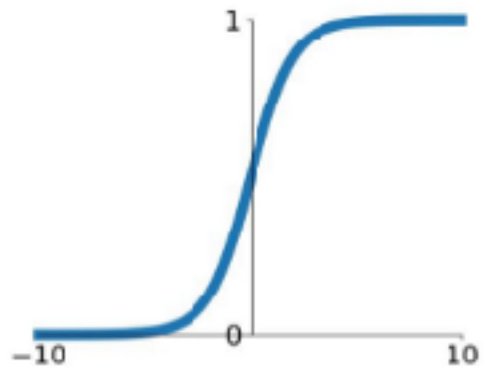
>0
<0



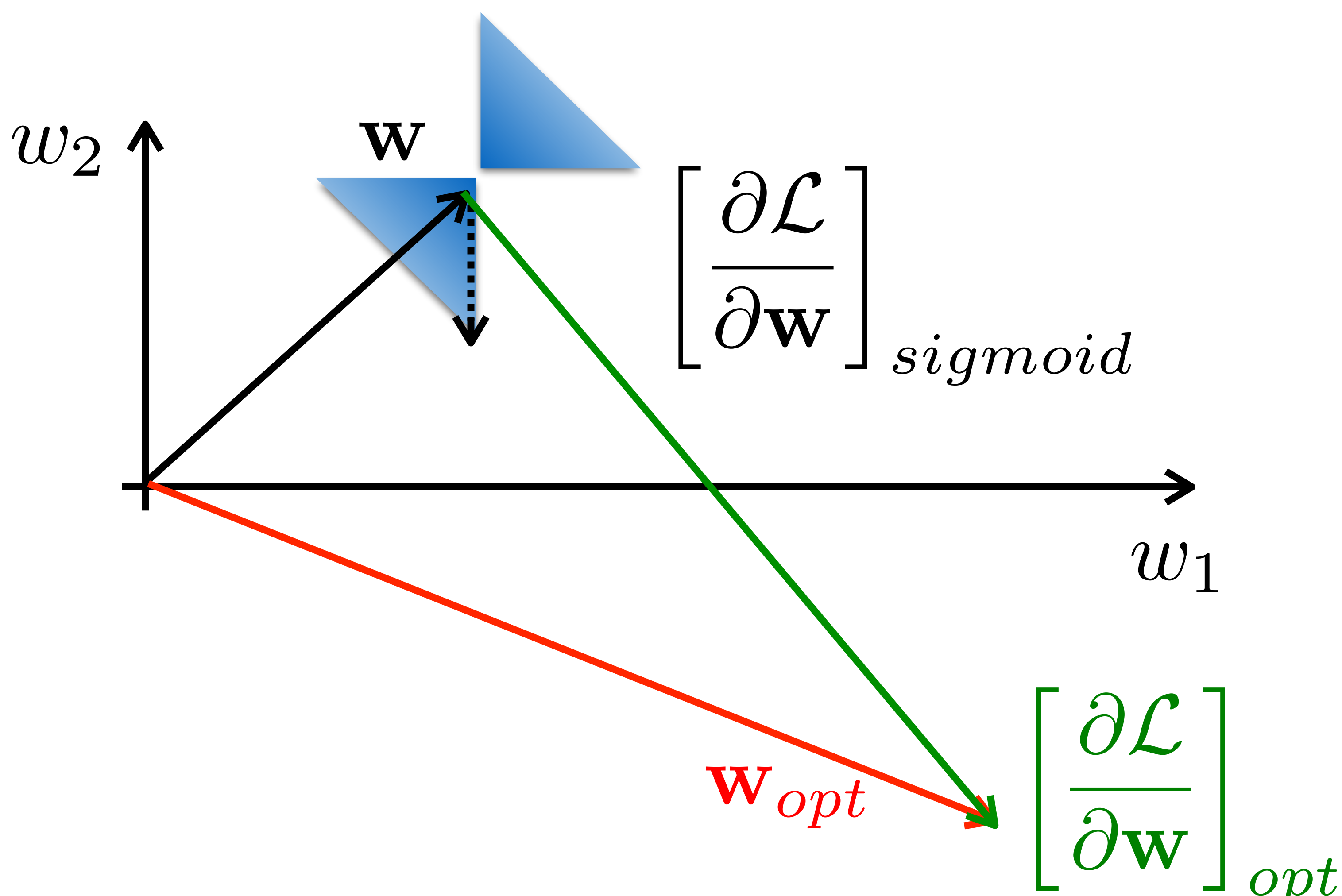
- what happens when sigmoid input is only positive?

Sigmoid

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



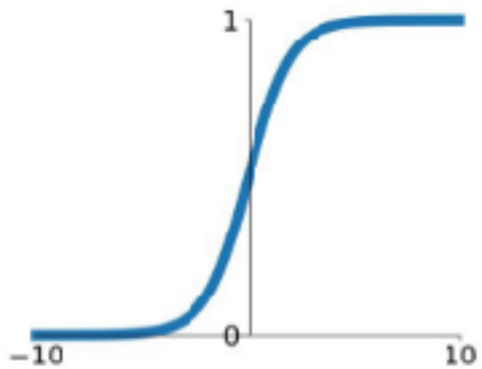
$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}(p)}{\partial p} \cdot \frac{\partial p}{\partial \mathbf{w}} \quad \begin{matrix} >0 \\ <0 \end{matrix}$$



- what happens when sigmoid input is only positive?

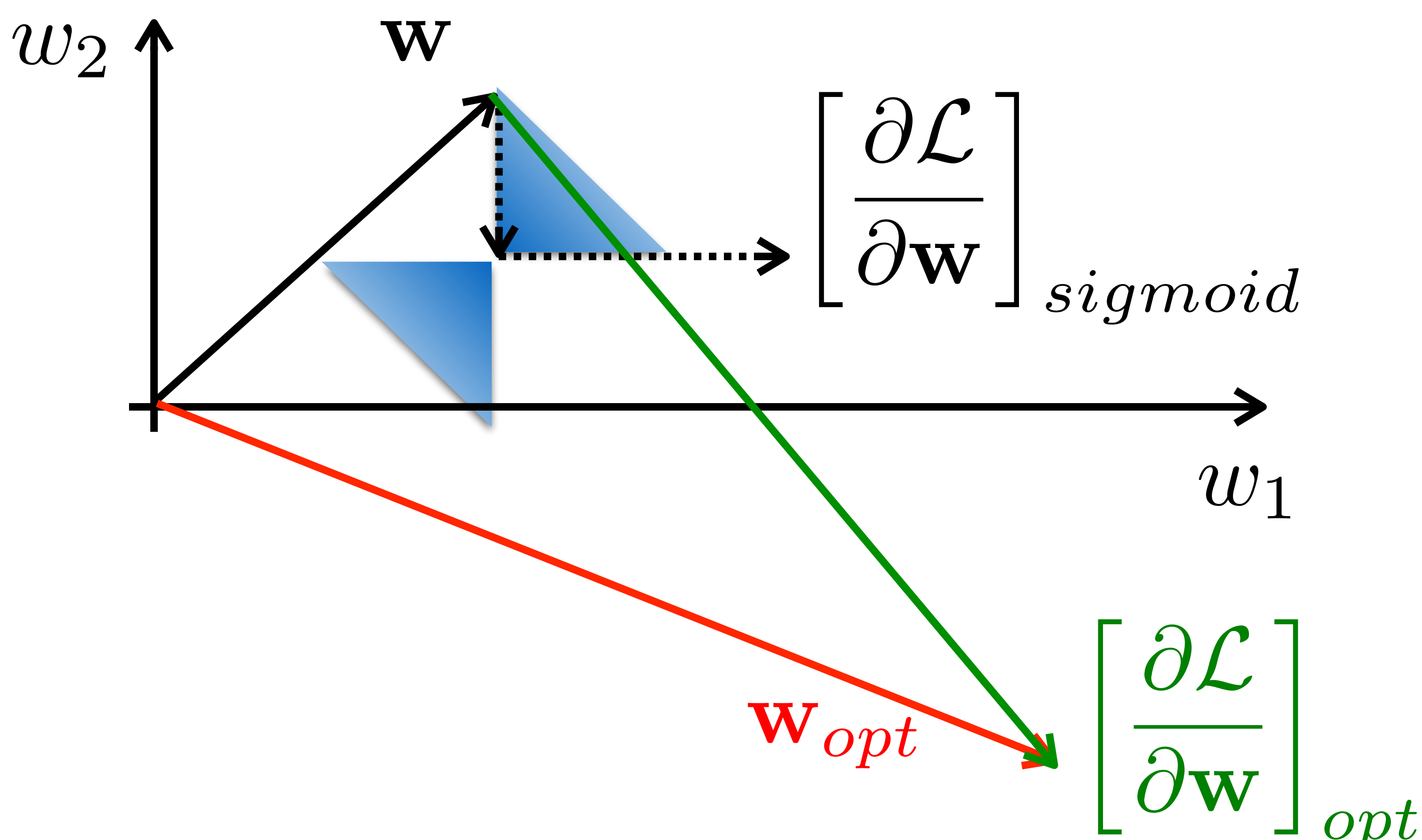
Sigmoid

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



$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}(p)}{\partial p} \cdot \frac{\partial p}{\partial \mathbf{w}}$$

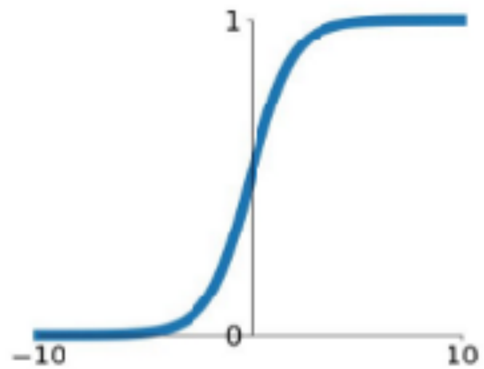
>0
<0



- what happens when sigmoid input is only positive?

Sigmoid

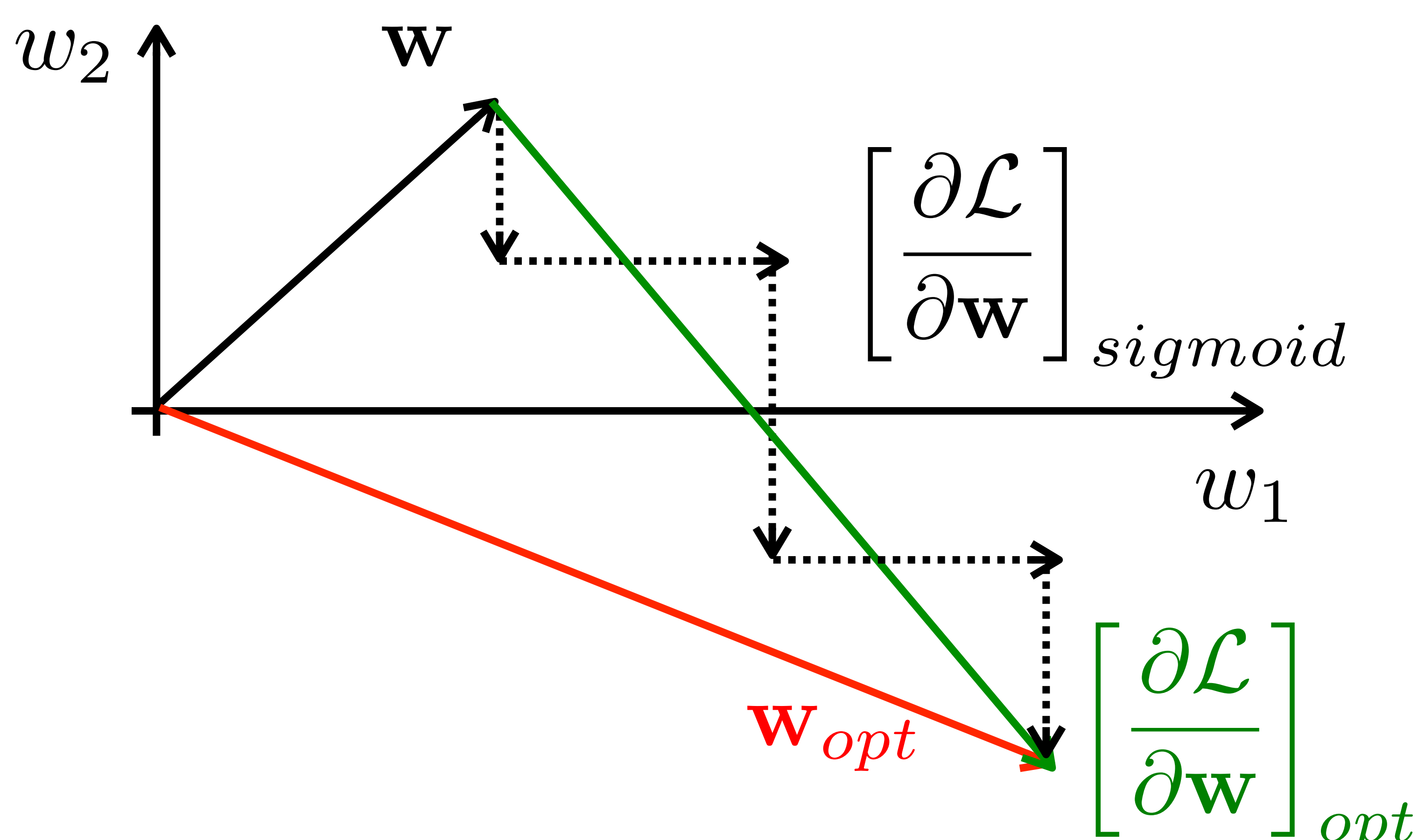
$$\sigma(x) = \frac{1}{1+e^{-x}}$$



$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}(p)}{\partial p} \cdot \frac{\partial p}{\partial \mathbf{w}}$$

>0

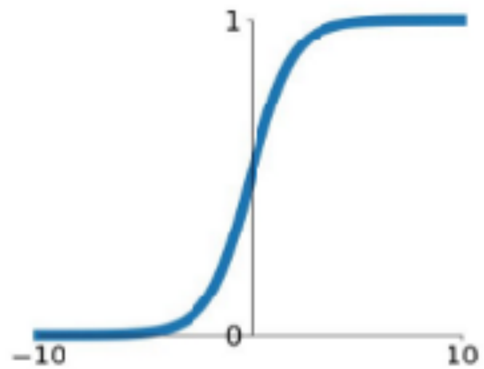
<0



- what happens when sigmoid input is only positive?

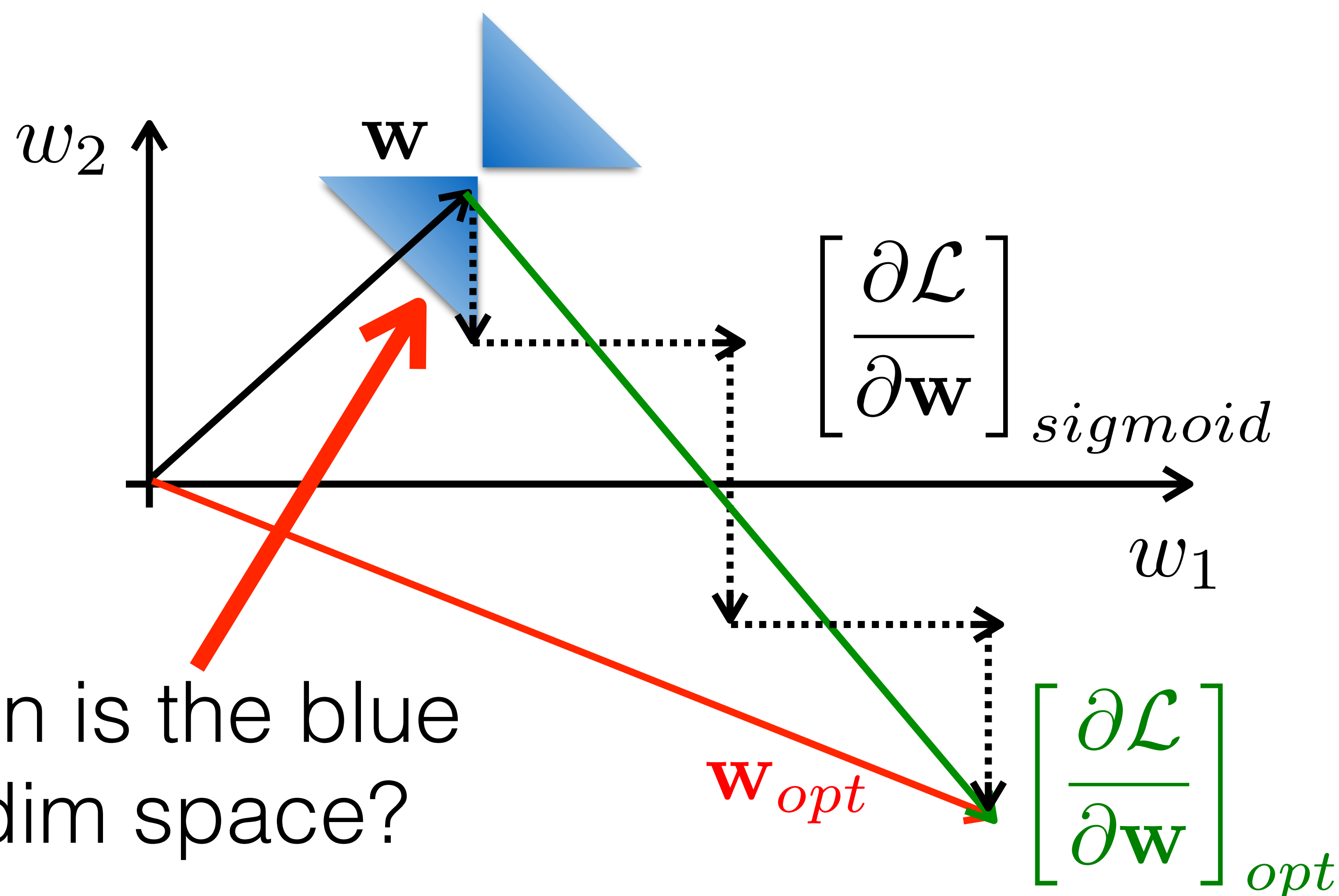
Sigmoid

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



$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}(p)}{\partial p} \cdot \frac{\partial p}{\partial \mathbf{w}}$$

>0
<0

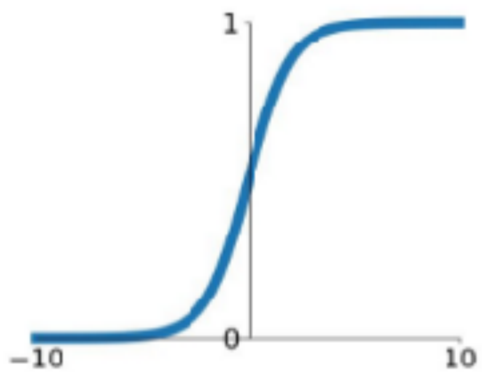


how big fraction is the blue region in 10-dim space?

- what happens when sigmoid input is only positive?

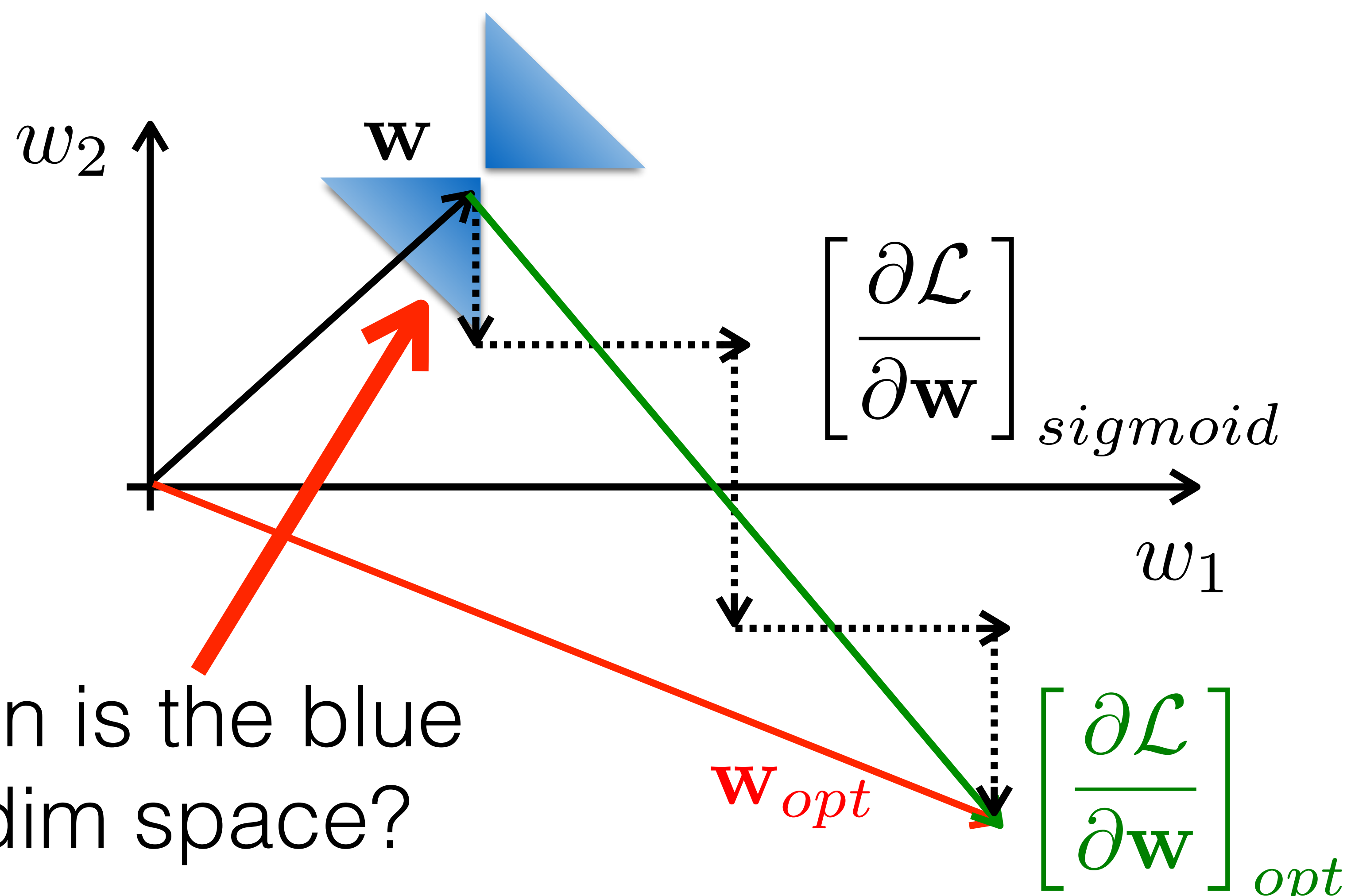
Sigmoid

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



$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}(p)}{\partial p} \cdot \frac{\partial p}{\partial \mathbf{w}}$$

