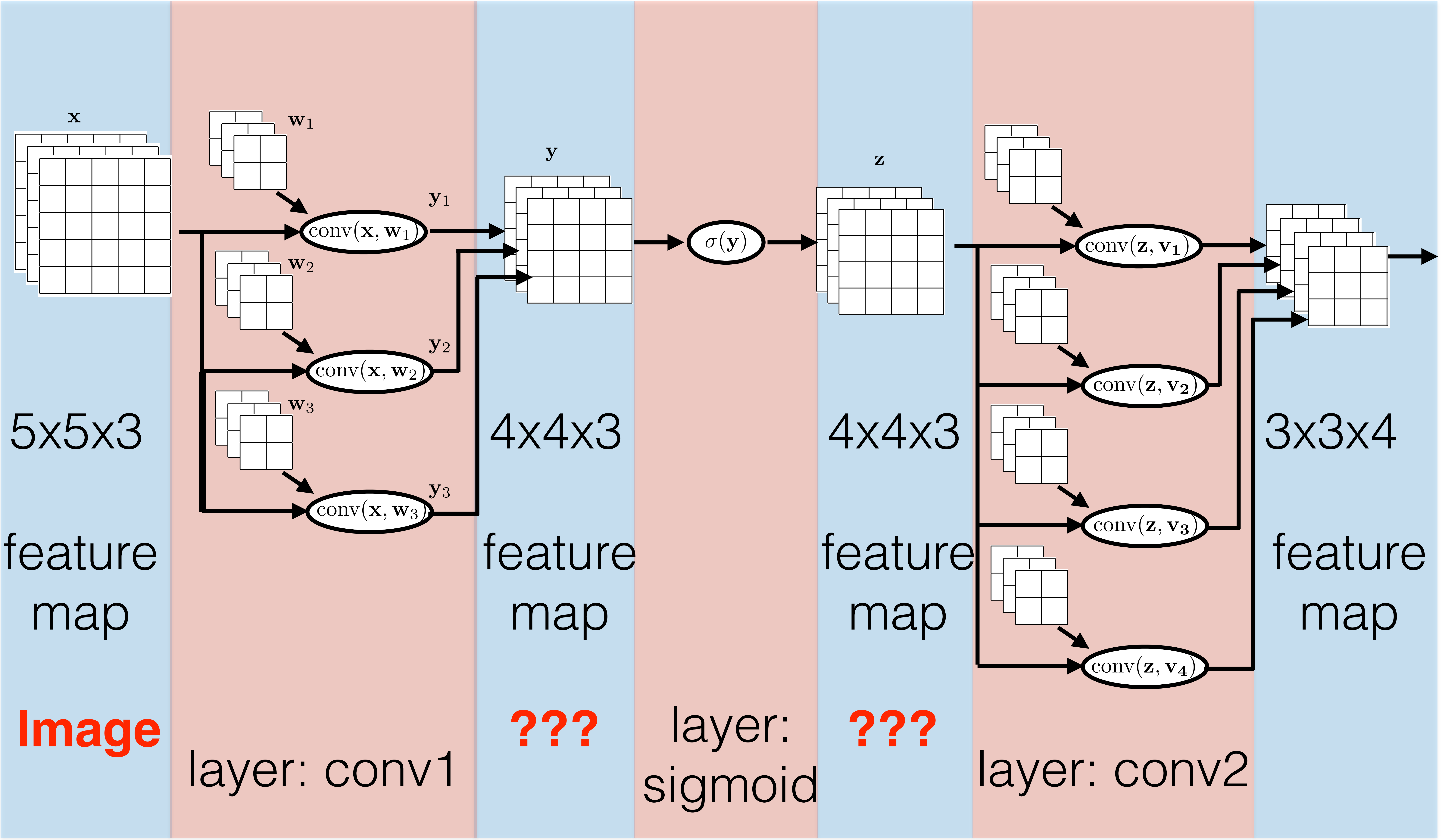
Elementary layers and their issues

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

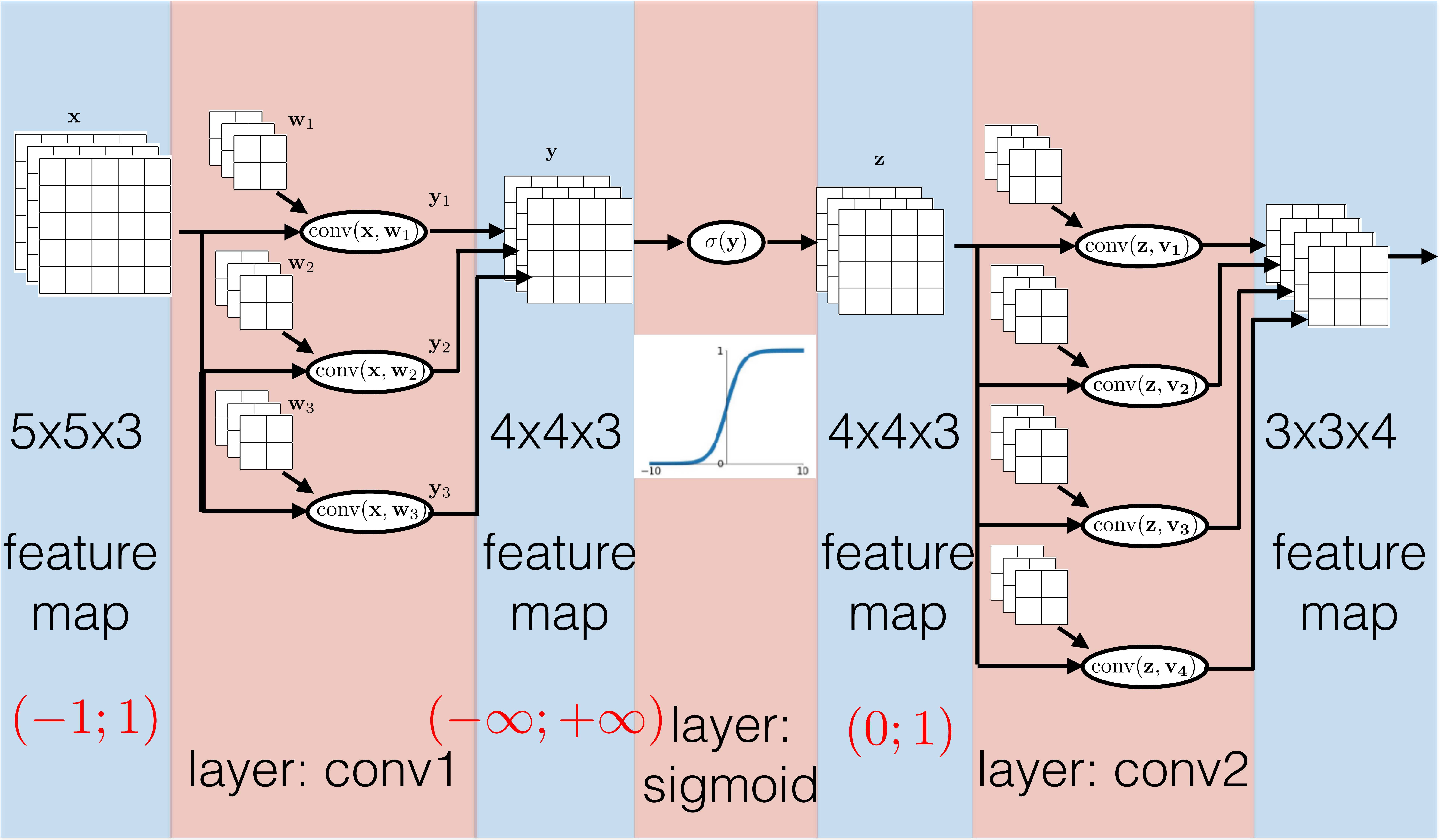
Learning

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



Learning

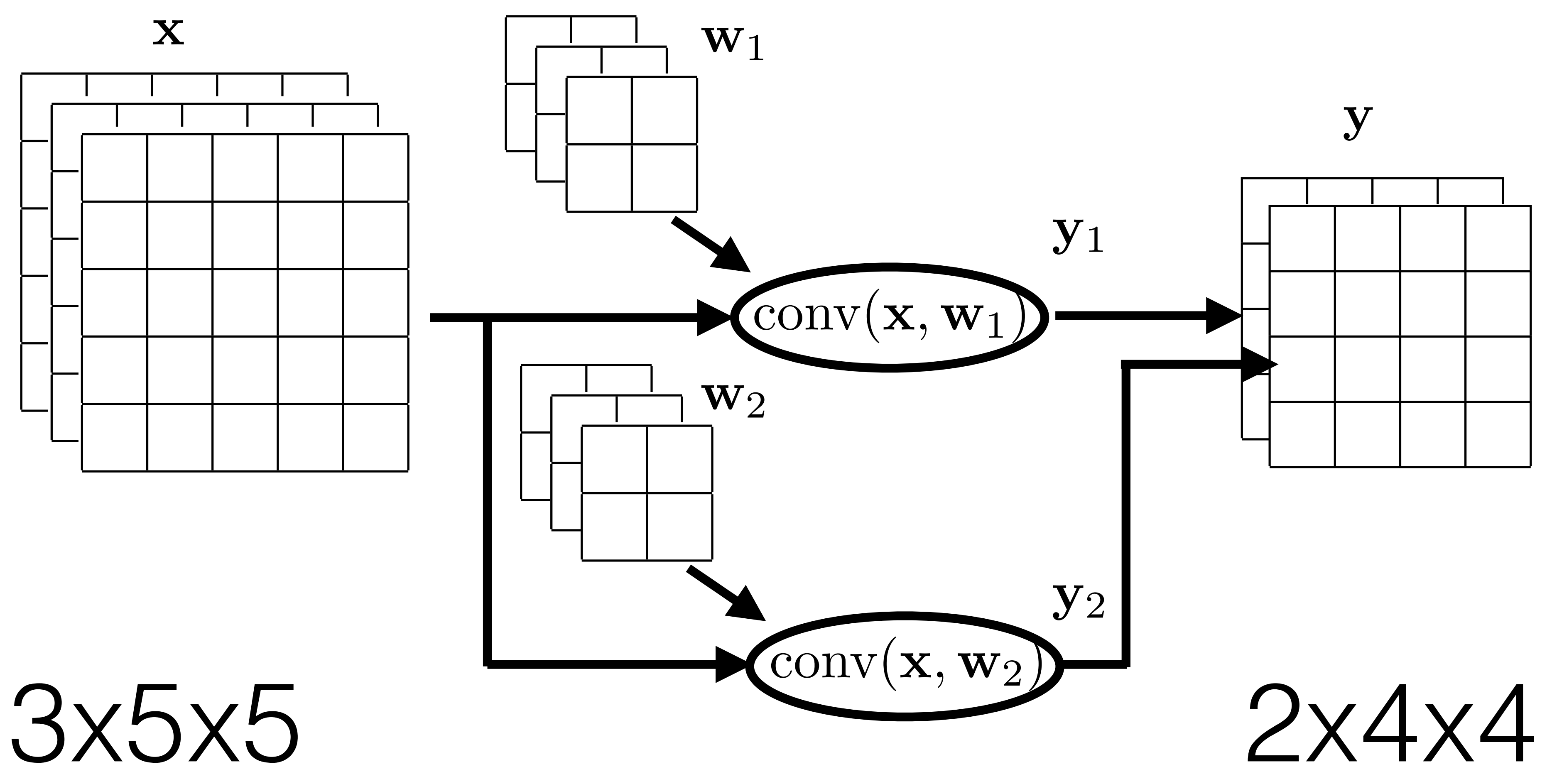
- let us plug image as input, what **values** are propagated?



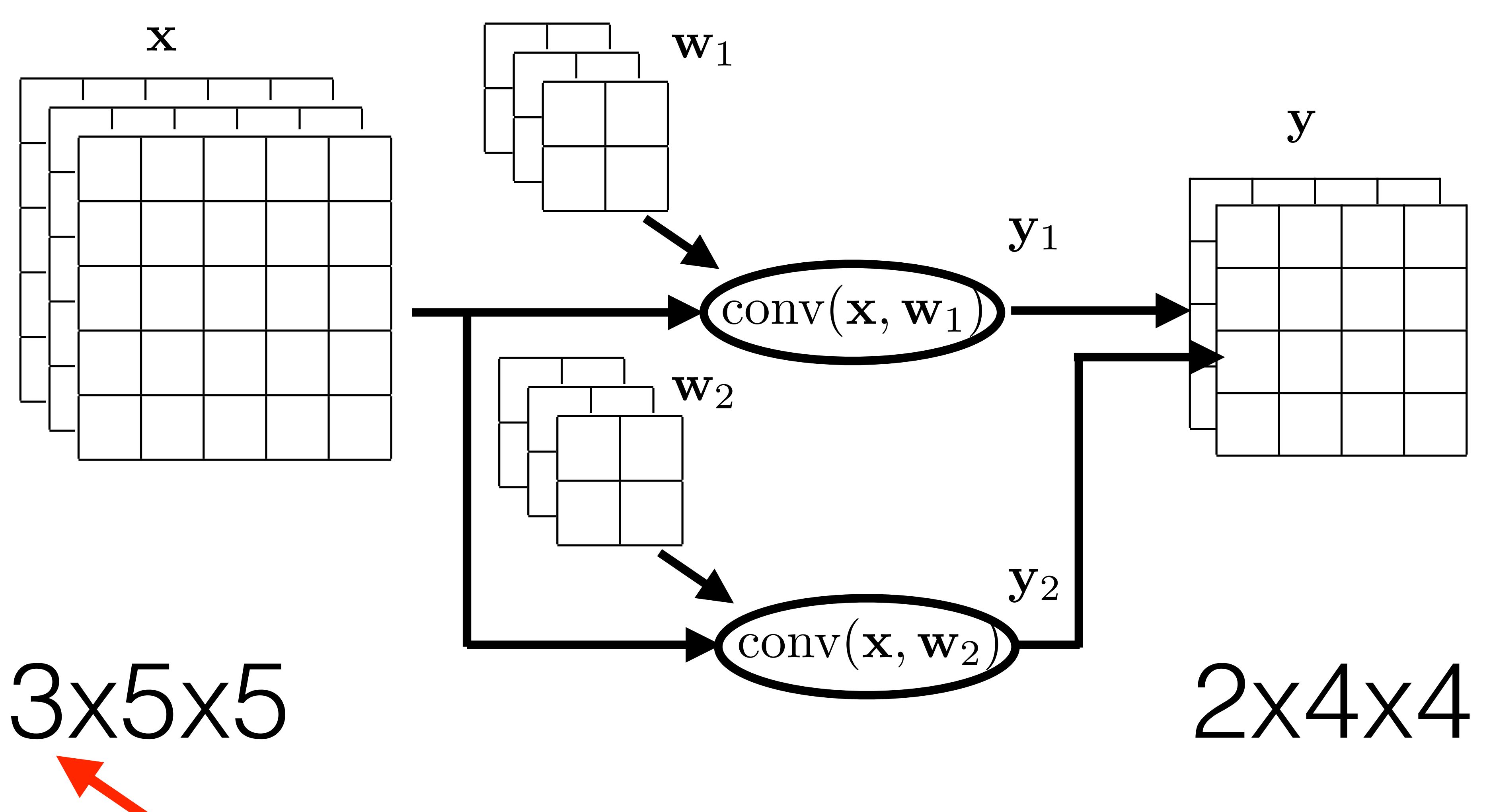
Outline

- 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

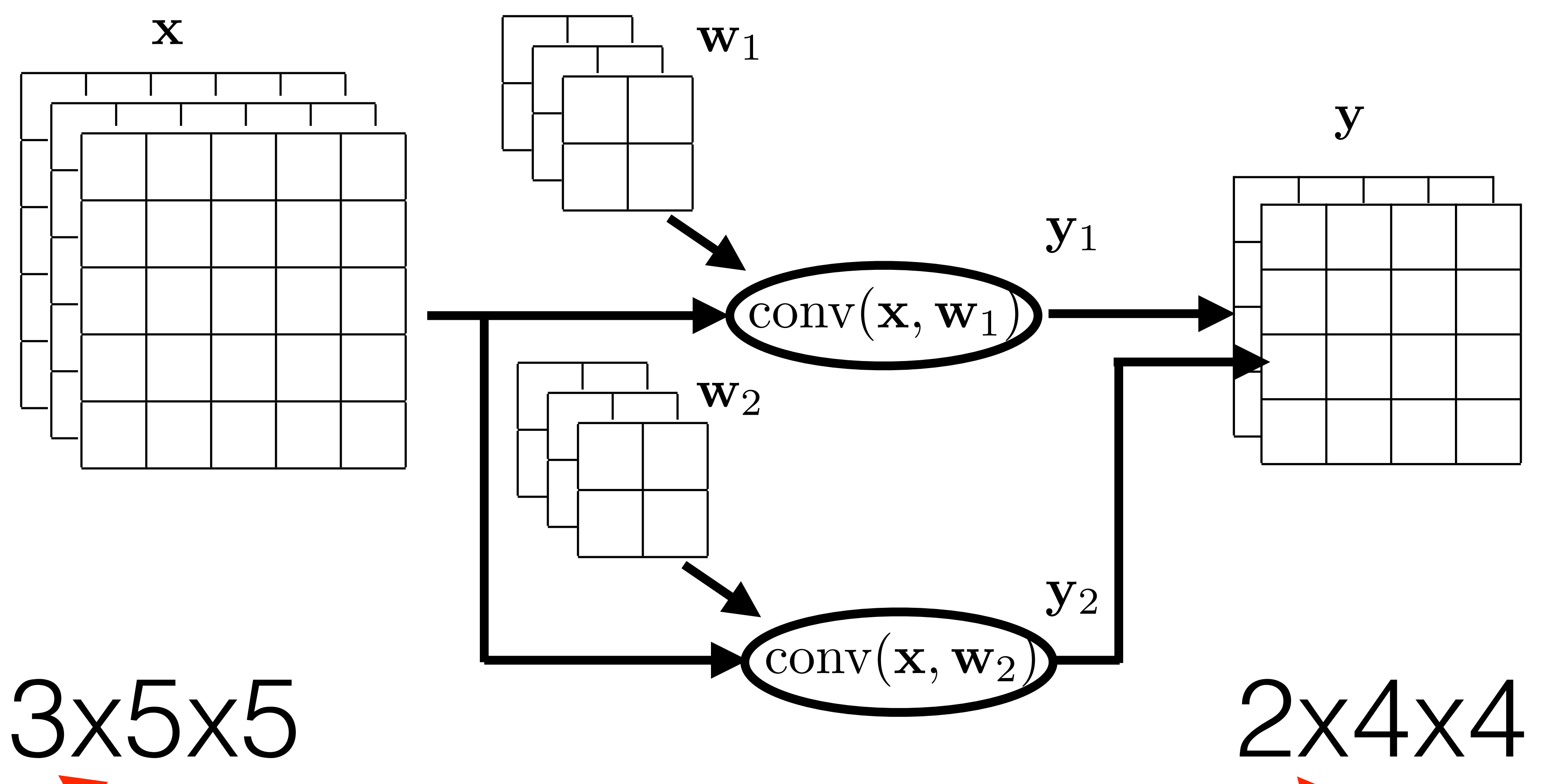


2D convolution forward pass




```
# 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)
```

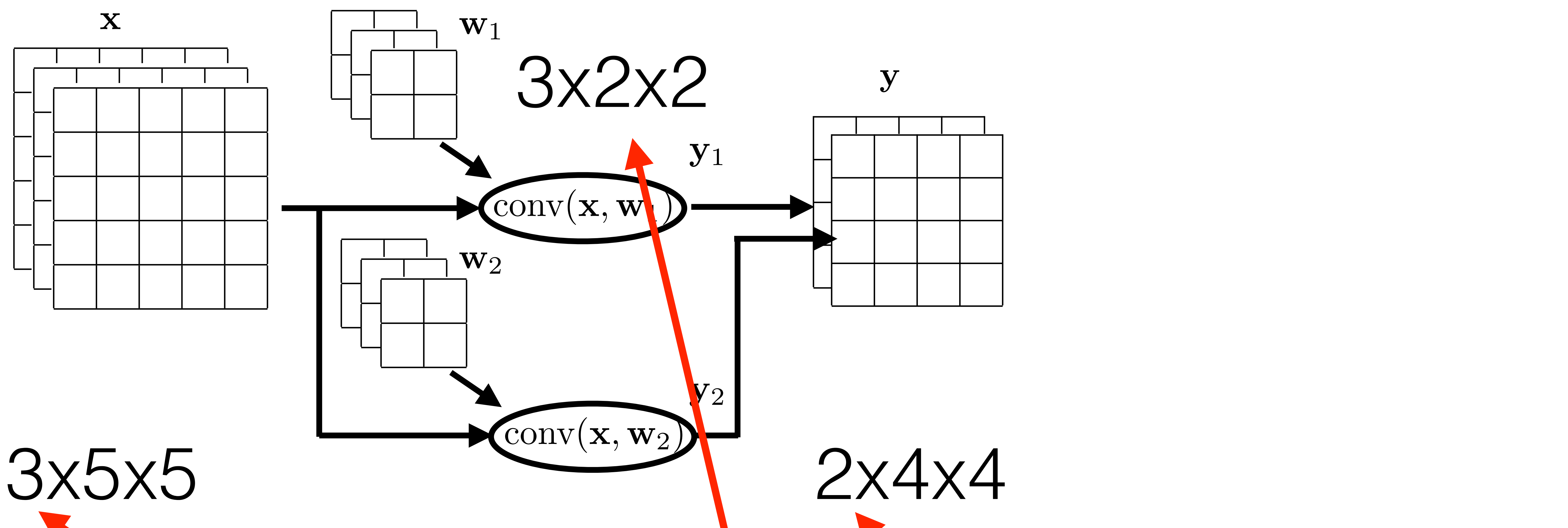
2D convolution forward pass



also number of kernels

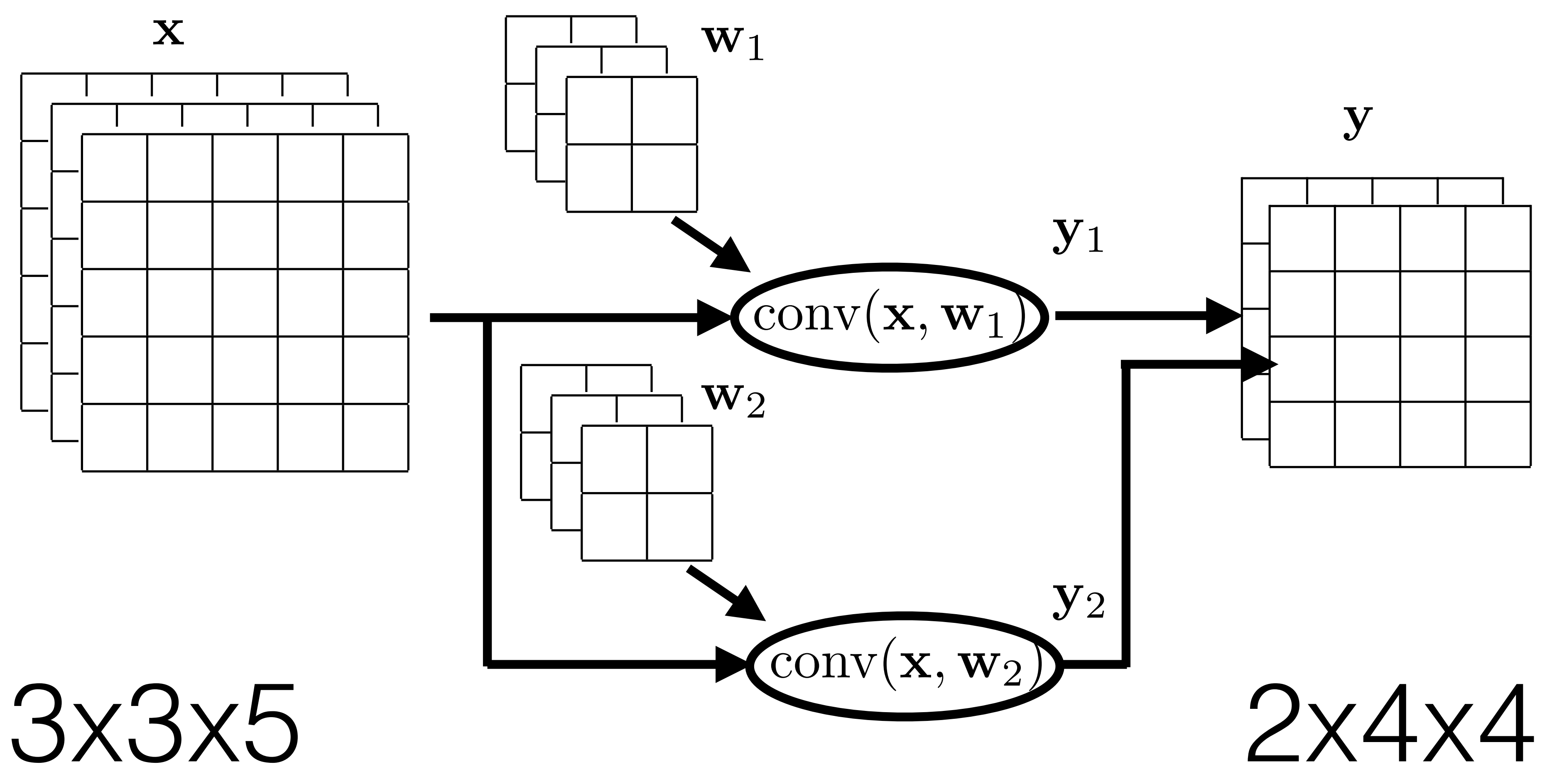
```
# initialise
import torch.nn as nn
# define 2D convolutional layer
first_layer = nn.Conv2d(in_channels=3, out_channels=3,
                        kernel_size=2, stride=1,
                        padding=1)
```

2D convolution forward pass




```
# 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



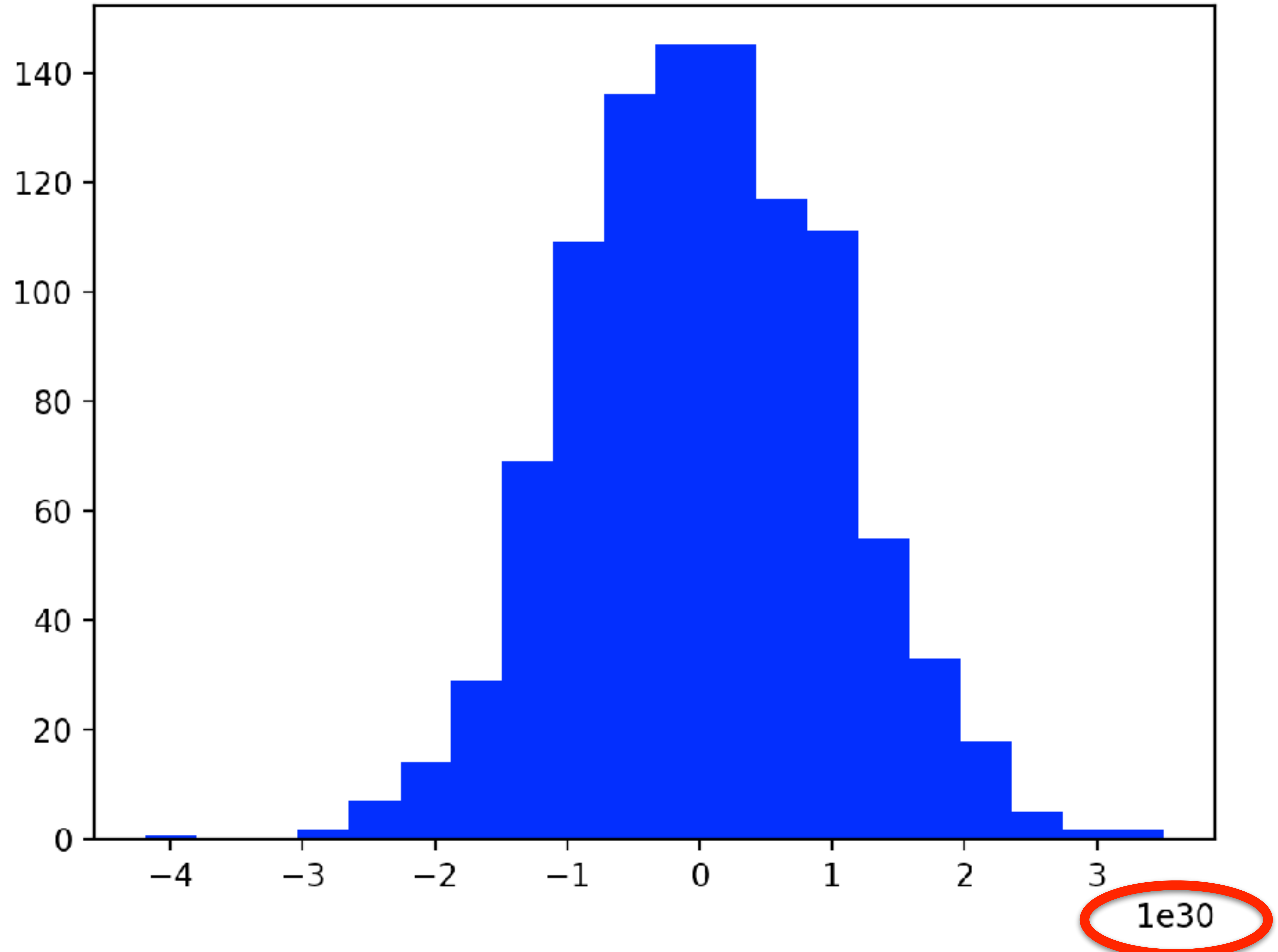
Very important property of convolutional layer is:

jvp 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
```