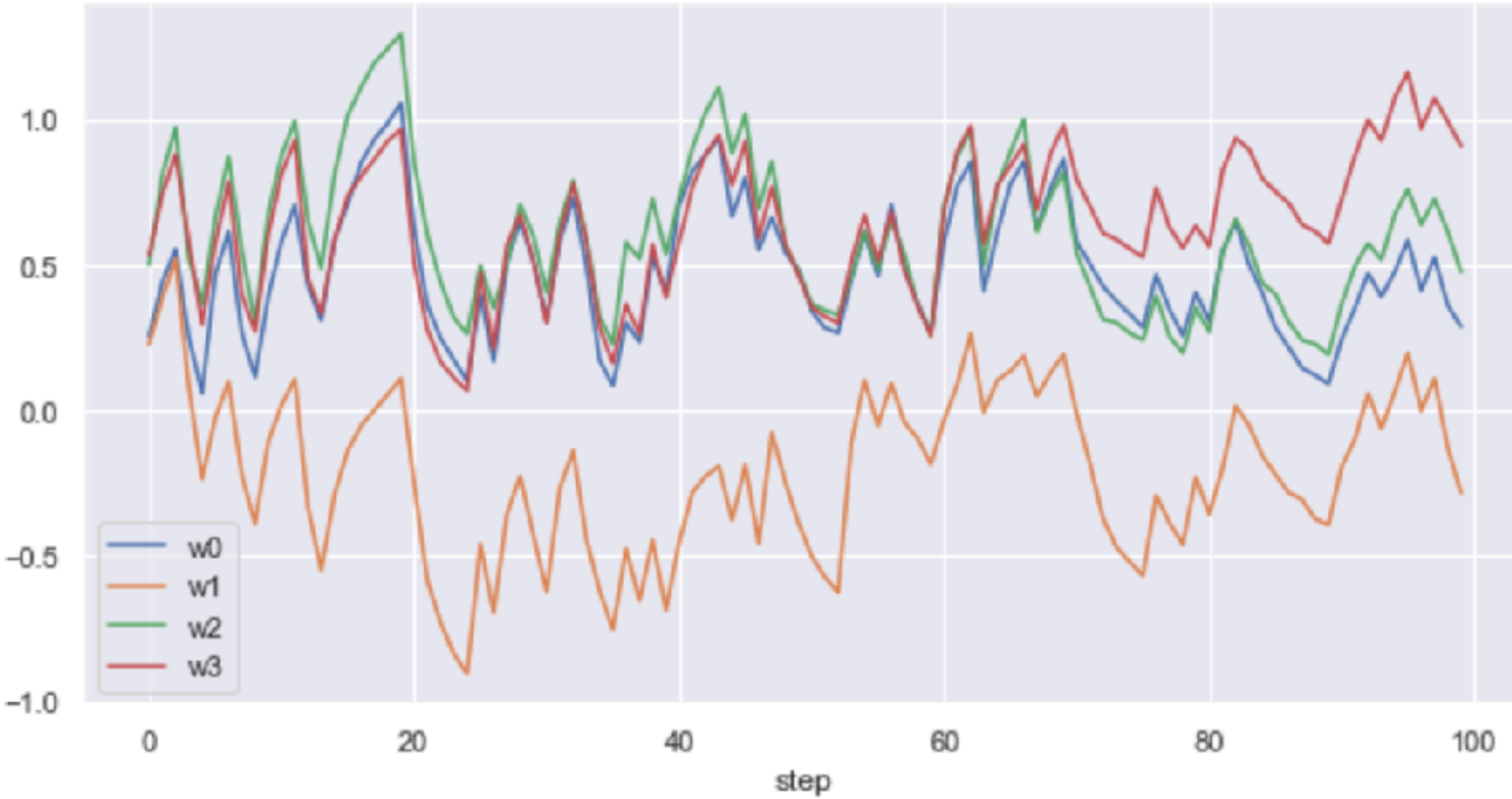
>0
<0



2/(2^10)

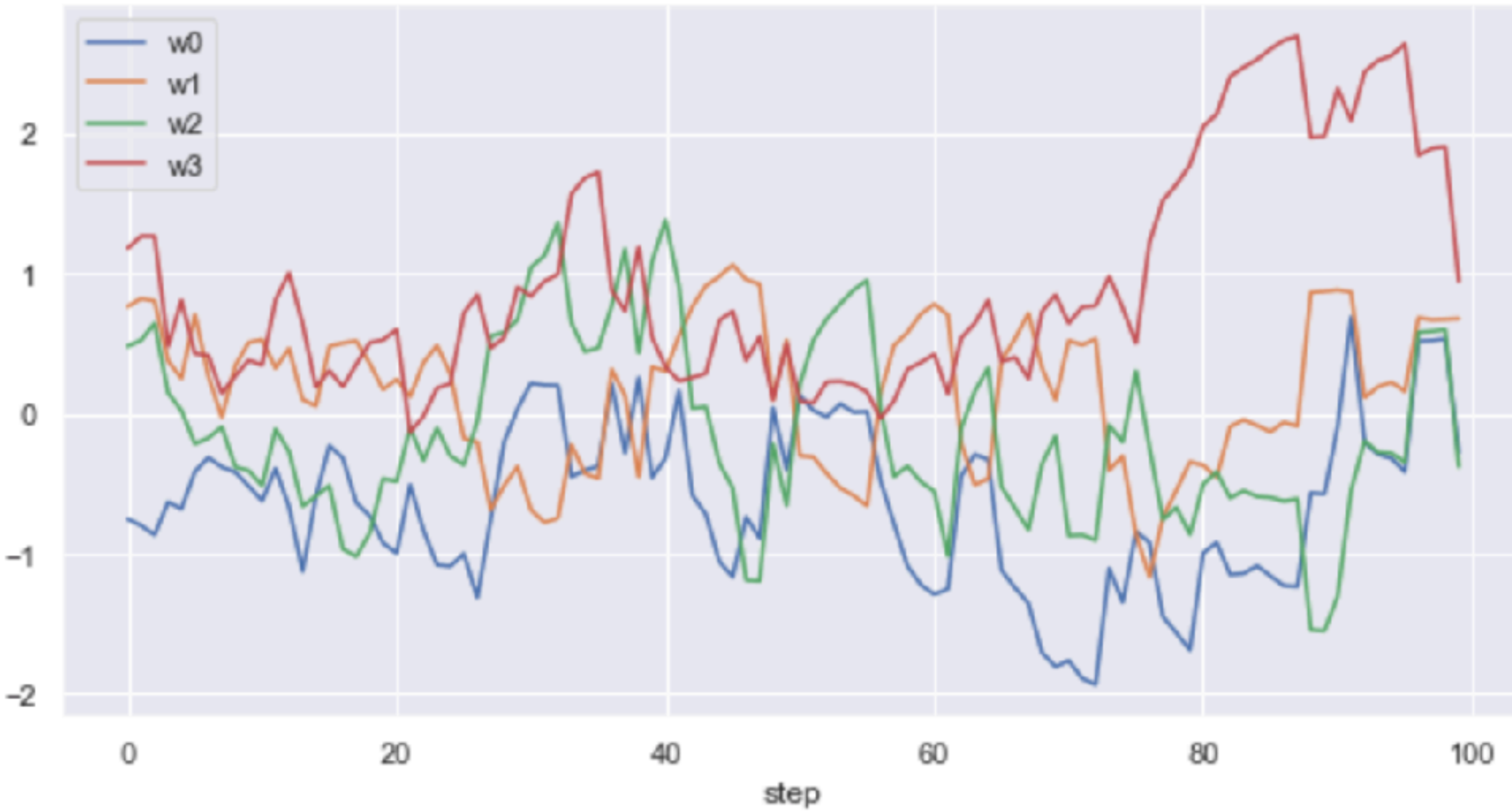
how big fraction is the blue region in 10-dim space?

- what happens when sigmoid input is only positive?



sigmoid activation function

- what happens when sigmoid input is only positive?

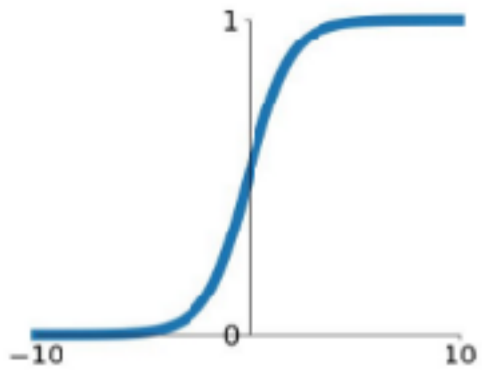


tanh activation function

Activation functions

Sigmoid

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

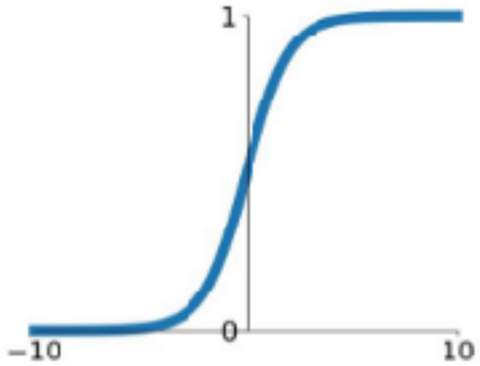


- zero gradient when saturated
- not zero-centered (pos. output)
- computationally expensive

Activation functions

Sigmoid

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



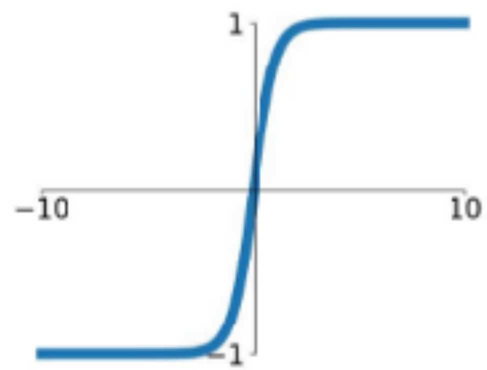
- zero gradient when saturated
- not zero-centered (pos. output)
- computationally expensive

PyTorch: `nn.Sigmoid()`

Activation functions

tanh

$$\tanh(x)$$



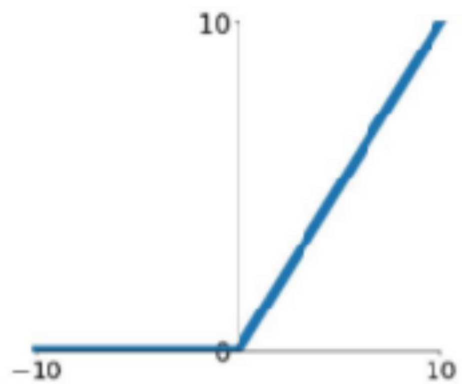
- zero gradient when saturated
- ~~not zero-centered (only positive ouputs)~~
- computationally expensive

- PyTorch: `nn.Tanh()`

Activation functions

ReLU

$\max(0, x)$



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

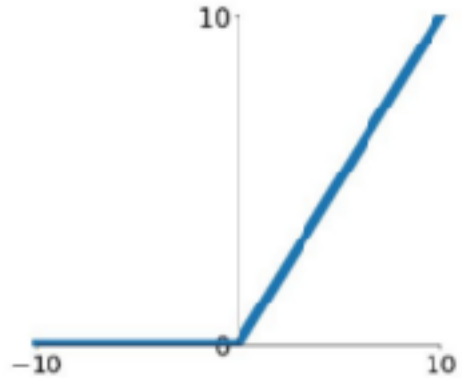
- PyTorch: `nn.ReLu()`

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

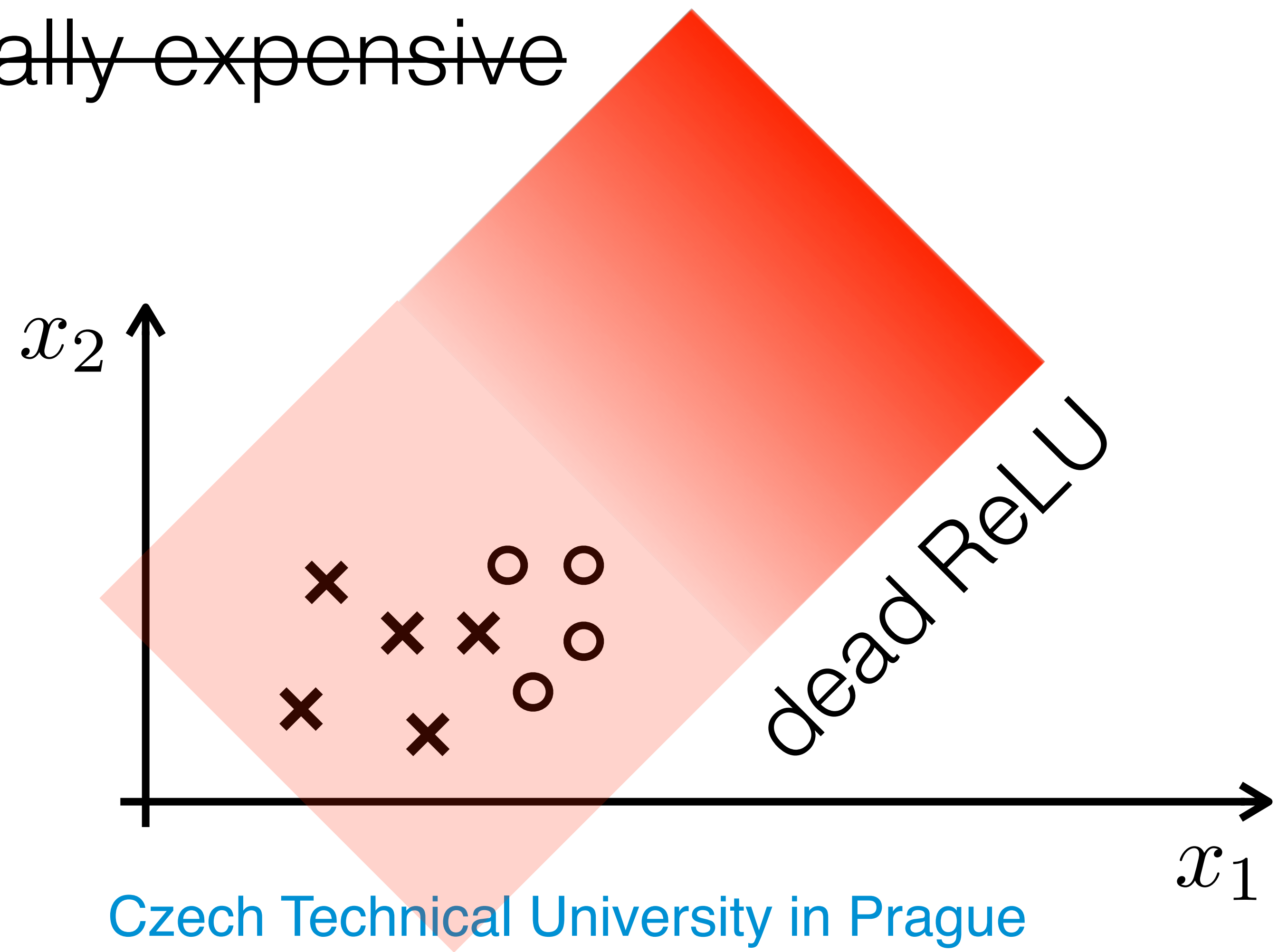
Activation functions

ReLU

$\max(0, x)$

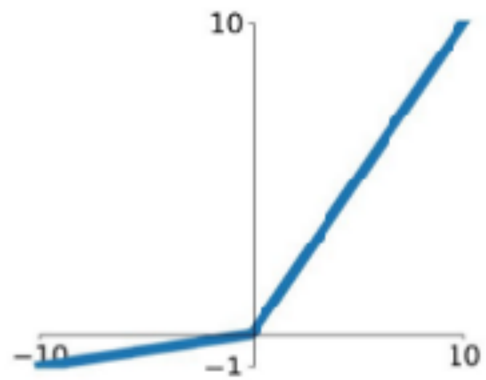


- ~~zero gradient when saturated~~ *(partially => dead ReLU!)*
- not zero-centered (only positive ouputs)
- ~~computationally expensive~~



Leaky ReLU

$\max(0.1x, x)$



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

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

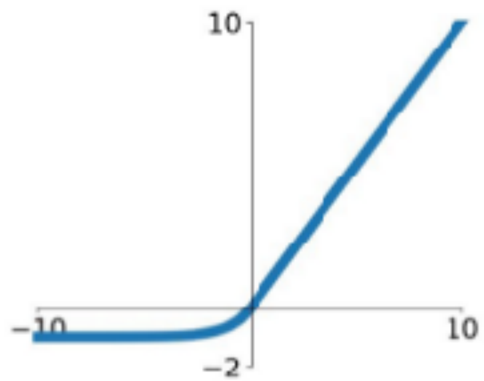
Small gradient for negative values give tiny chance to recover

- backprop: $$\frac{\partial \max(0.1x, x)}{\partial x} = \begin{cases} 0.1 & x < 0 \\ 1 & \text{otherwise} \end{cases}$$

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

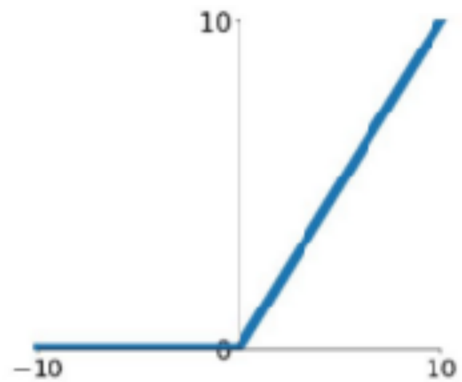
- PyTorch: `nn.LeakyReLU(alpha=1)`

Summary

- Use ReLU and avoid undesired properties by
  - good weight initialization
  - data preprocessing
  - batch normalization

ReLU

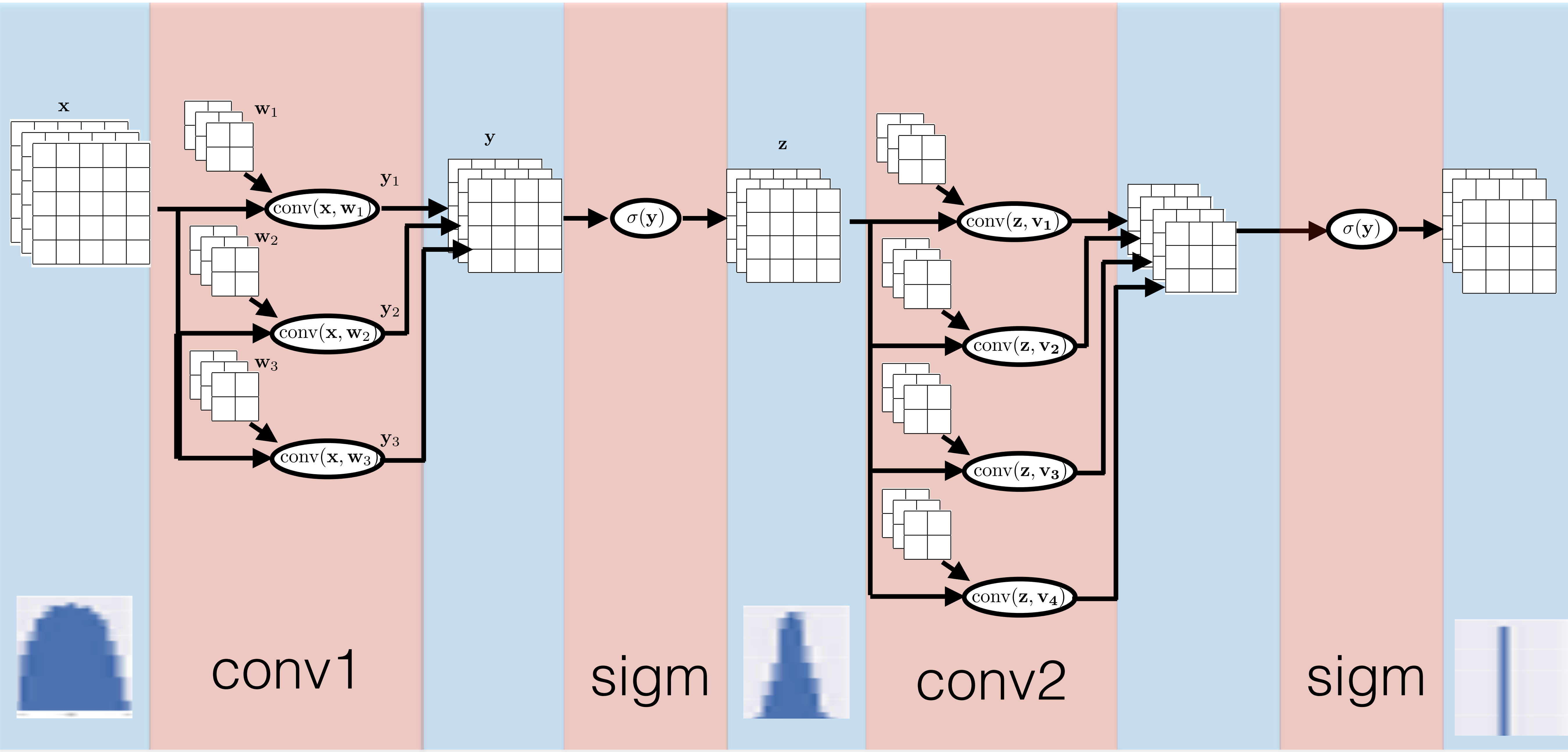
$\max(0, x)$



- Still you want to keep "reasonable values" to avoid:
  - diminishing/exploding gradient
  - dead ReLu or saturated sigmoid

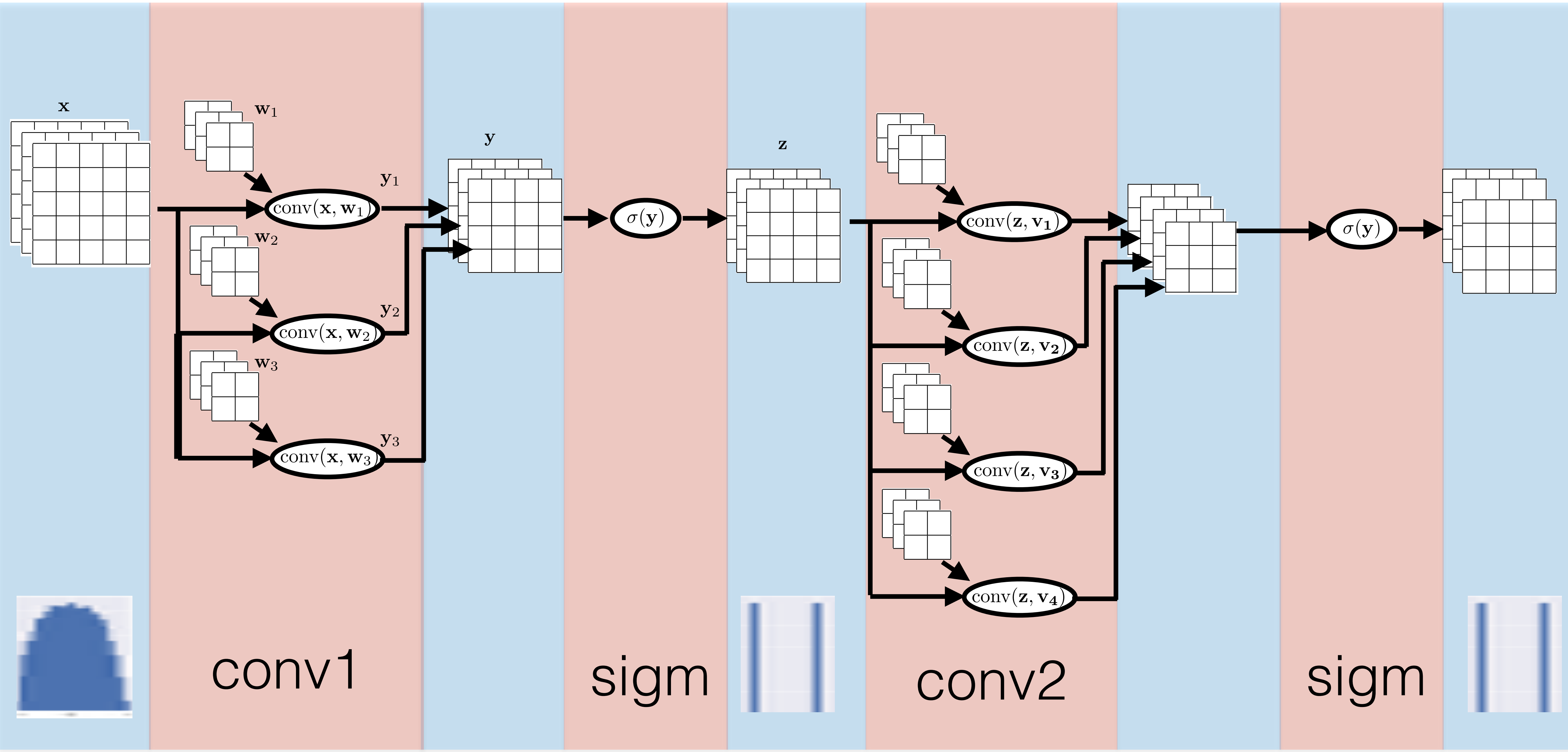
Learning

- what happens to **sigm outputs** when weights are **small**?



Learning

- what happens to **sigm outputs** when weights are **huge**?



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

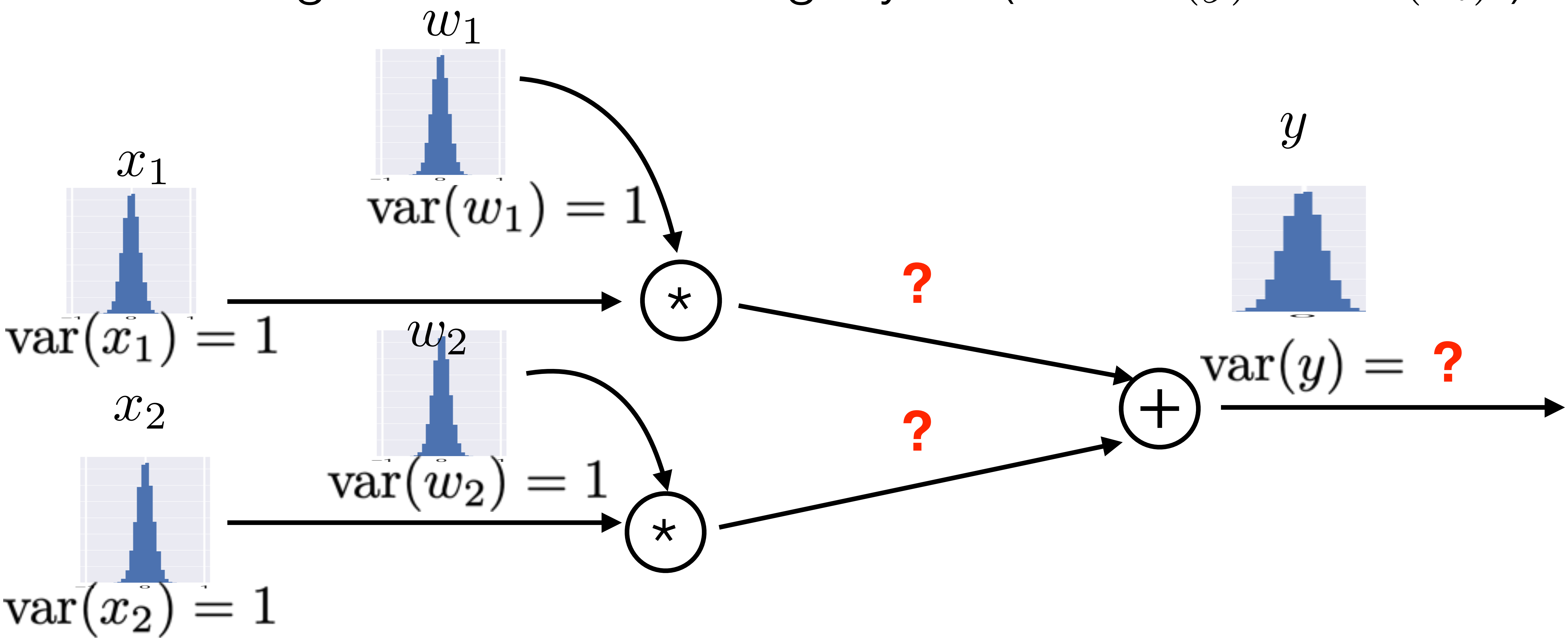
Data preprocessing & initializations

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

Data preprocessing & initializations

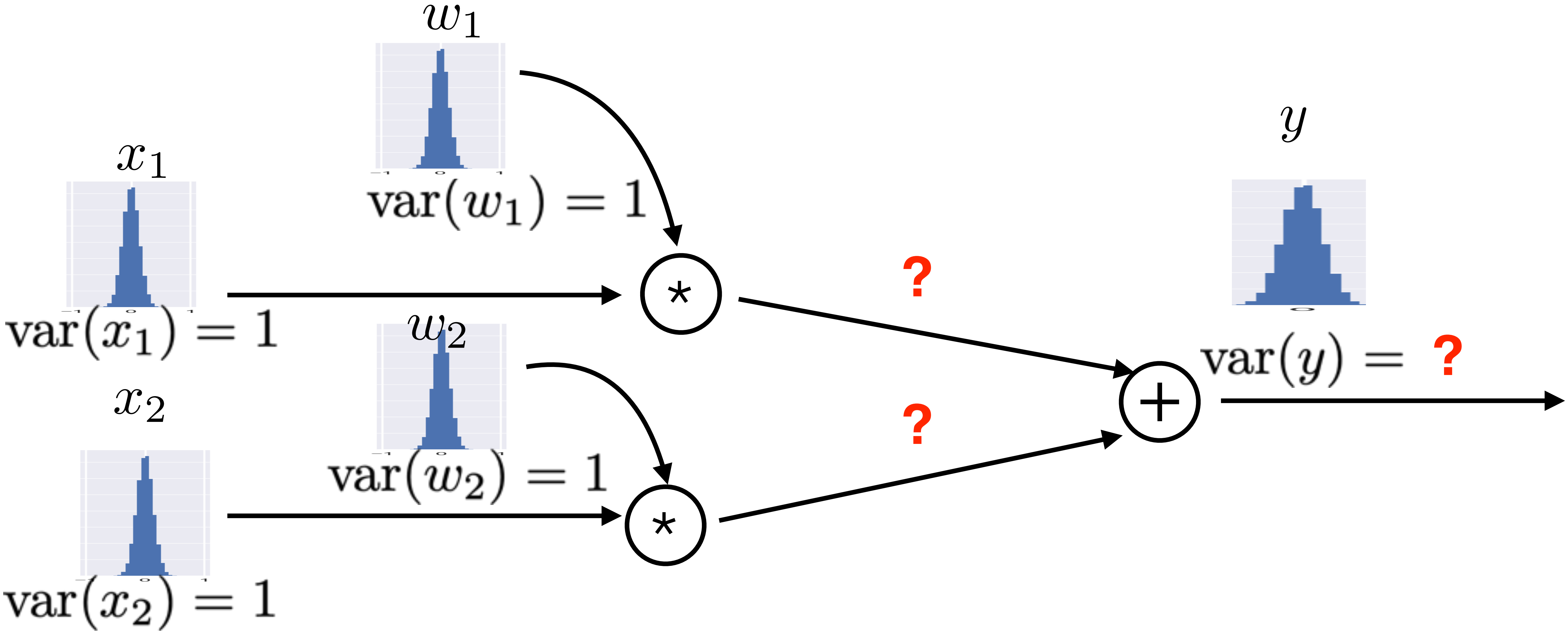
- Input preprocessing:
  - Pixels values shifted zero mean to avoid only positive inputs and the unwanted “zig-zag” behaviour
- Weight initialization:
  - $\mathbf{w} = 0$ all gradients the same
  - $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \sigma)$ diminishing/exploding values
  - $\mathbf{w}^{(i)} \sim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

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



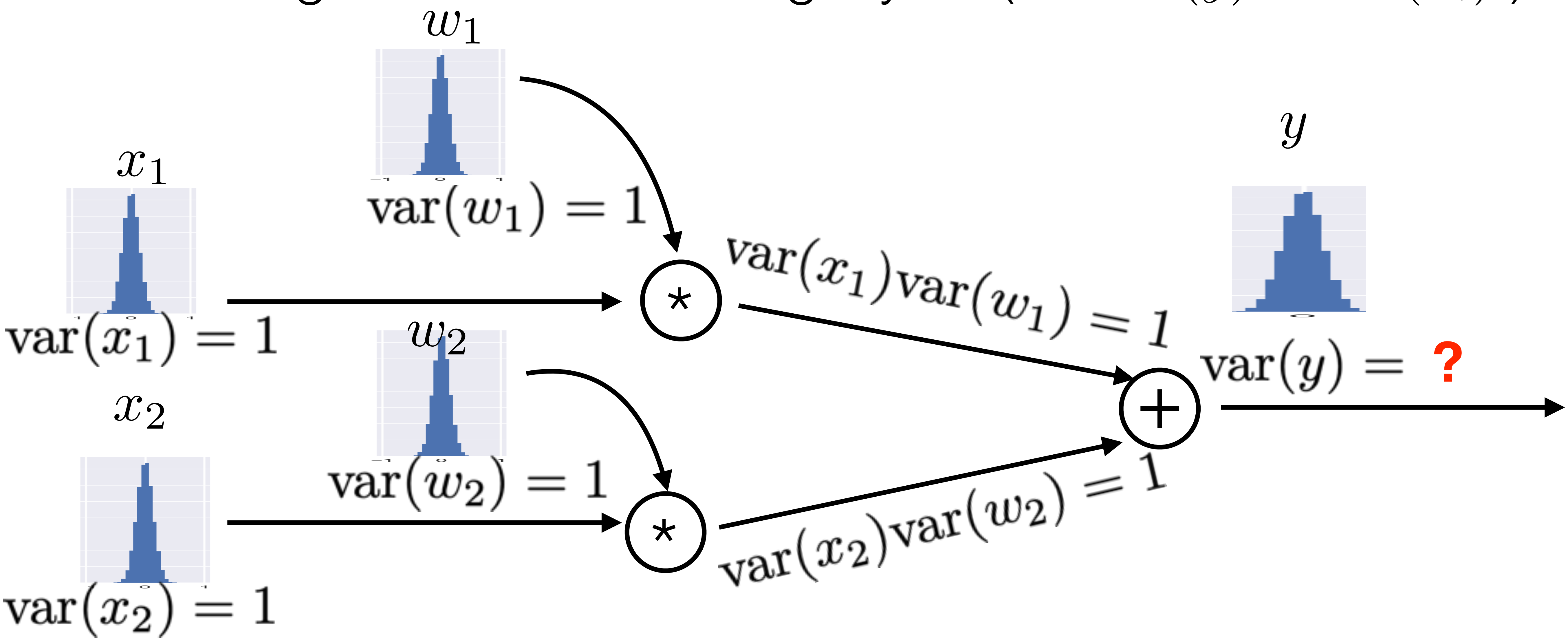
$$\mathrm{var}(x_1 w_1) = (\mathrm{var}(x_1) + \mu_{x_1}^2)(\mathrm{var}(w_1) + \mu_{w_1}^2) - \mu_{x_1}^2 \mu_{w_1}^2$$

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



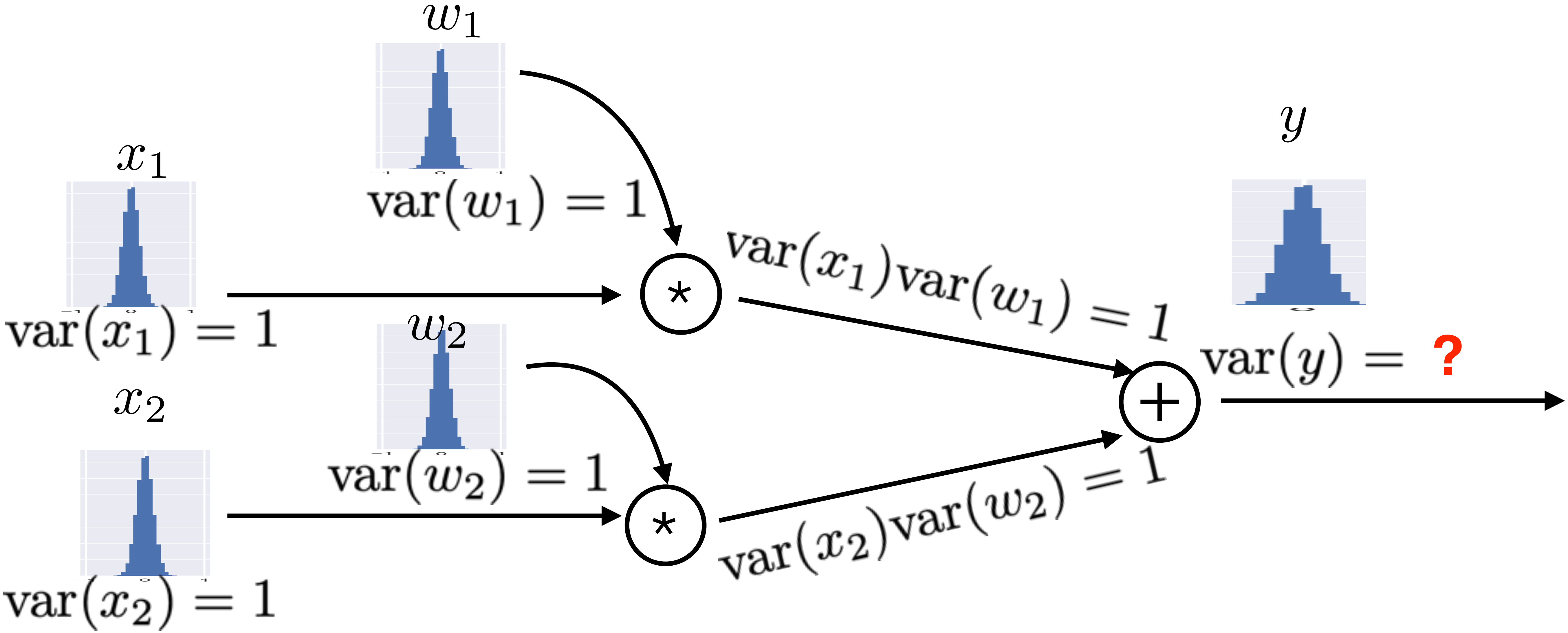
$$\mathrm{var}(x_1 w_1) = (\mathrm{var}(x_1) + \mu_{x_1}^2)(\mathrm{var}(w_1) + \mu_{w_1}^2) - \mu_{x_1}^2 \mu_{w_1}^2$$
$$= \mathrm{var}(x_1)\mathrm{var}(w_1) = 1$$

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



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

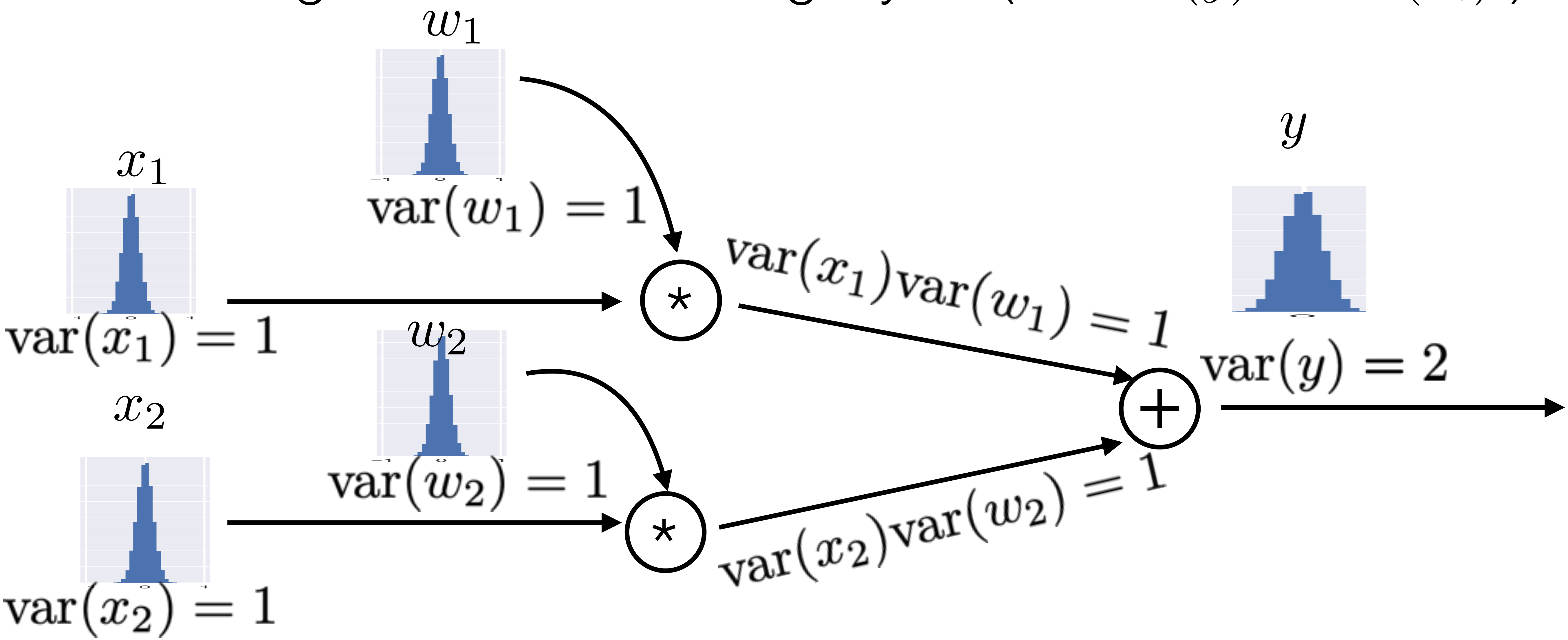
Preserve signal variance among layers (i.e. $\text{var}(y) = \text{var}(x_i)$)



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

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

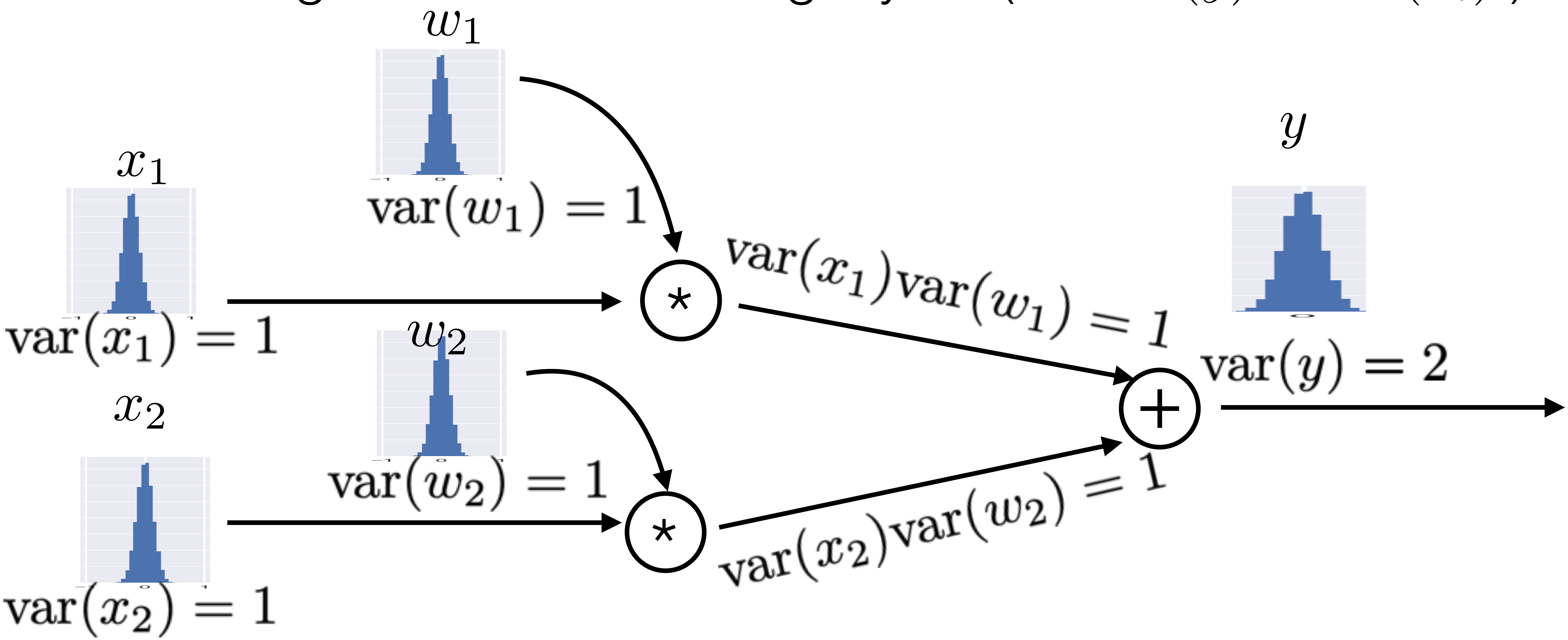
Preserve signal variance among layers (i.e. $\mathrm{var}(y) = \mathrm{var}(x_i)$)



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

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

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



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

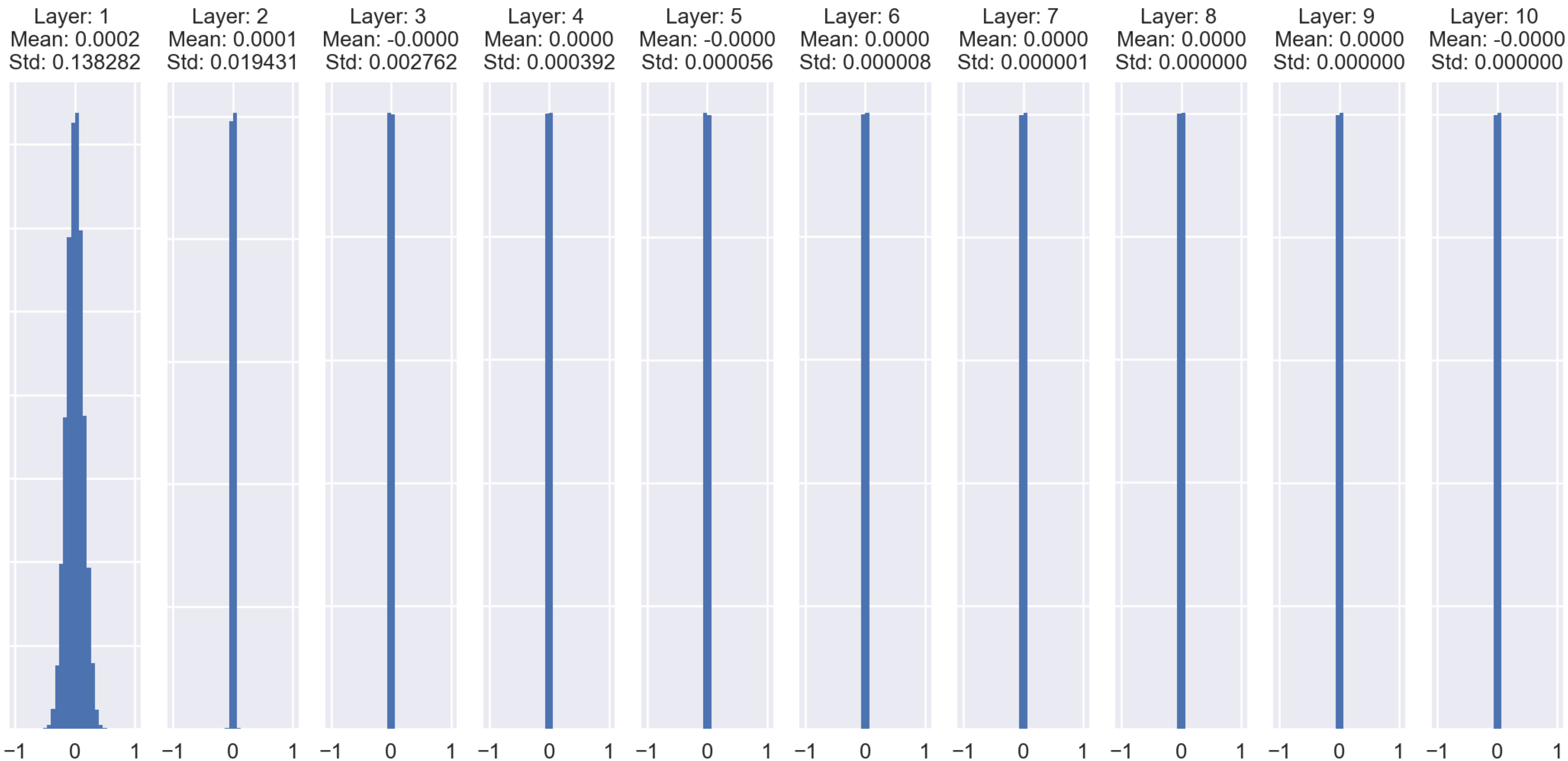
$$\mathrm{var}(y) = \mathrm{var}(x_1 w_1 + x_2 w_2) = \mathrm{var}(x_1 w_1) + \mathrm{var}(x_2 w_2)$$

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

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

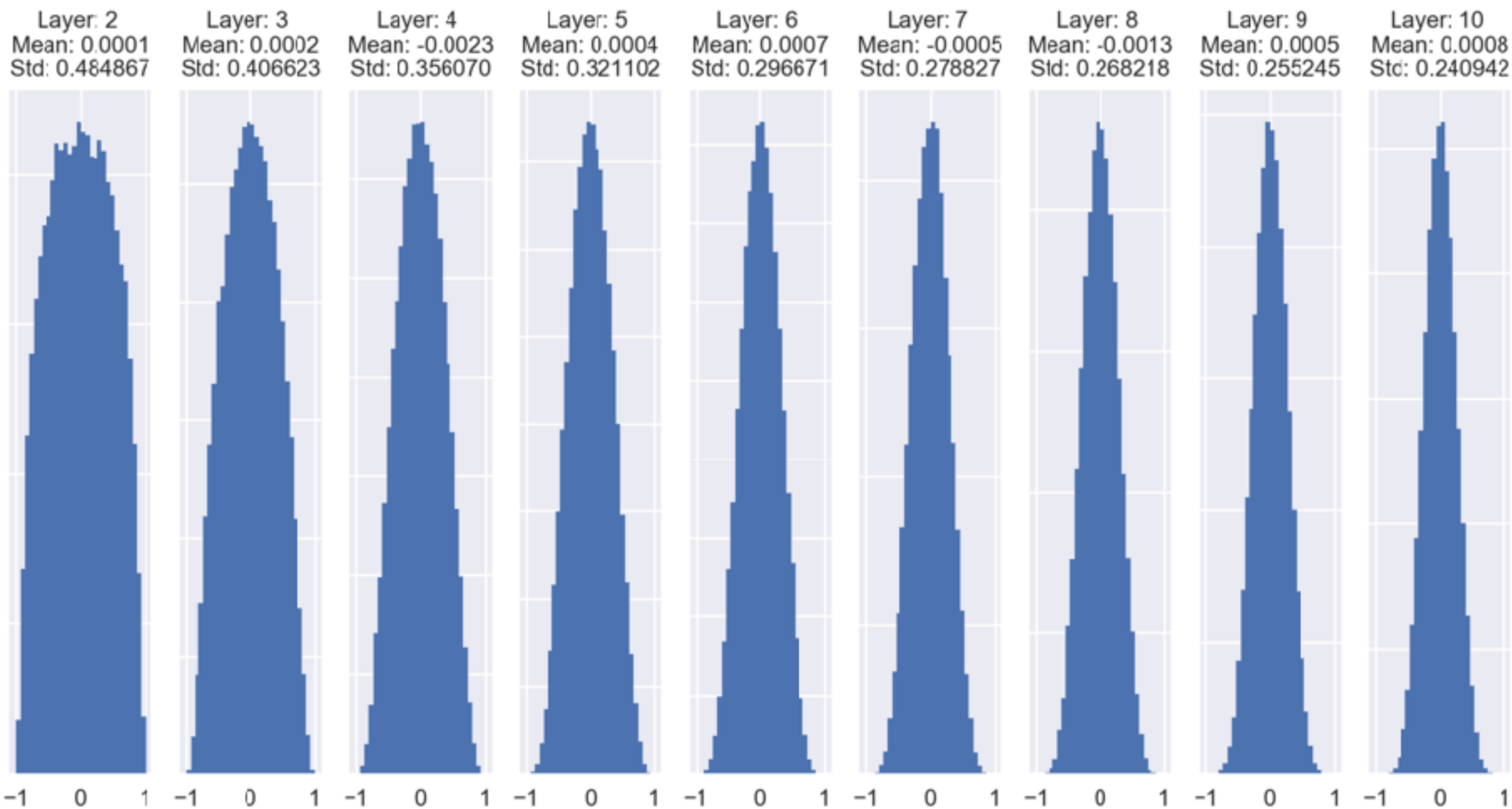
Xavier initialization [Glorot 2010]