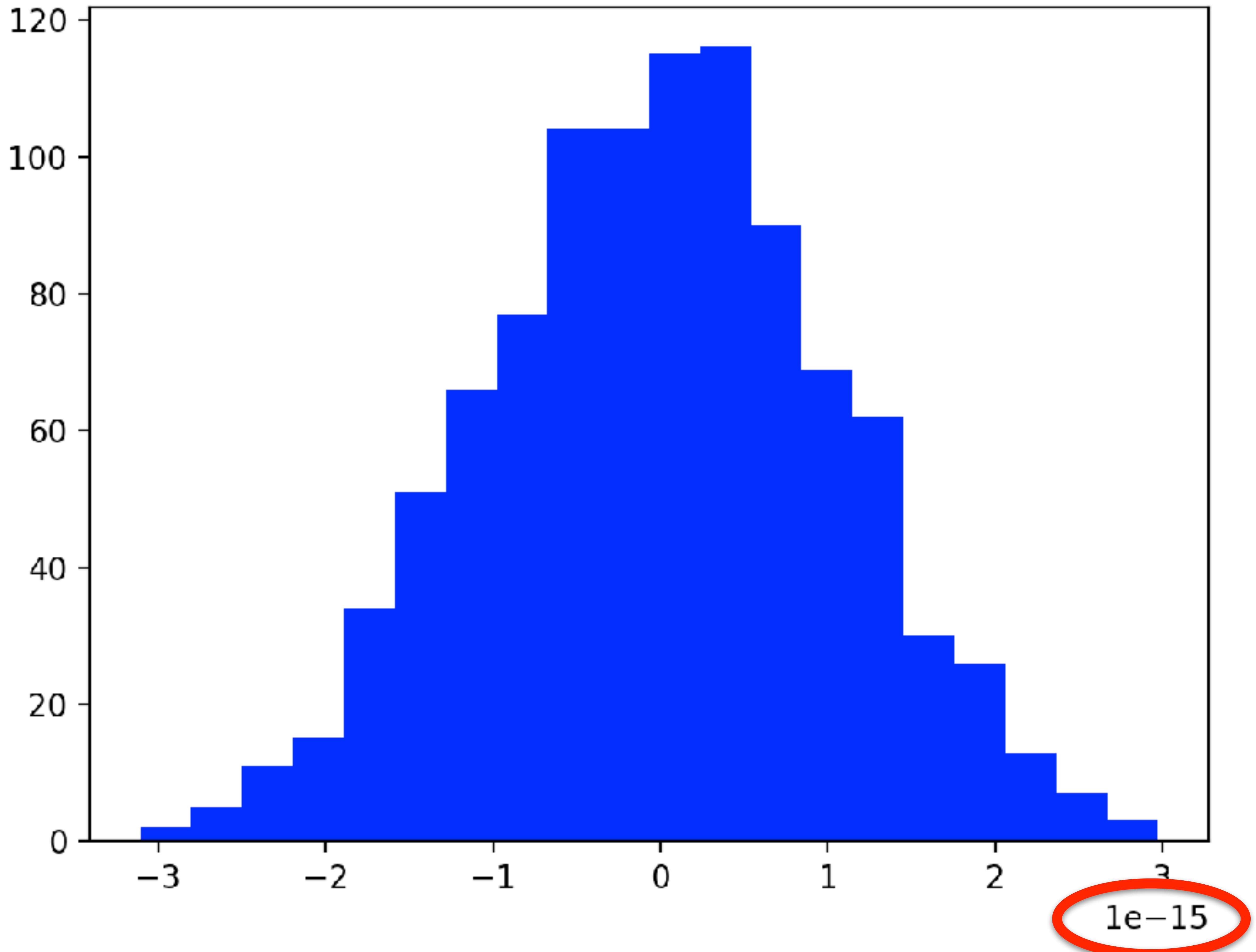
=>Gradient clipping

Value-based
vs
Norm-based

Learning

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

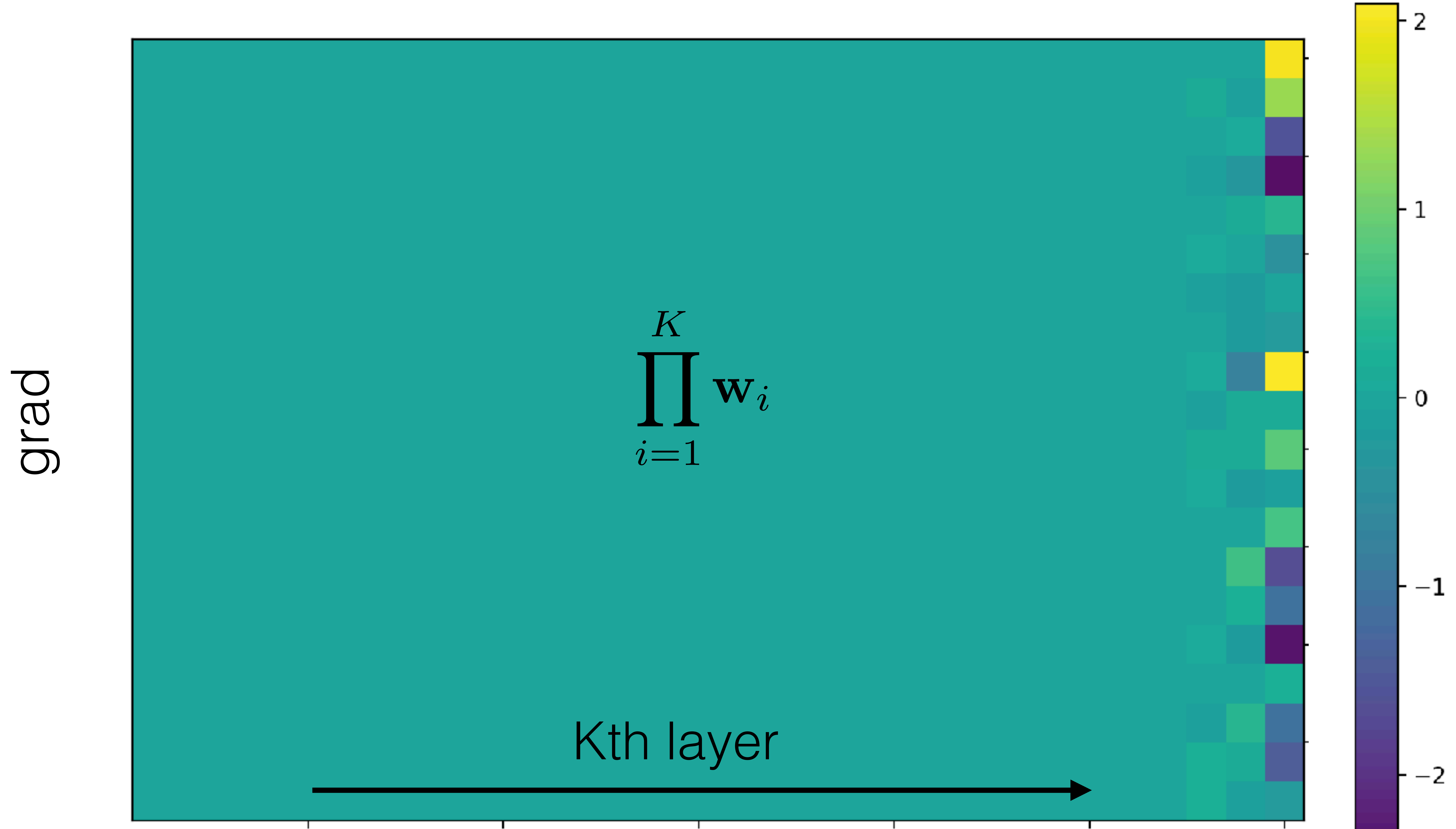
```
y = torch.randn(1000,1)
for i in range(30):
    weights = torch.randn(1000,1000)/100
    y =  weights @ y
```



Learning

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

```
y.sum().backward()
x.grad
```



$$\prod_{i=1}^{K} \mathbf{w}_i$$

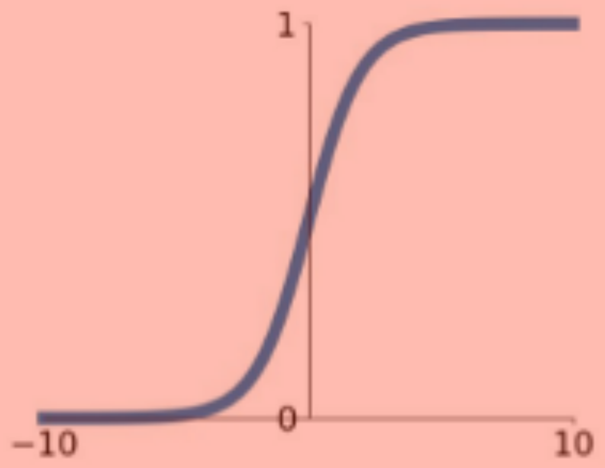
Outline

- 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

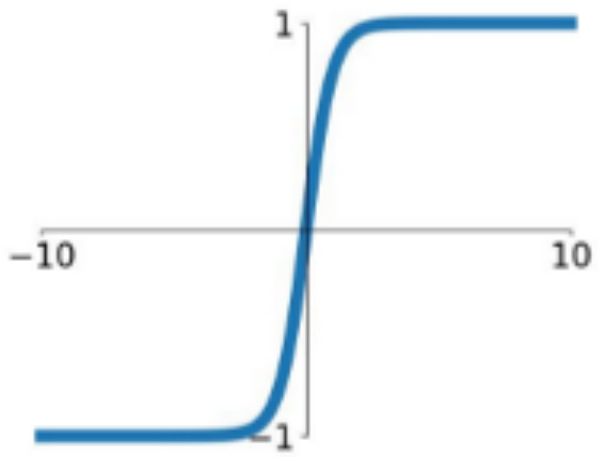
Sigmoid

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



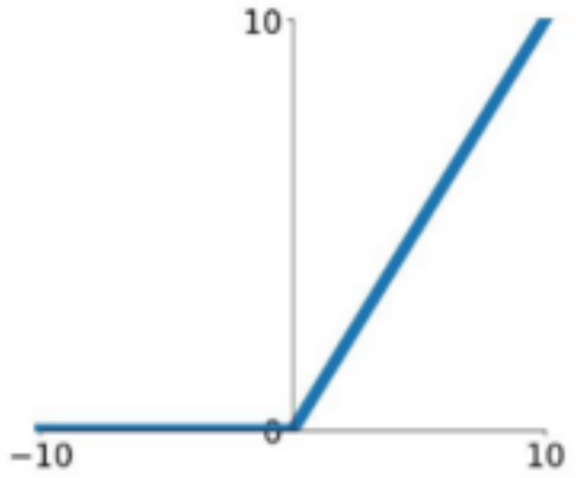
tanh

$\tanh(x)$



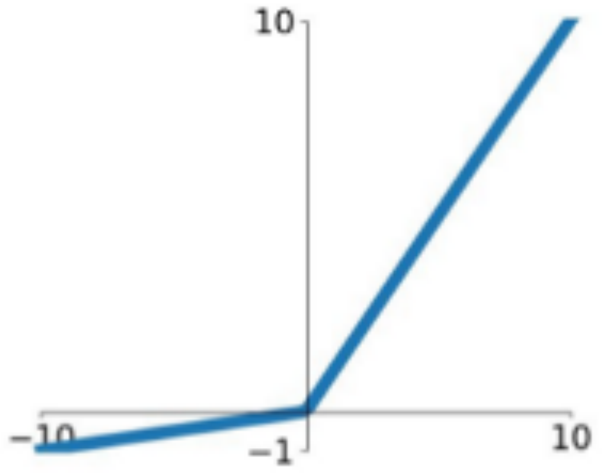
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