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



Xavier initialization [Glorot 2010]

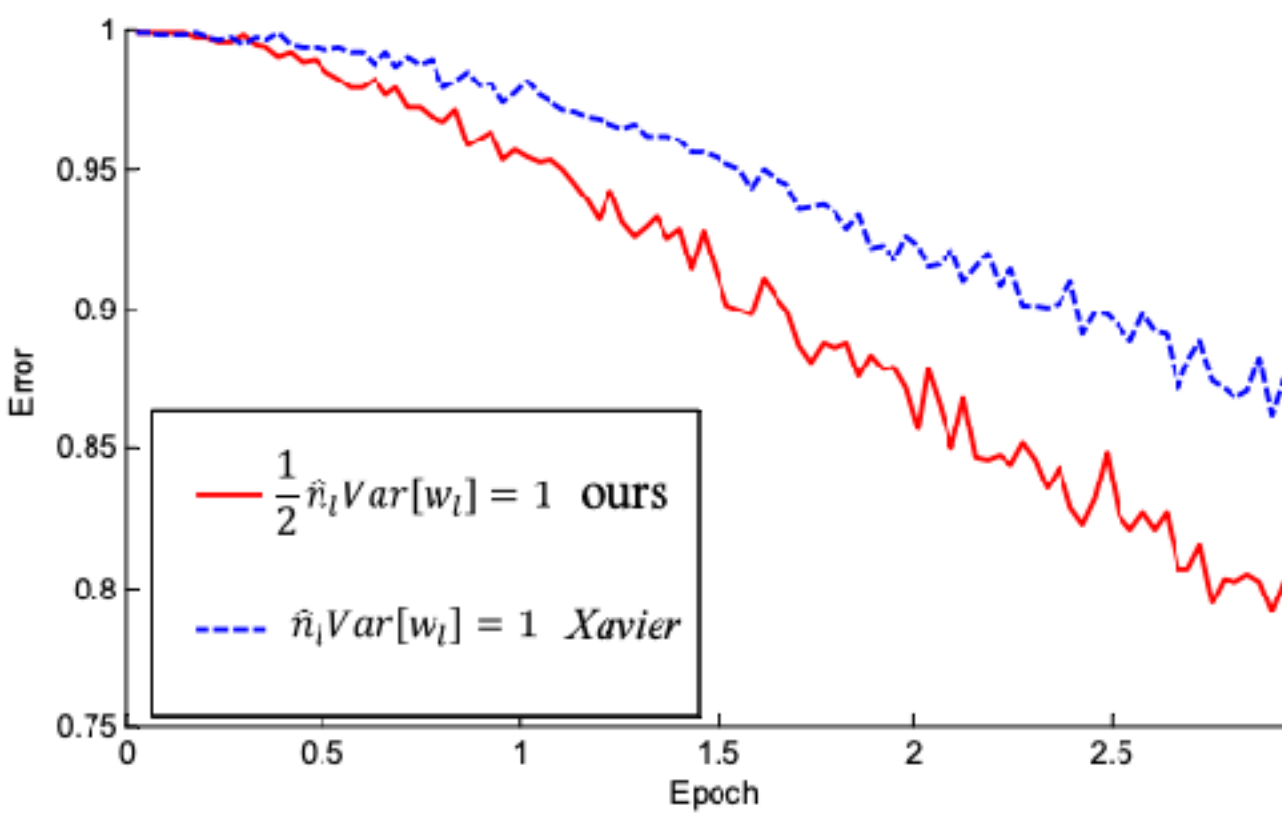
Signal in Xavier initialized weights $\mathbf{w}^{(i)} \sim \mathcal{N}(\mathbf{0}, 1/N^{(i)})$ forward (and backward) pass (better but not ideal)



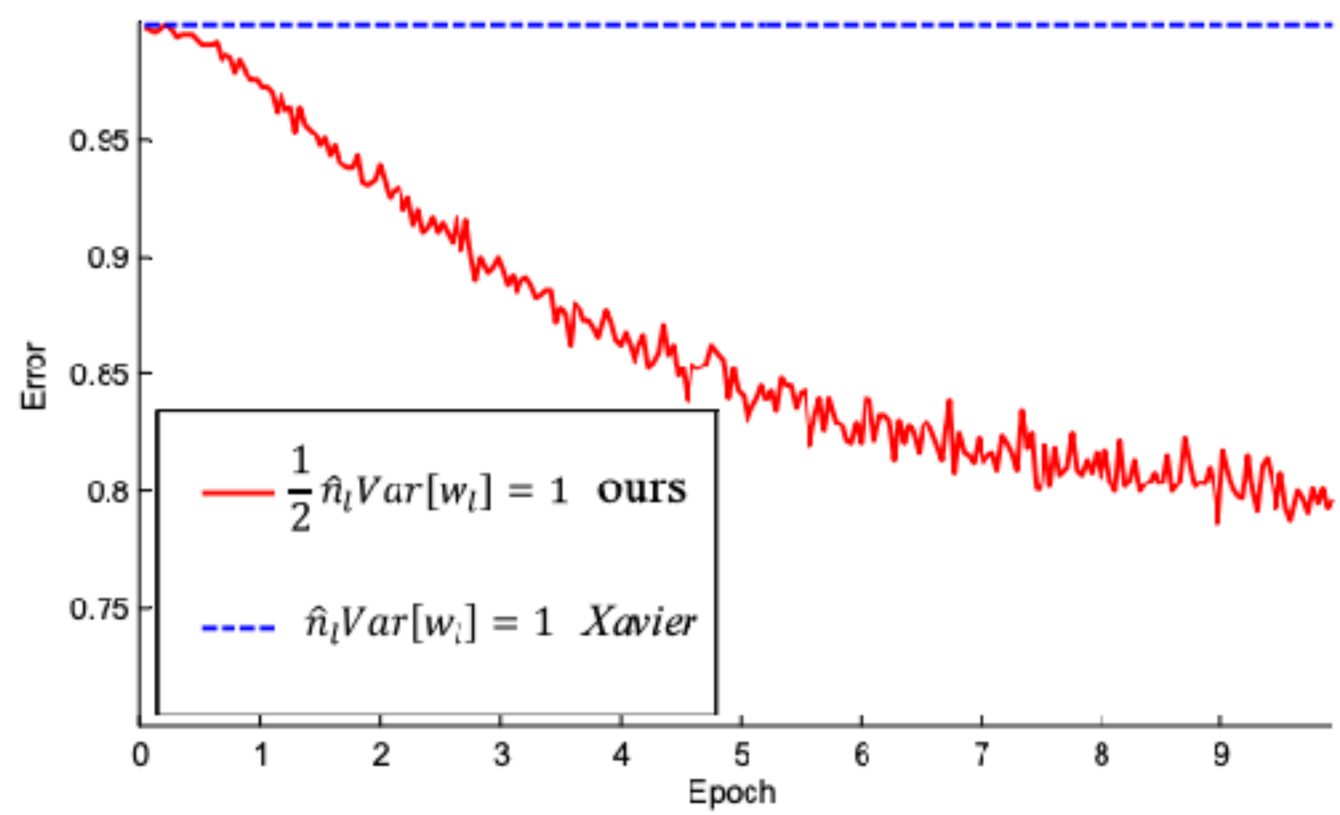
Kaimimg initialization
https://arxiv.org/pdf/1502.01852.pdf

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

22 layers



30 layers



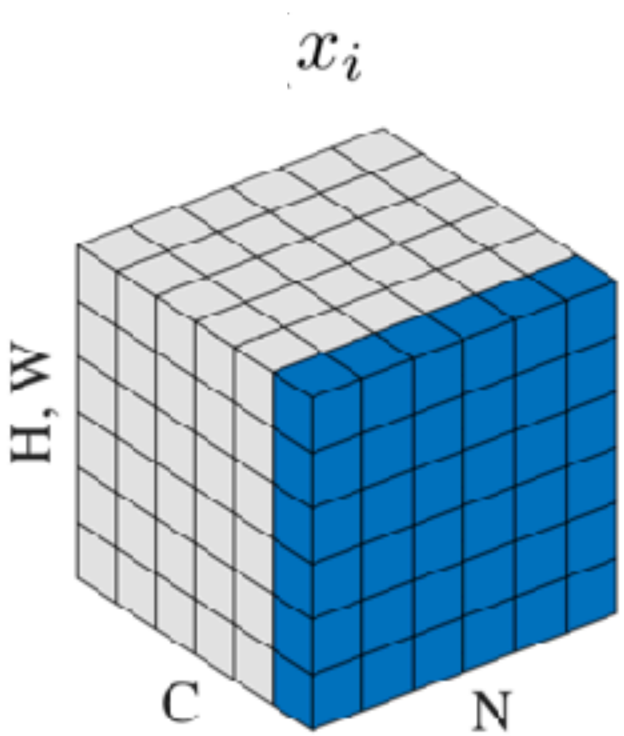
- PyTorch: `nn.init.xavier_uniform(conv1.weight)`
  `nn.init.calculate_gain('sigmoid')`

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)

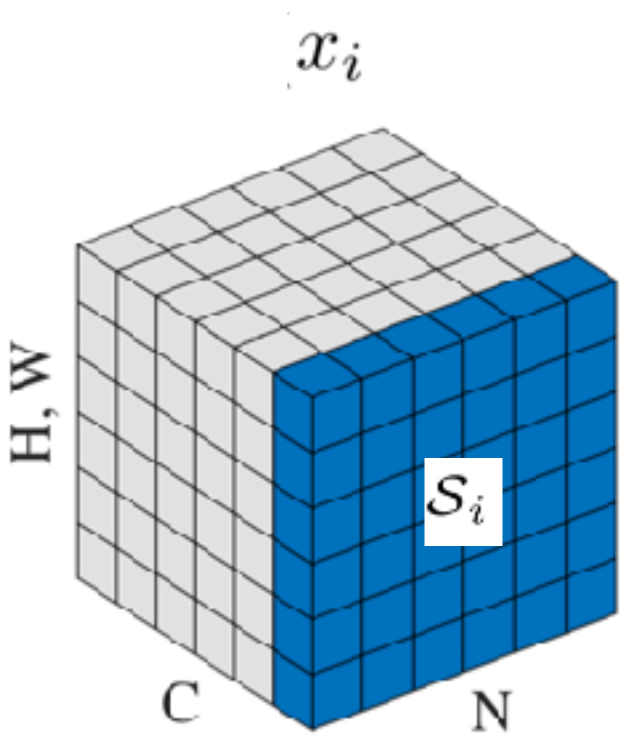


$$i = (i_N, i_C, i_H, i_W)$$

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

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

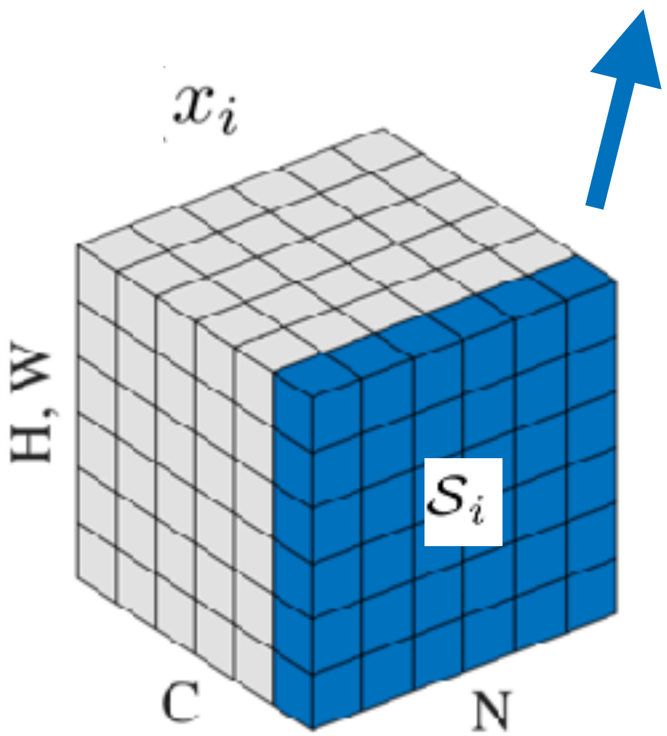
$$\mathcal{S}_{1,1,1,1} = \{(1,1,1,1), (2,1,1,1), \dots (N,1,H,W)\}$$

$$\vdots \qquad \vdots \qquad \vdots \qquad \vdots$$

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

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

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

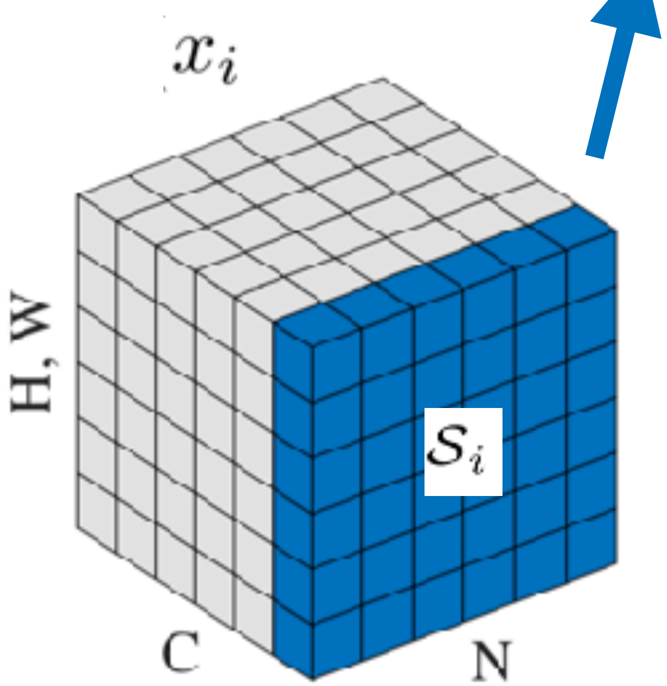


For each channel i compute mean a std

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

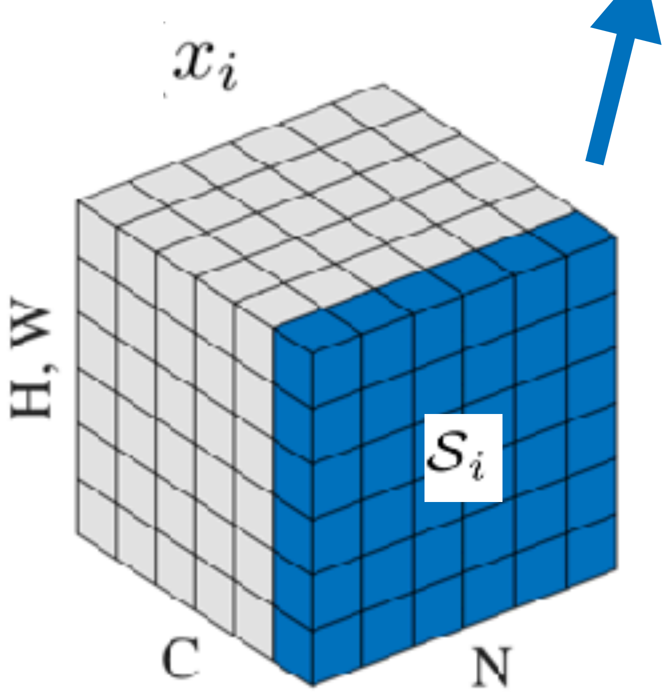
$$\mu_i = \frac{1}{m}\sum_{k\in\mathcal{S}_i} x_k, \quad \sigma_i = \sqrt{\frac{1}{m}\sum_{k\in\mathcal{S}_i}(x_k-\mu_i)^2+\epsilon},$$

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



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)



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

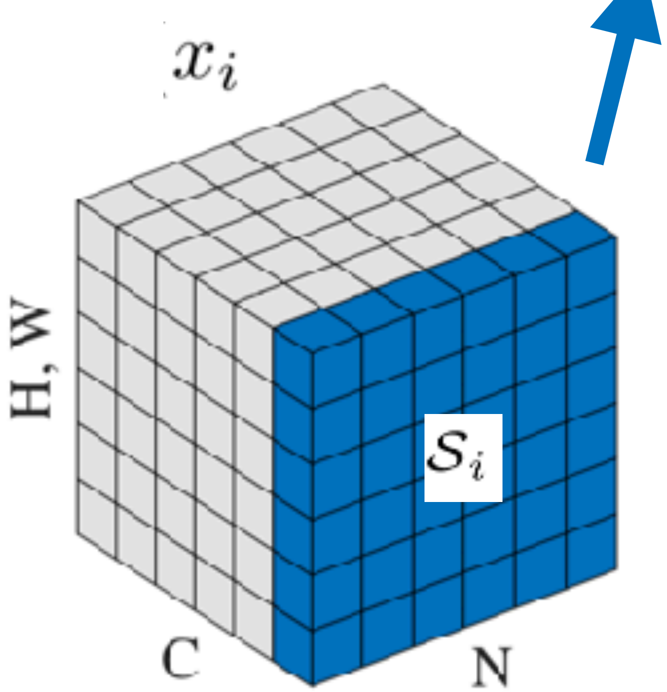
$$\hat{x}_i = \frac{1}{\sigma_i}(x_i - \mu_i)$$

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

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

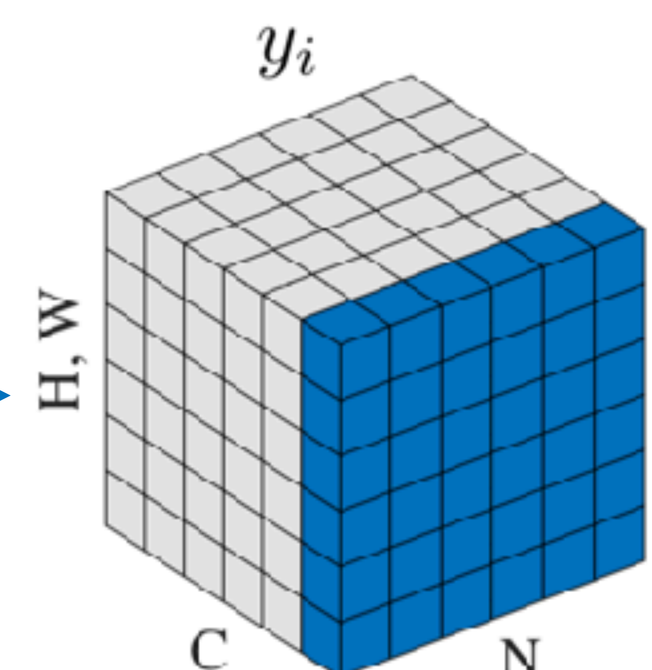
Batch normalization layer [Ioffe and Szegedy 2015]
https://arxiv.org/pdf/1502.03167.pdf (over 6k citation)

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



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

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



- 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)

batch size → 20, channels → 100, width → 35, height → 45

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

What is dimensionality of the output?

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?

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)
```

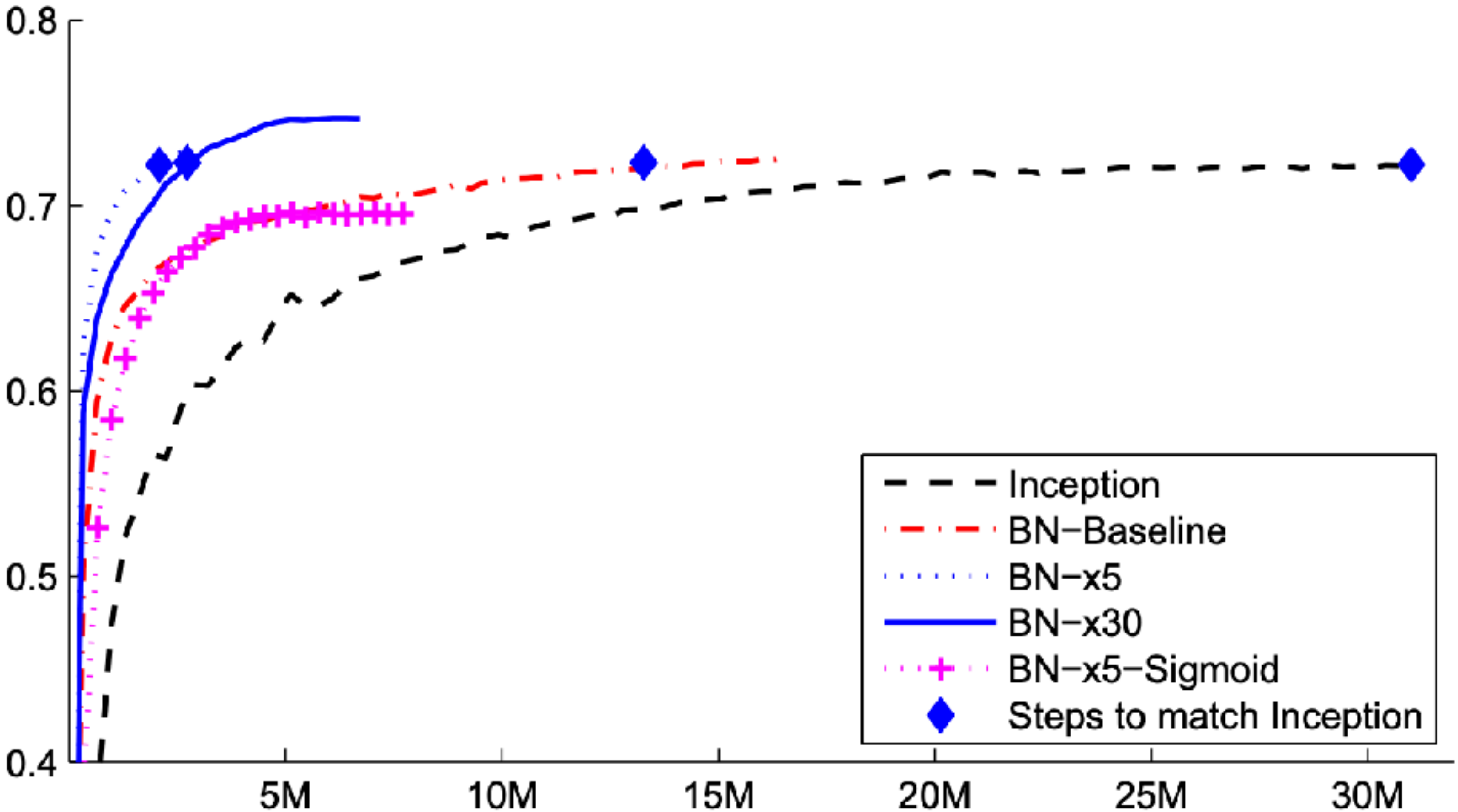
What is dimensionality of the output?

the same: 20x100x35x45

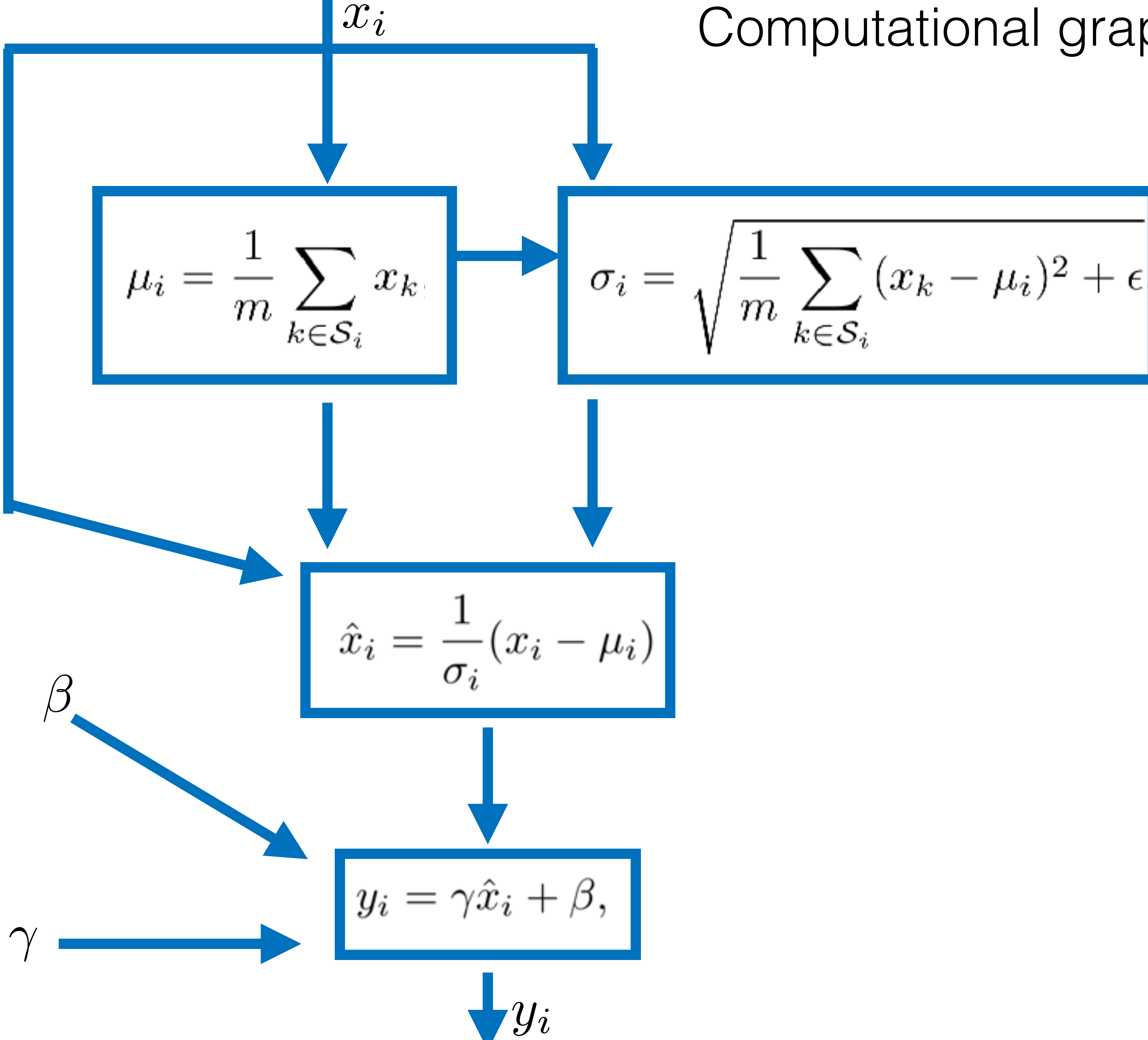
What is dimensionality of mean μ?

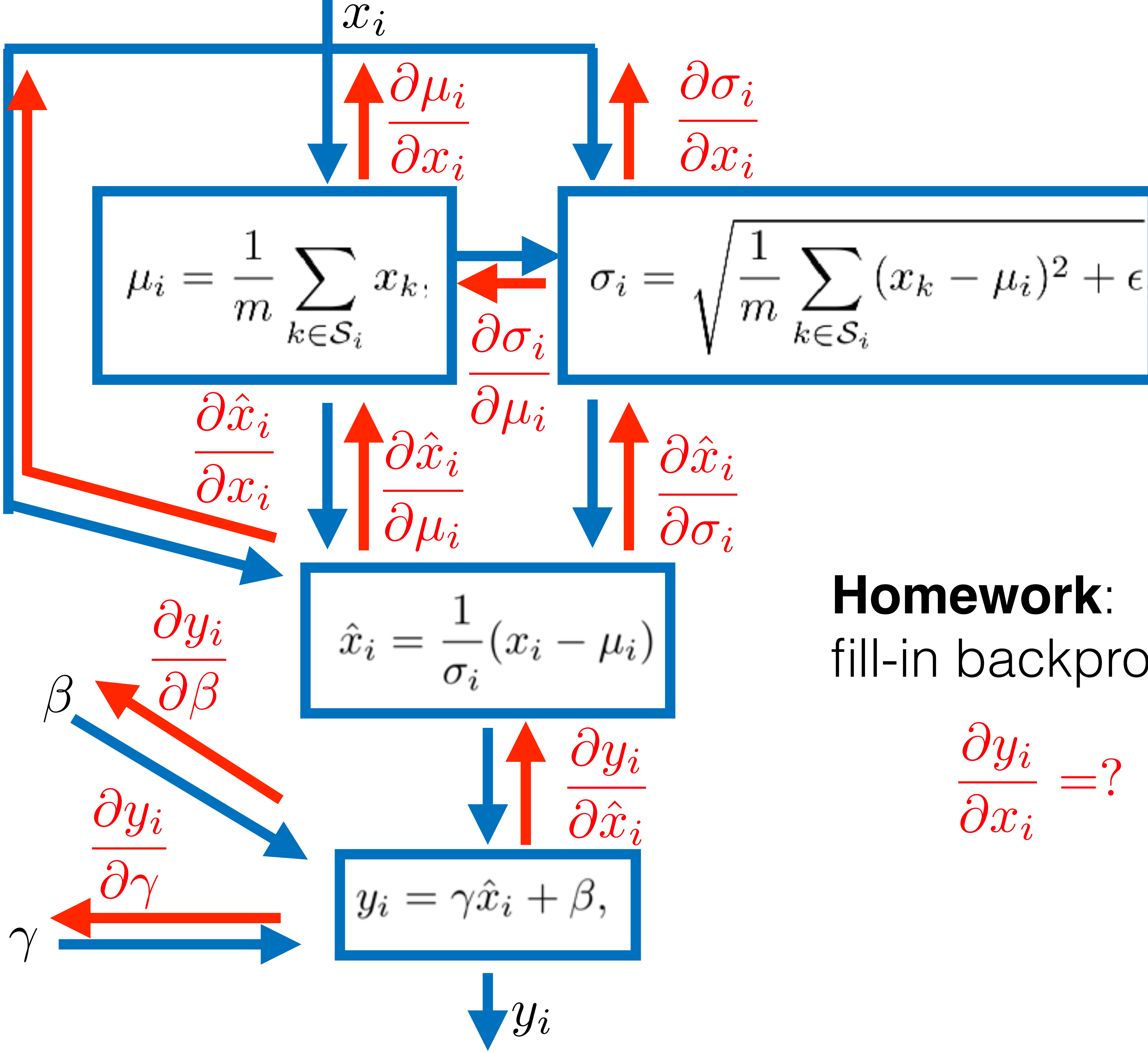
100 dimensional vector

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



Computational graph of BN



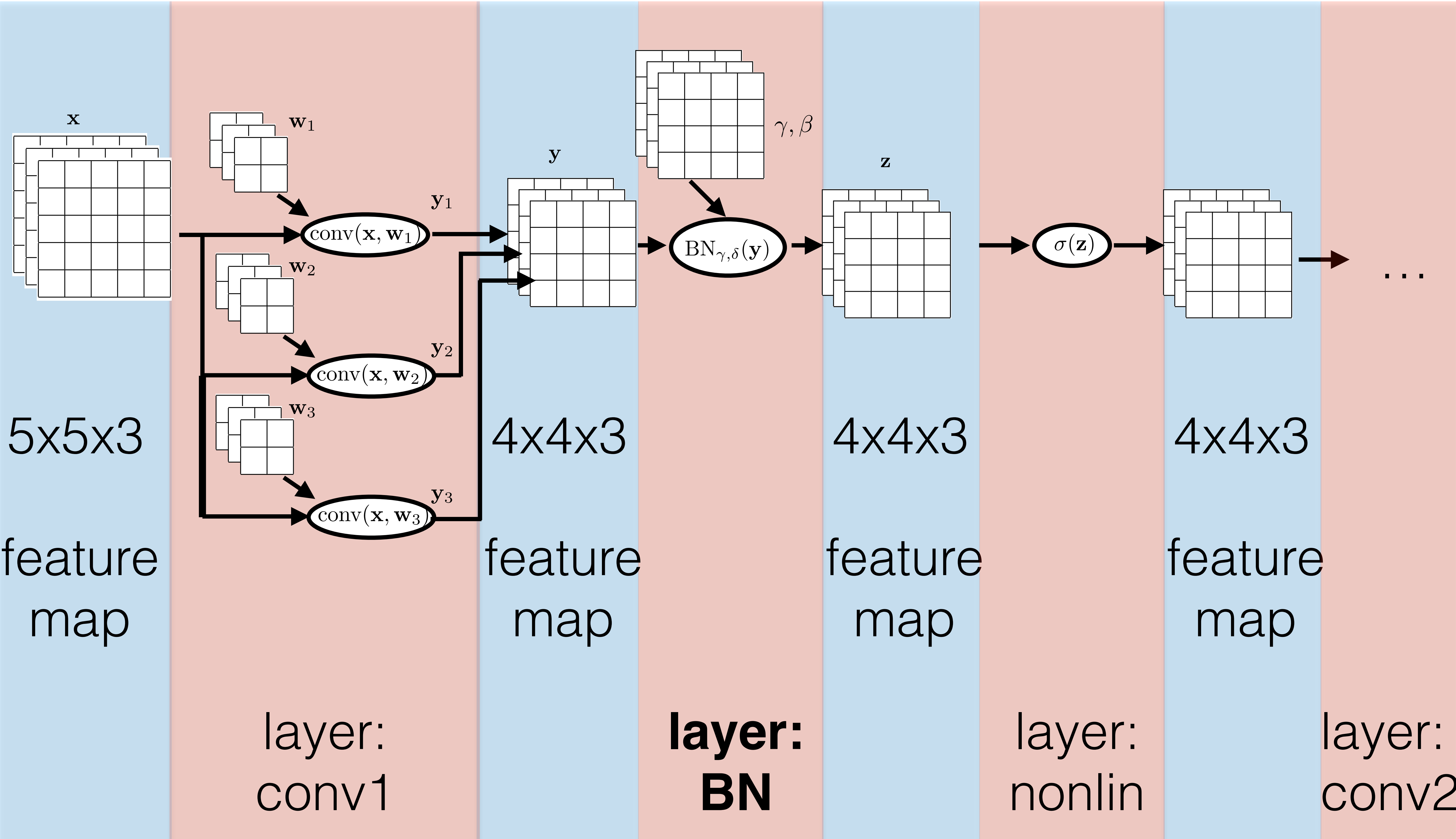


Homework:

fill-in backprop of BN

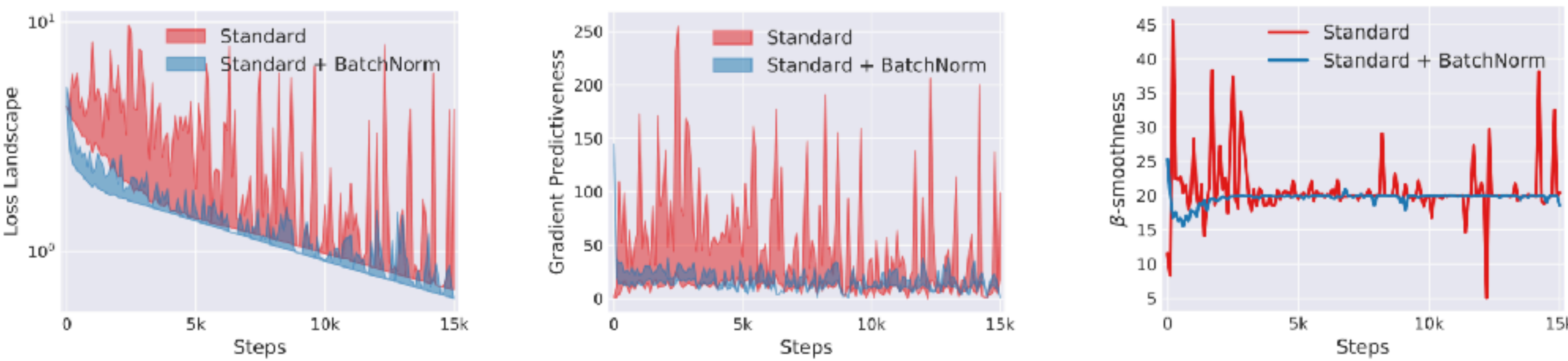
$$\frac{\partial y_i}{\partial x_i} = ?$$

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



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.

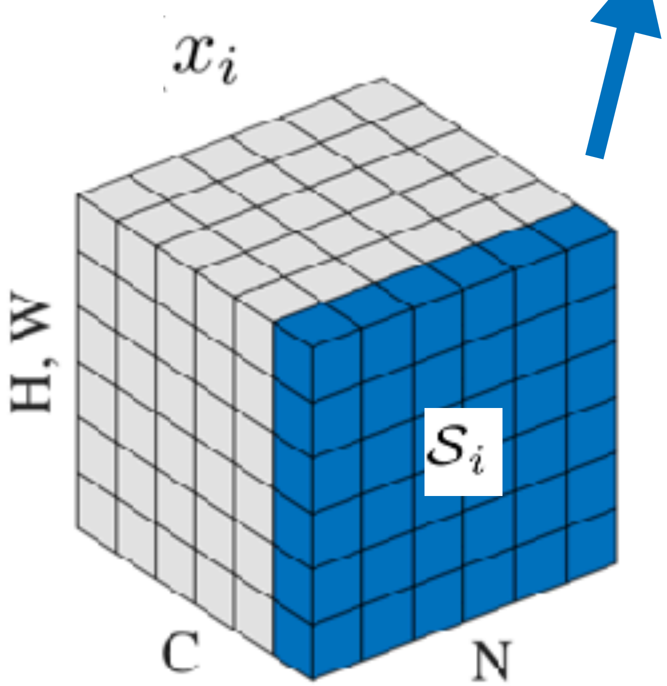


(a) loss landscape (b) gradient predictiveness (c) "effective" β-smoothness

Batch Normalization - conclusions

- **Testing data** (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)

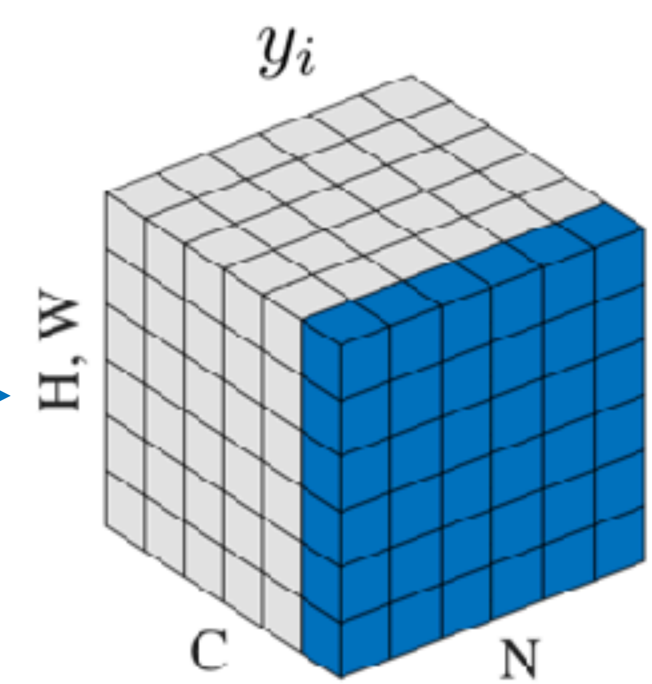


$$\mu_i = \frac{1}{m}\sum_{k \in \mathcal{S}_i} x_k, \quad \sigma_i = \sqrt{\frac{1}{m}\sum_{k \in \mathcal{S}_i}(x_k - \mu_i)^2 + \epsilon},$$

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

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

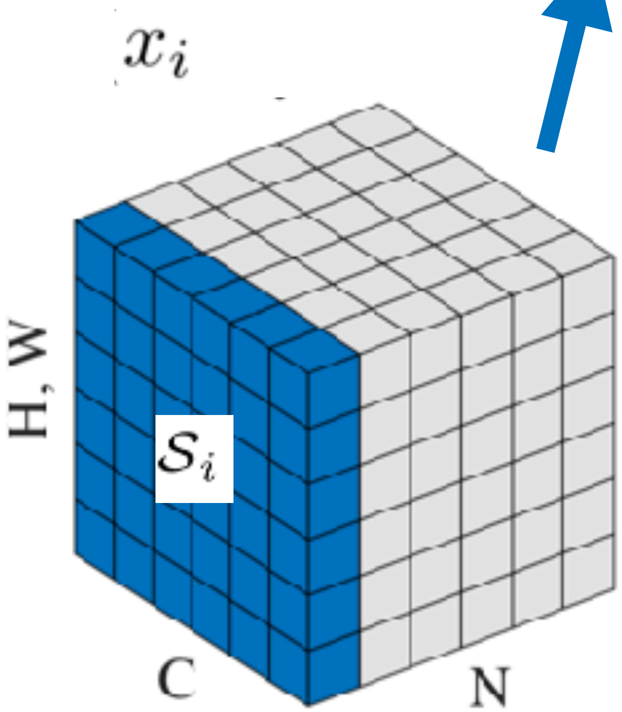
$$y_i = \gamma \hat{x}_i + \beta,$$



Layer normalization [Ba, Kiros, Hinton 2016]

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

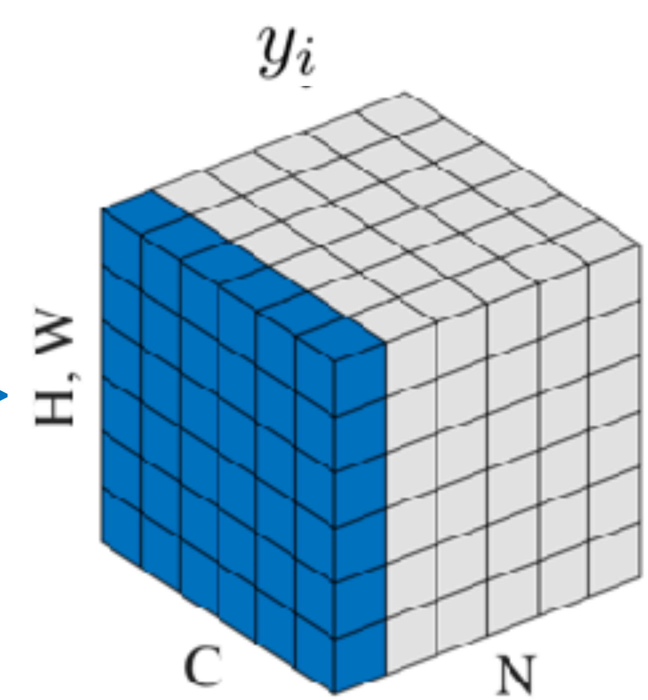
$$\mu_i = \frac{1}{m} \sum_{k \in \mathcal{S}_i} x_k, \quad \sigma_i = \sqrt{\frac{1}{m} \sum_{k \in \mathcal{S}_i} (x_k - \mu_i)^2 + \epsilon},$$



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

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

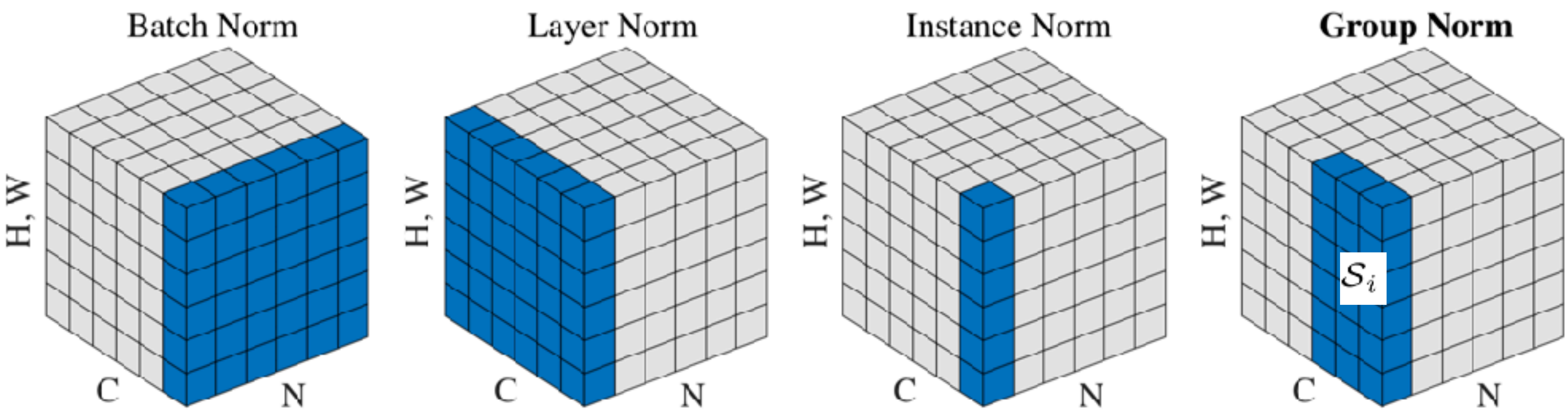
$$y_i = \gamma \hat{x}_i + \beta,$$



Layer normalization performs well on RNN

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

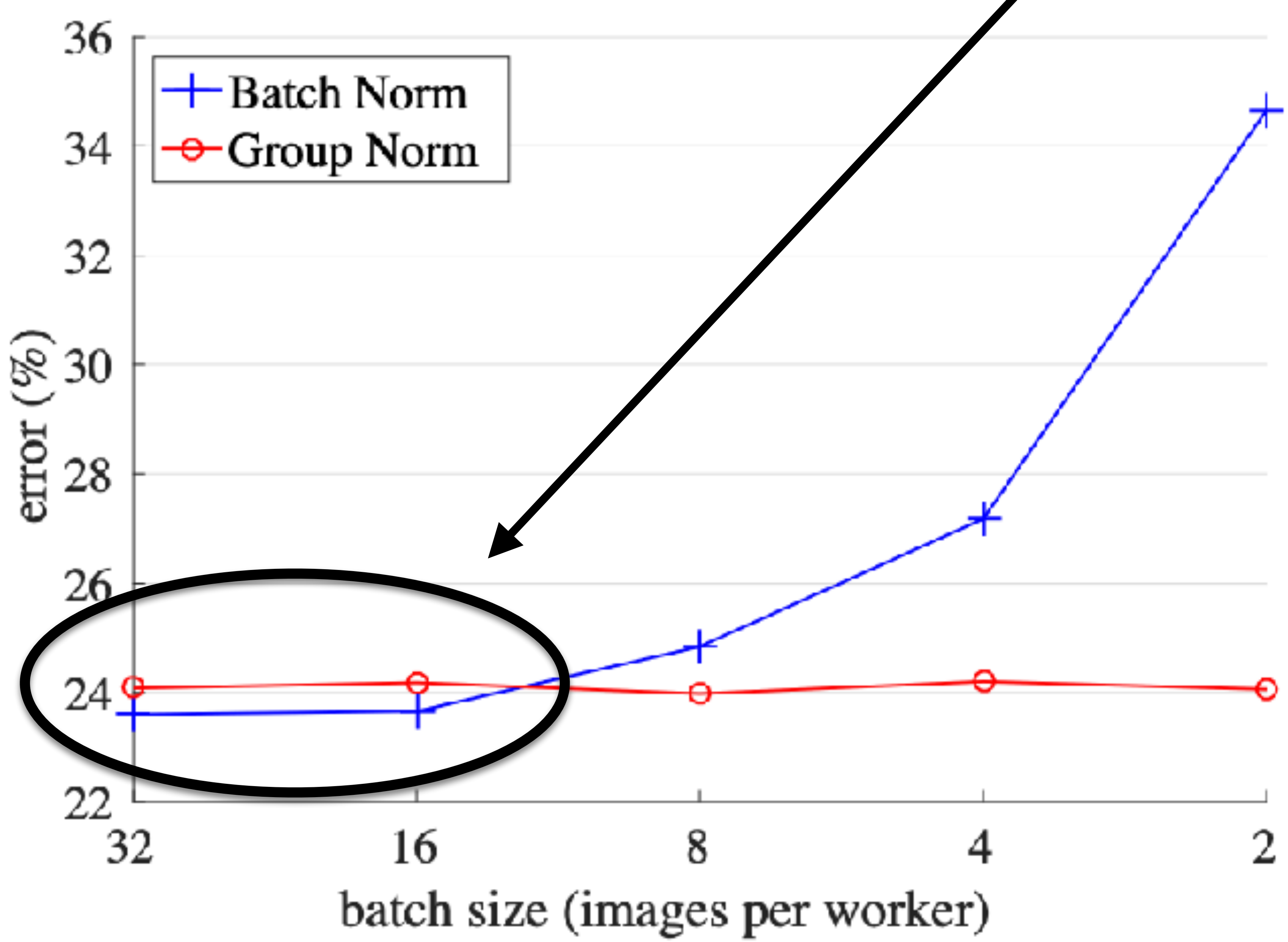
Group normalization performs well for style transfer (GANs) and RNN but does not outperform BN for image classification



Classification RNN Style transfer

Group Normalization - conclusions