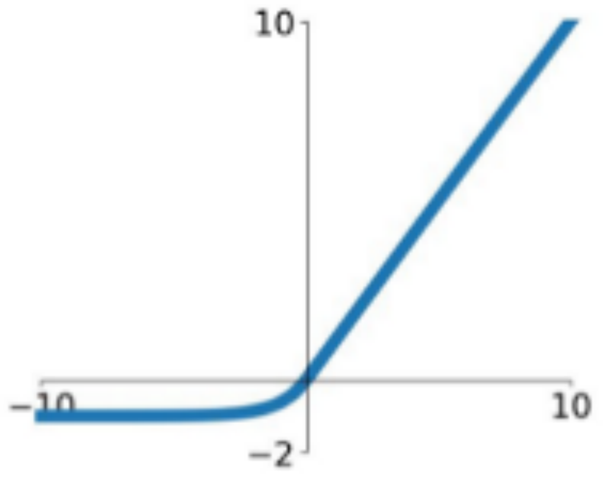


Maxout

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

ELU

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

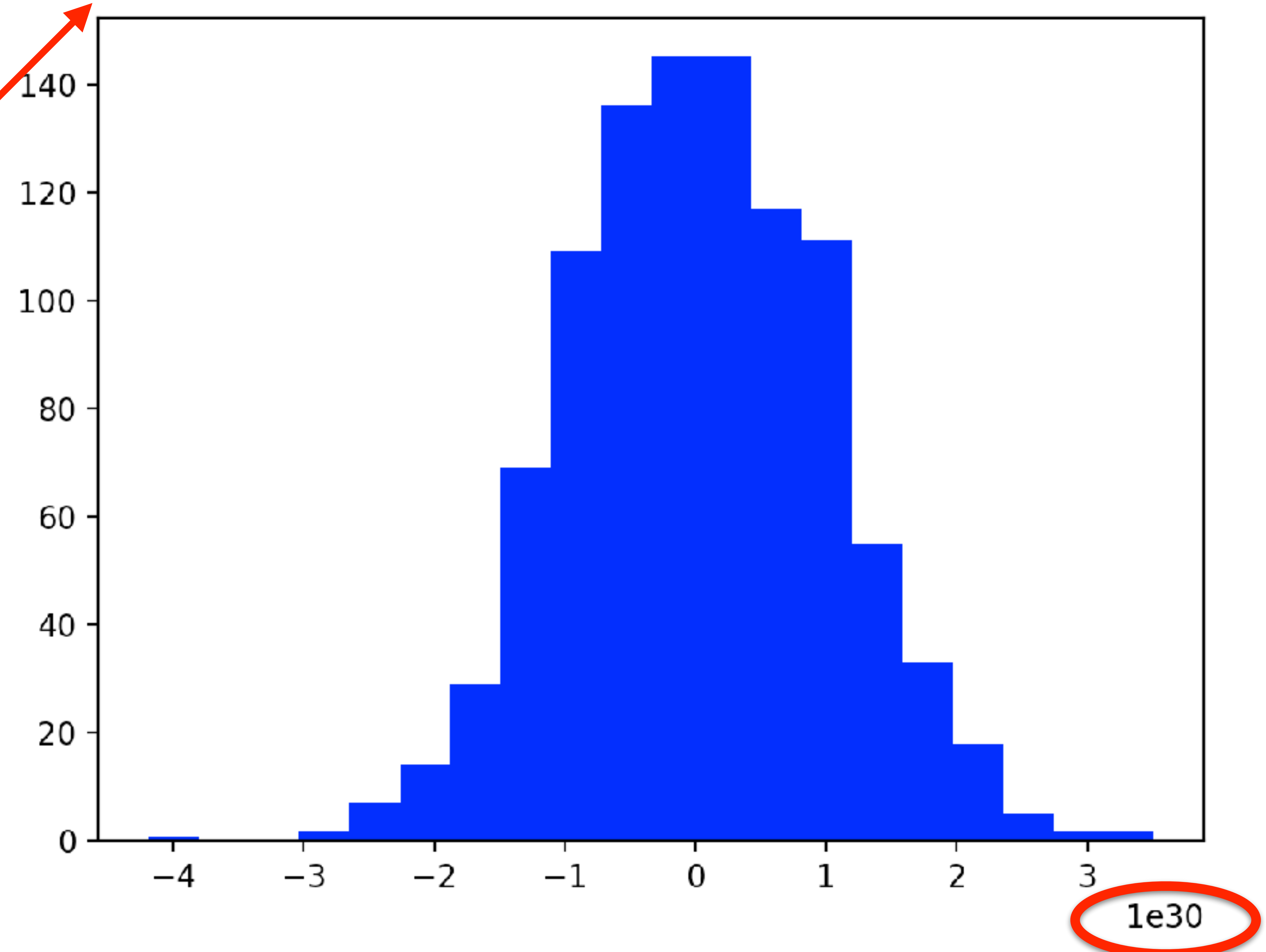


Learning

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
```

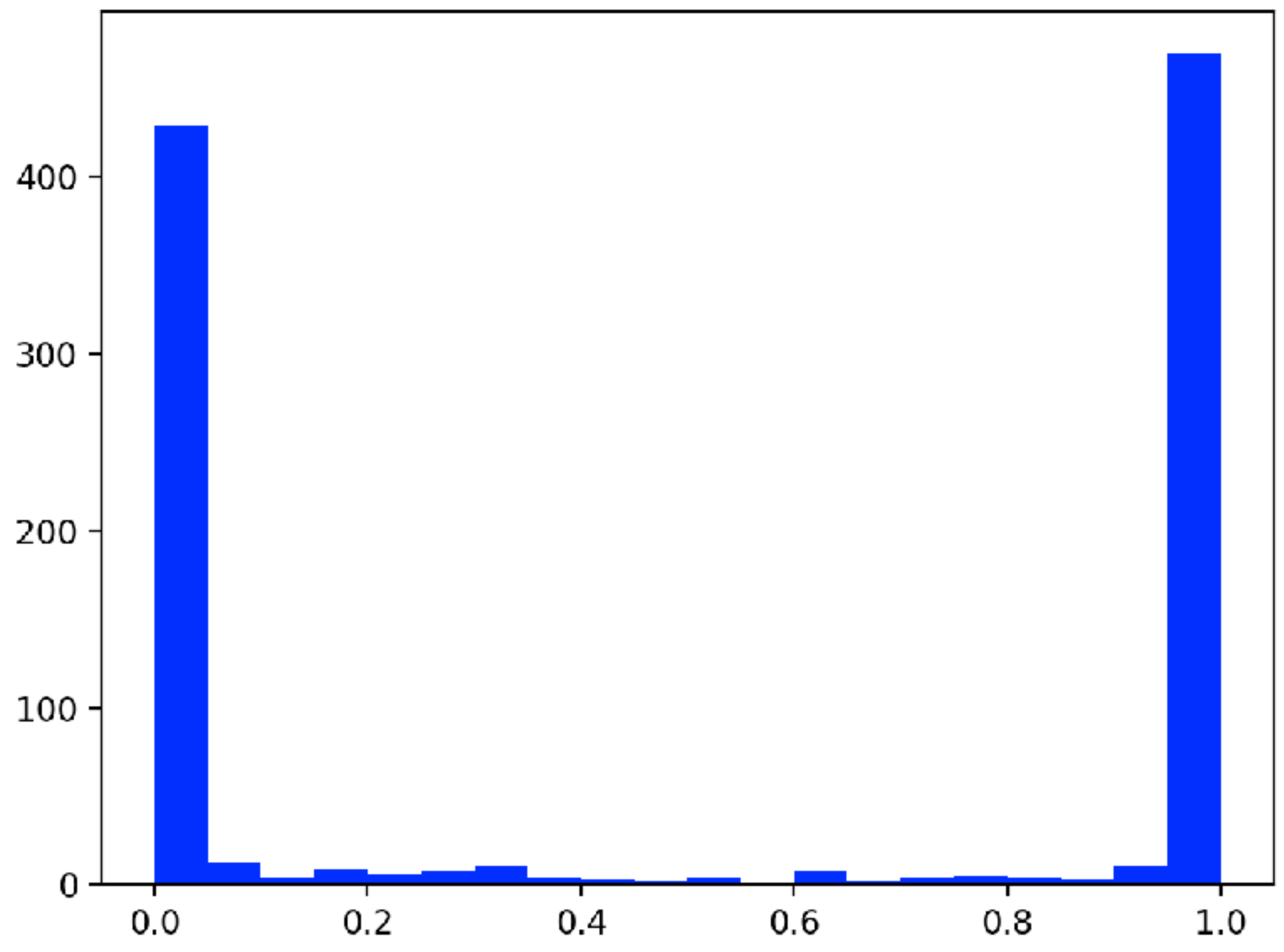
sigmoid



Learning

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

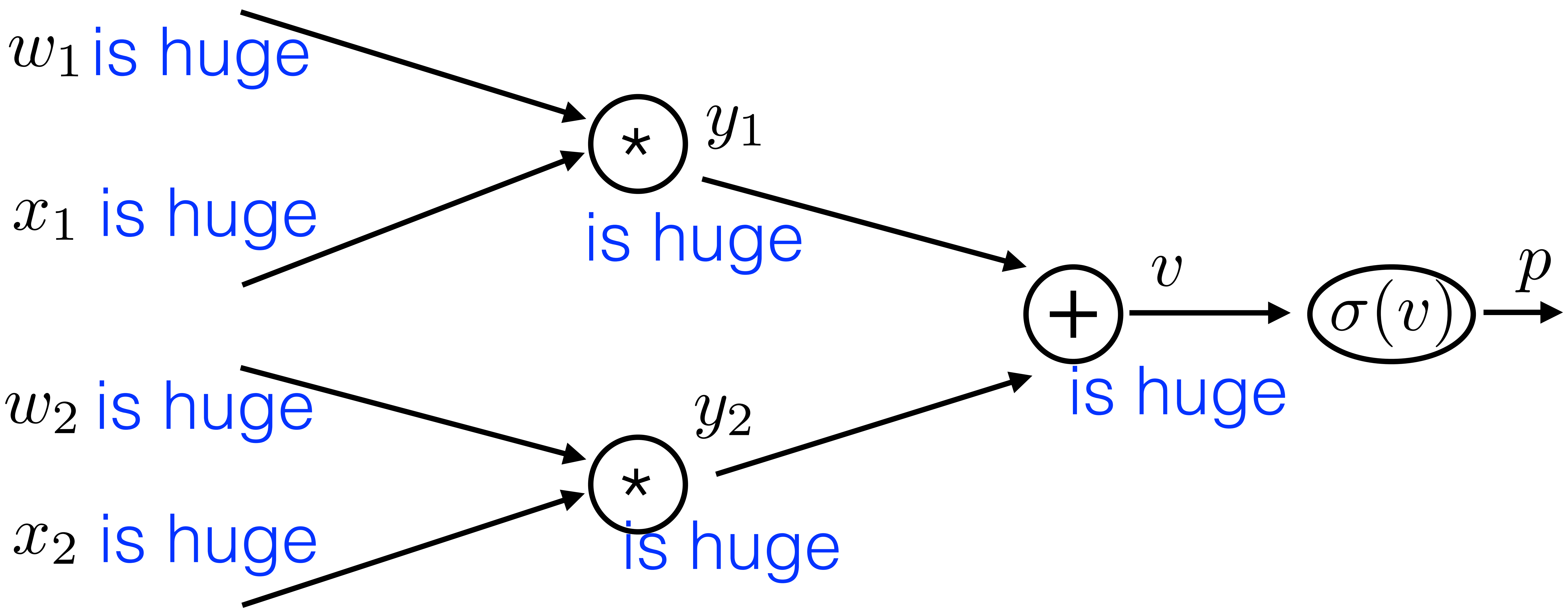
```
y = torch.randn(1000,1)
for i in range(30):
    weights = torch.randn(1000,1000)
    y =  torch.sigmoid(weights @ y)
```

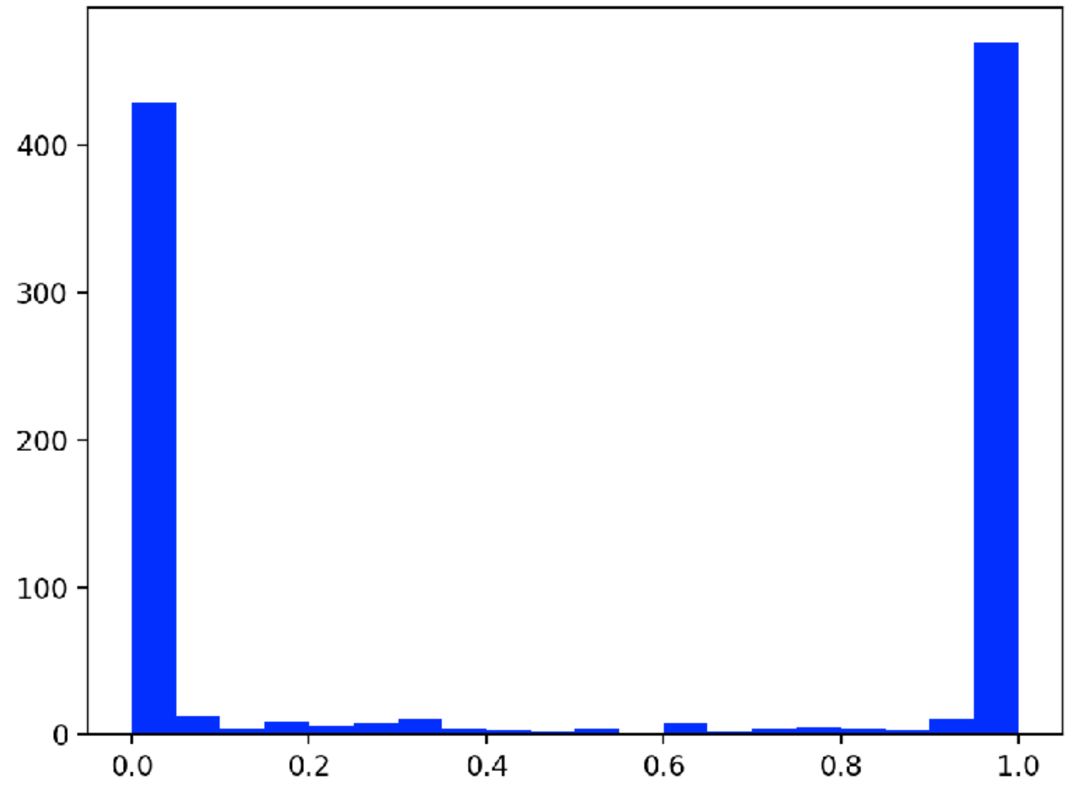


- what happen to **backprop gradient** when weights are **huge**?

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

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

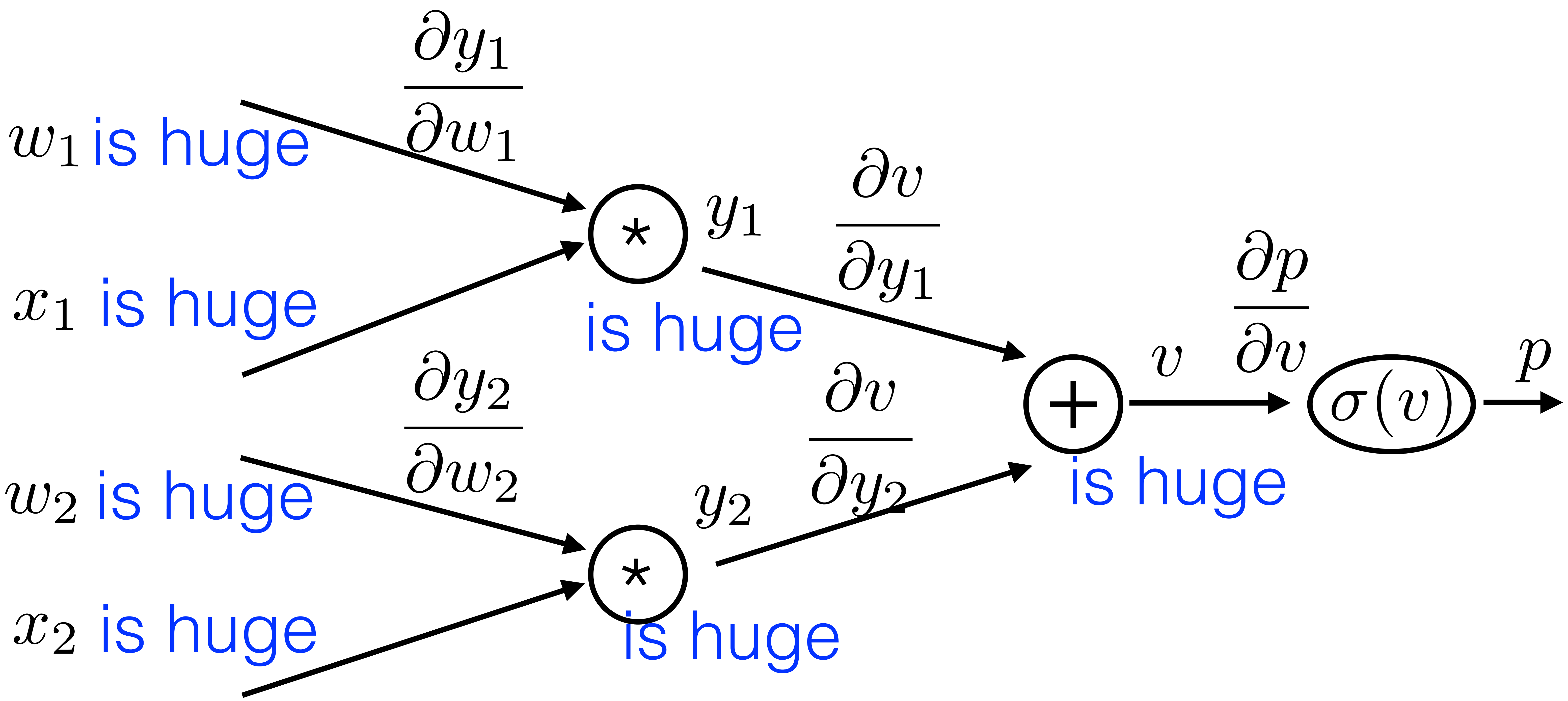


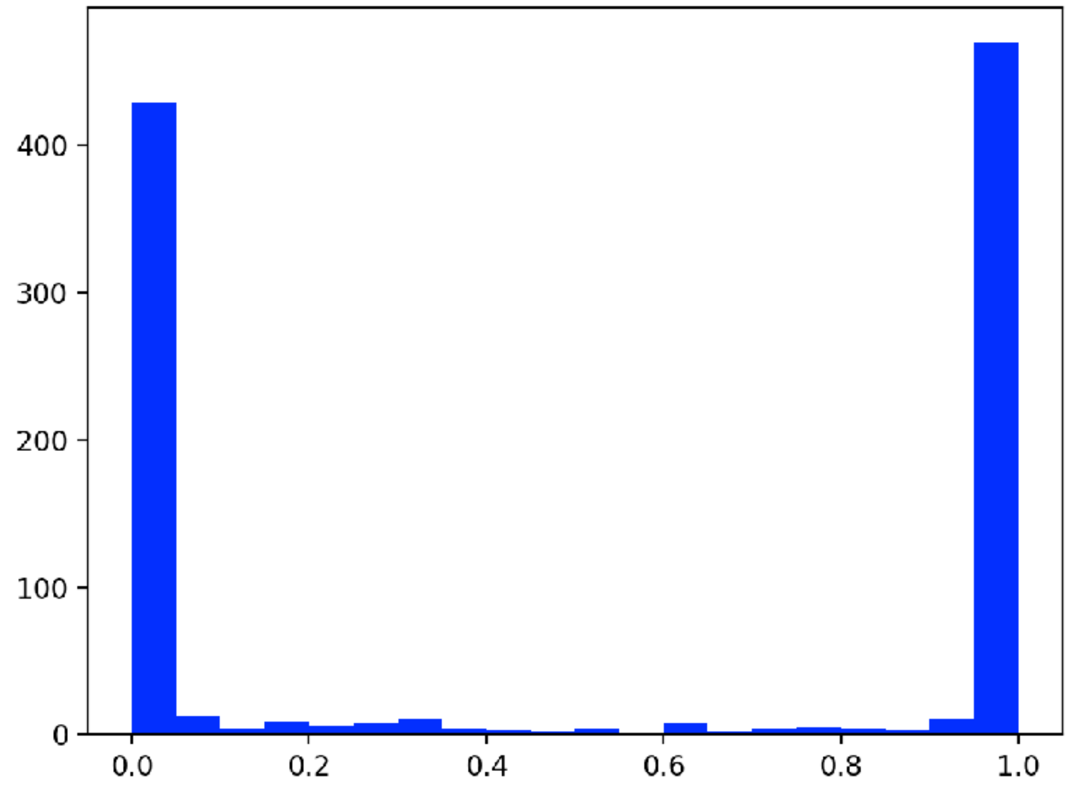


- what happen to **backprop gradient** when weights are **huge**?

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

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

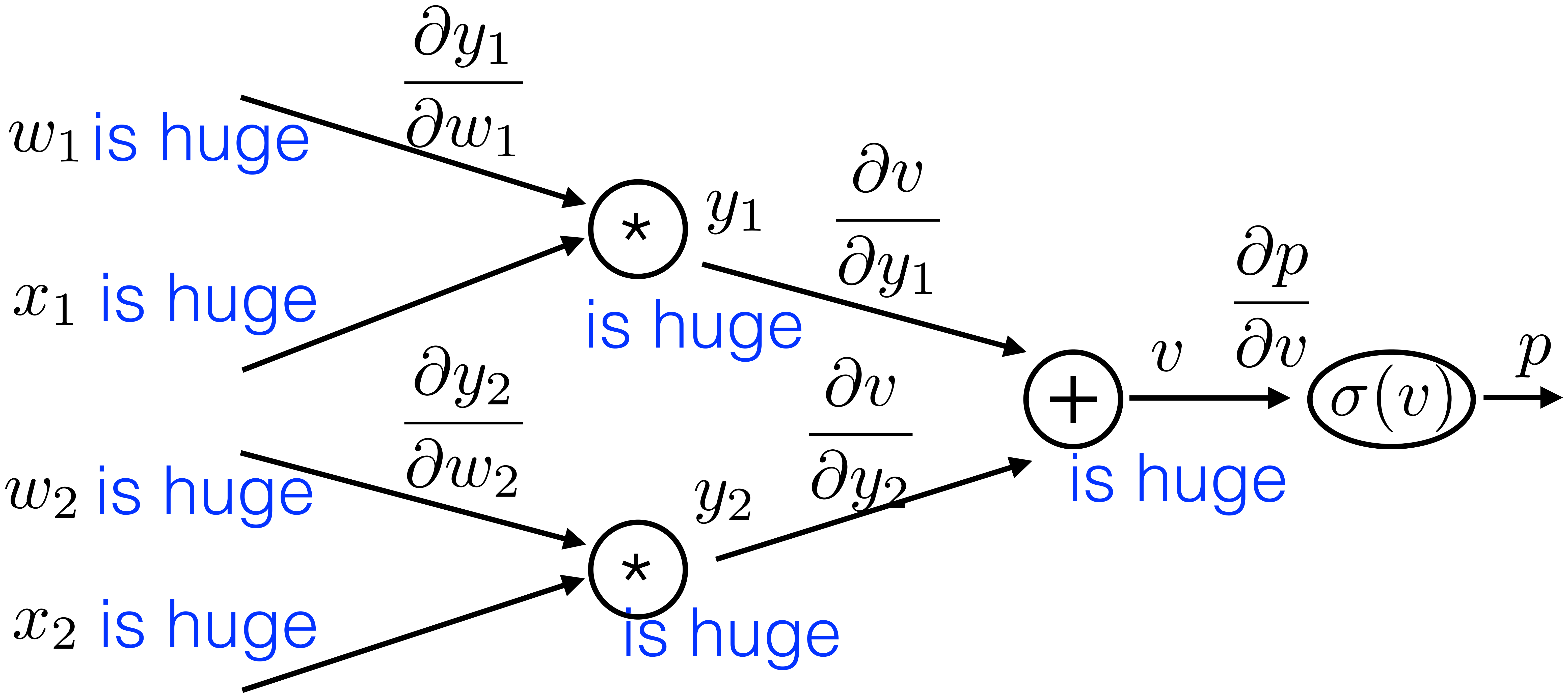


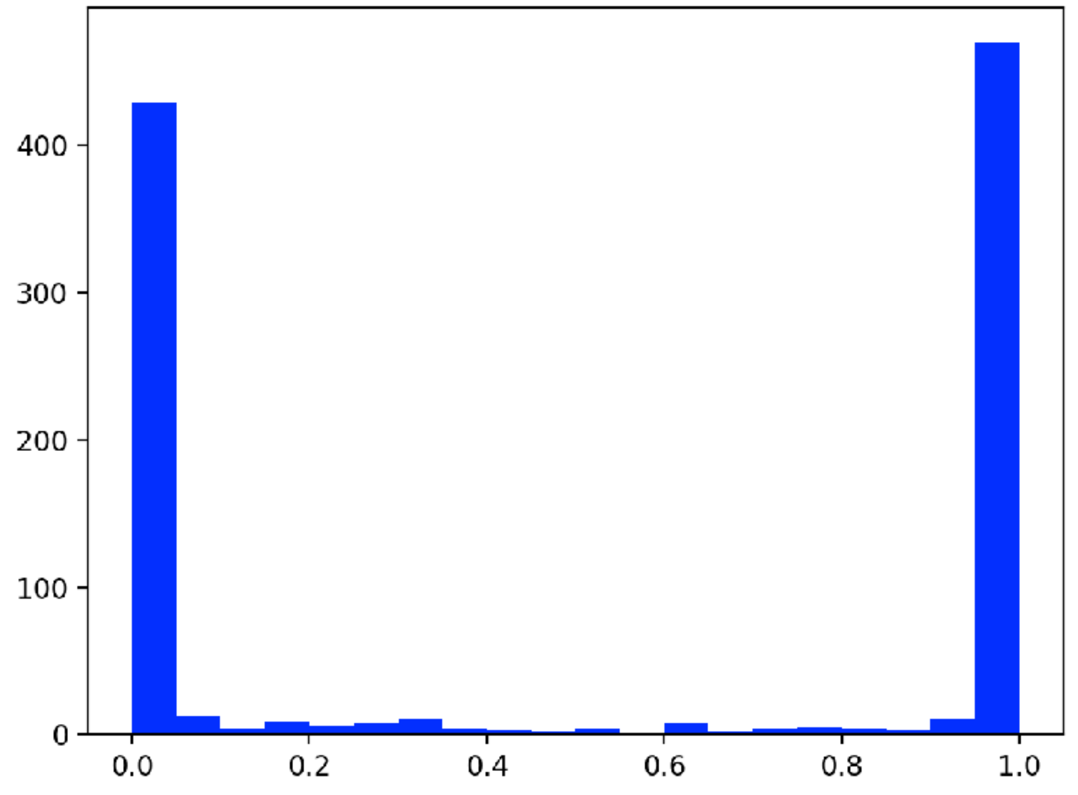


- what happen to **backprop gradient** when weights are **huge**?

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

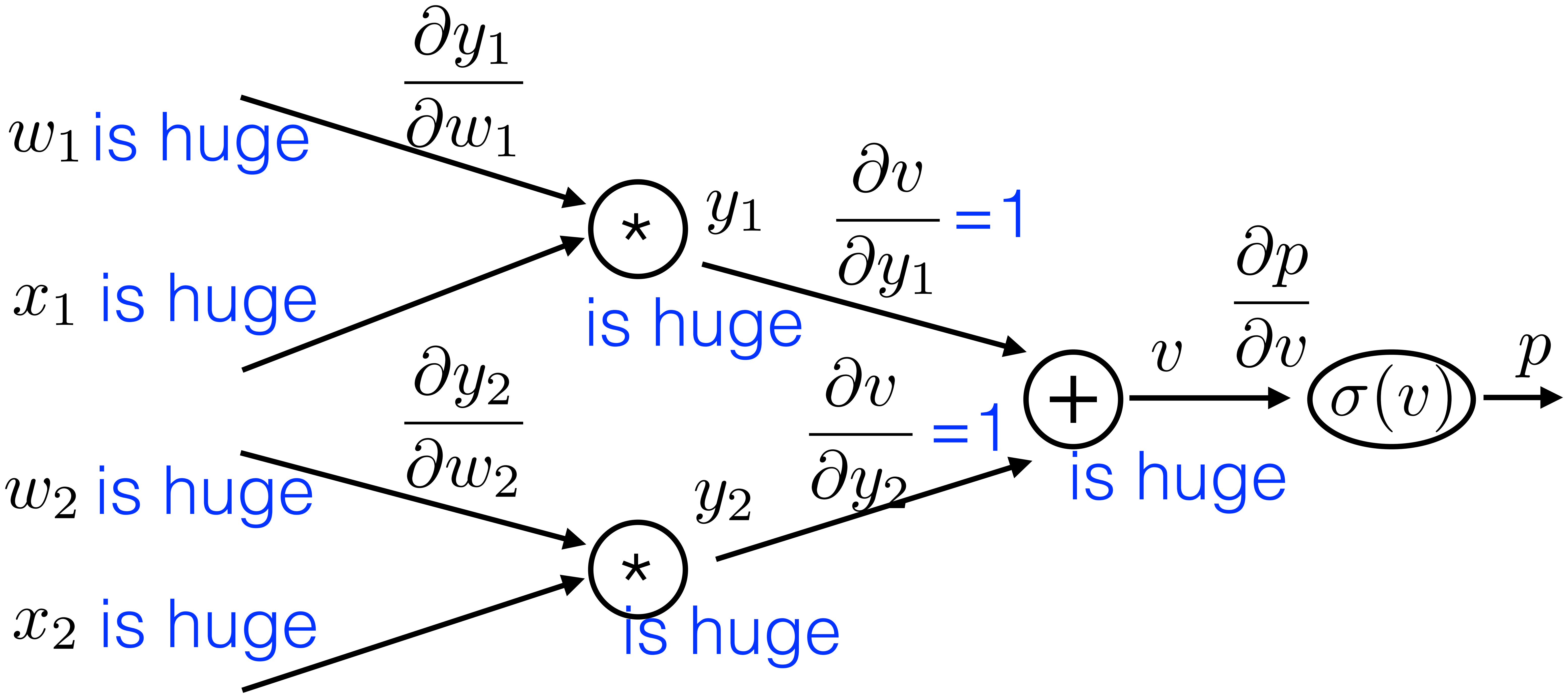
$$\frac{\partial p}{\partial w_2} = \frac{\partial y_2}{\partial w_2}\frac{\partial v}{\partial y_1}\frac{\partial p}{\partial v} =?$$

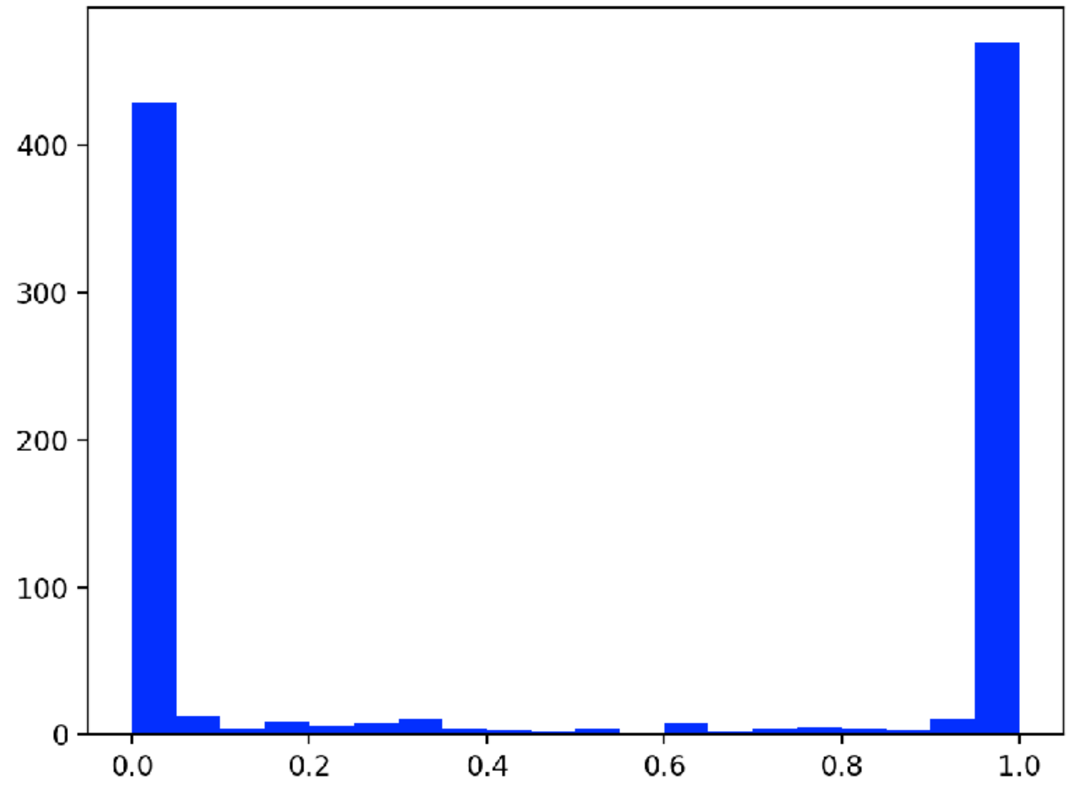




- what happen to **backprop gradient** when weights are **huge**?

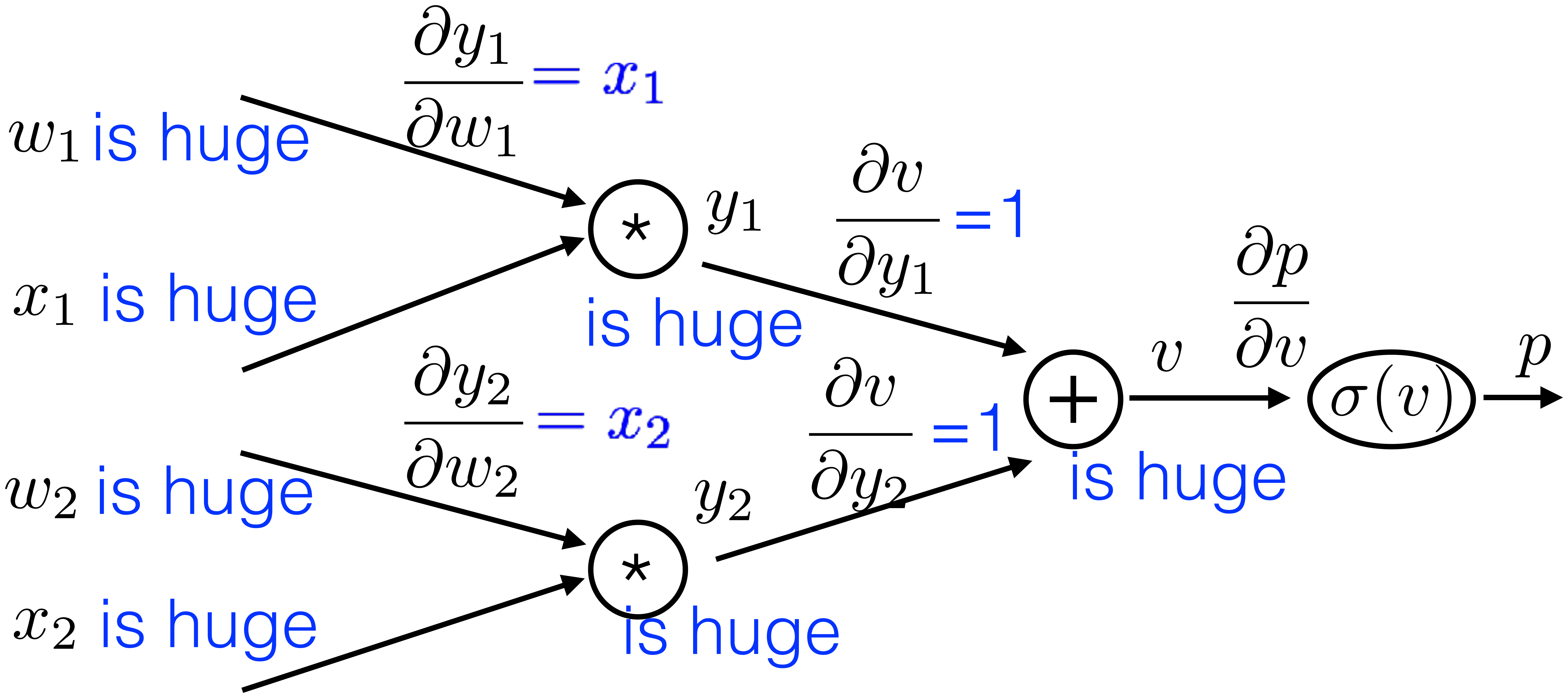
$$\frac{\partial p}{\partial w_1} = \frac{\partial y_1}{\partial w_1} \cdot 1 \quad \frac{\partial p}{\partial v} = ? \qquad \frac{\partial p}{\partial w_2} = \frac{\partial y_2}{\partial w_2} \cdot 1 \quad \frac{\partial p}{\partial v} = ?$$

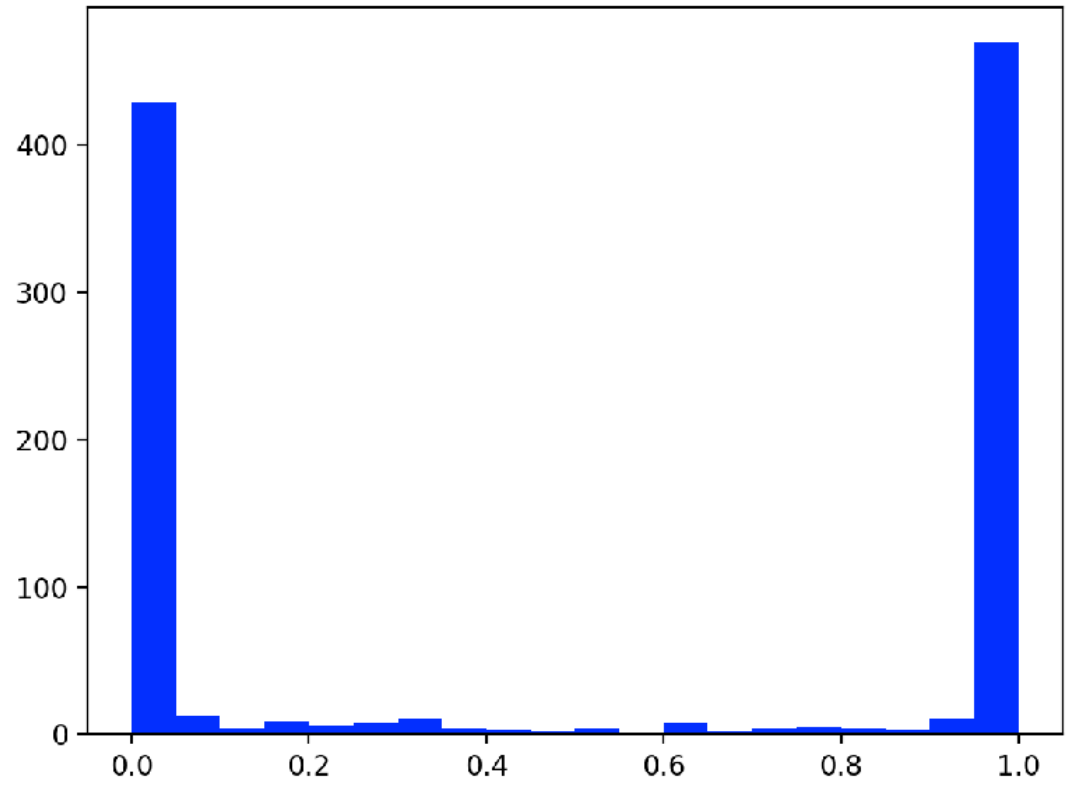




- what happen to **backprop gradient** when weights are **huge**?

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

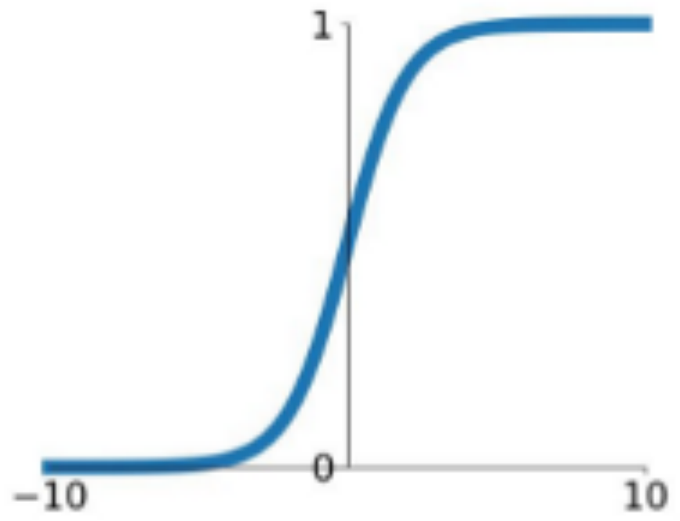




- what happen to **backprop gradient** when weights are **huge**?

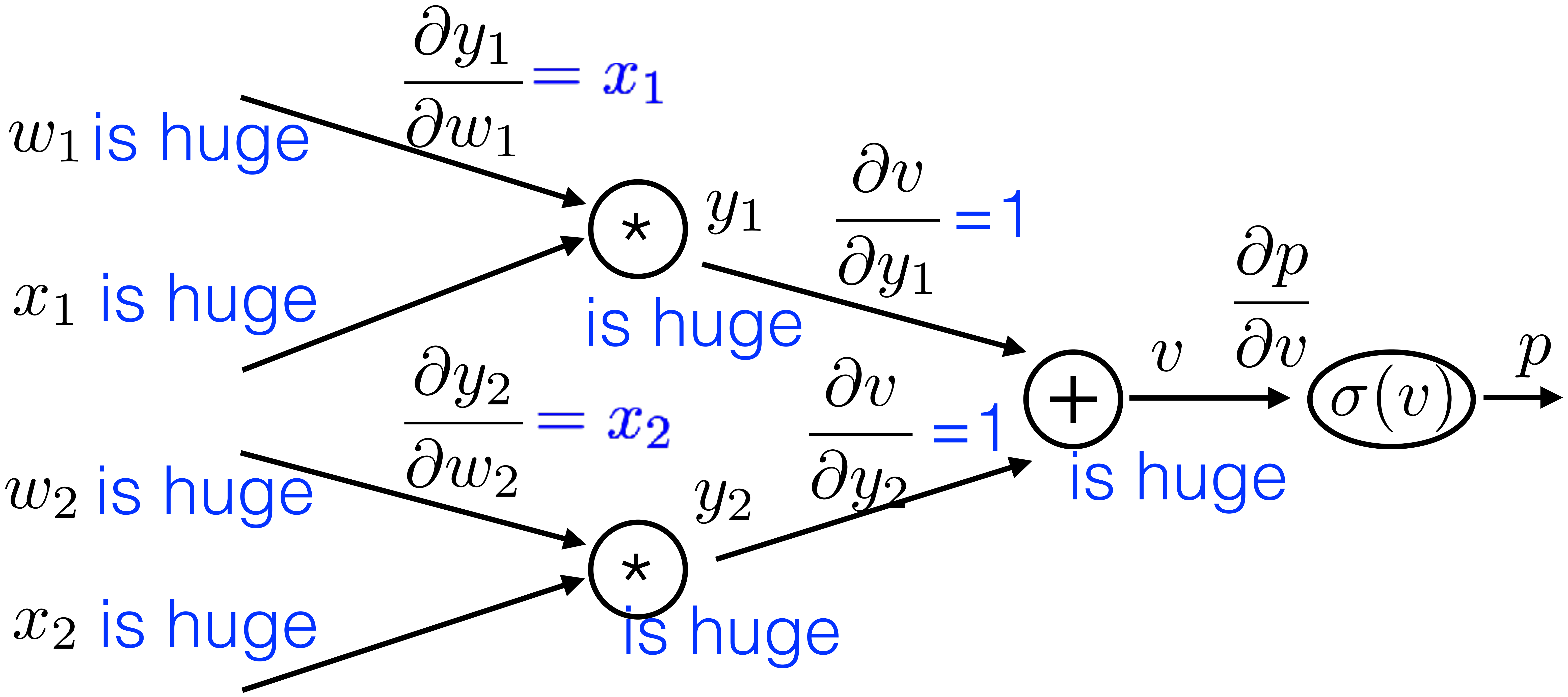
Sigmoid

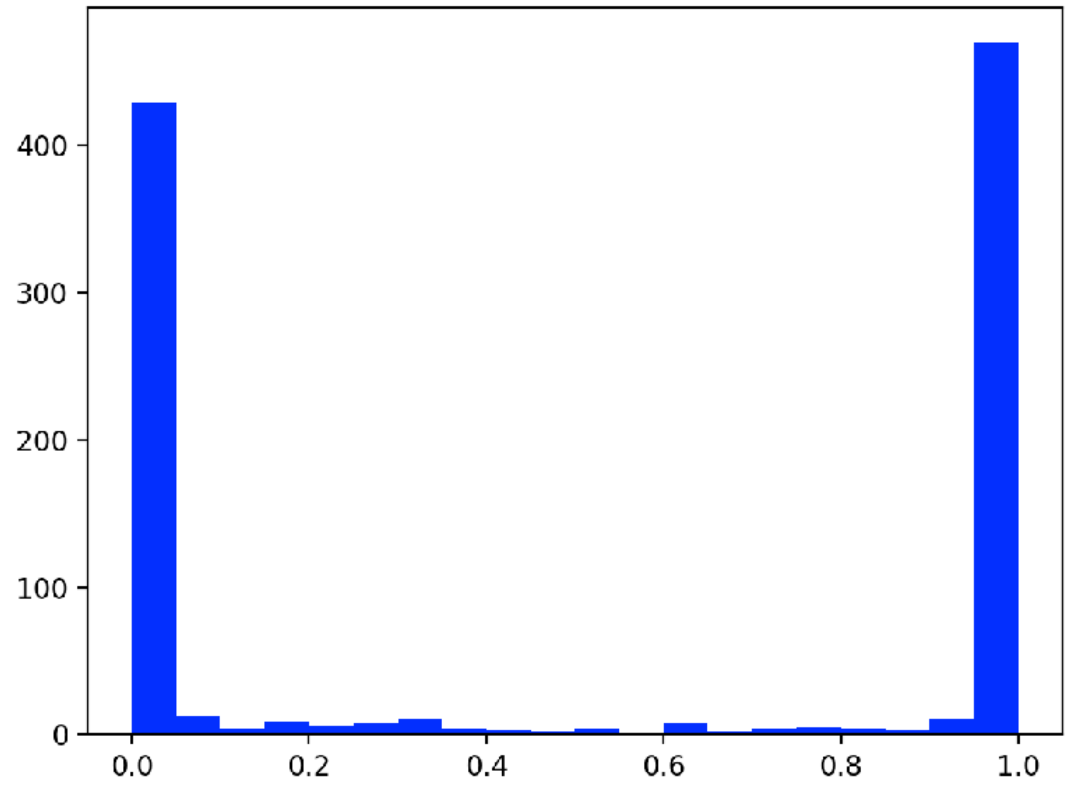
$\sigma(x) = \frac{1}{1+e^{-x}}$



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

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

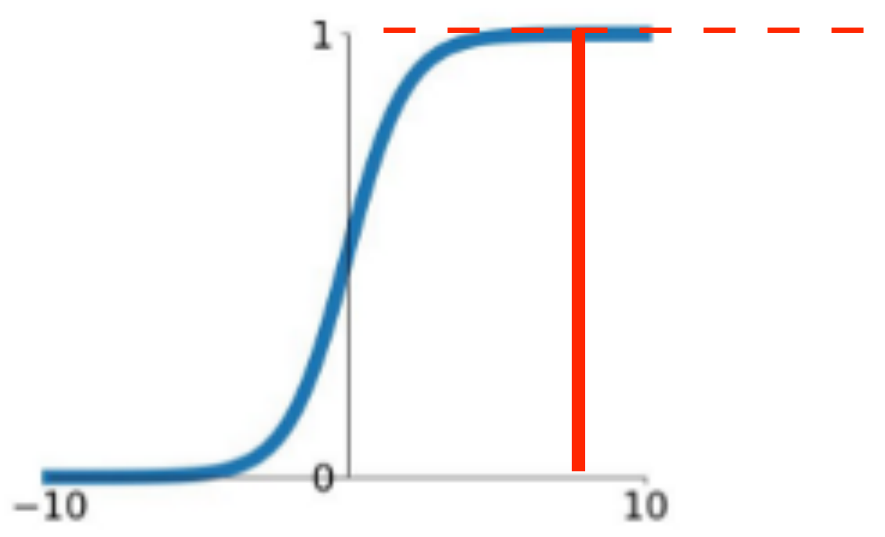




- what happen to **backprop gradient** when weights are **huge**?

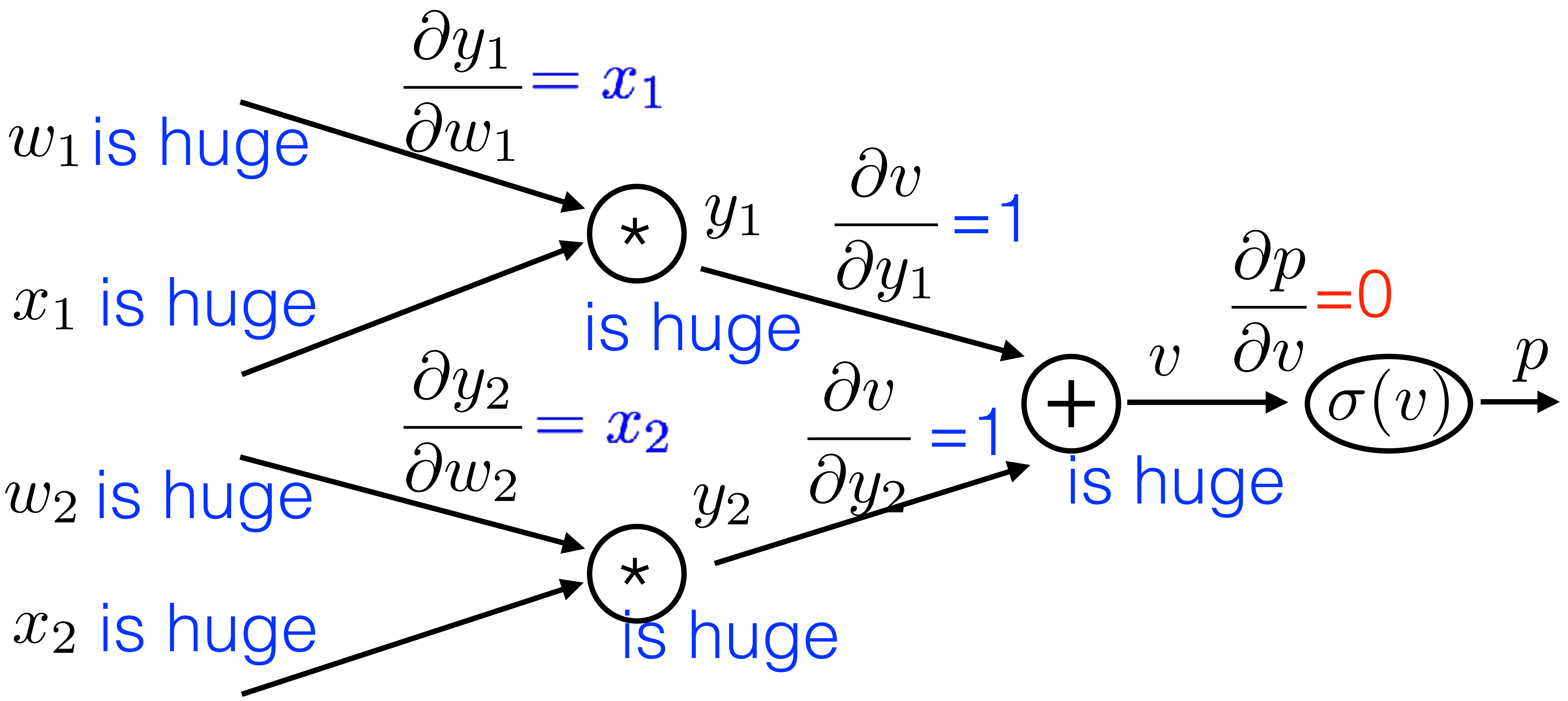
Sigmoid

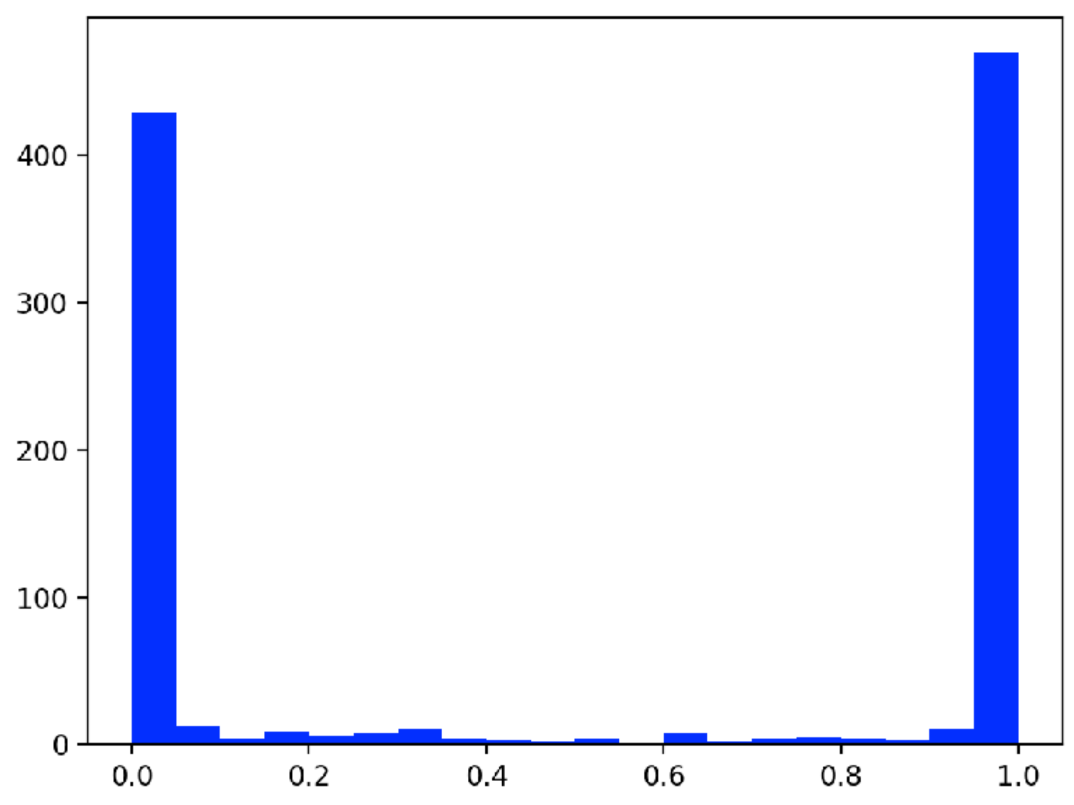
$\sigma(x) = \frac{1}{1+e^{-x}}$



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

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

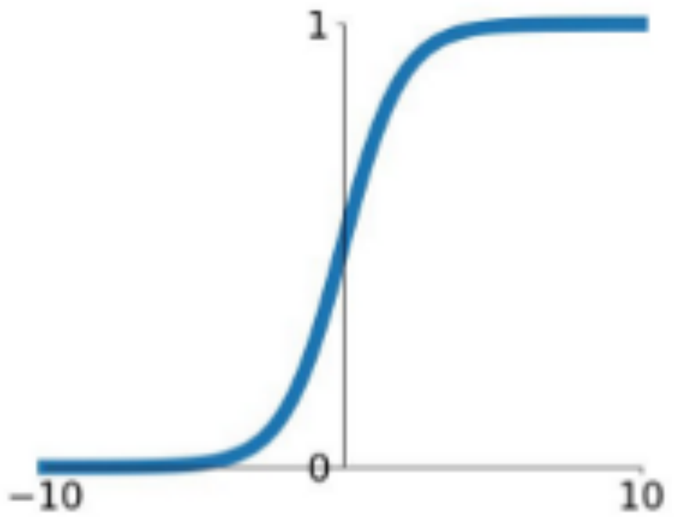




- what happen to **backprop gradient** when weights are **huge**?

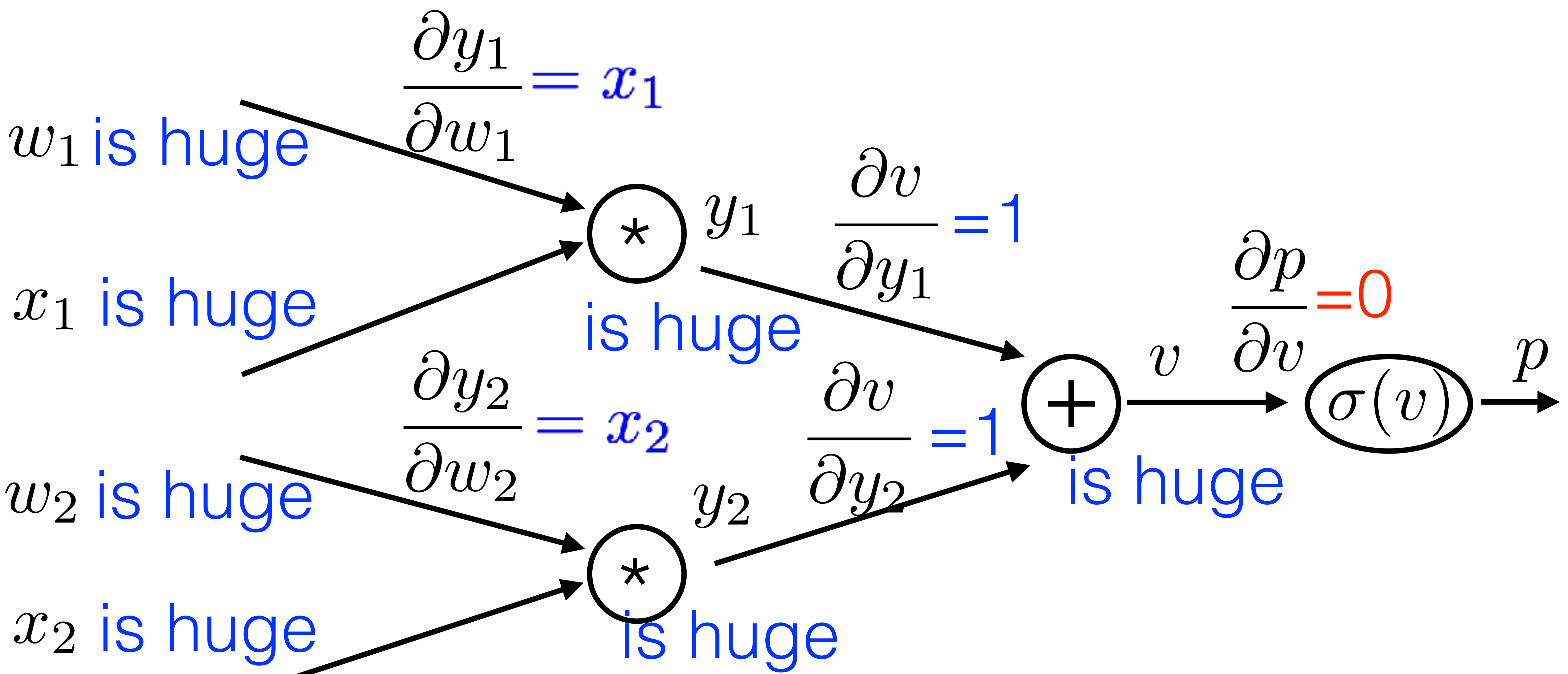
Sigmoid

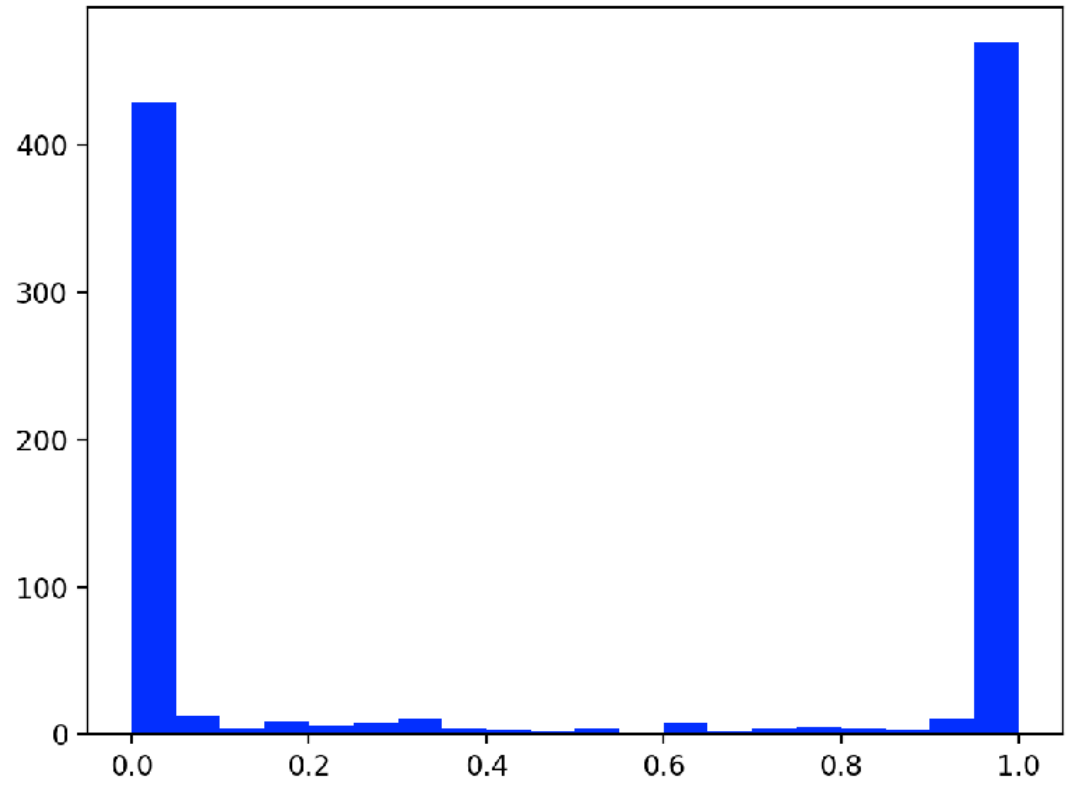
$\sigma(x) = \frac{1}{1+e^{-x}}$



- zero gradient when saturated

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

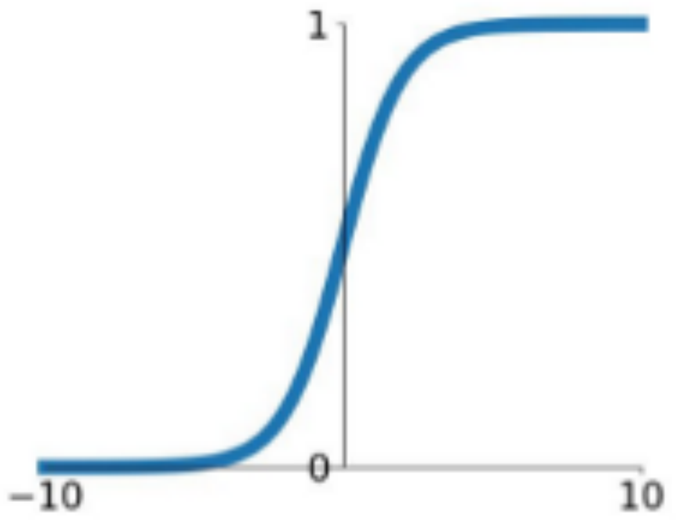




- what happens when sigmoid **input is only positive**?

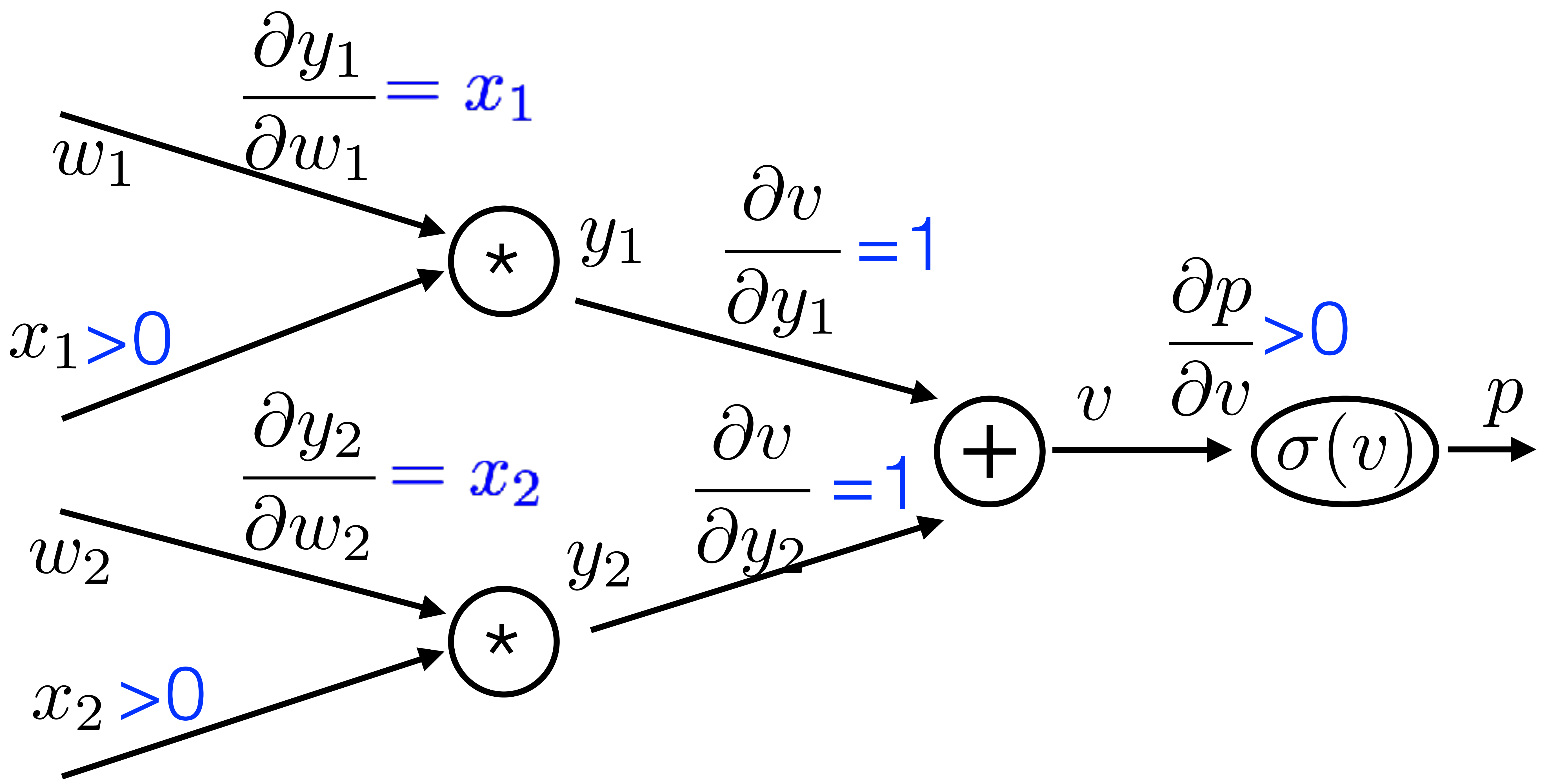
Sigmoid

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



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

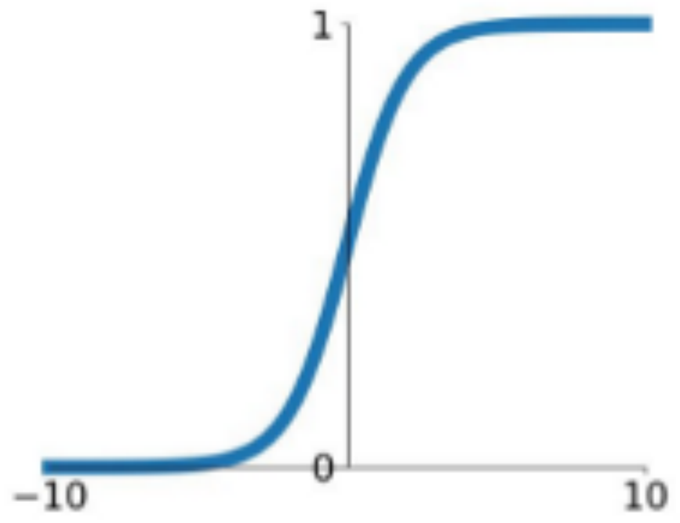
$$\frac{\partial p}{\partial w_2} = \underset{>0}{x_2} \cdot 1 \cdot \frac{\partial p}{\partial v} = ?$$



- what happens when sigmoid input is only positive?

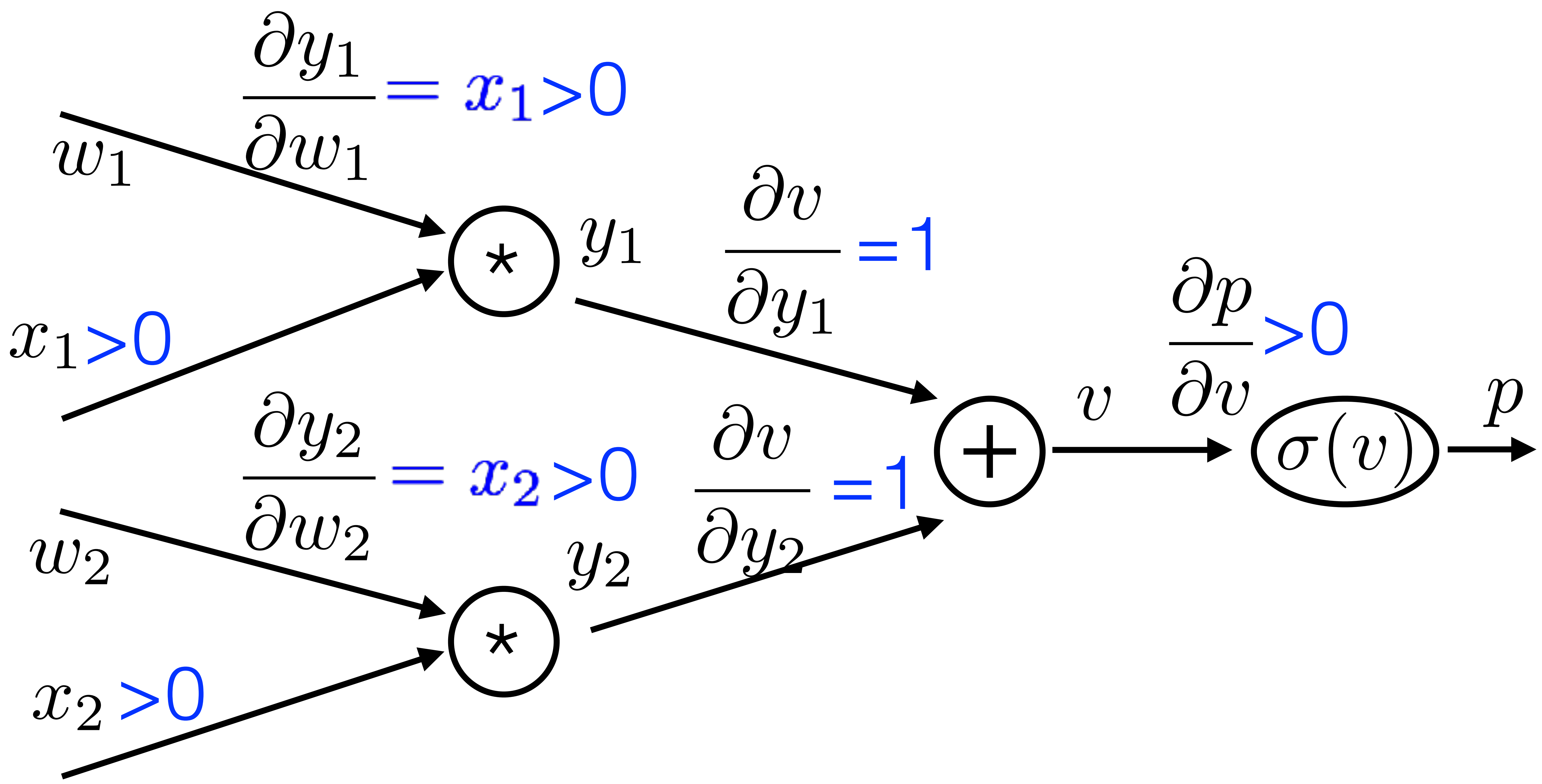
Sigmoid

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



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

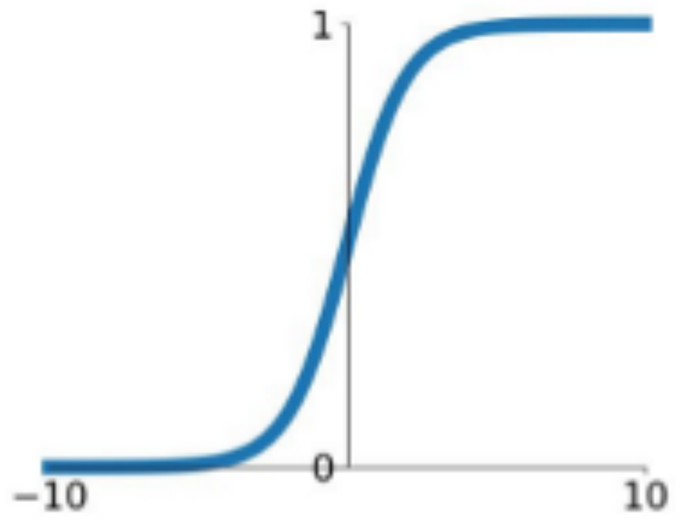
$$\frac{\partial p}{\partial w_2} = \underset{>0}{x_2} \cdot 1 \cdot \underset{>0}{\frac{\partial p}{\partial v}} = ?$$



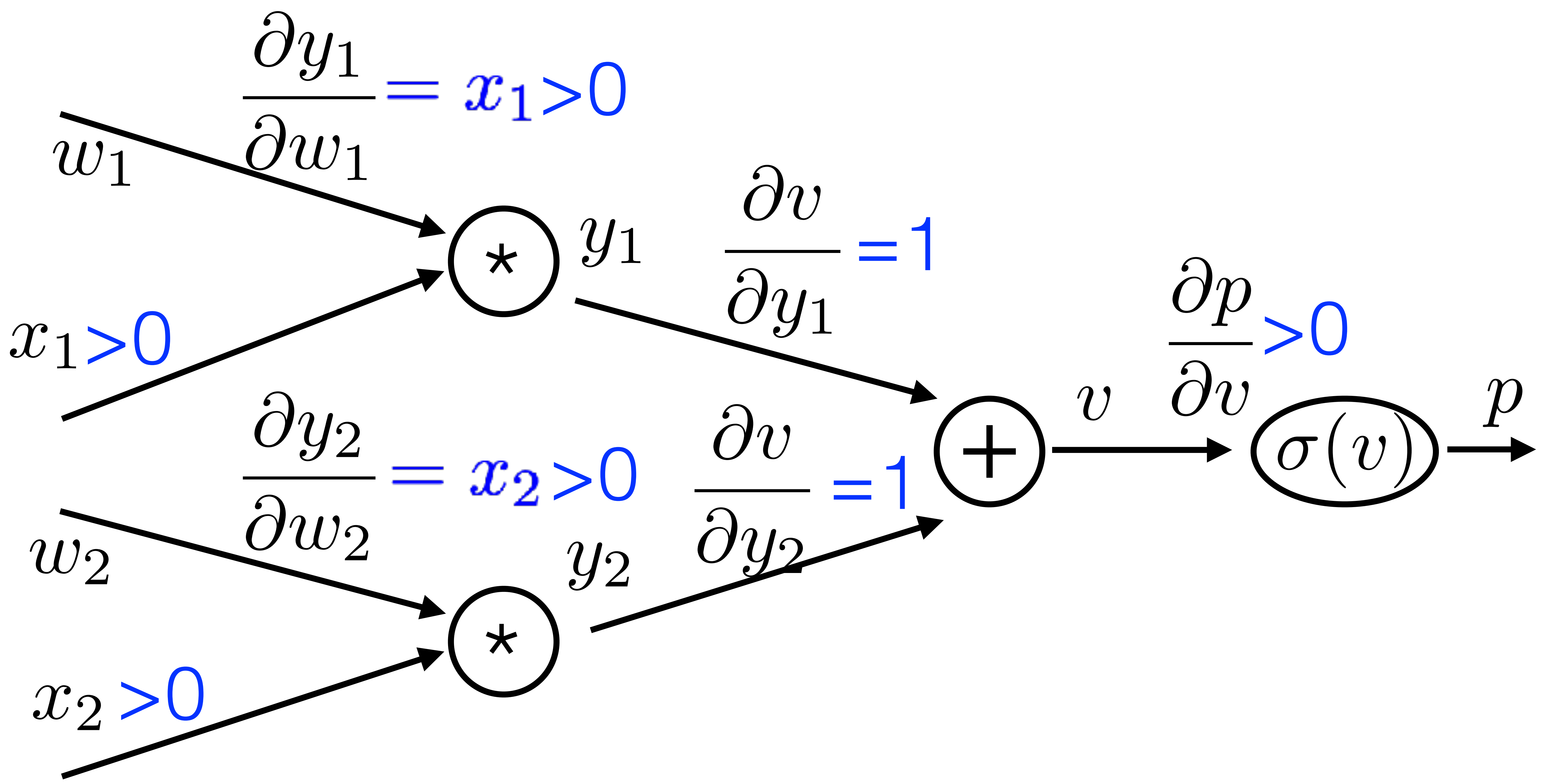
- what happens when sigmoid input is only positive?

Sigmoid

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



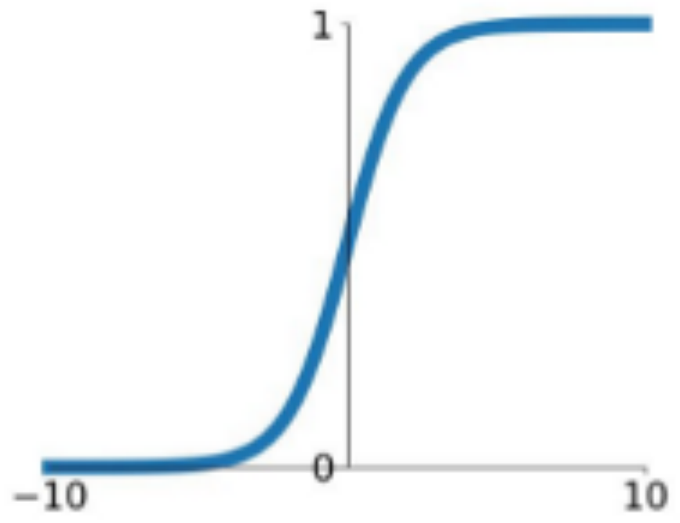
$$\frac{\partial p}{\partial w_1} = \underset{>0}{x_1} \cdot 1 \cdot \underset{>0}{\frac{\partial p}{\partial v}} \; >0 \qquad \frac{\partial p}{\partial w_2} = \underset{>0}{x_2} \cdot 1 \cdot \underset{>0}{\frac{\partial p}{\partial v}} \; >0 \; => \; \frac{\partial p}{\partial \mathbf{w}} \; >0$$



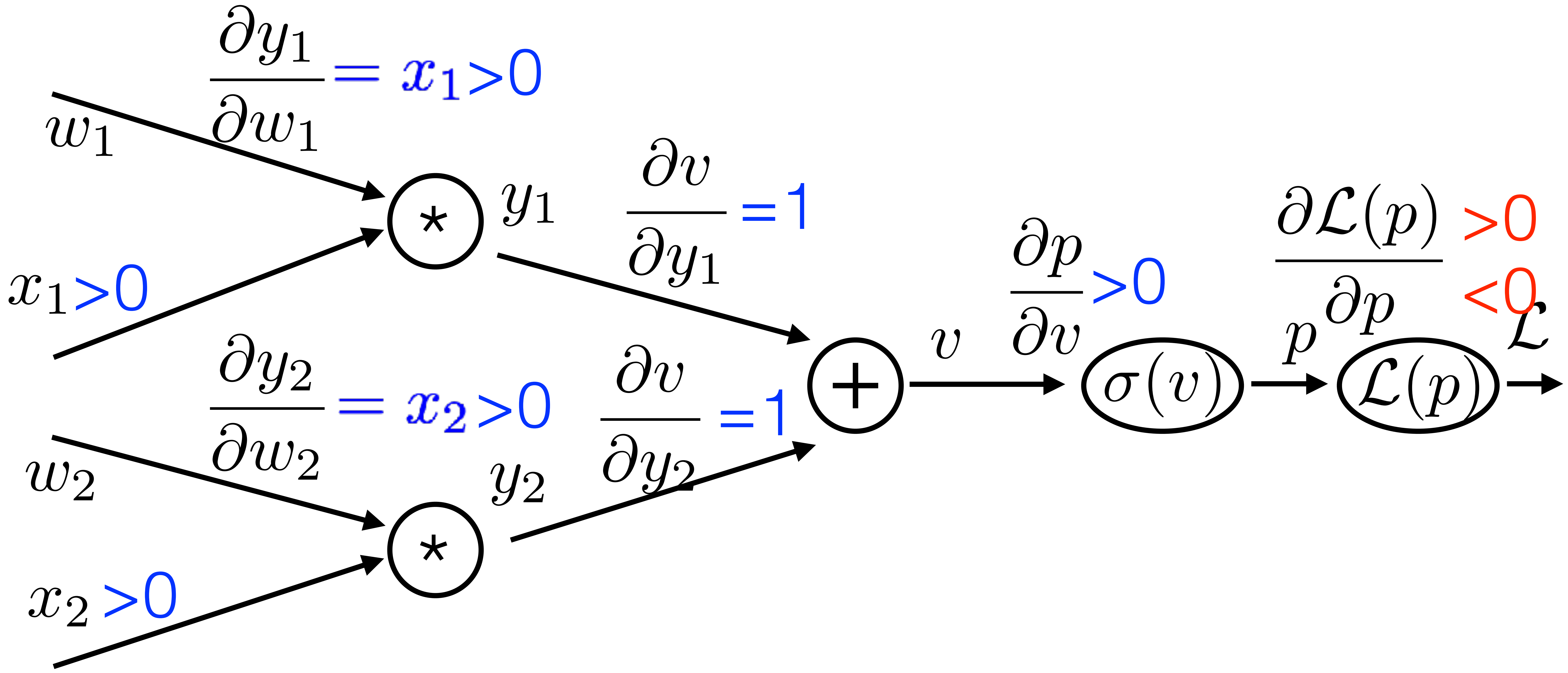
- what happens when sigmoid input is only positive?

Sigmoid

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



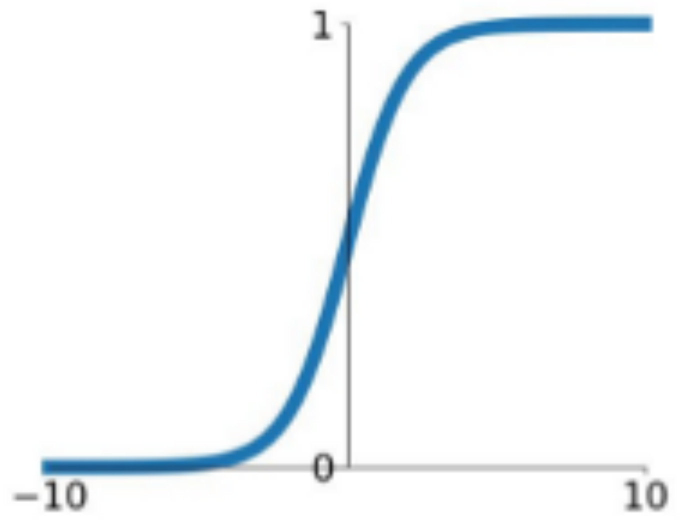
$$\frac{\partial p}{\partial w_1} = x_1 \cdot 1 \cdot \frac{\partial p}{\partial v} > 0 \qquad \frac{\partial p}{\partial w_2} = x_2 \cdot 1 \cdot \frac{\partial p}{\partial v} > 0 \Rightarrow \frac{\partial p}{\partial \mathbf{w}} > 0$$



- what happens when sigmoid input is only positive?

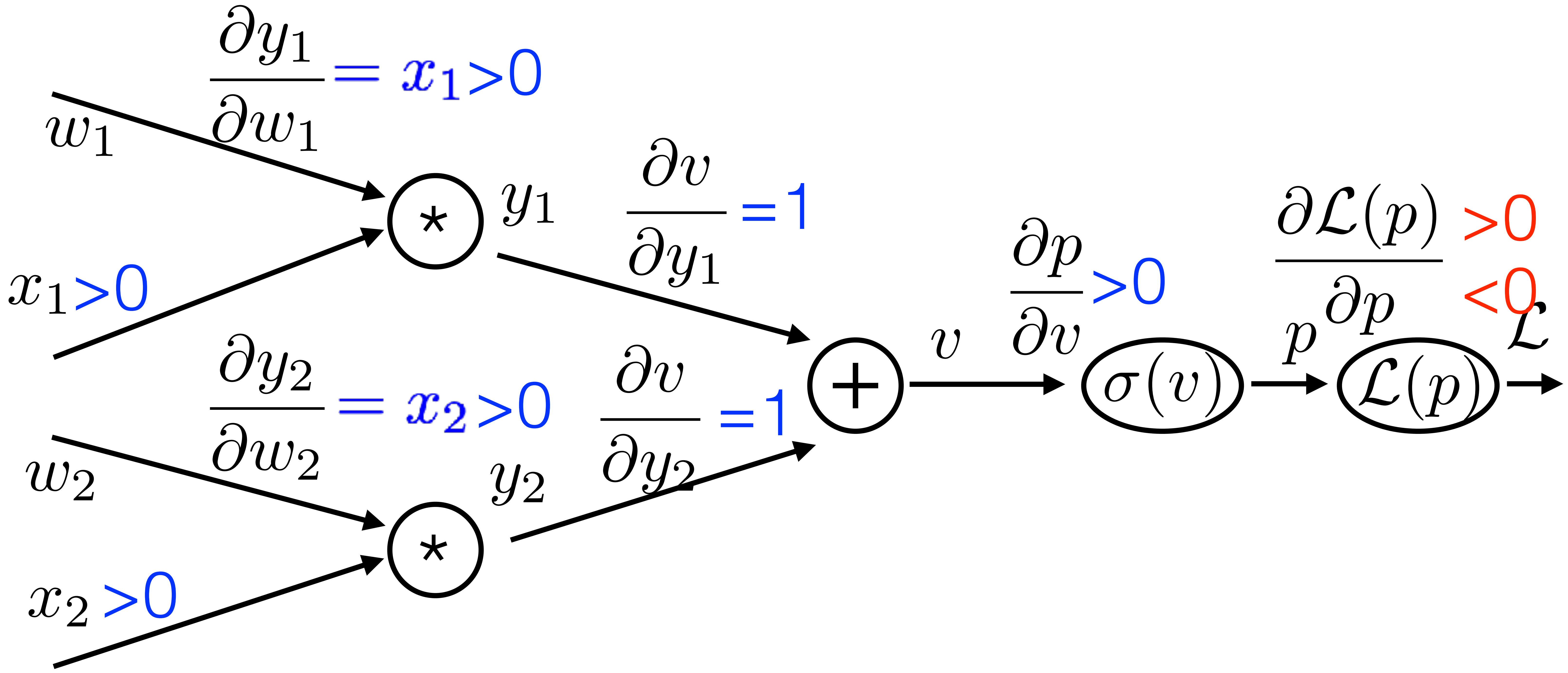