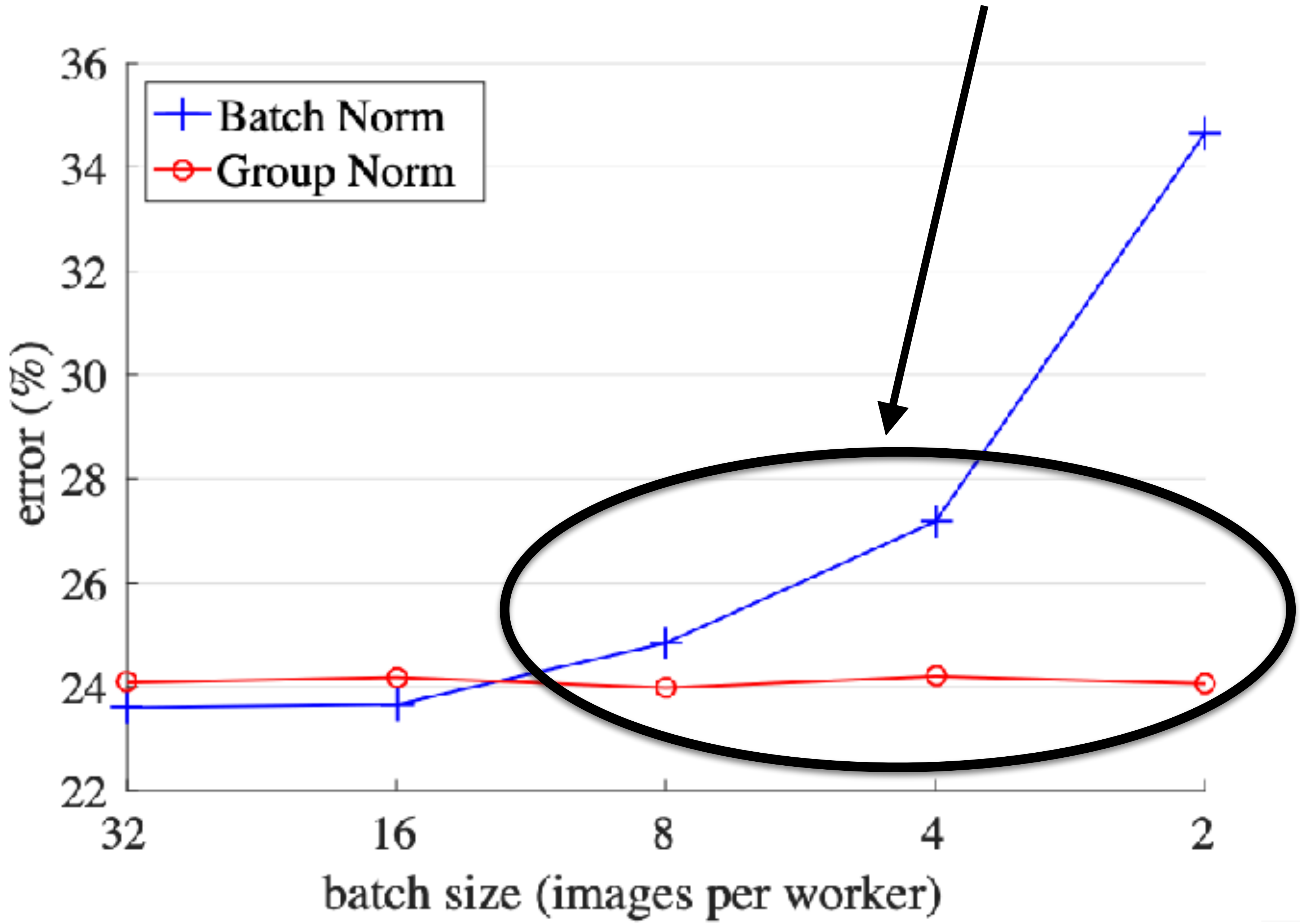
- GN achieves performance comparable with BN on image classification tasks.



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

Group Normalization - conclusions

- GN achieves performance comparable with BN on image classification tasks.
- For smaller mini-batches GN outperforms BN significantly

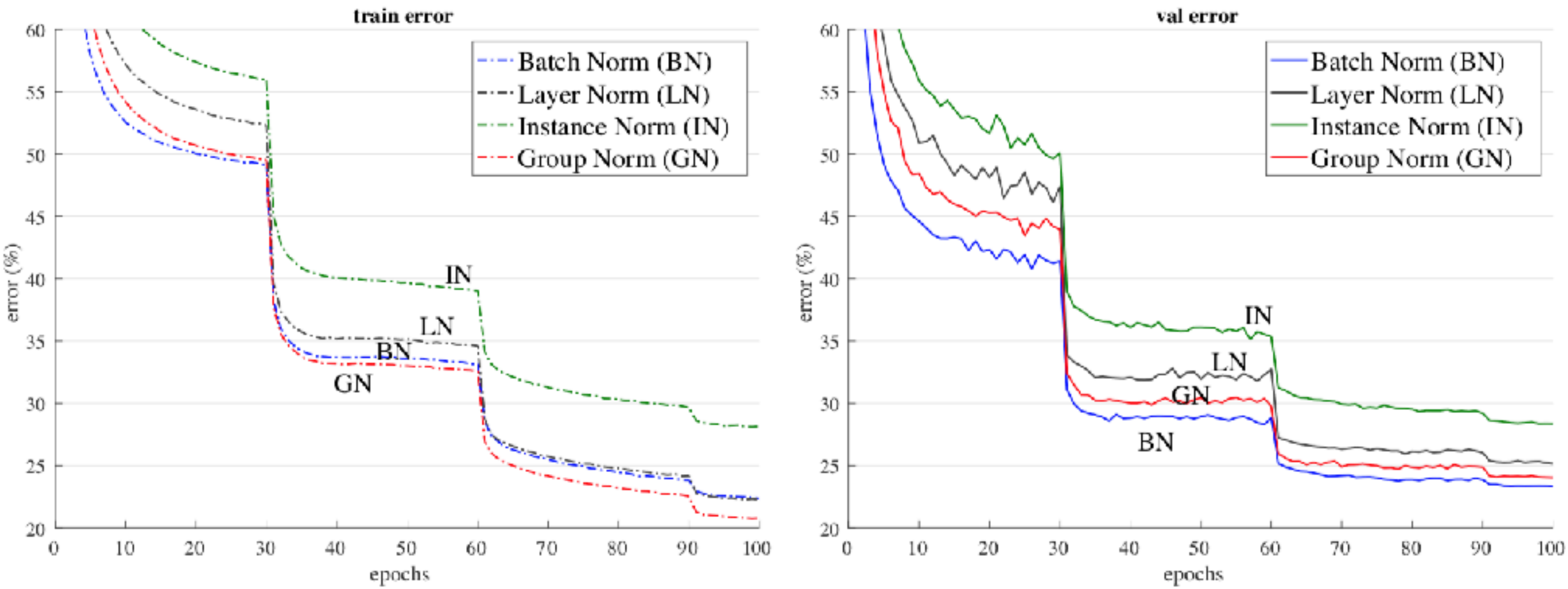


[Wu, He, CVPR, 2018] https://arxiv.org/pdf/1803.08494.pdf

Group Normalization - conclusions

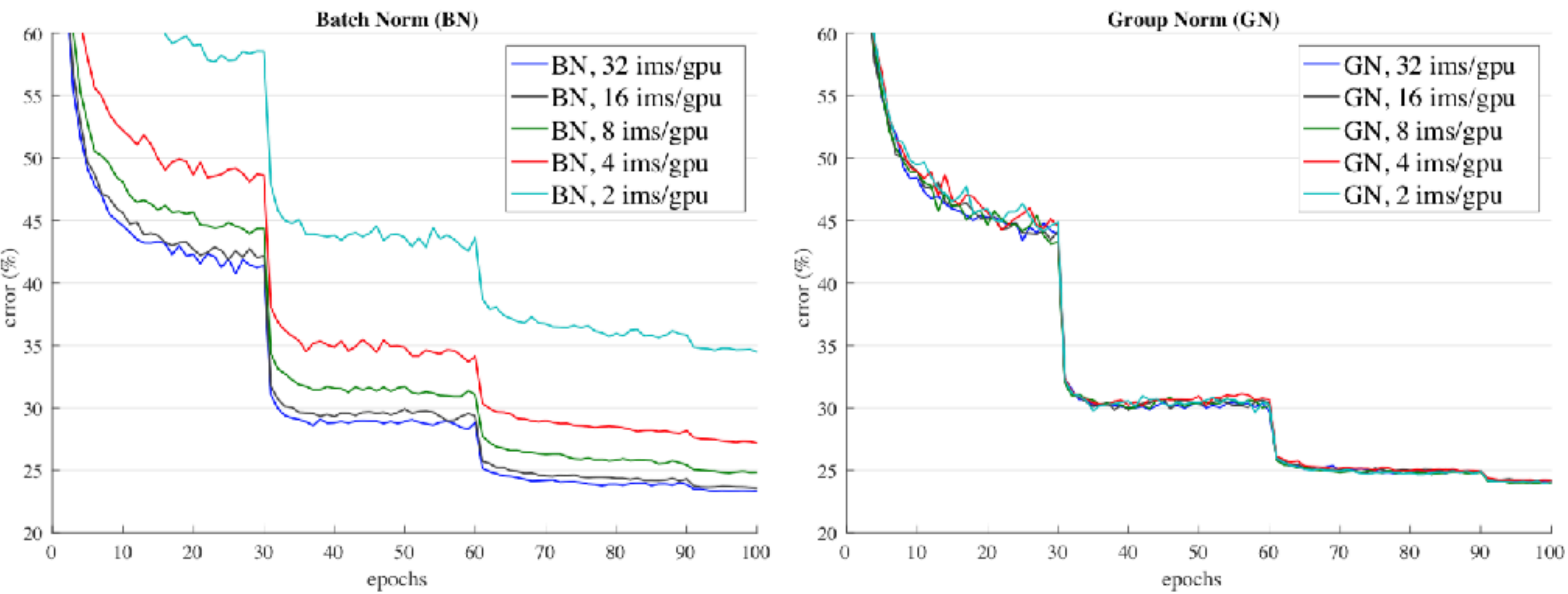
- GN achieves performance comparable with BN on image classification tasks.

Sufficiently large mini-batch size = 32



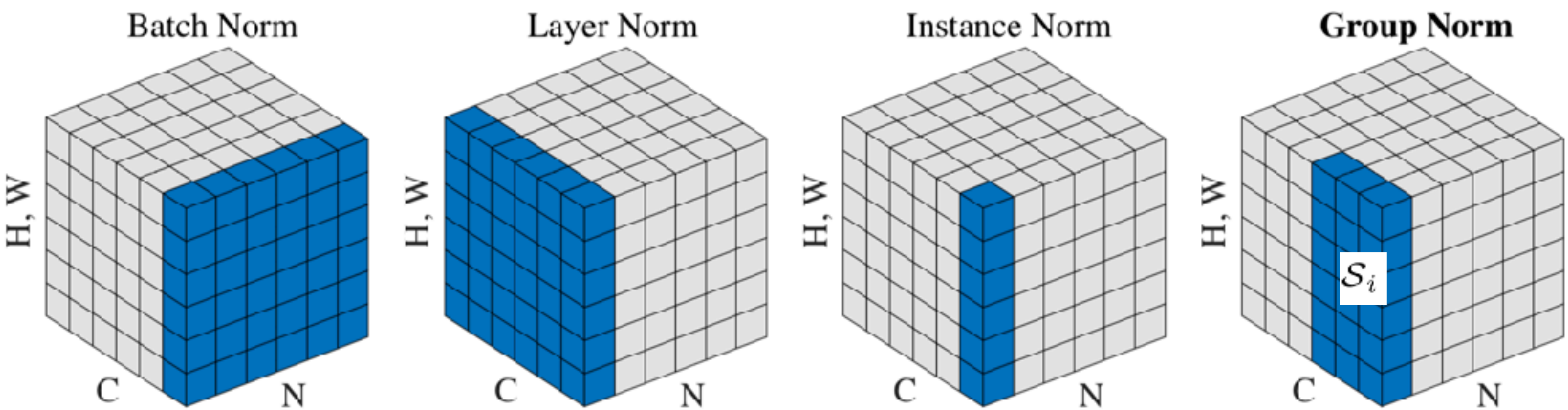
Group Normalization - conclusions

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



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

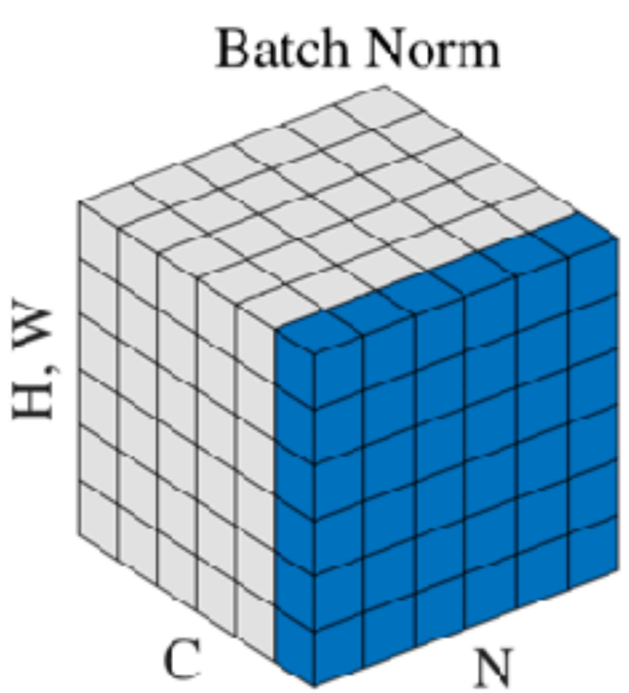


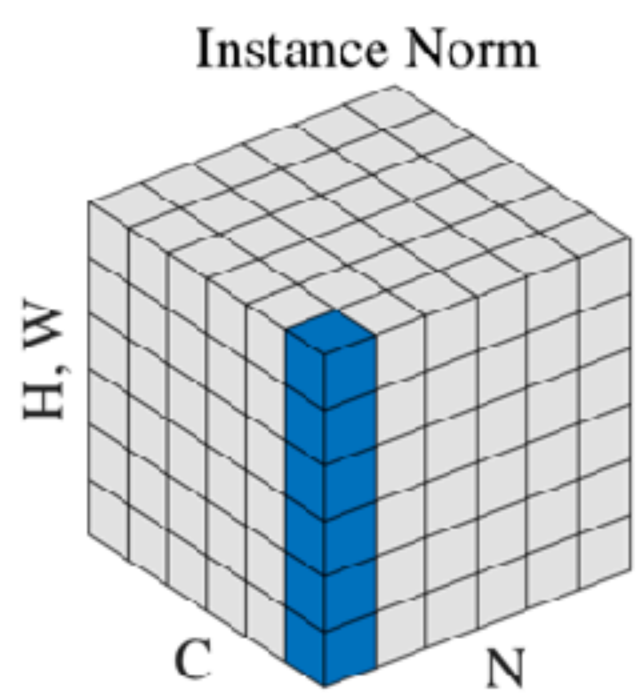
Classification RNN Style transfer

Batch-Instance normalization

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

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





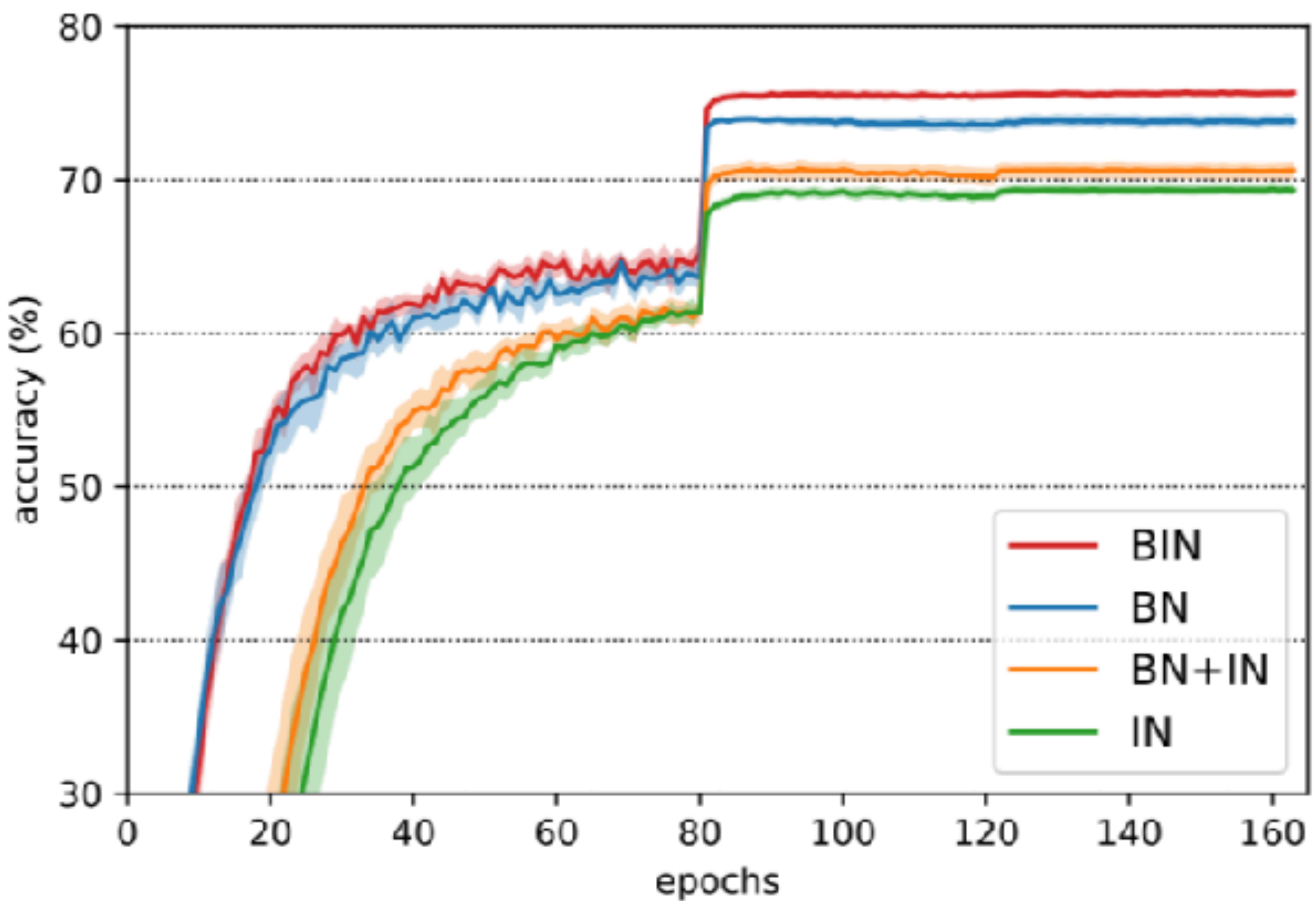
Batch-Instance normalization

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

Classification results: ResNet-101 on CIFAR-100



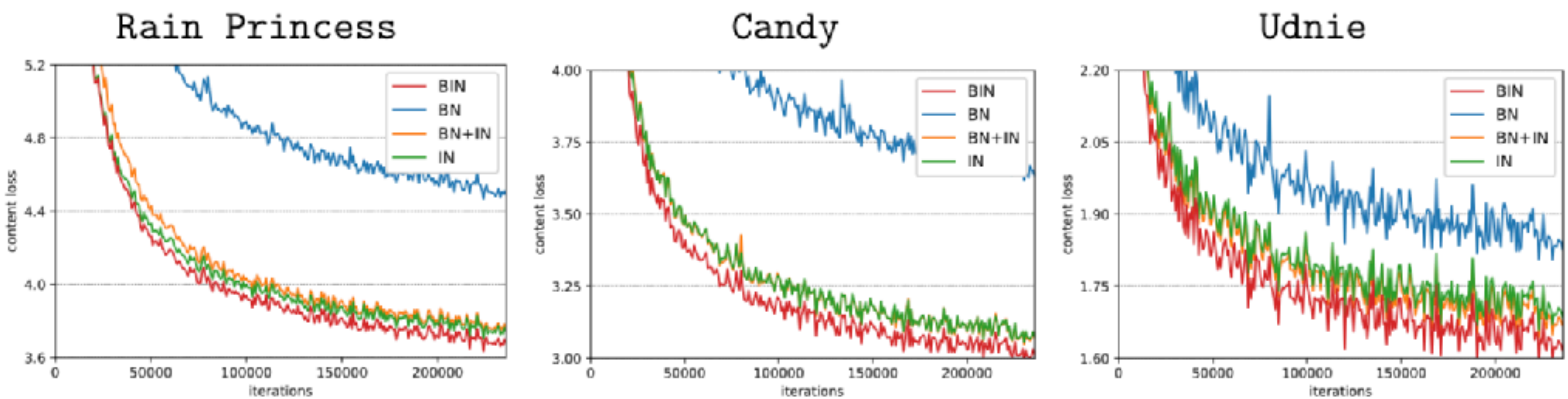
Batch-Instance normalization

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

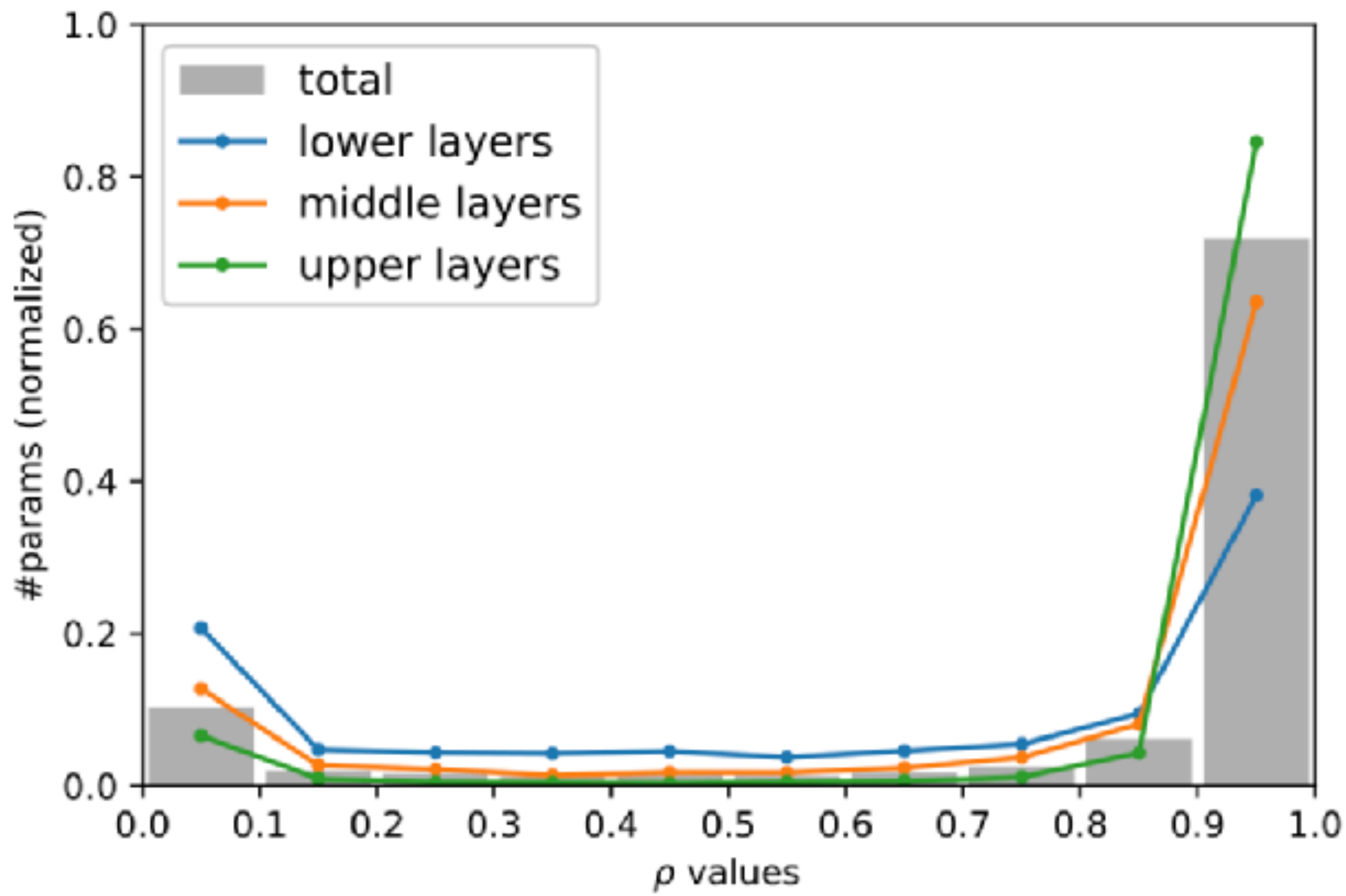


Batch-Instance normalization
https://arxiv.org/pdf/1805.07925.pdf

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

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

Classification results: ResNet-101 on CIFAR-100



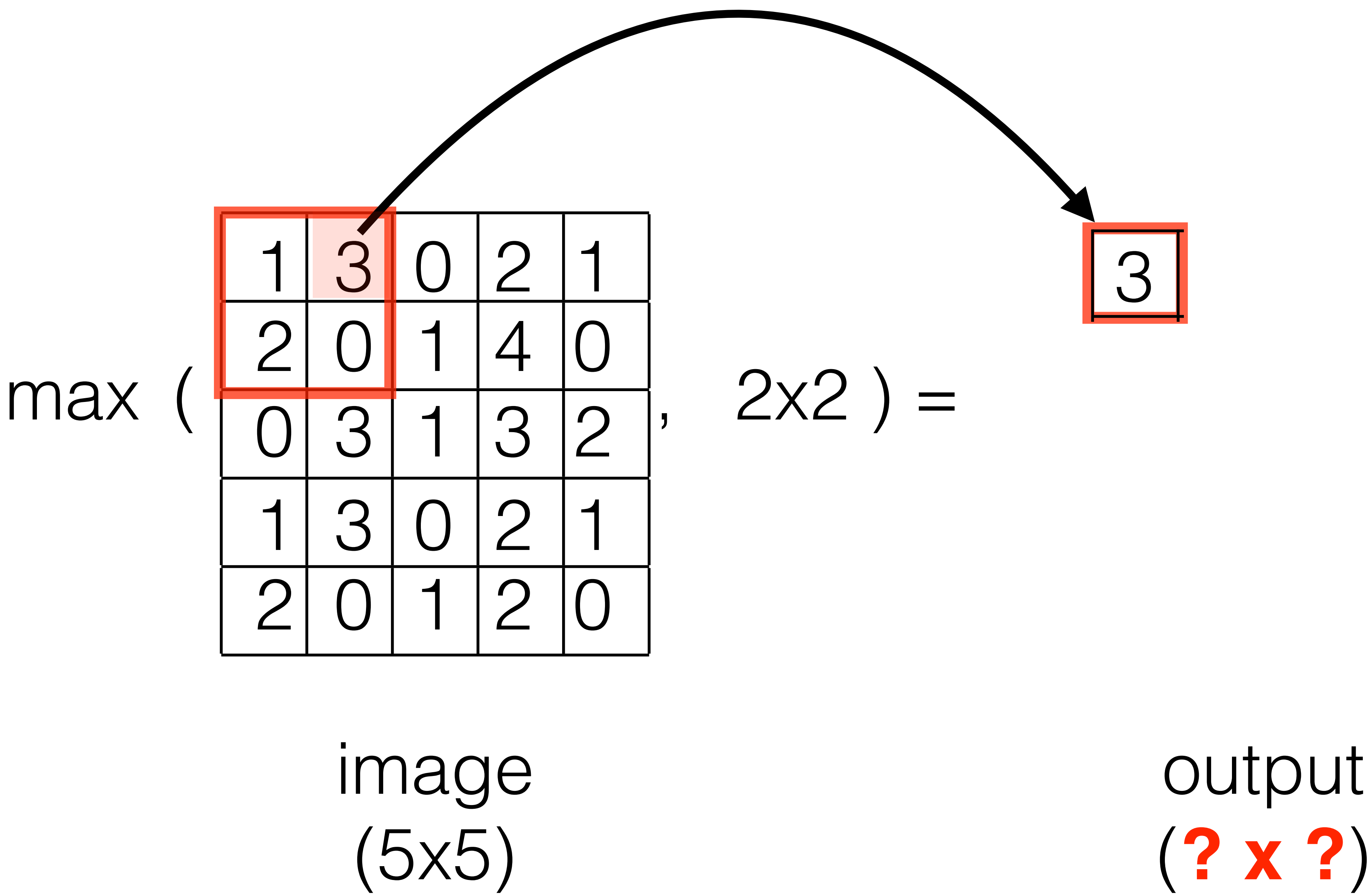
Normalization layers - Summary

- BN: works for classification, suffers from small mini-batch.
- LN: works for recurrent nets
- IN/GN: works for style transfer nets and are littlebit weaker on classification than BN (with large minibatch).
- BIN: sufficiently flexible to work best for both: classification and style transfer nets, but it has more parameters to learn.