Sigmoid

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



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

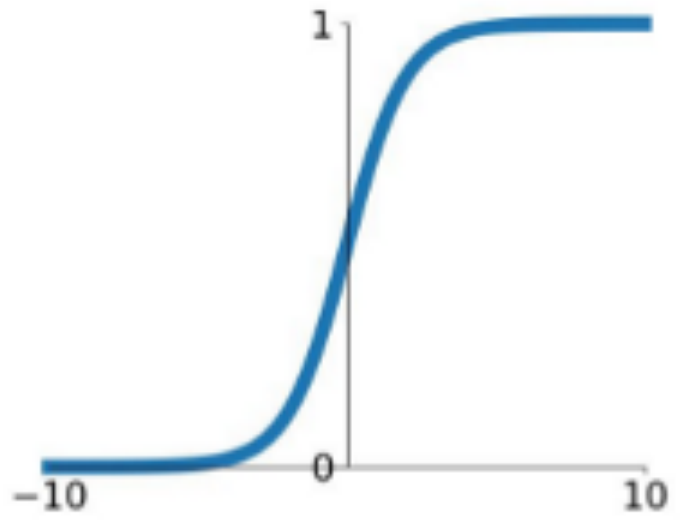
$$\frac{\partial p}{\partial w_1} = x_1 \cdot 1 \cdot \frac{\partial p}{\partial v} > 0 \qquad \frac{\partial p}{\partial w_2} = x_2 \cdot 1 \cdot \frac{\partial p}{\partial v} > 0 \Rightarrow \frac{\partial p}{\partial \mathbf{w}} > 0$$



- what happens when sigmoid input is only positive?

Sigmoid

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



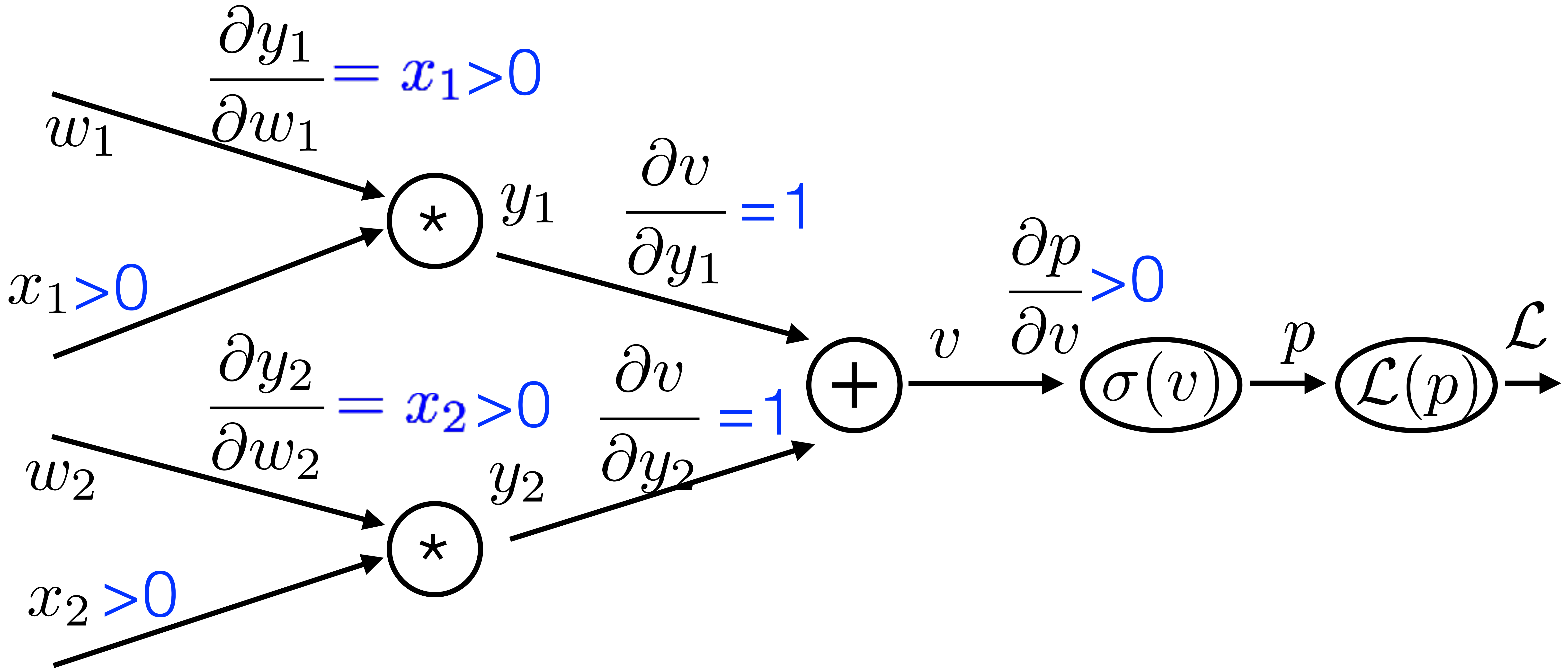
$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}(p)}{\partial p} \cdot \frac{\partial p}{\partial \mathbf{w}}$$

>0

<0

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

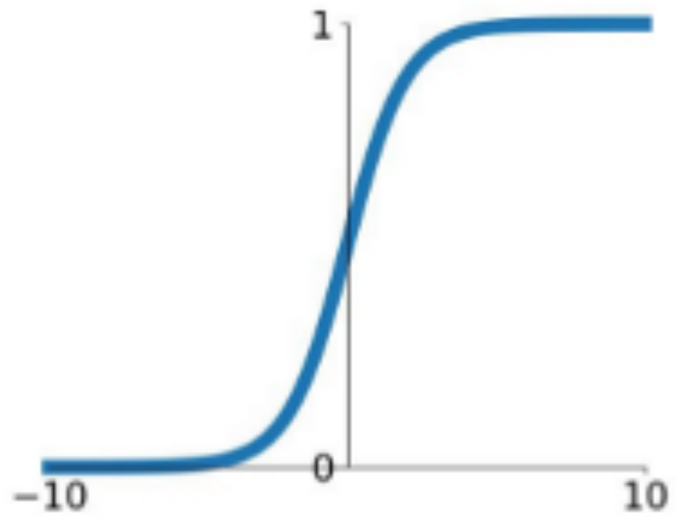
$$\frac{\partial p}{\partial w_2} = x_2 \cdot 1 \cdot \frac{\partial p}{\partial v}$$ >0 => $$\frac{\partial p}{\partial \mathbf{w}}$$ >0



- what happens when sigmoid input is only positive?

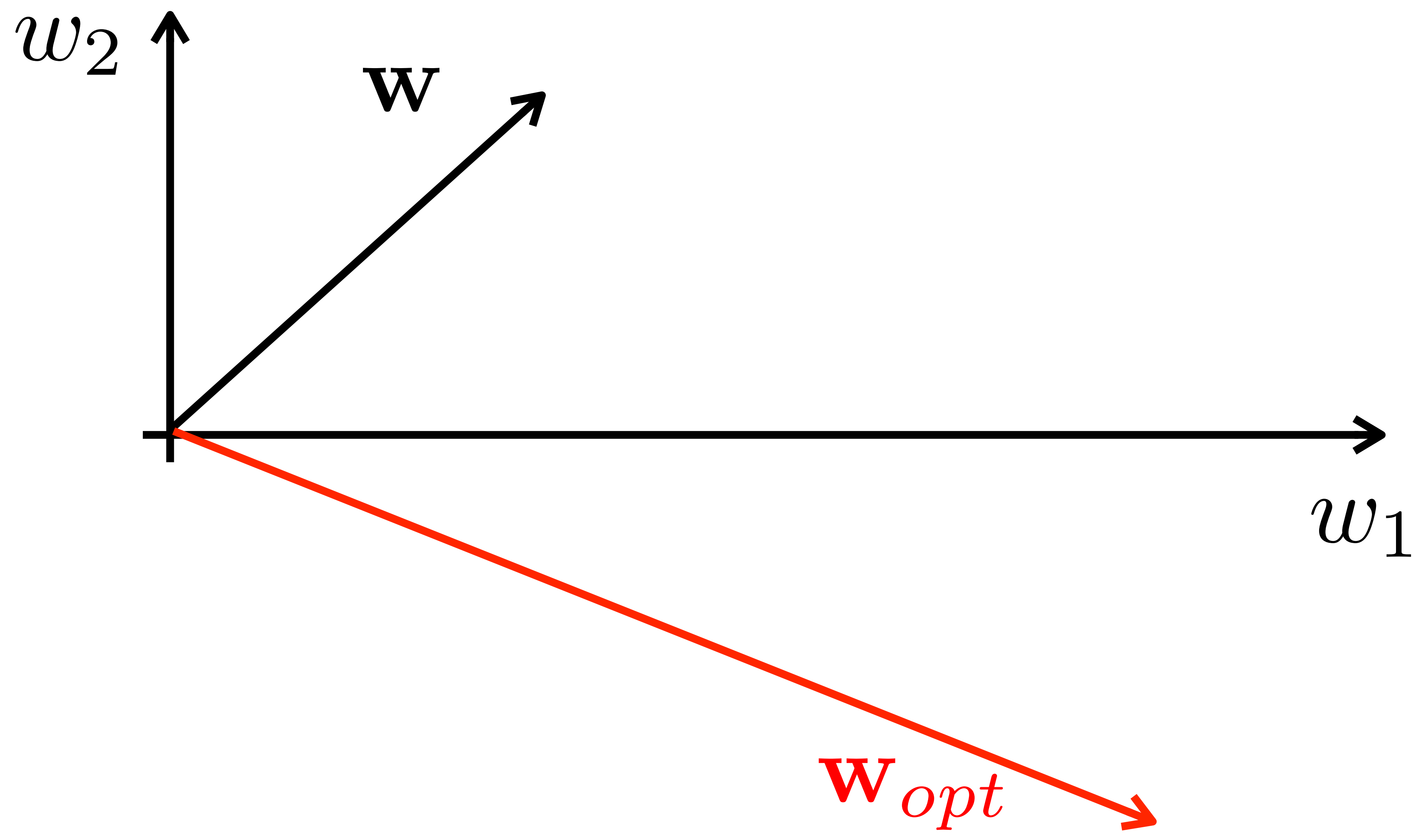
Sigmoid

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



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

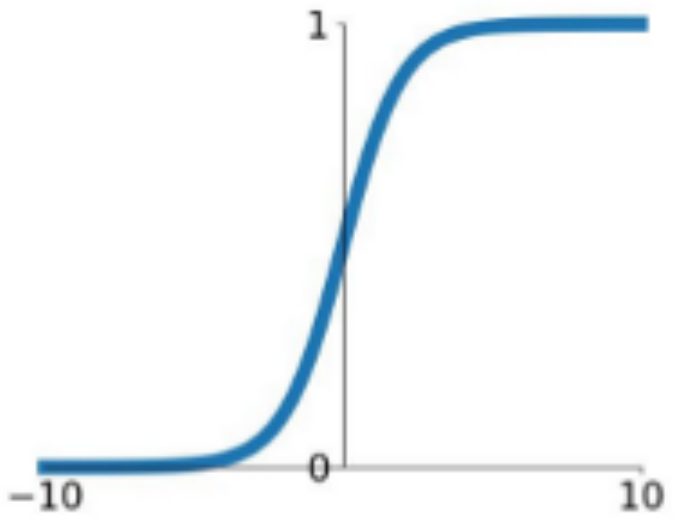
>0
<0



- what happens when sigmoid input is only positive?

Sigmoid

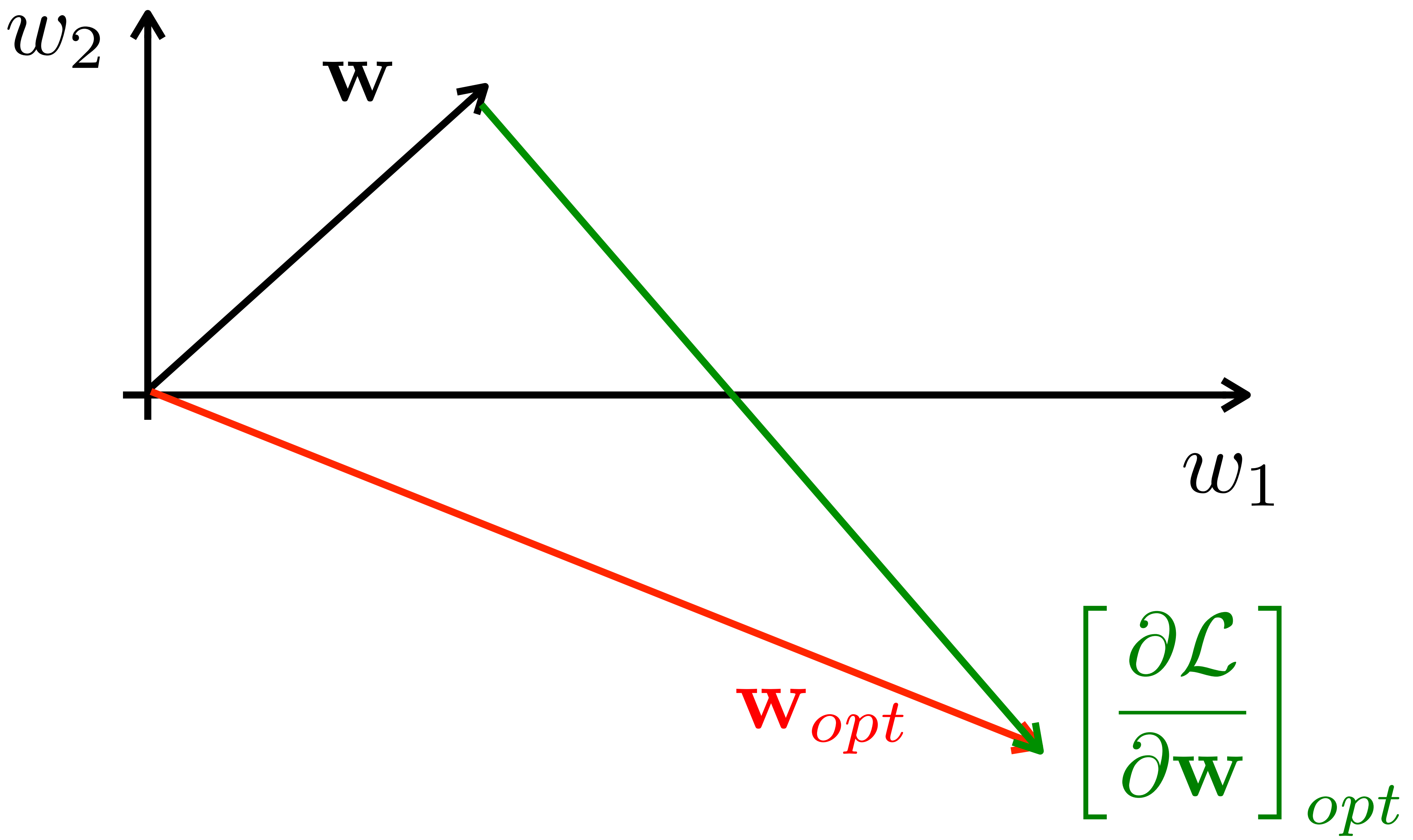
$$\sigma(x) = \frac{1}{1+e^{-x}}$$



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

>0

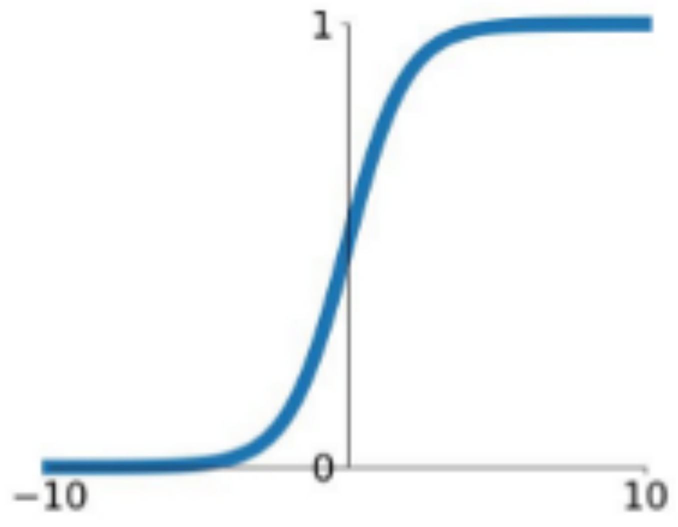
<0



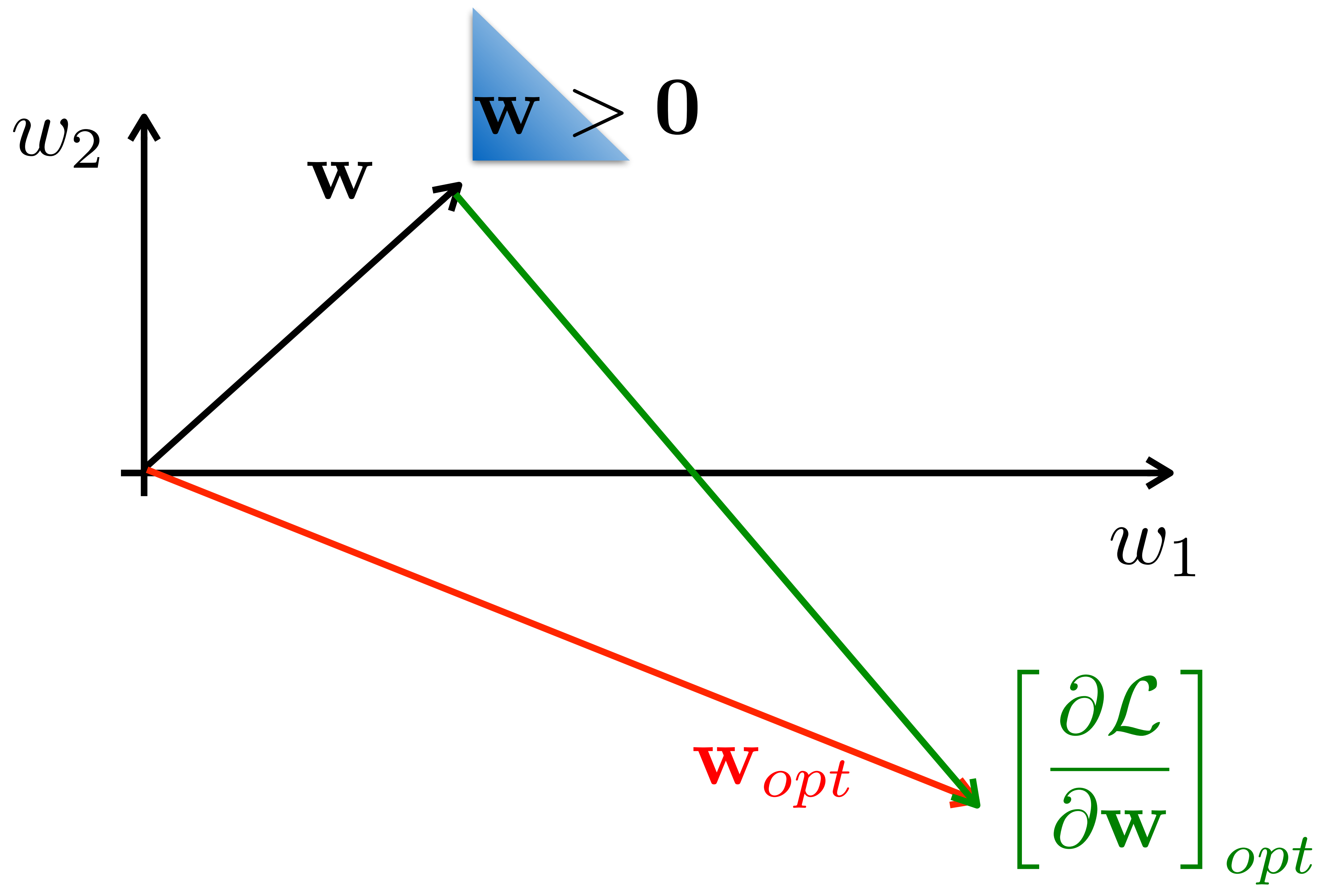
- what happens when sigmoid input is only positive?

Sigmoid

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



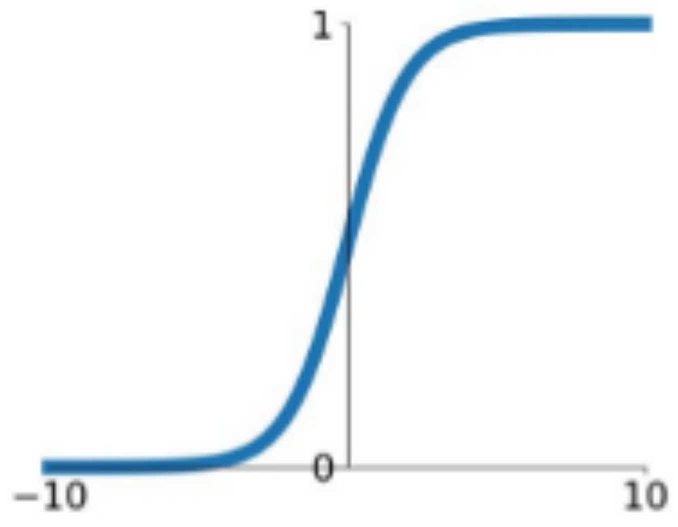
$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}(p)}{\partial p} \cdot \frac{\partial p}{\partial \mathbf{w}} \quad \begin{matrix} >0 \\ <0 \end{matrix}$$



- what happens when sigmoid input is only positive?

Sigmoid

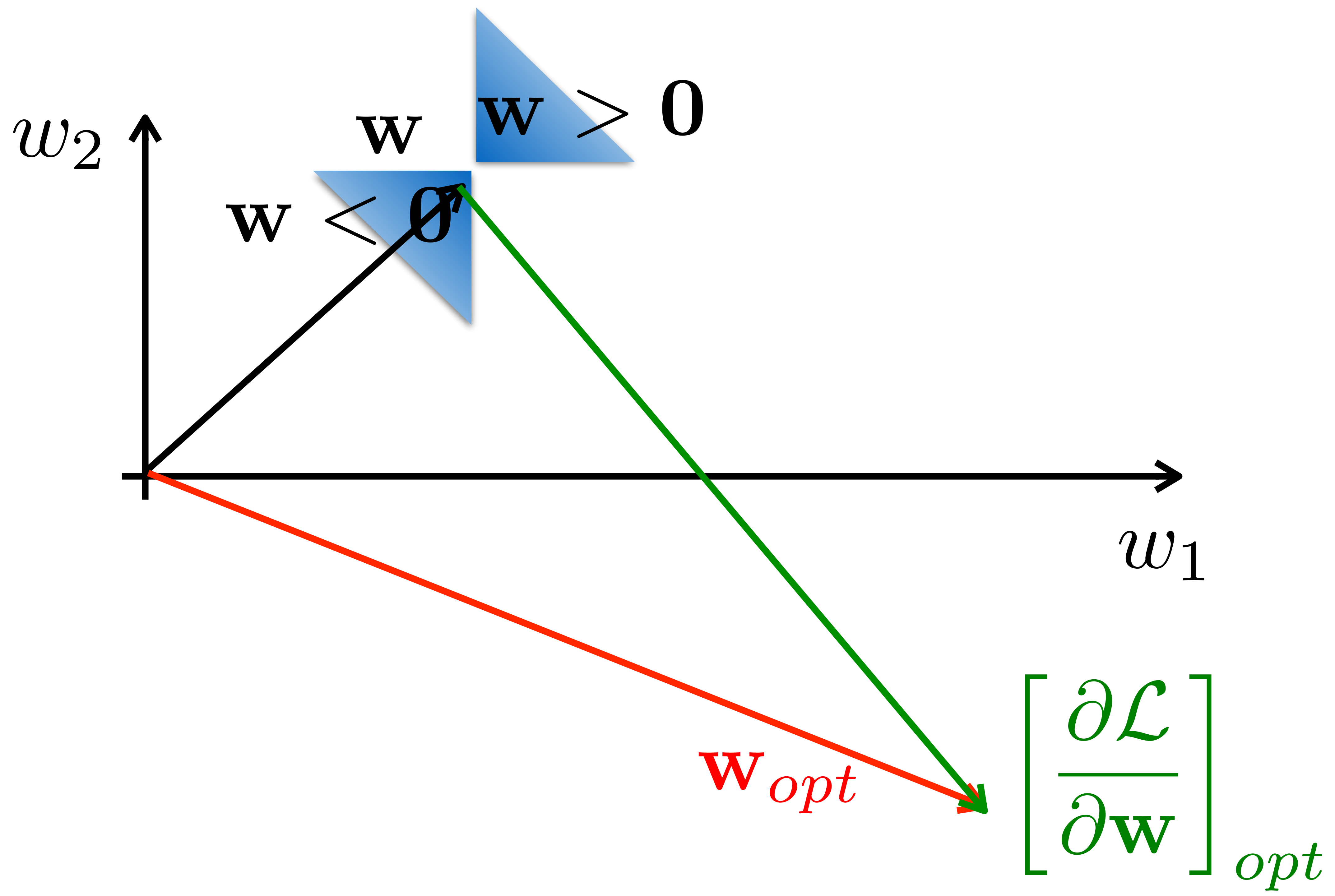
$\sigma(x) = \frac{1}{1+e^{-x}}$



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

>0

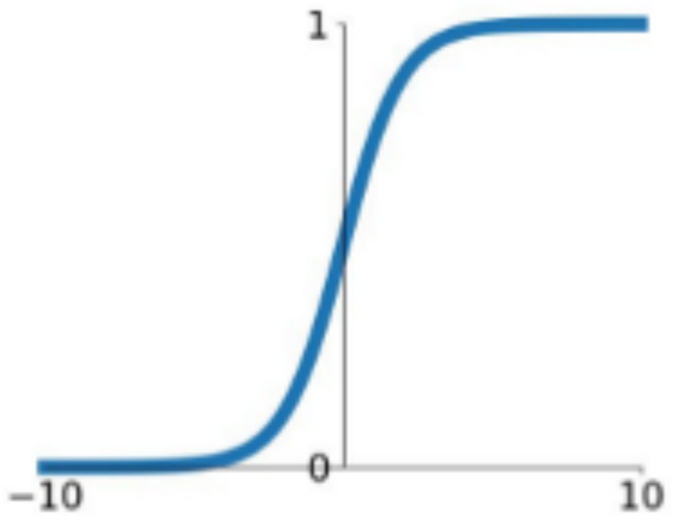
<0



- what happens when sigmoid input is only positive?

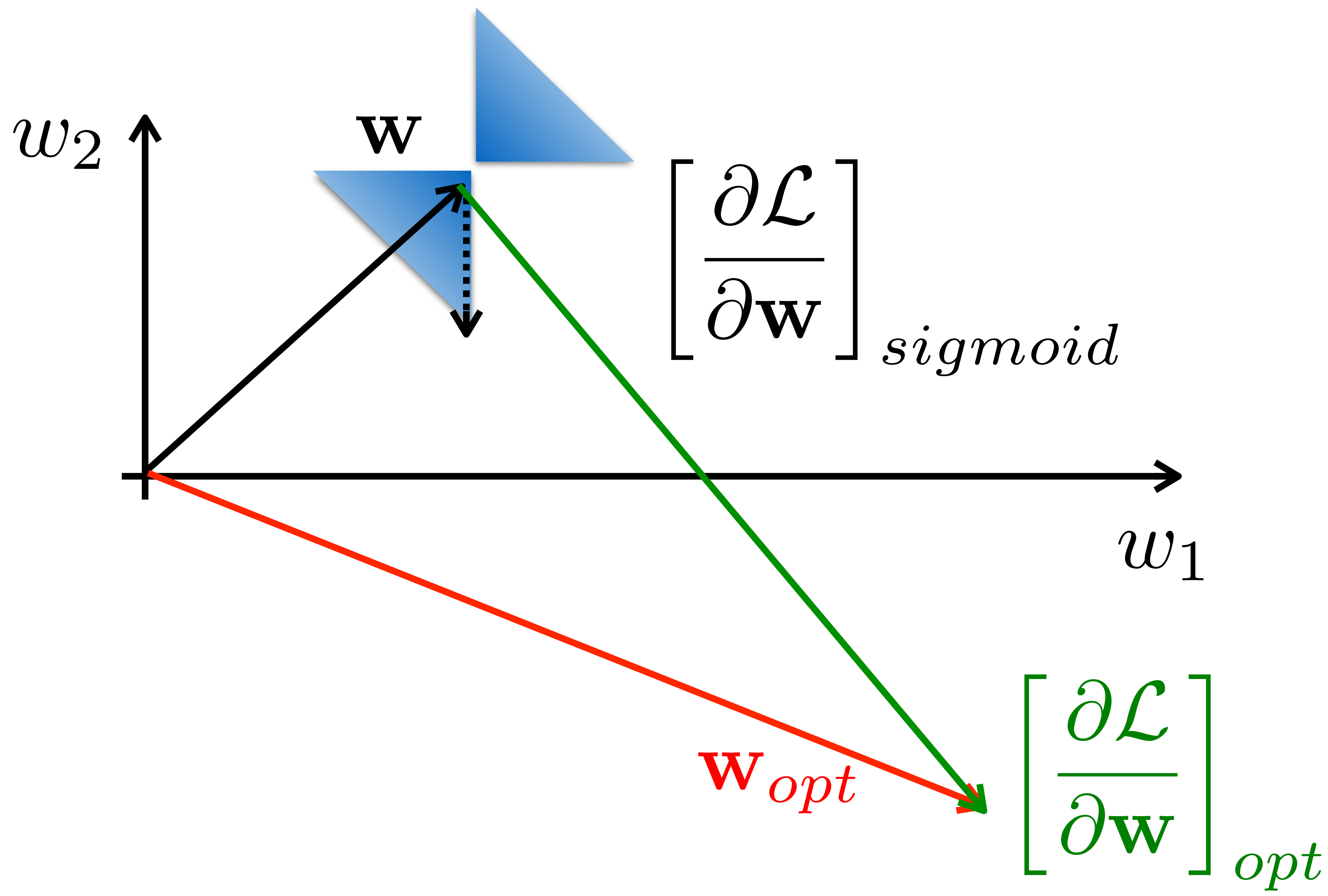
Sigmoid

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



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

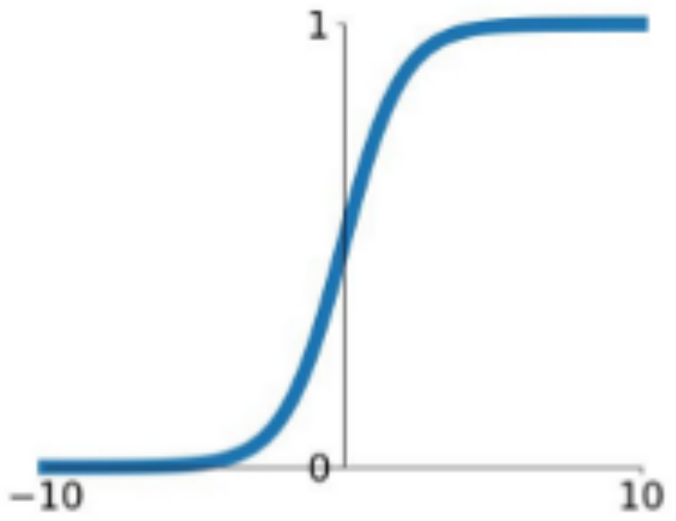
>0
<0



- what happens when sigmoid input is only positive?

Sigmoid

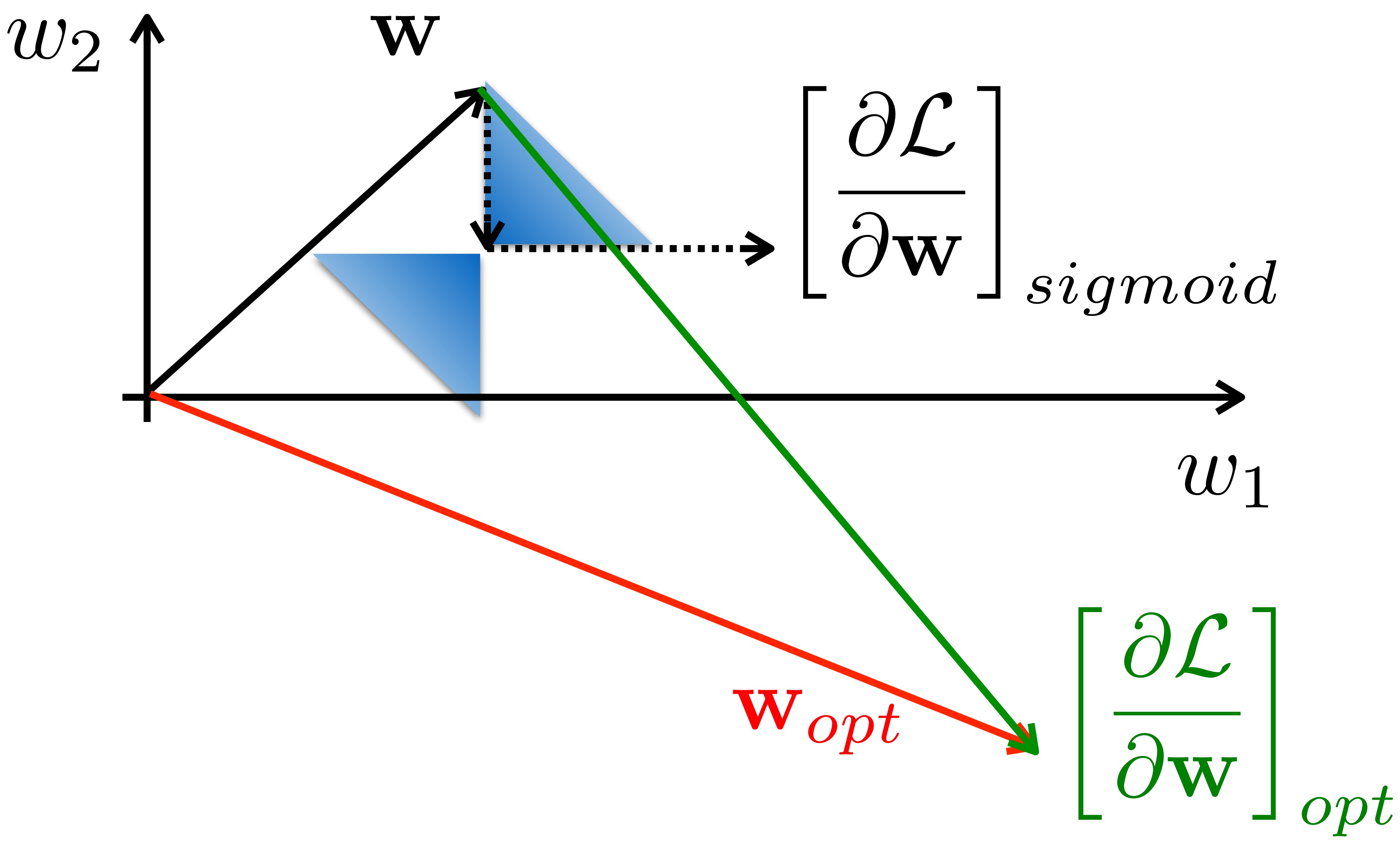
$\sigma(x) = \frac{1}{1+e^{-x}}$



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

>0

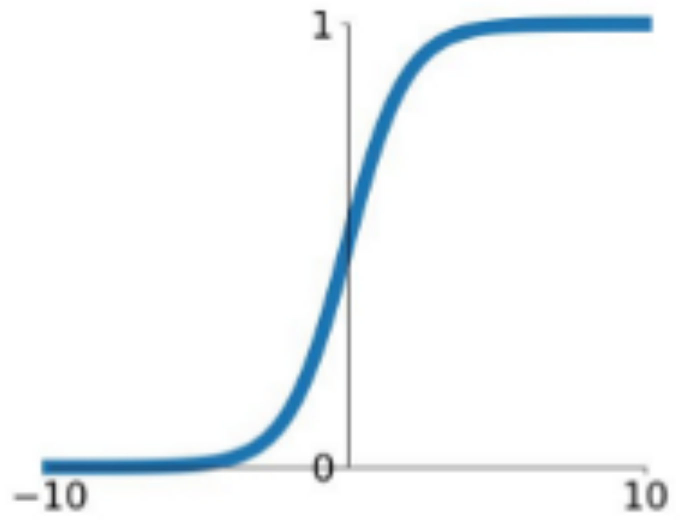
<0



- what happens when sigmoid input is only positive?

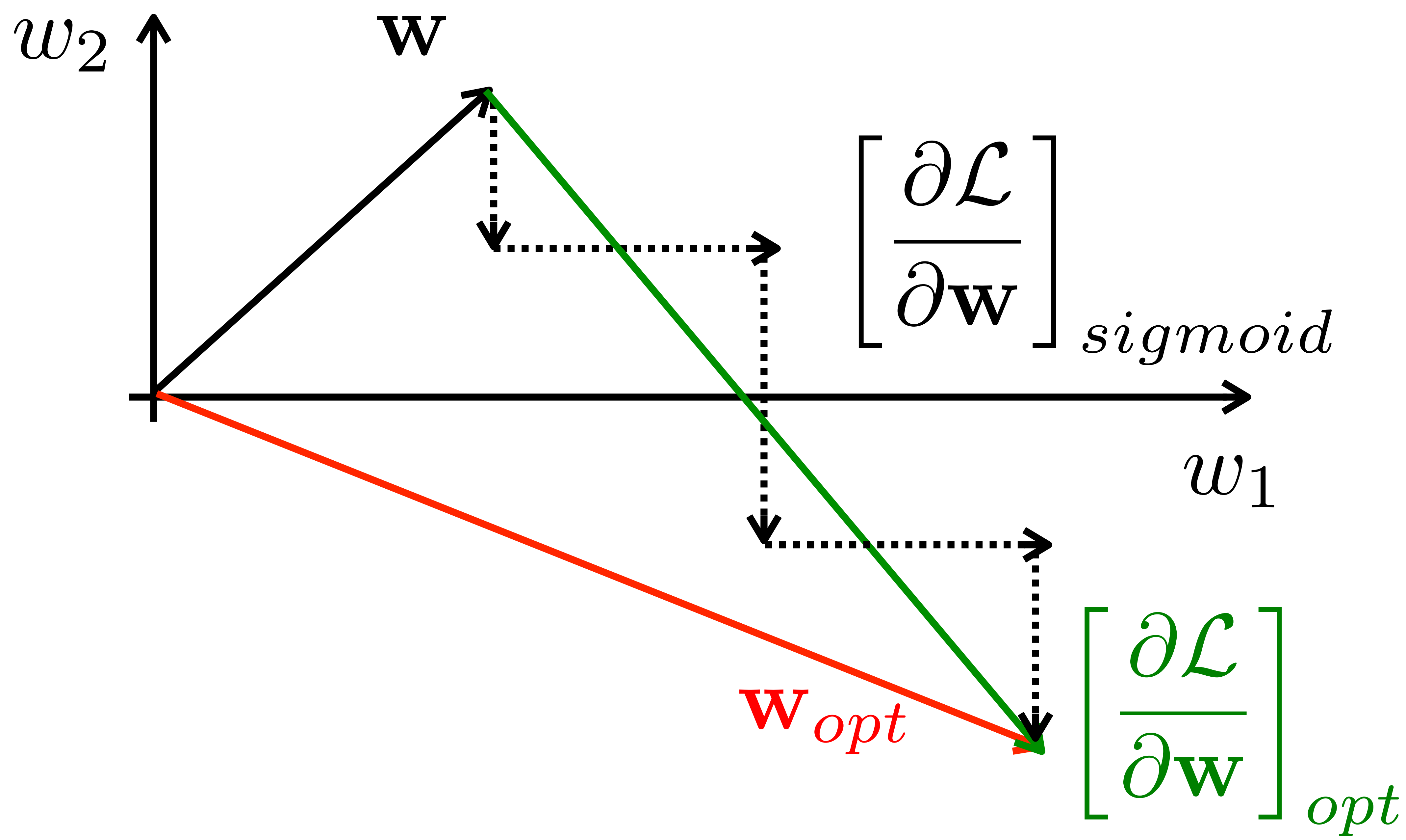
Sigmoid

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



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

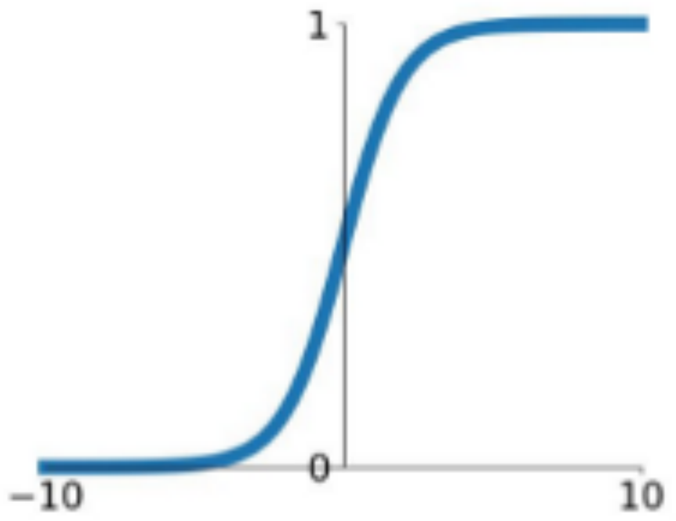
>0
<0



- what happens when sigmoid input is only positive?

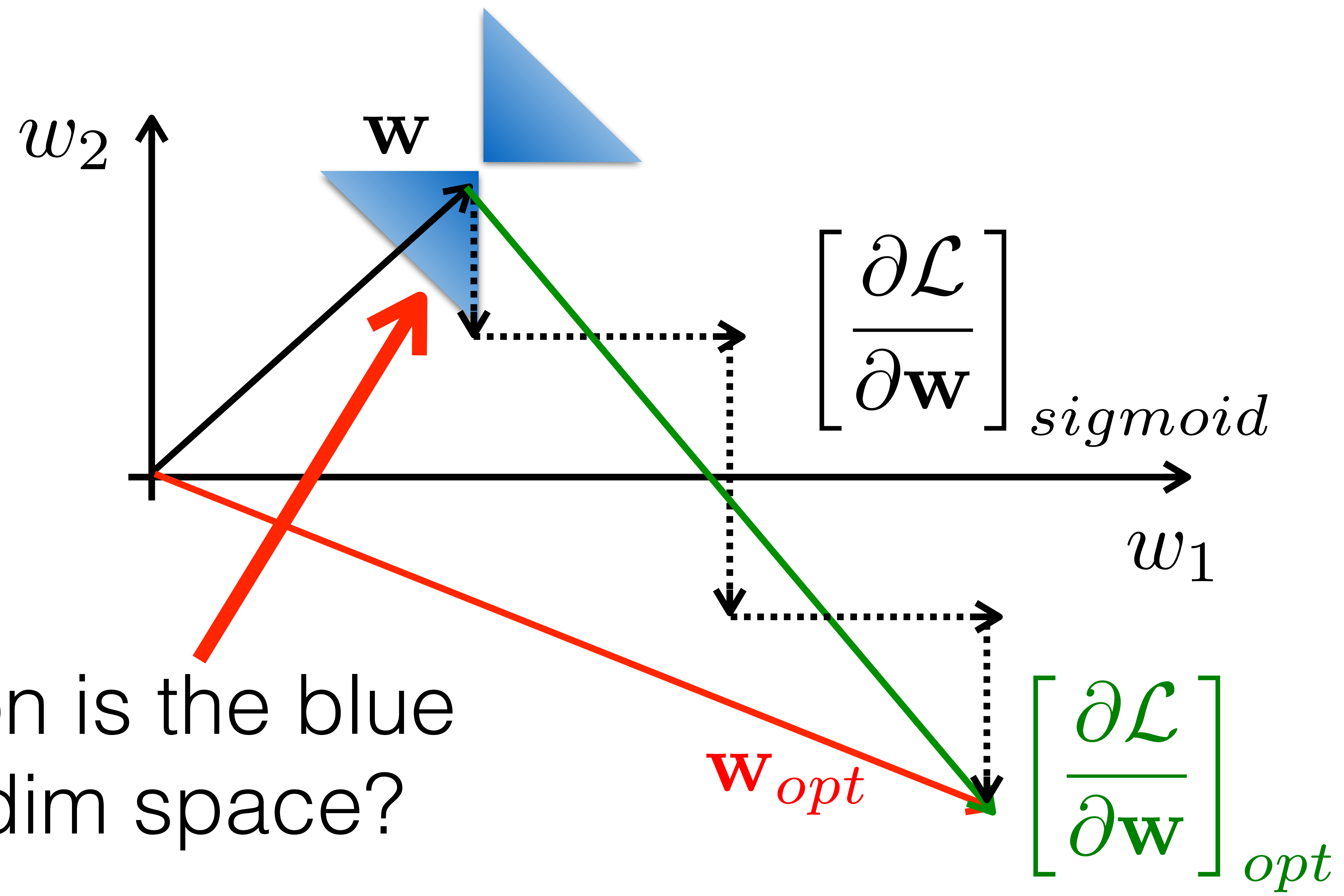
Sigmoid

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



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

>0
<0

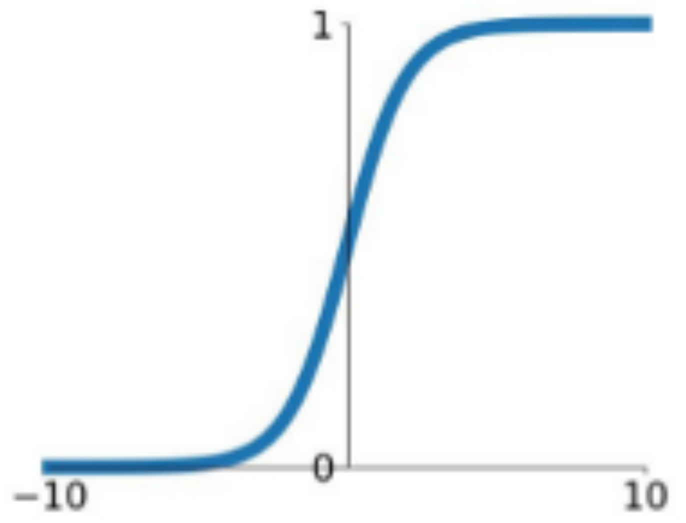


how big fraction is the blue region in 10-dim space?

- what happens when sigmoid input is only positive?

Sigmoid

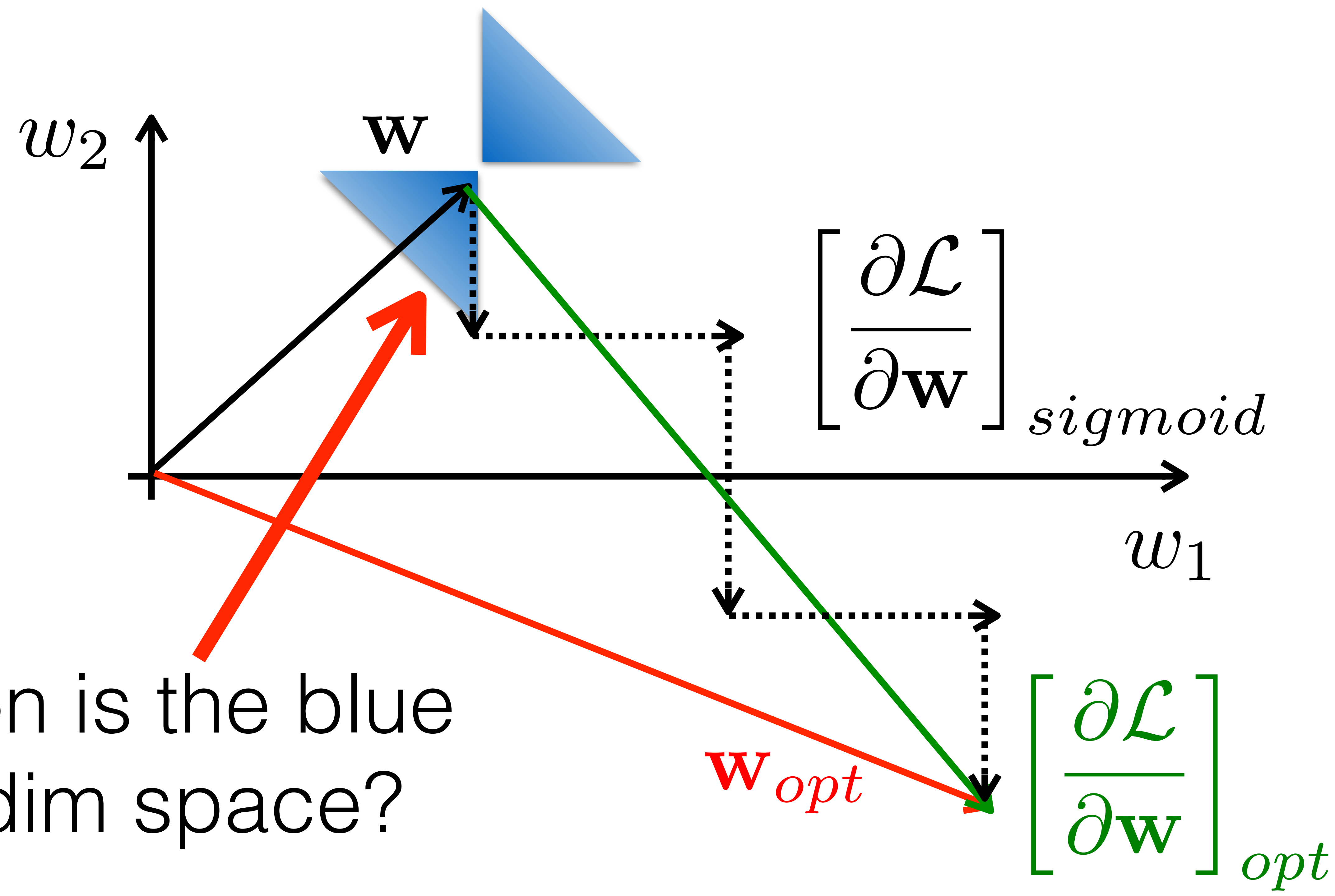
$$\sigma(x) = \frac{1}{1+e^{-x}}$$



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

>0

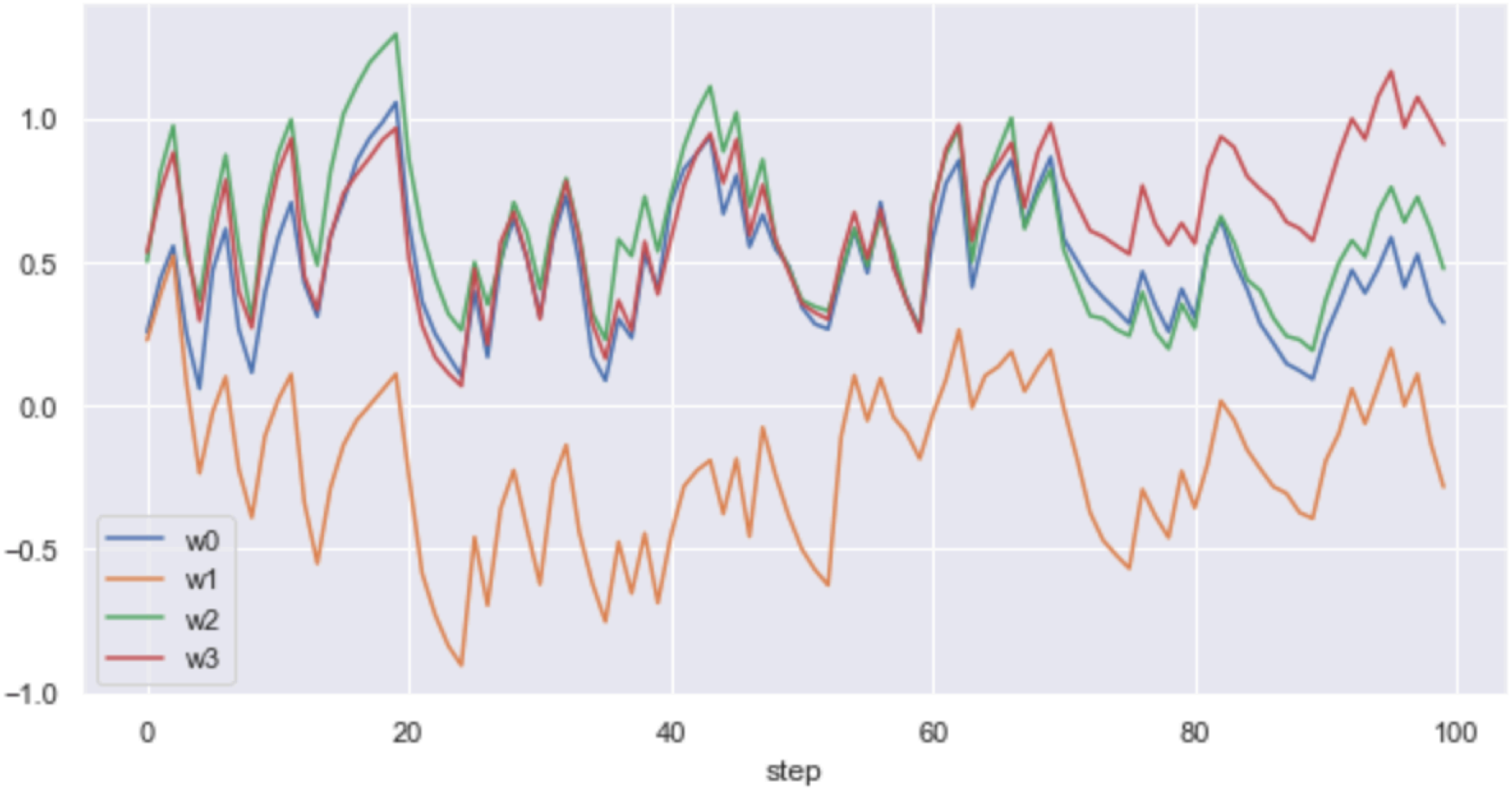
<0



2/(2^10)

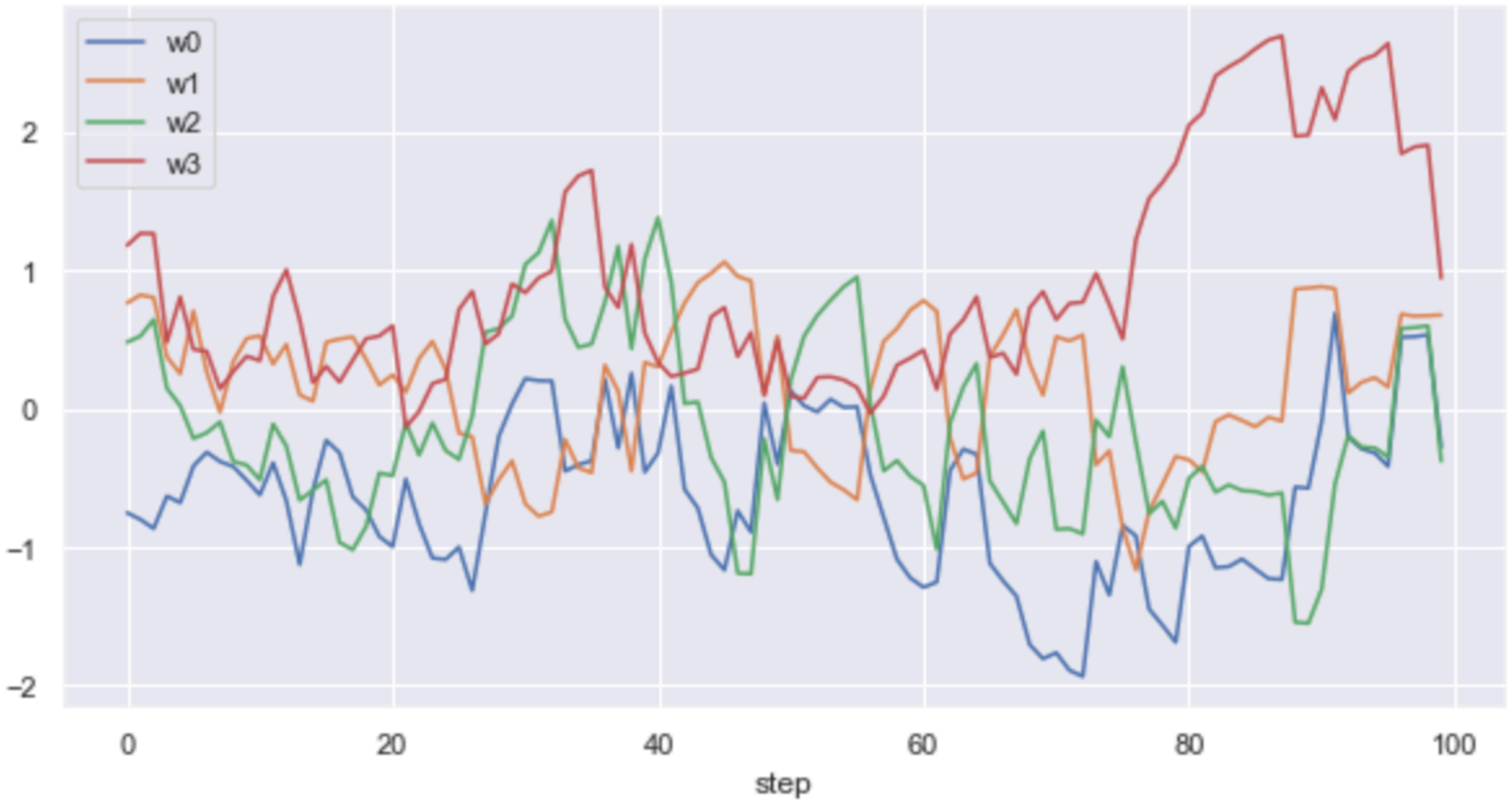
how big fraction is the blue region in 10-dim space?

- what happens when sigmoid input is only positive?



sigmoid activation function

- what happens when sigmoid input is only positive?

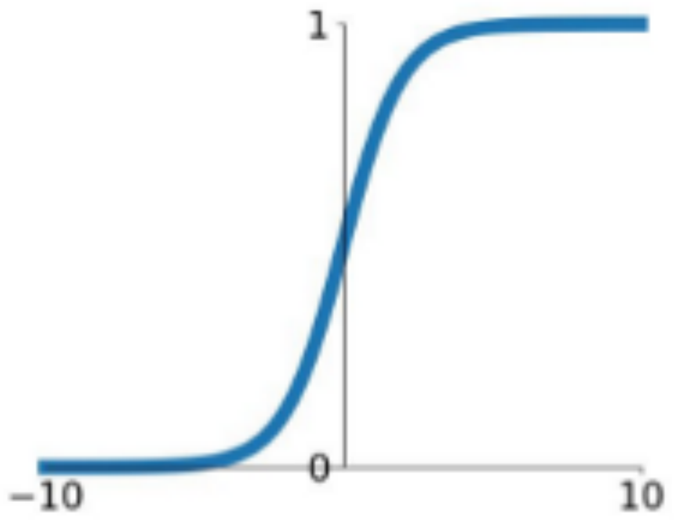


tanh activation function

Activation functions

Sigmoid

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

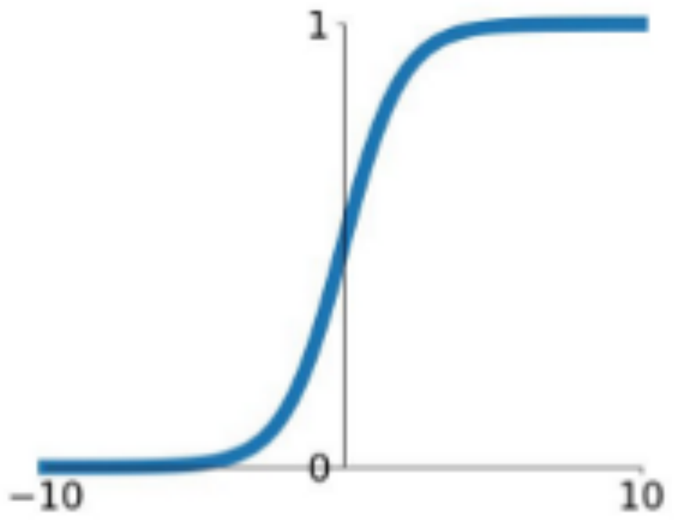


- zero gradient when saturated
- not zero-centered (pos. output)
- computationally expensive

Activation functions

Sigmoid

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



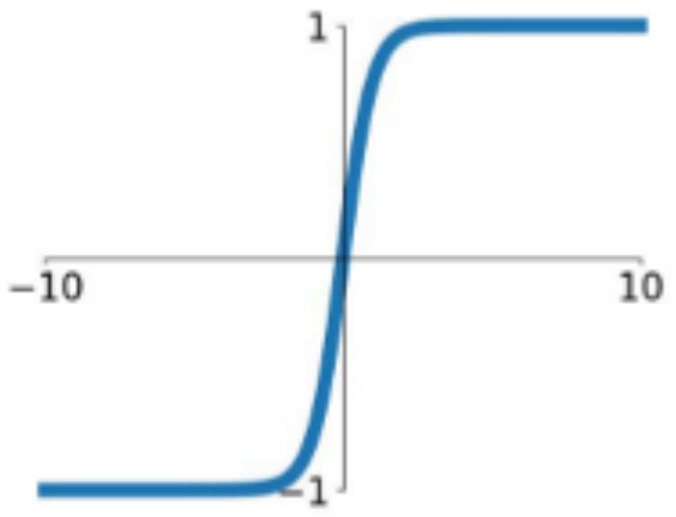
- zero gradient when saturated
- not zero-centered (pos. output)
- computationally expensive

PyTorch: `nn.Sigmoid()`

Activation functions

tanh

$\tanh(x)$



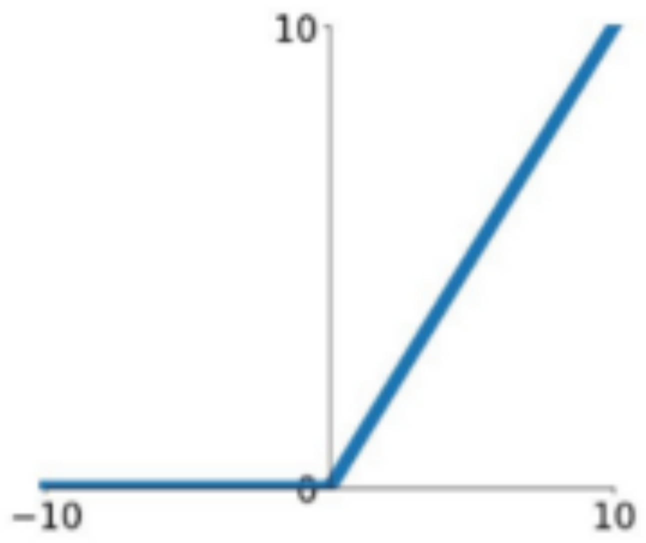
- zero gradient when saturated
- ~~not zero-centered (only positive ouputs)~~
- computationally expensive

- PyTorch: `nn.Tanh()`

Activation functions

ReLU

$\max(0, x)$



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

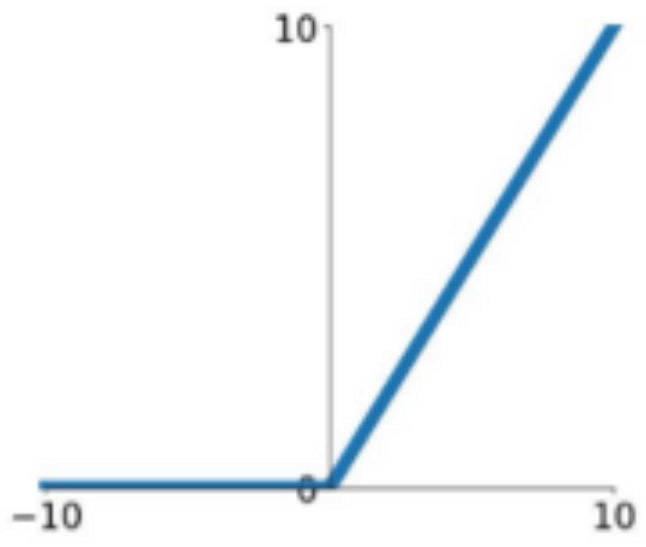
- PyTorch: `nn.ReLu()`

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

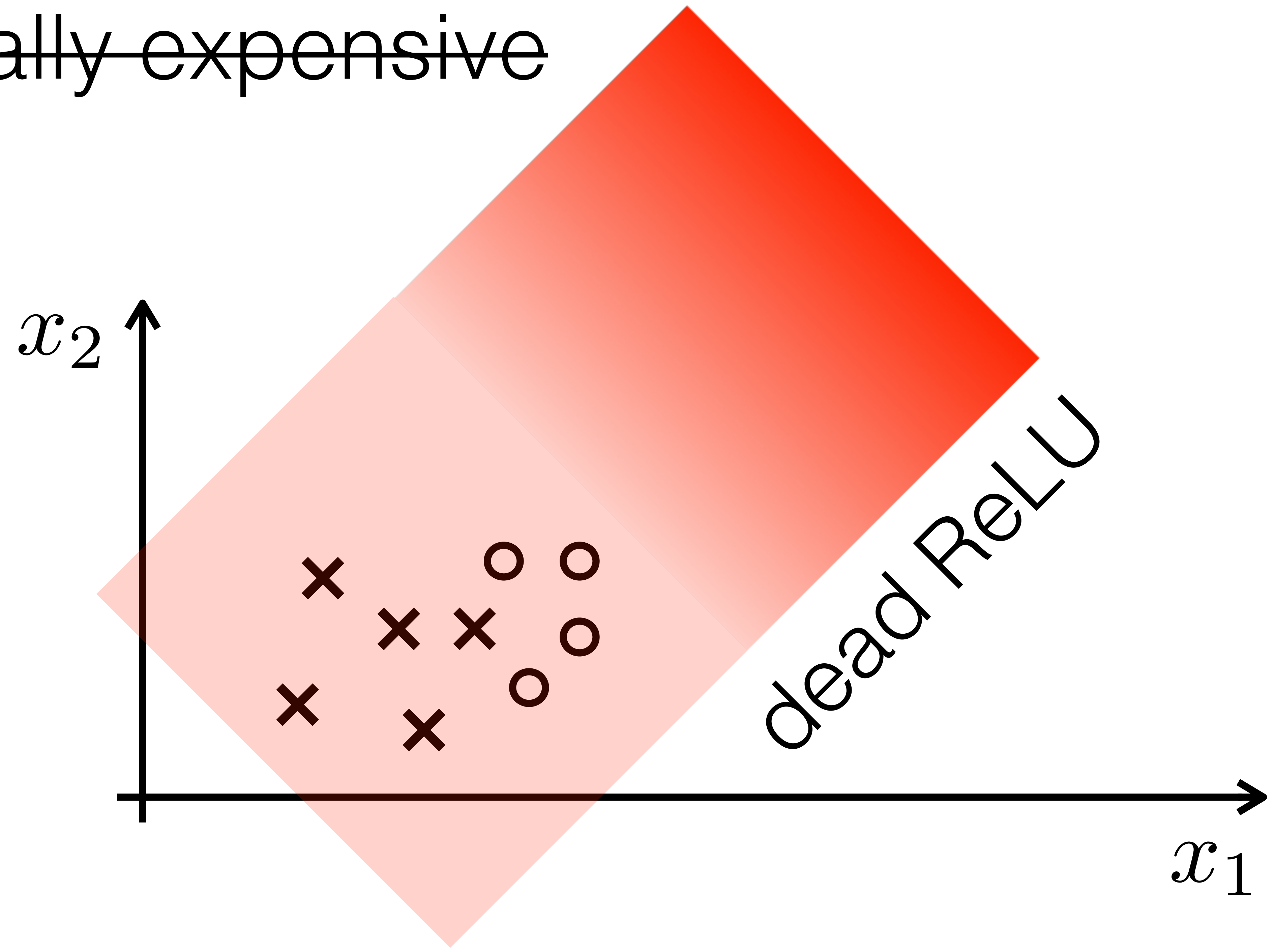
Activation functions

ReLU

$\max(0, x)$



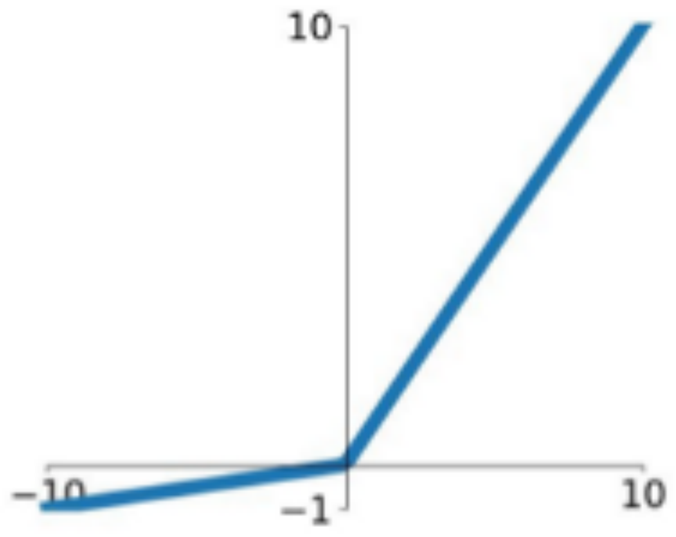
- ~~zero gradient when saturated~~ *(partially => dead ReLU!)*
- not zero-centered (only positive ouputs)