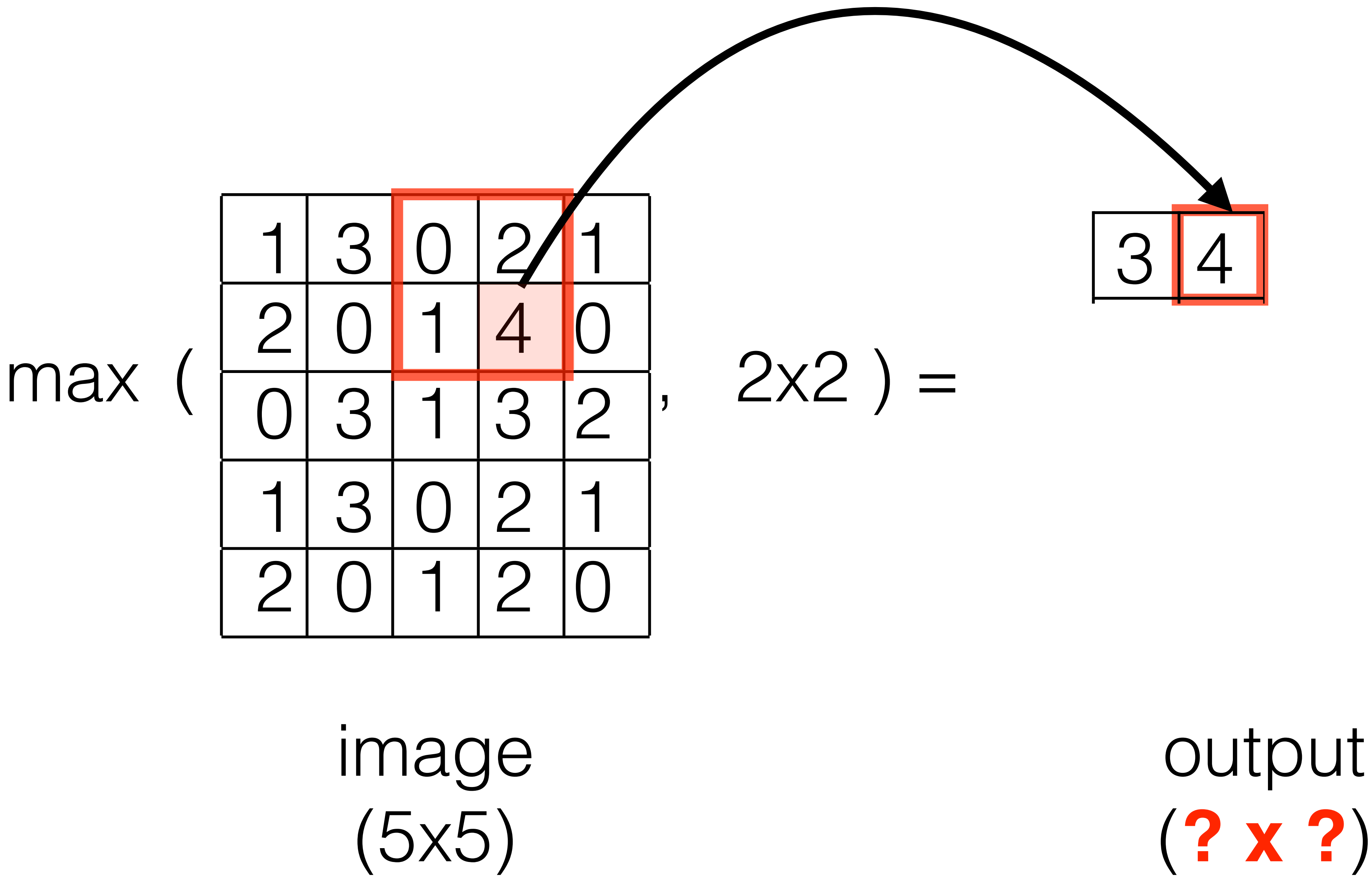
Outline

- 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



Max-pooling

max (

1	3	0	2	1
2	0	1	4	0
0	3	1	3	2
1	3	0	2	1
2	0	1	2	0

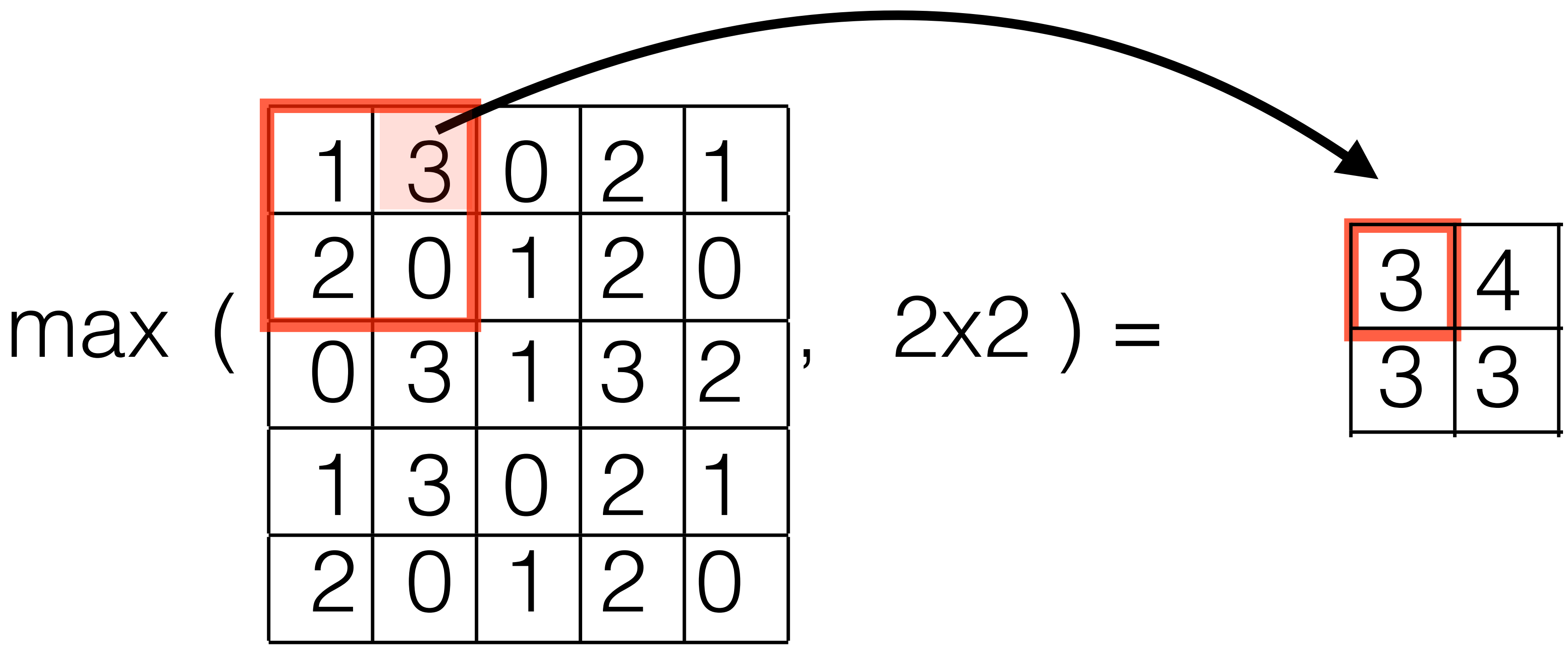
, 2x2) =

3	4
3	3

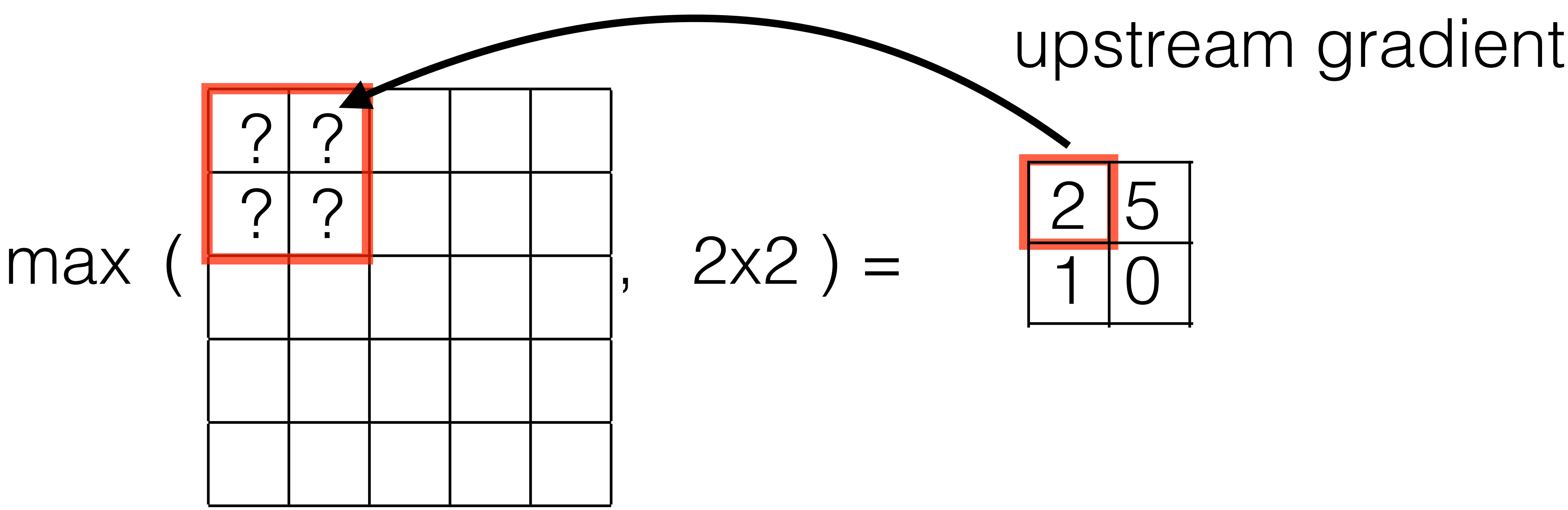
image
(5x5)

output
(**? x ?**)

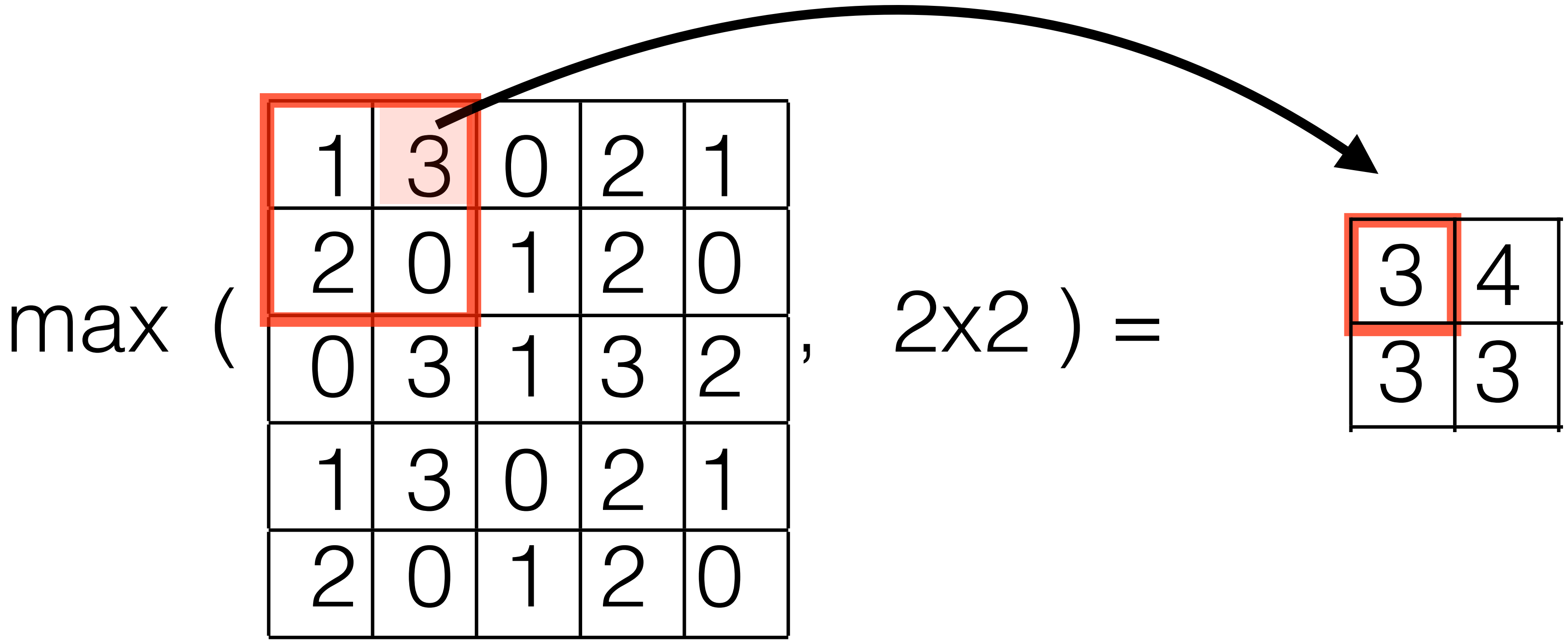
Max-pooling feed-forward



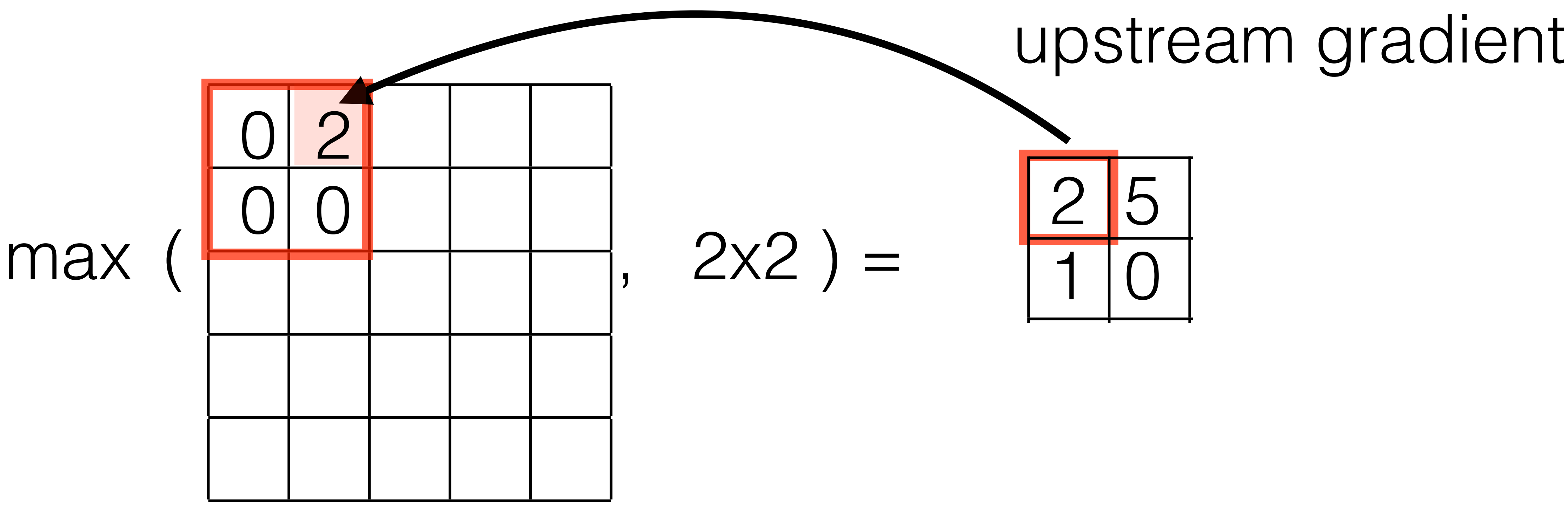
Max-pooling Backprop



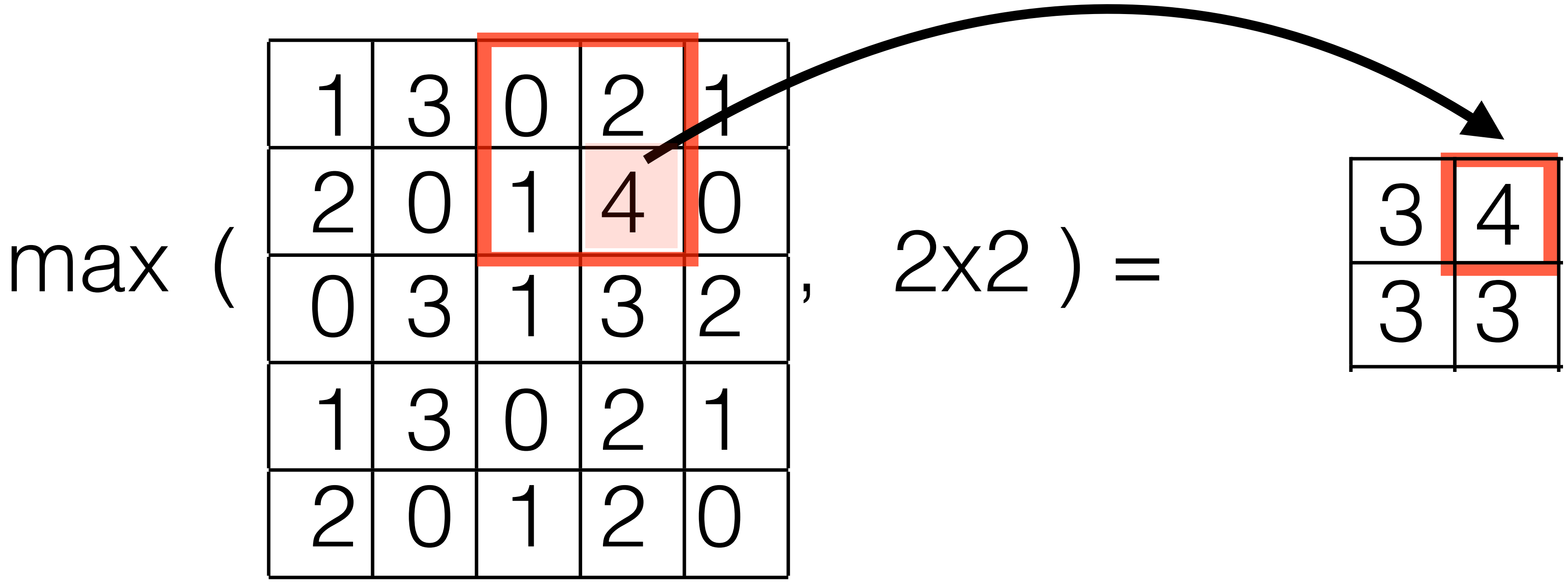
Max-pooling feed-forward



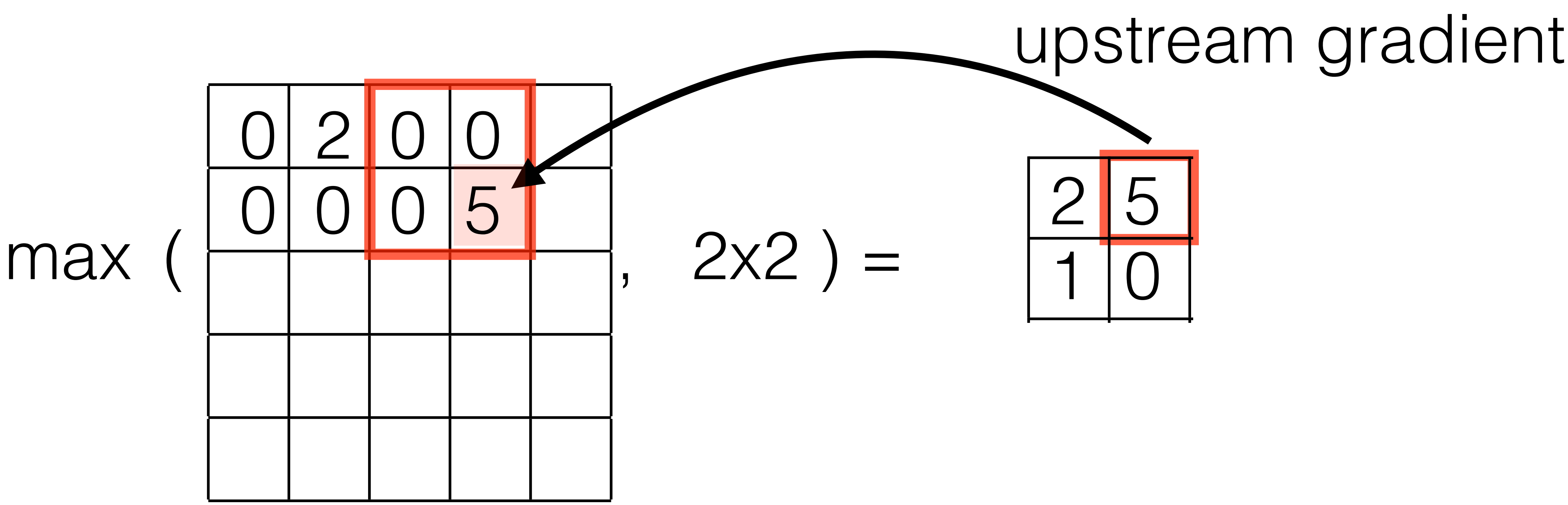
Max-pooling Backprop



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)

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

(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

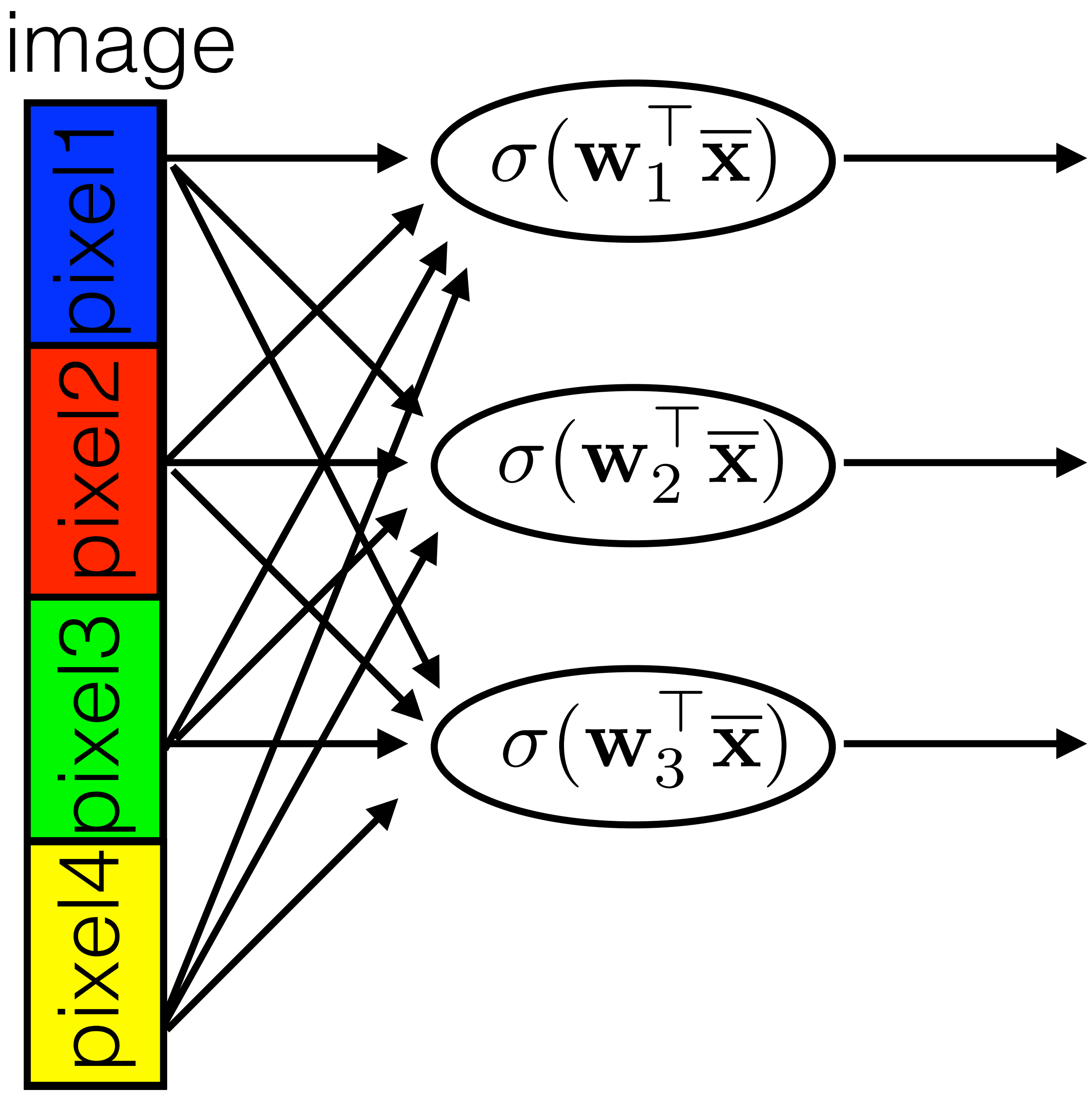
- Regression:
  - L2 loss
  - L1 loss
- Classification:
  - cross entropy loss (N-output classifier $\mathbf{f}(\mathbf{x}, \mathbf{w})$)
  - logistic loss (single output dichotomy classifier $f(\mathbf{x}, \mathbf{w})$)
  - Kulback-Leibler loss
- Ranking:
  - Ranking loss

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

PyTorch: `torch.nn.MarginRankingLoss()`

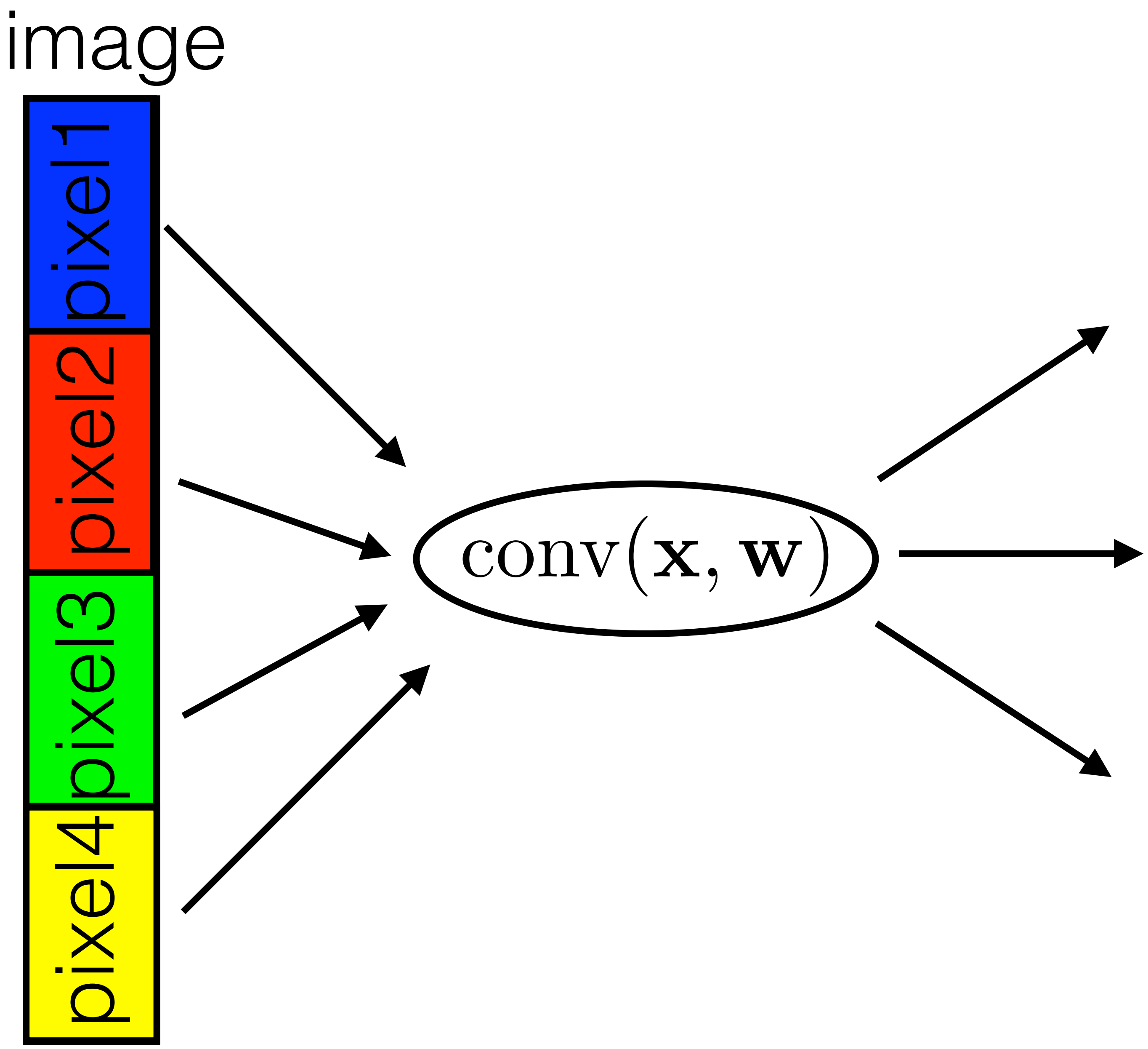
Regularization & overfitting

- Best regularization is using the right structure of the network



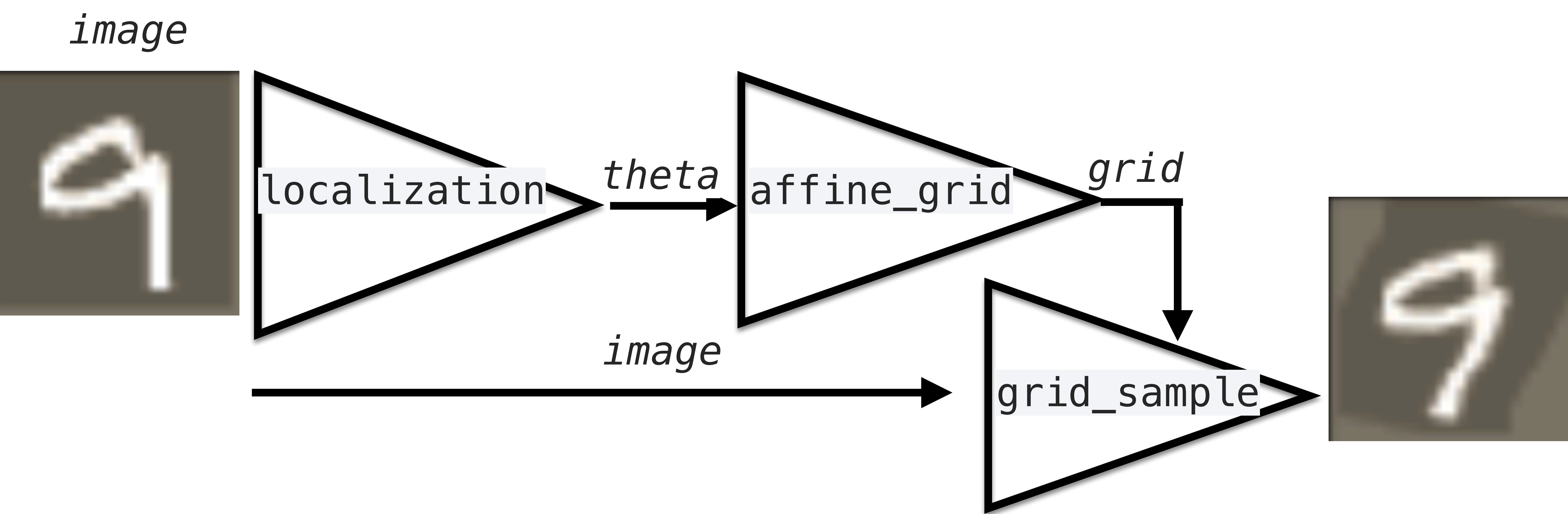
Regularization & overfitting

- Best regularization is using the right structure of the network



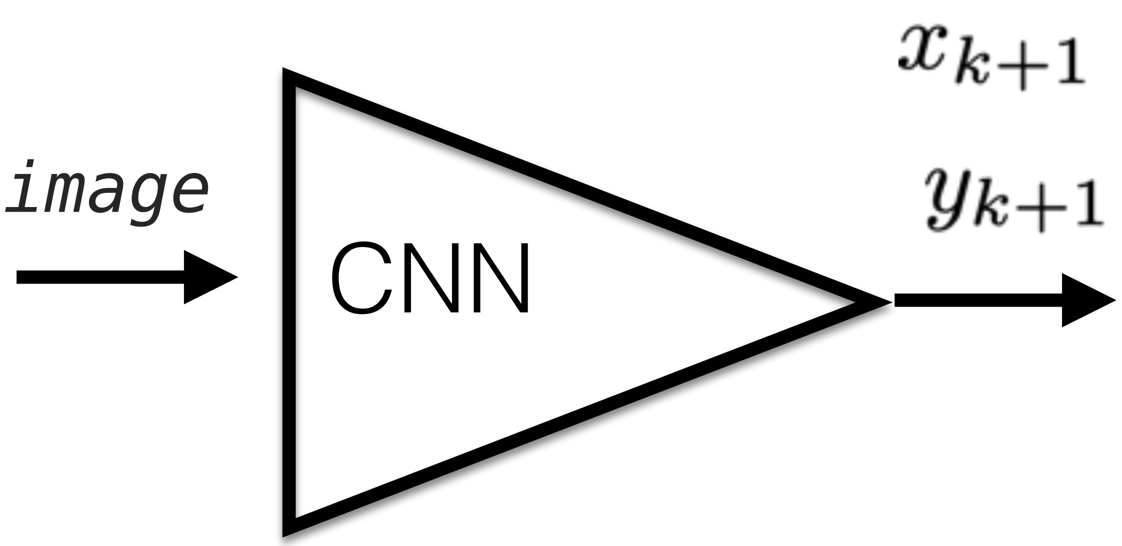
Regularization & overfitting

- Best regularization is using the right structure of the network



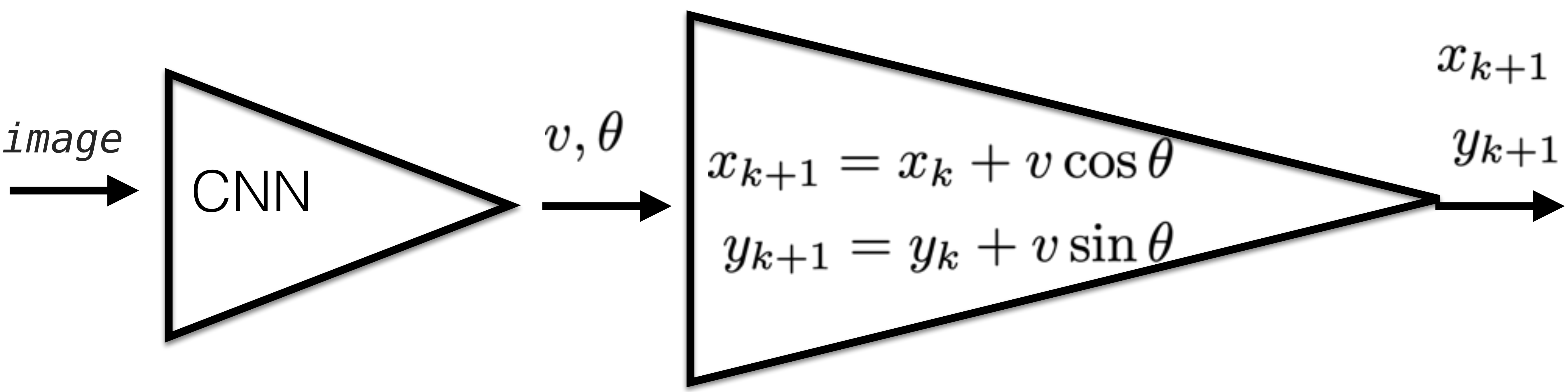
Regularization & overfitting

- Best regularization is using the right structure of the network



Regularization & overfitting

- Best regularization is using the right structure of the network



Regularization & overfitting

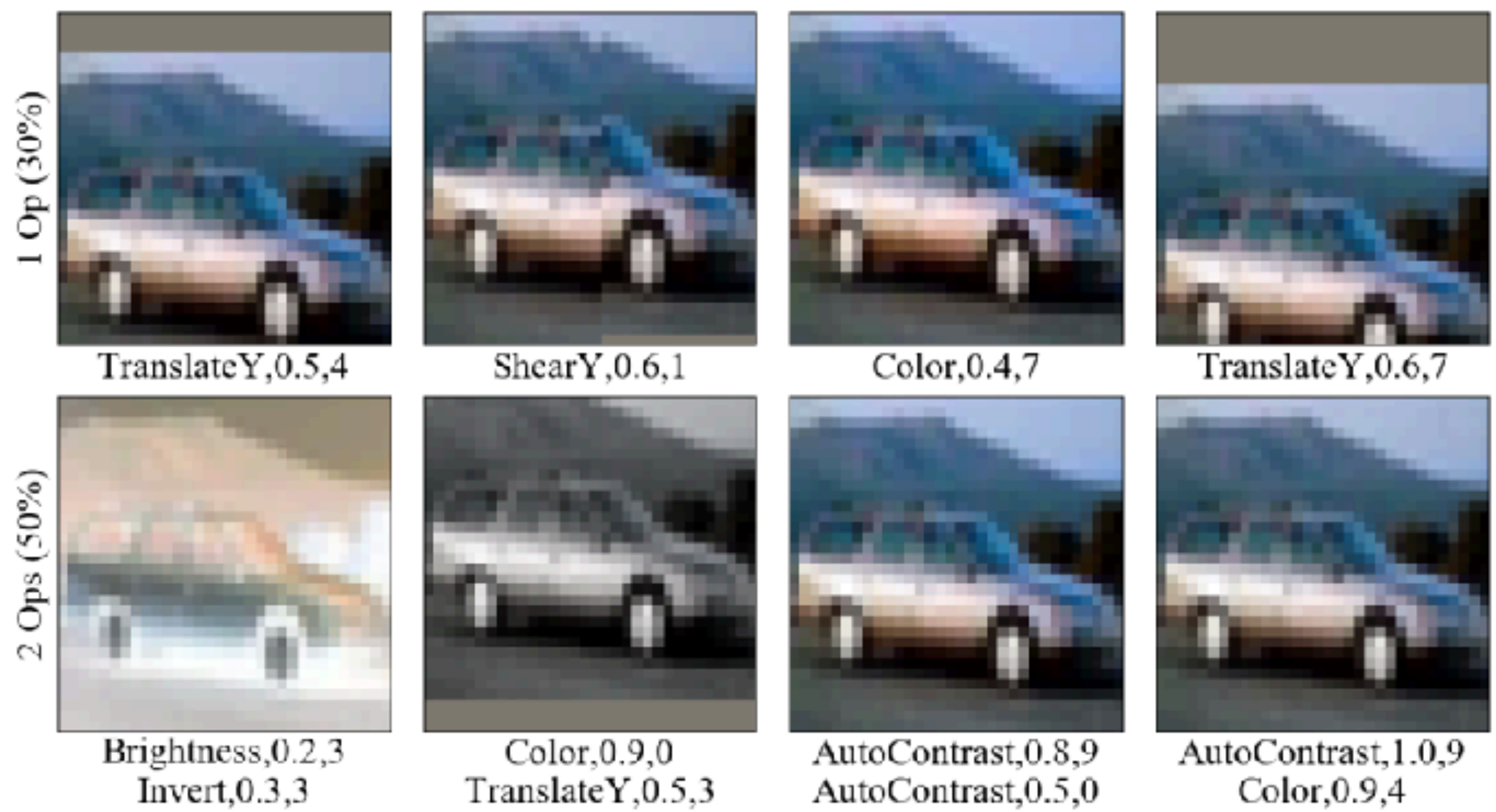
- Best regularization is using the right structure of the network
- L2, L1 norms on weights
  - avoids overfitting and exploding gradient
  - implemented via weight_decay parameter in PyTorch