- ~~computationally expensive~~



Activation functions

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)`

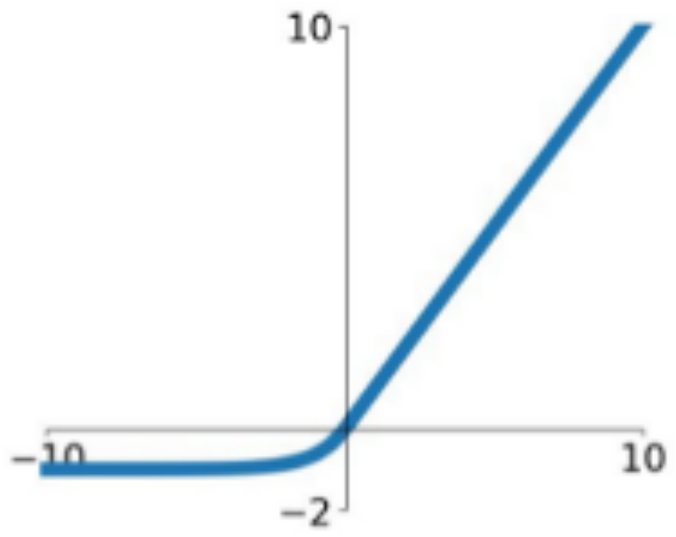
Small gradient for negative values gives tiny chance to recover

- 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

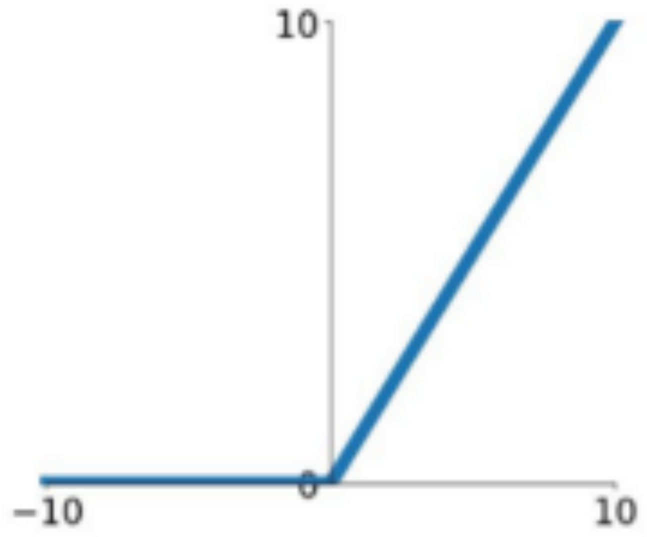
- PyTorch: `nn.LeakyReLU(alpha=1)`

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

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 to zero mean to avoid only positive inputs (and the unwanted “zig-zag” behaviour) - no PCA used!

Data preprocessing & initializations

- Input preprocessing:
  - Pixels values shifted to zero mean to avoid only positive inputs (and the unwanted "zig-zag" behaviour) - no PCA used!
- Weight initialization:
  - $\mathbf{w} = 0$ all gradients the same

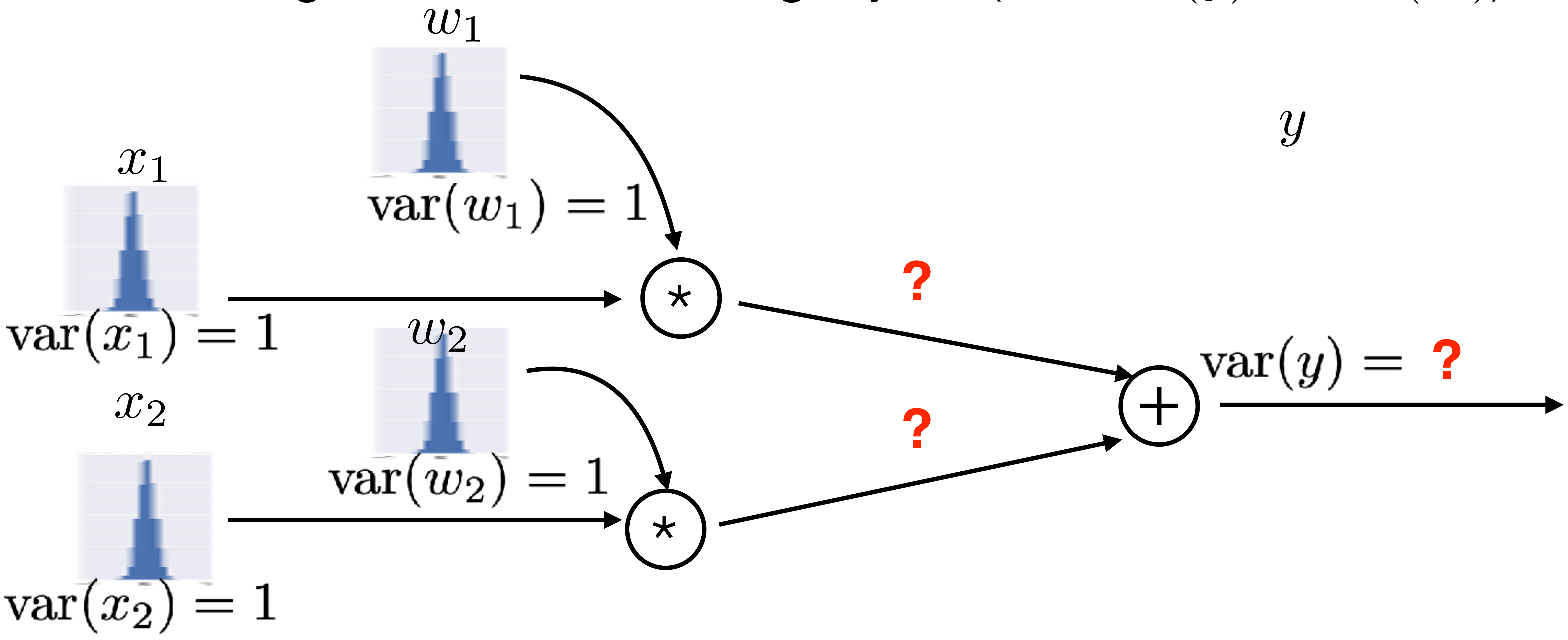
Data preprocessing & initializations

- Input preprocessing:
  - Pixels values shifted to zero mean to avoid only positive inputs (and the unwanted "zig-zag" behaviour) - no PCA used!
- Weight initialization:
  - $\mathbf{w} = 0$ all gradients the same
  - $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \sigma)$ diminishing/exploding values

Data preprocessing & initializations

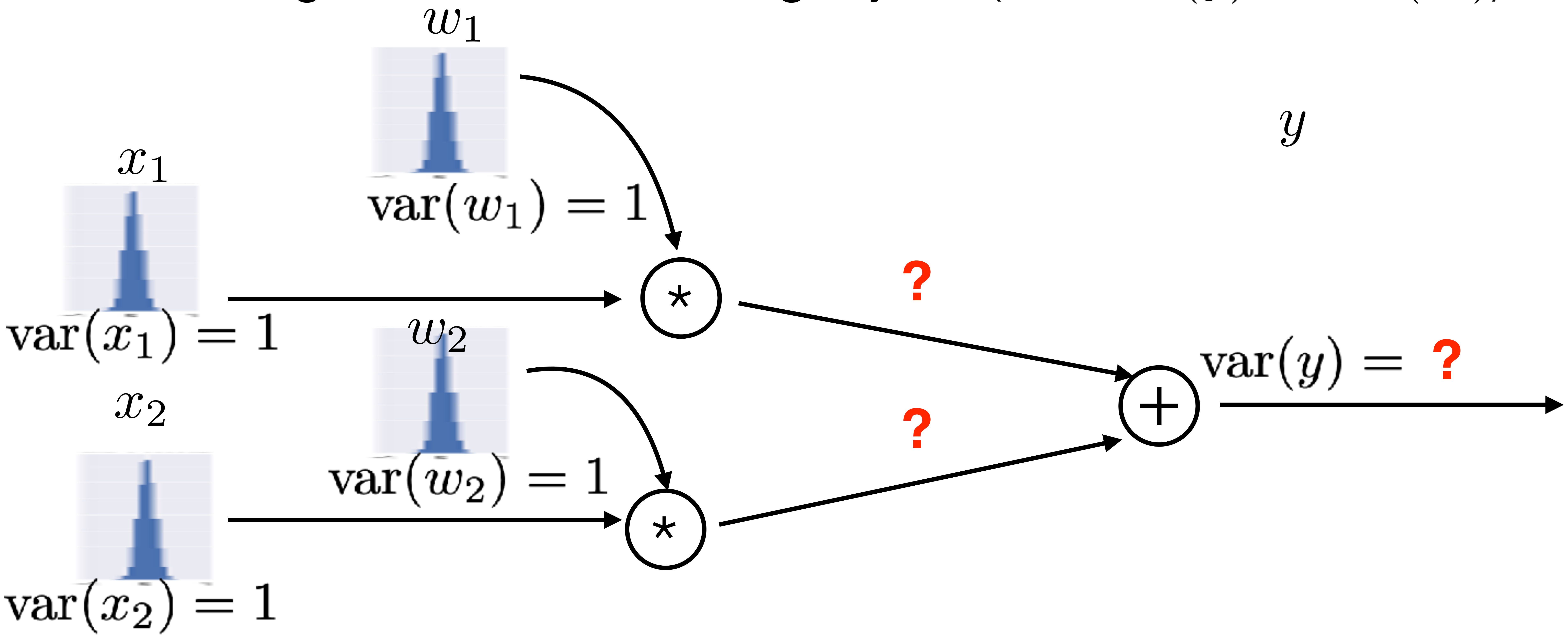
- Input preprocessing:
  - Pixels values shifted to zero mean to avoid only positive inputs (and the unwanted “zig-zag” behaviour) - no PCA used!
- Weight initialization:
  - $\mathbf{w} = 0$ all gradients the same
  - $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \sigma)$ diminishing/exploding values
  - $\mathbf{w}^{(i)} \sim \mathcal{N}(\mathbf{0}, 1/N^{(i)})$ preserves variance of signal among layers

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



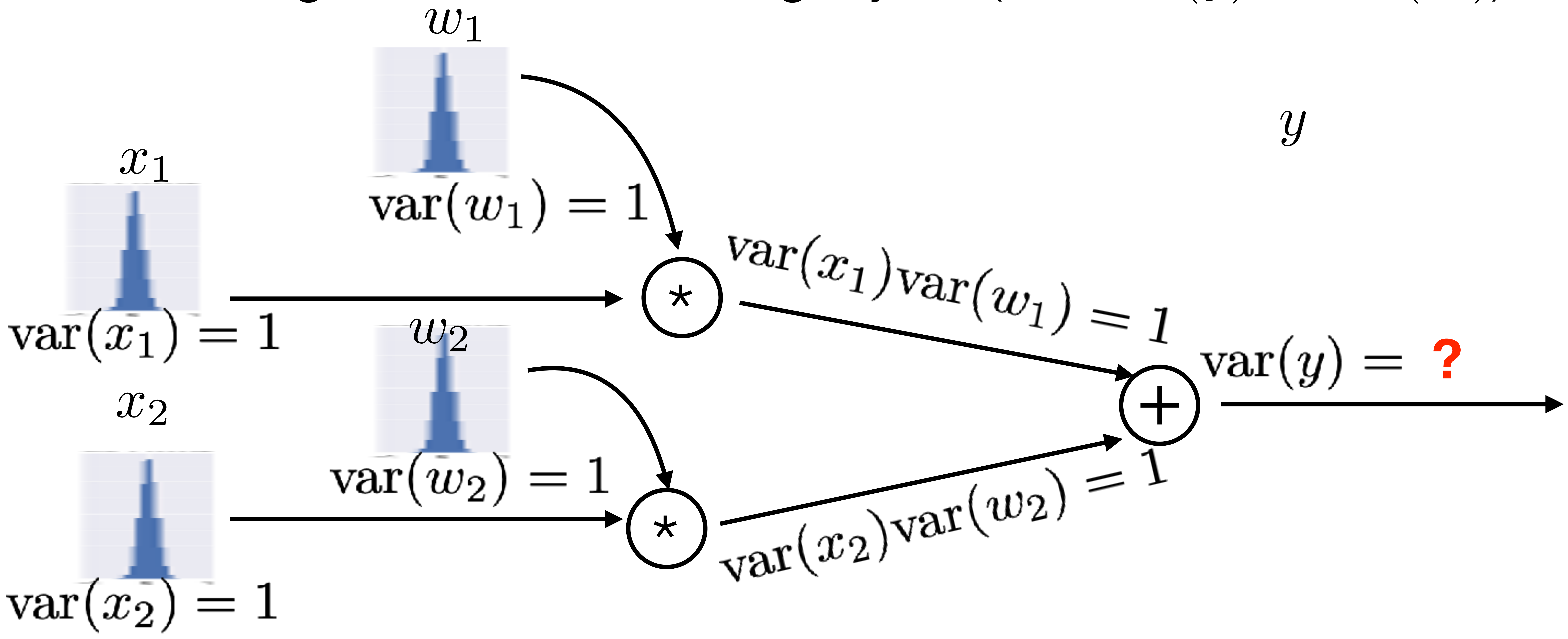
$$\mathrm{var}(x_1 w_1) = (\mathrm{var}(x_1) + \mu_{x_1}^2)(\mathrm{var}(w_1) + \mu_{w_1}^2) - \mu_{x_1}^2 \mu_{w_1}^2$$

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



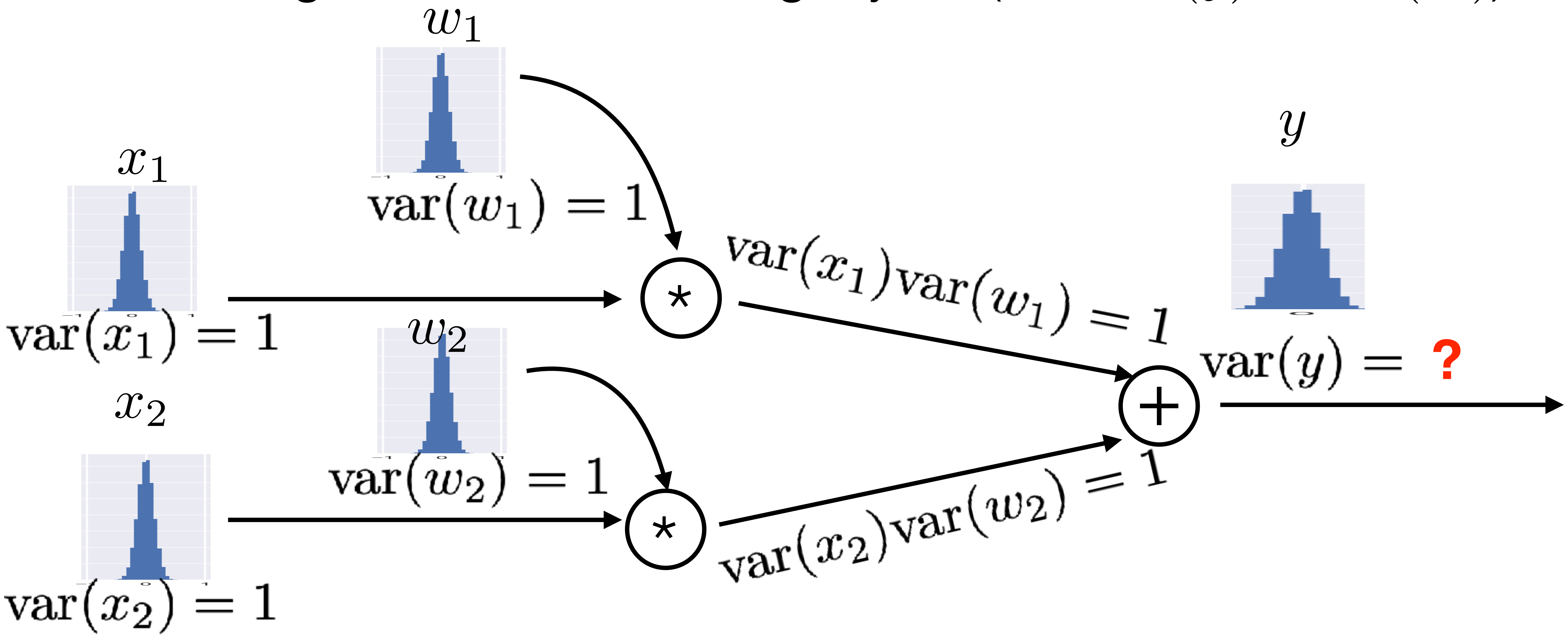
$$\mathrm{var}(x_1 w_1) = (\mathrm{var}(x_1) + \mu_{x_1}^2)(\mathrm{var}(w_1) + \mu_{w_1}^2) - \mu_{x_1}^2 \mu_{w_1}^2$$
$$= \mathrm{var}(x_1)\mathrm{var}(w_1) = 1$$

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



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

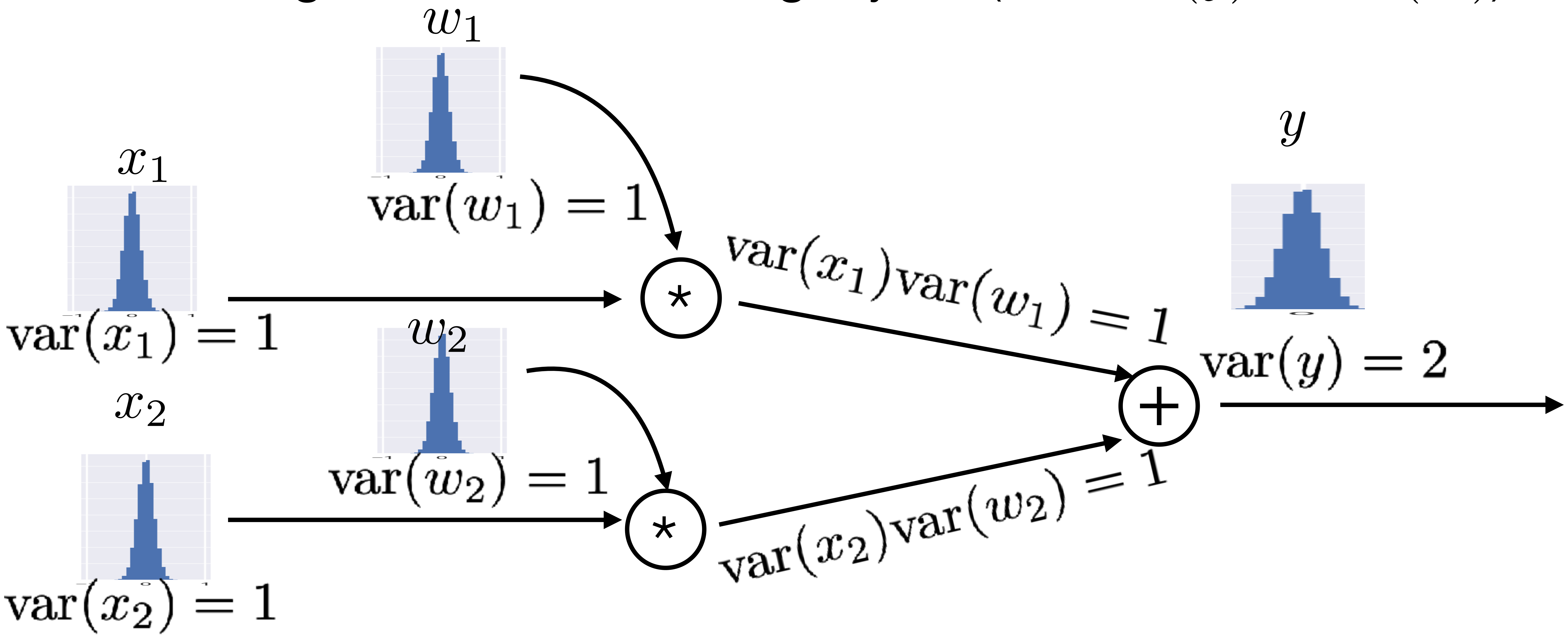
Preserve signal variance among layers (i.e. $\mathrm{var}(y) = \mathrm{var}(x_i)$)