```
optimizer = torch.optim.Adam(model.parameters(),
lr=1e-3, weight_decay=1e-4)
```

Regularization & overfitting

- Training set augmentation (jittering, mirroring, occlusions, brightness/contrast/color variations)
- Learn augmentation policy (AutoAugment, PBA), which provides good generalization

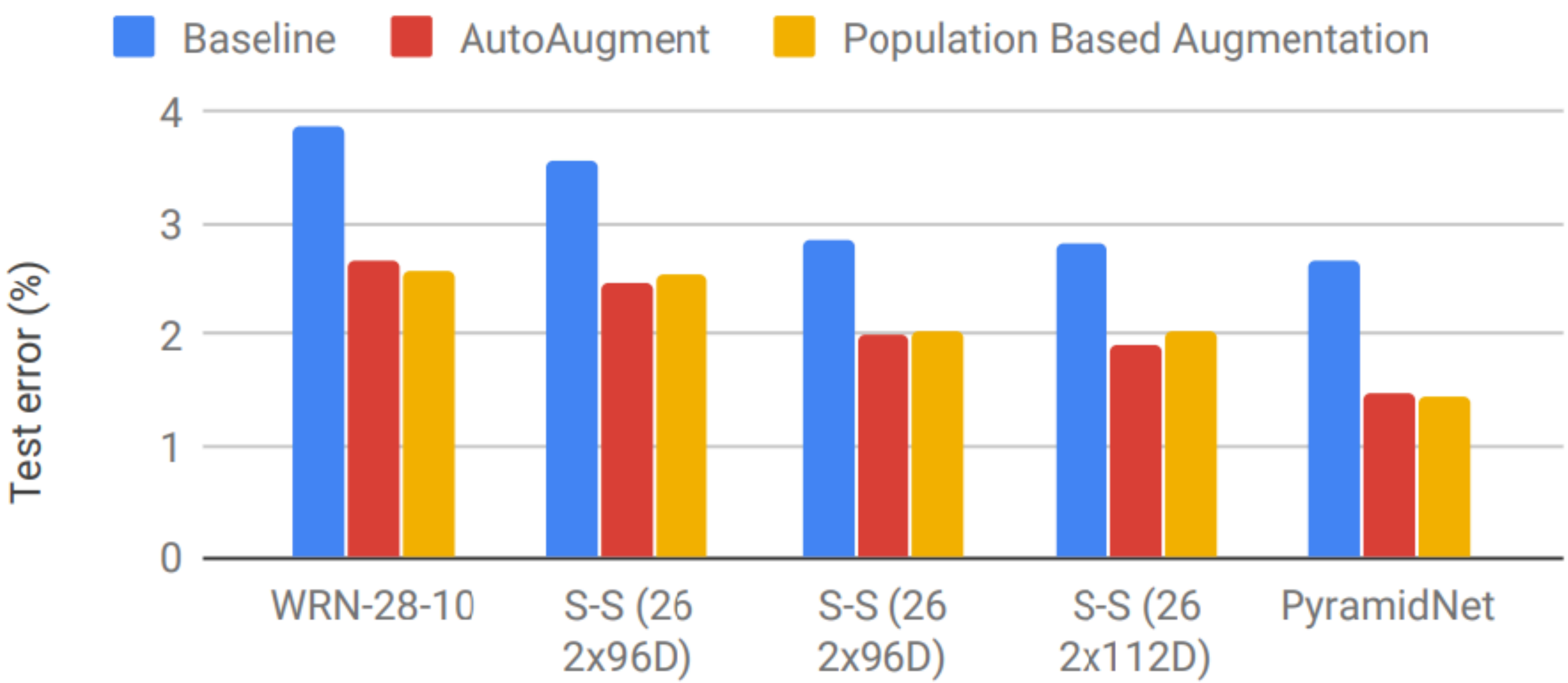
https://arxiv.org/pdf/1905.05393.pdf



Regularization & overfitting

- Training set augmentation (jittering, mirroring, occlusions, brightness/contrast/color variations)
- Learn augmentation policy (AutoAugment, PBA), which provides good generalization

https://arxiv.org/pdf/1905.05393.pdf



Regularization & overfitting

- Batch norm is regularization
  - each iteration it provides different brightness/contrast perturbation
- Ensemble:
  - learn multiple networks=> average of outputs is more stable and allow to predict confidence
- Drop-out layer:
  - suppress layer outputs at random
  - force random subnetworks to work well

```
m = nn.Dropout(p=0.2)
```

  - avoid combination with batch norm !
- Training on pre-trained network
- Weak-supervision and meta-learning (lecture by Patrik)