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

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

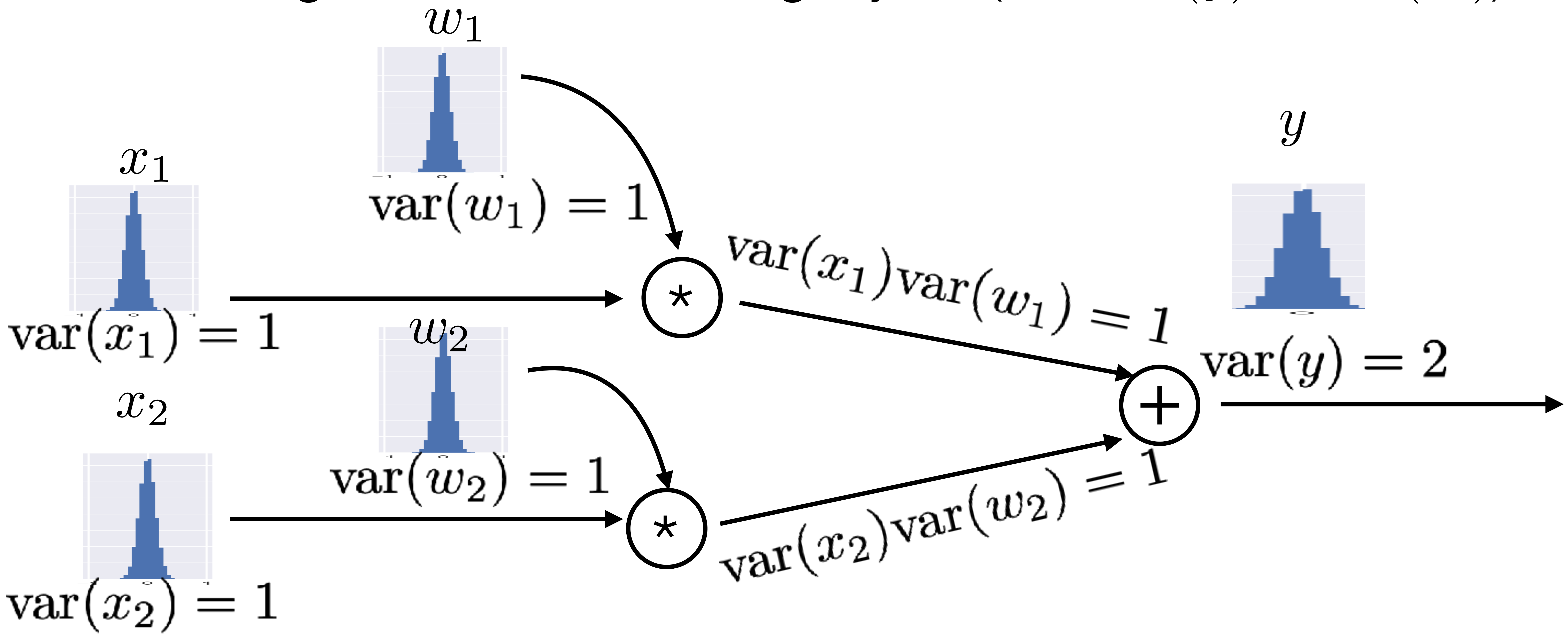
Preserve signal variance among layers (i.e. $\mathrm{var}(y) = \mathrm{var}(x_i)$)



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

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

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



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

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

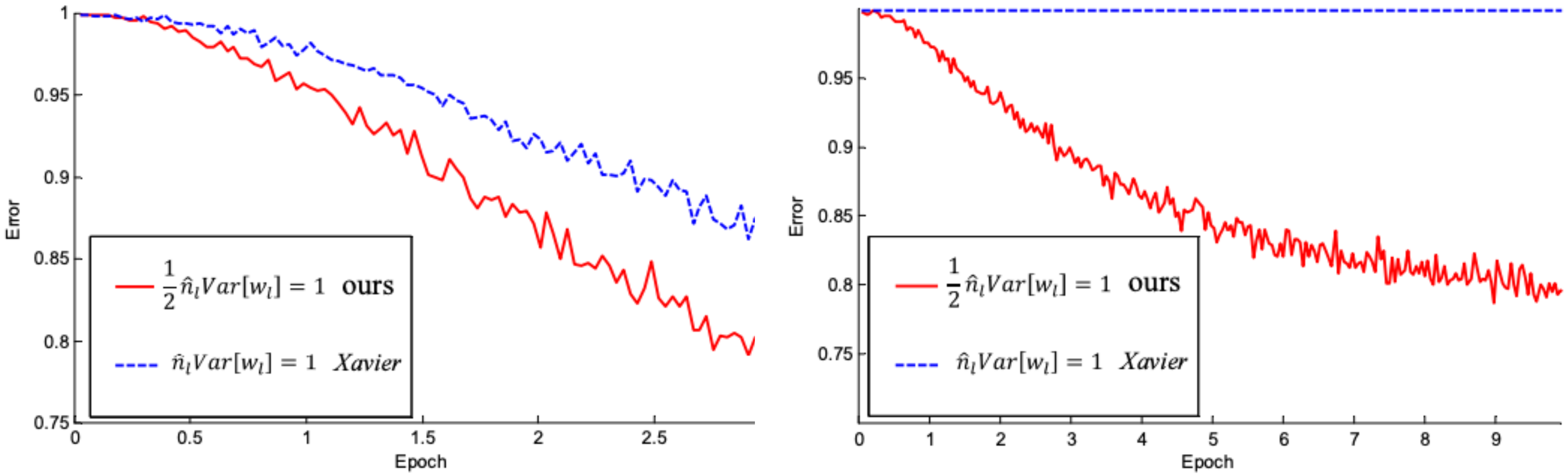
$$\mathrm{var}(y) = \mathrm{var}(w_1 x_1 + w_2 x_2 + \cdots + w_N x_N) = \\ = \sum_{i=1}^{N} \mathrm{var}(w_i)\mathrm{var}(x_i) \approx N * \mathrm{var}(w_i)\mathrm{var}(x_i) \Rightarrow \mathrm{var}(w_i) = \frac{1}{N}$$

Kaimimg initialization

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

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

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

Learning with mini-batches

input 4D tensor: batch_size x channels x height x width

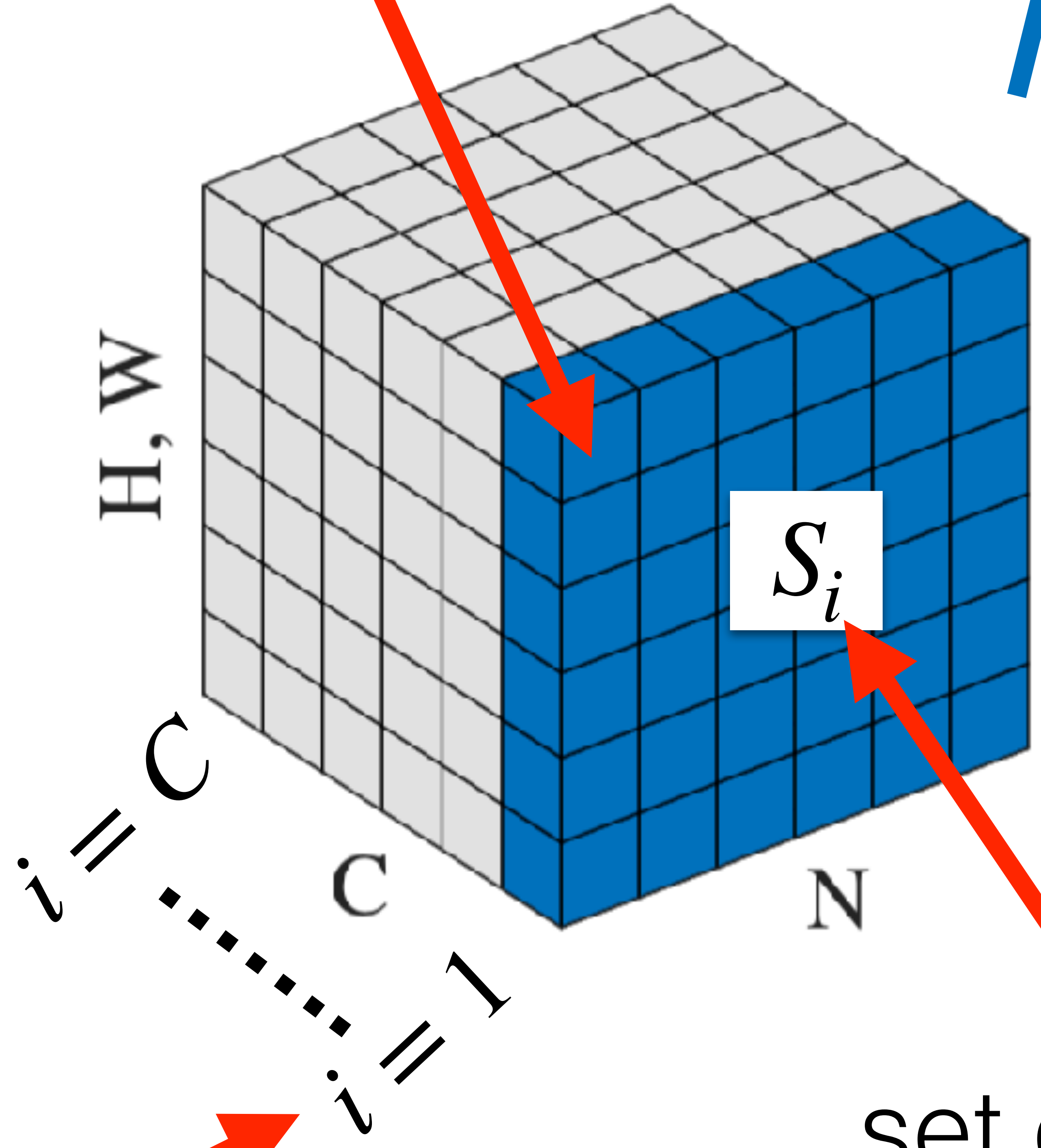
Batch normalization layer [Ioffe and Szegedy 2015]
https://arxiv.org/pdf/1502.03167.pdf (over 6k citation)

internal index within channel $\boldsymbol{i}$
k

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

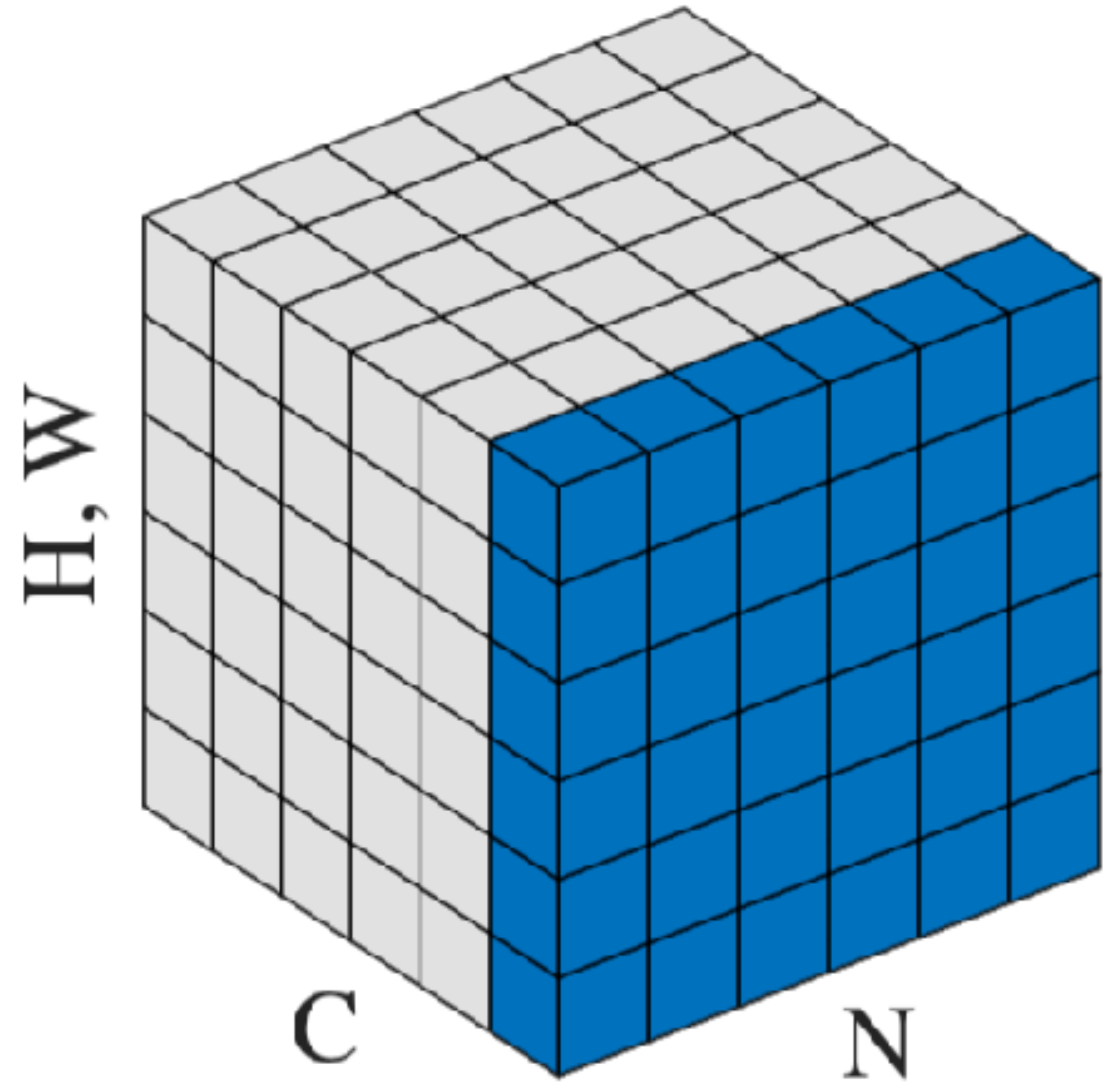
$$\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - \mu_i}{\sigma_i}$$

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

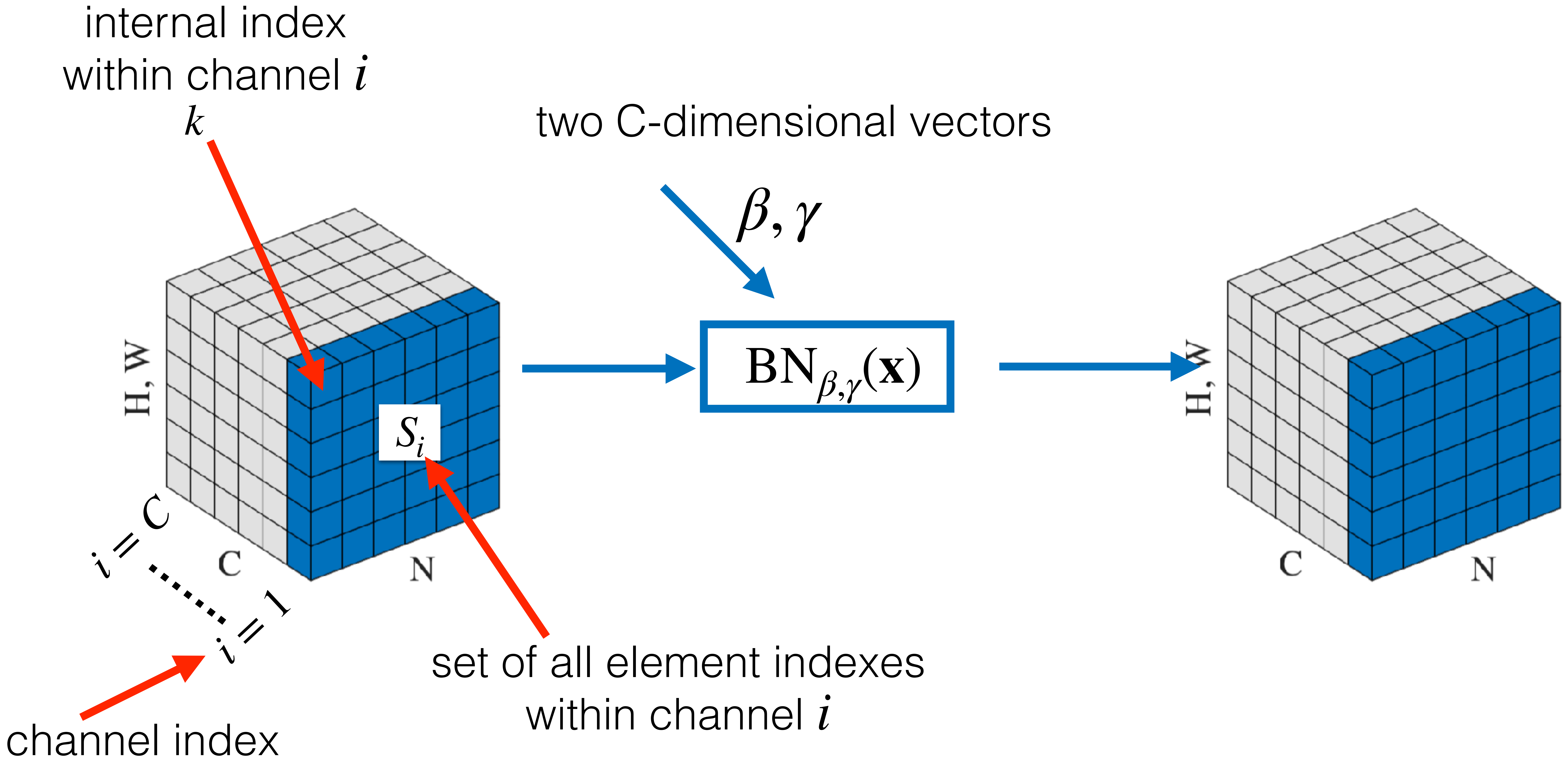


set of all element indexes within channel $\boldsymbol{i}$

channel index



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

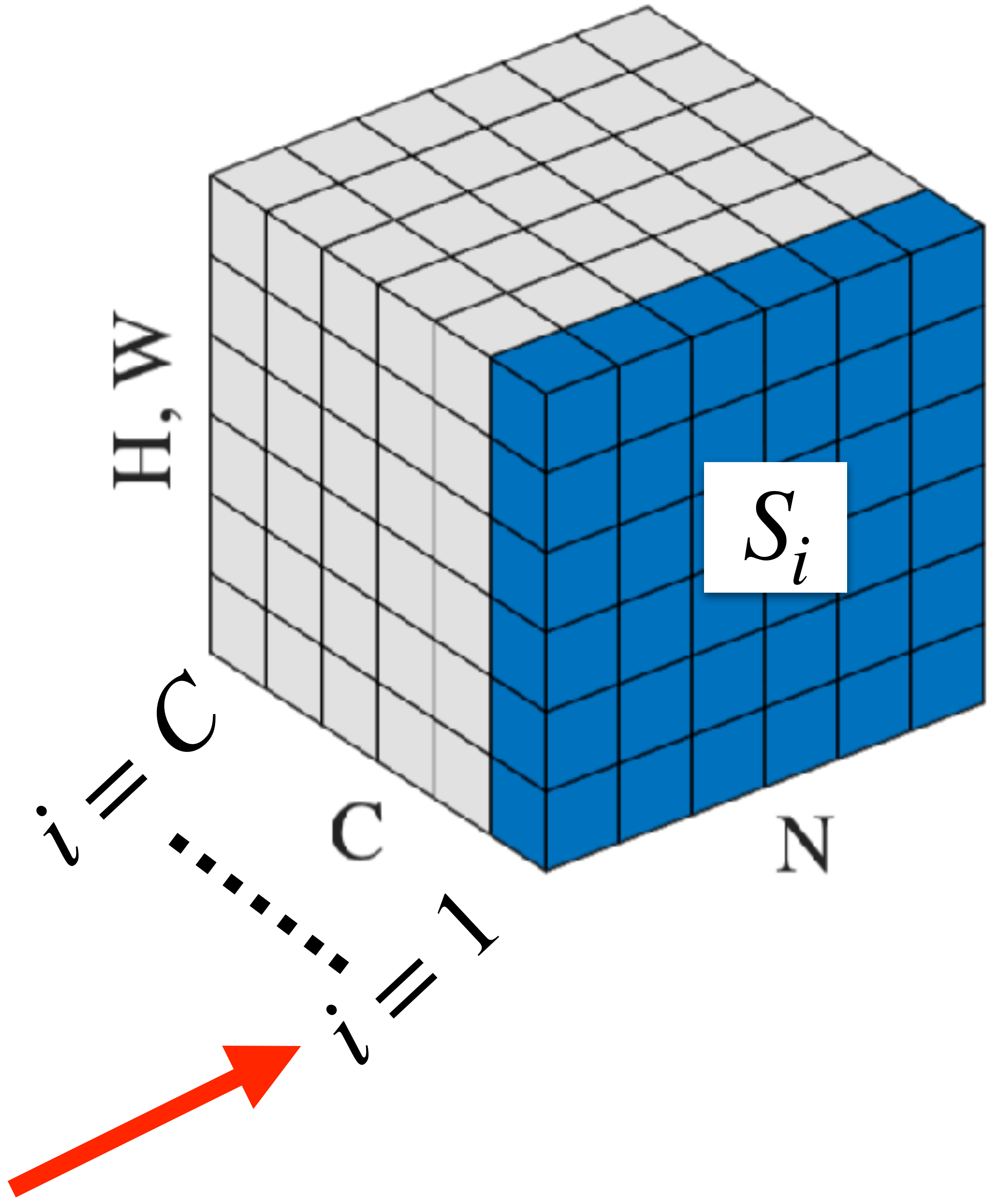


Batch normalization layer [Ioffe and Szegedy 2015]

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

batch size channels width height

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



channel index

What is dimensionality of the output?

the same: 20x100x35x45

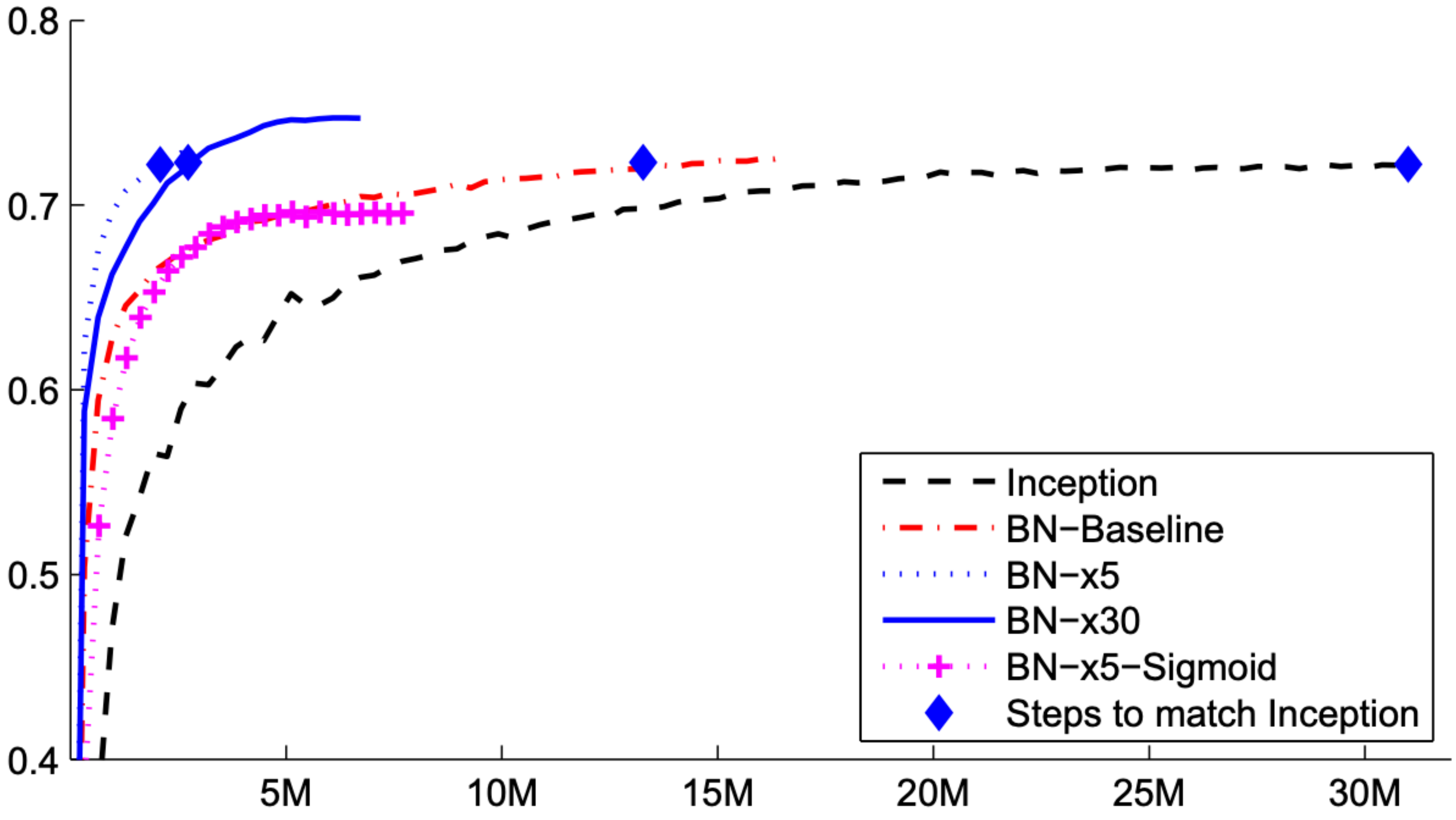
What is dimensionality of mean μ ?

100 dimensional vector

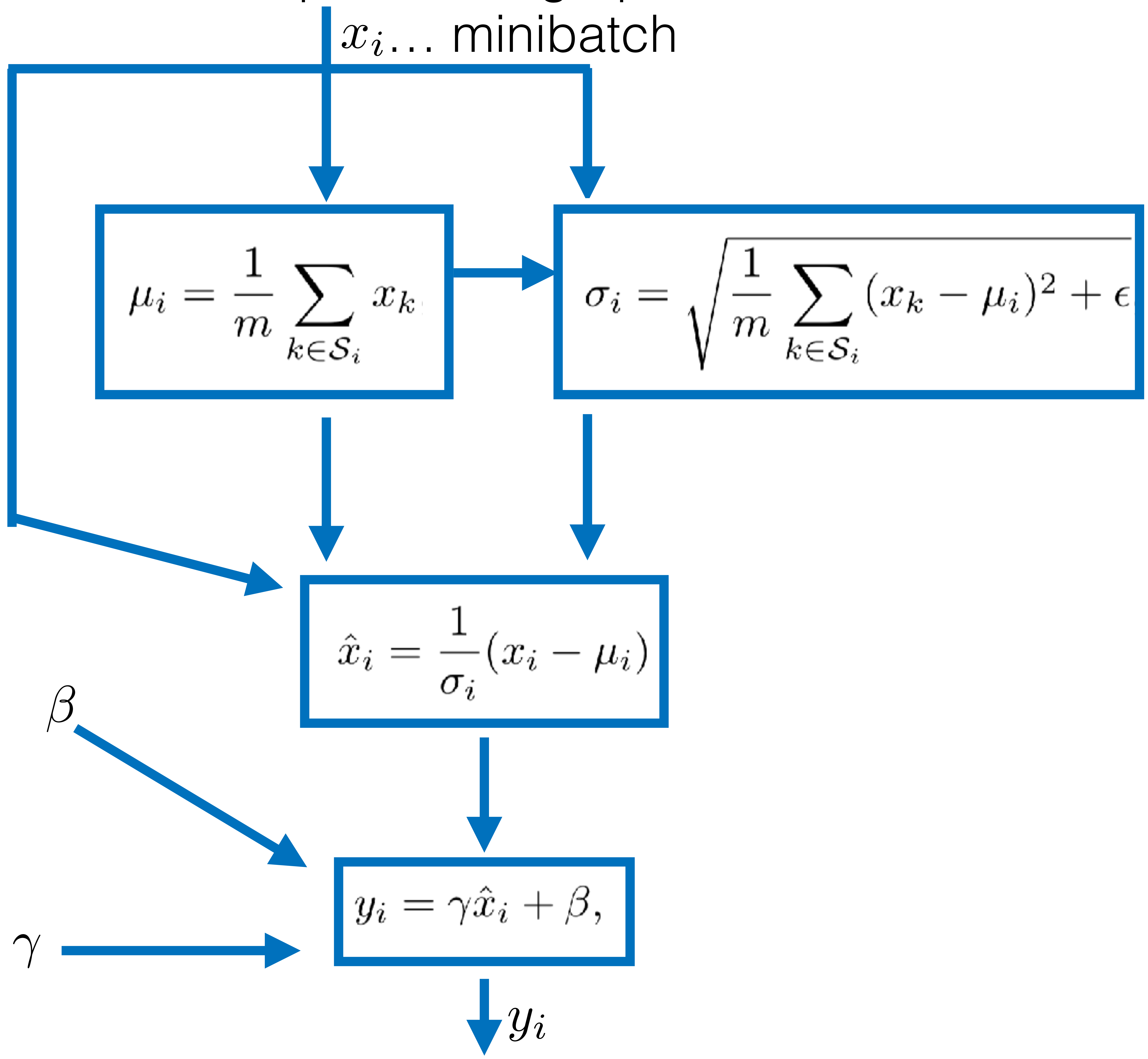
What is dimensionality of mean γ ?

100 dimensional vector

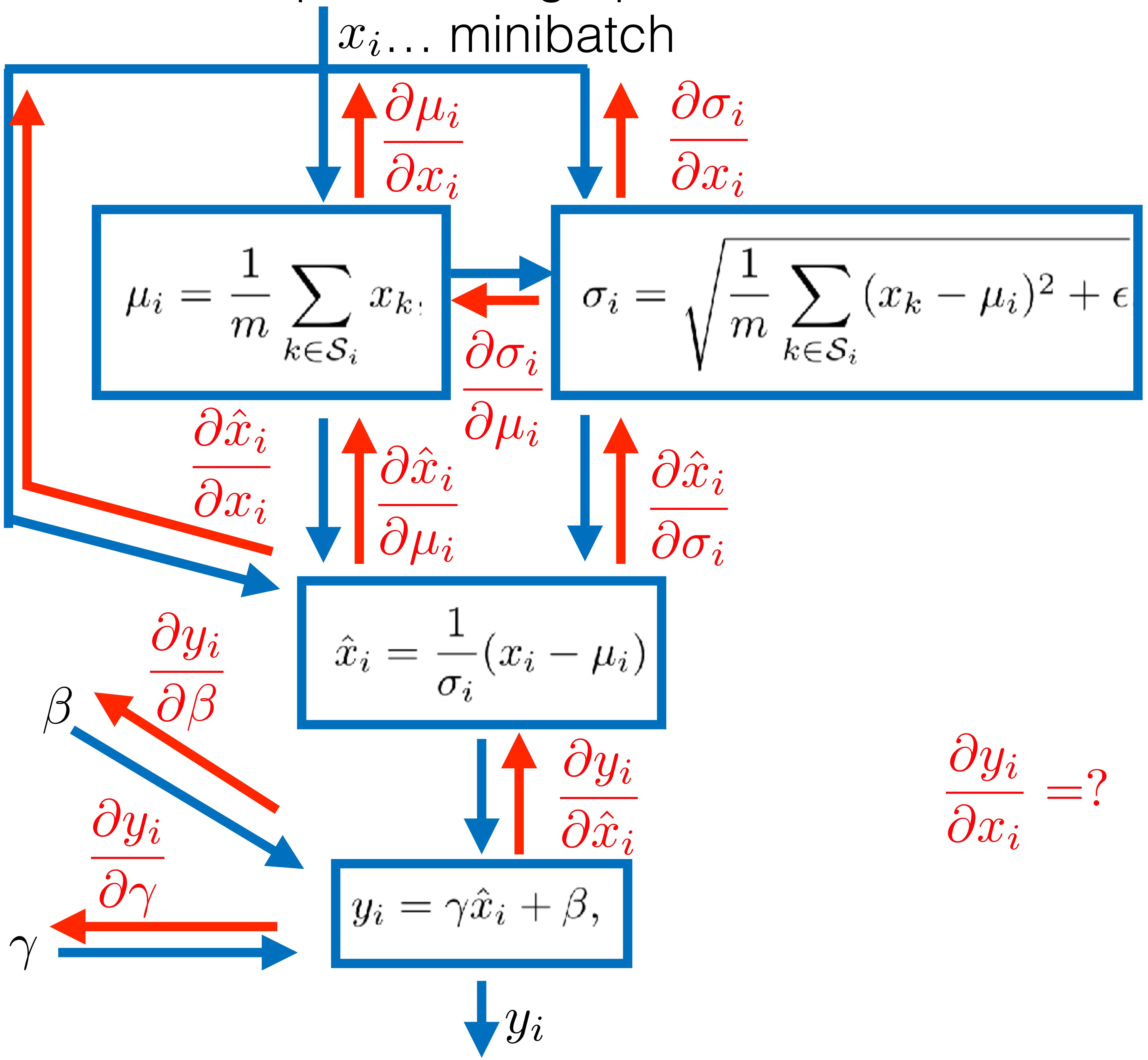
Batch normalization layer [Ioffe and Szegedy 2015]
https://arxiv.org/pdf/1502.03167.pdf (over 6k citation)



Computational graph of batch-norm

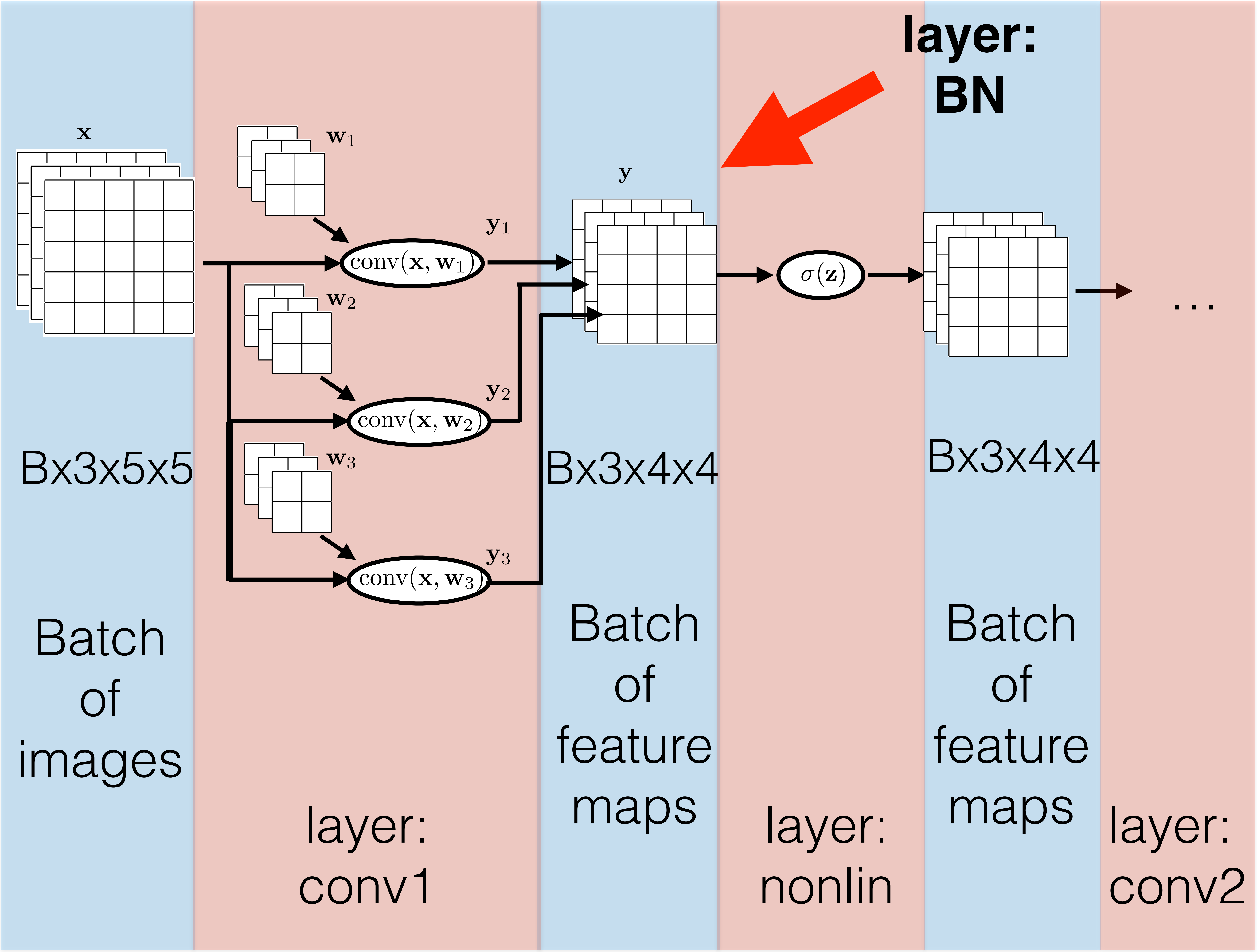


Computational graph of batch-norm

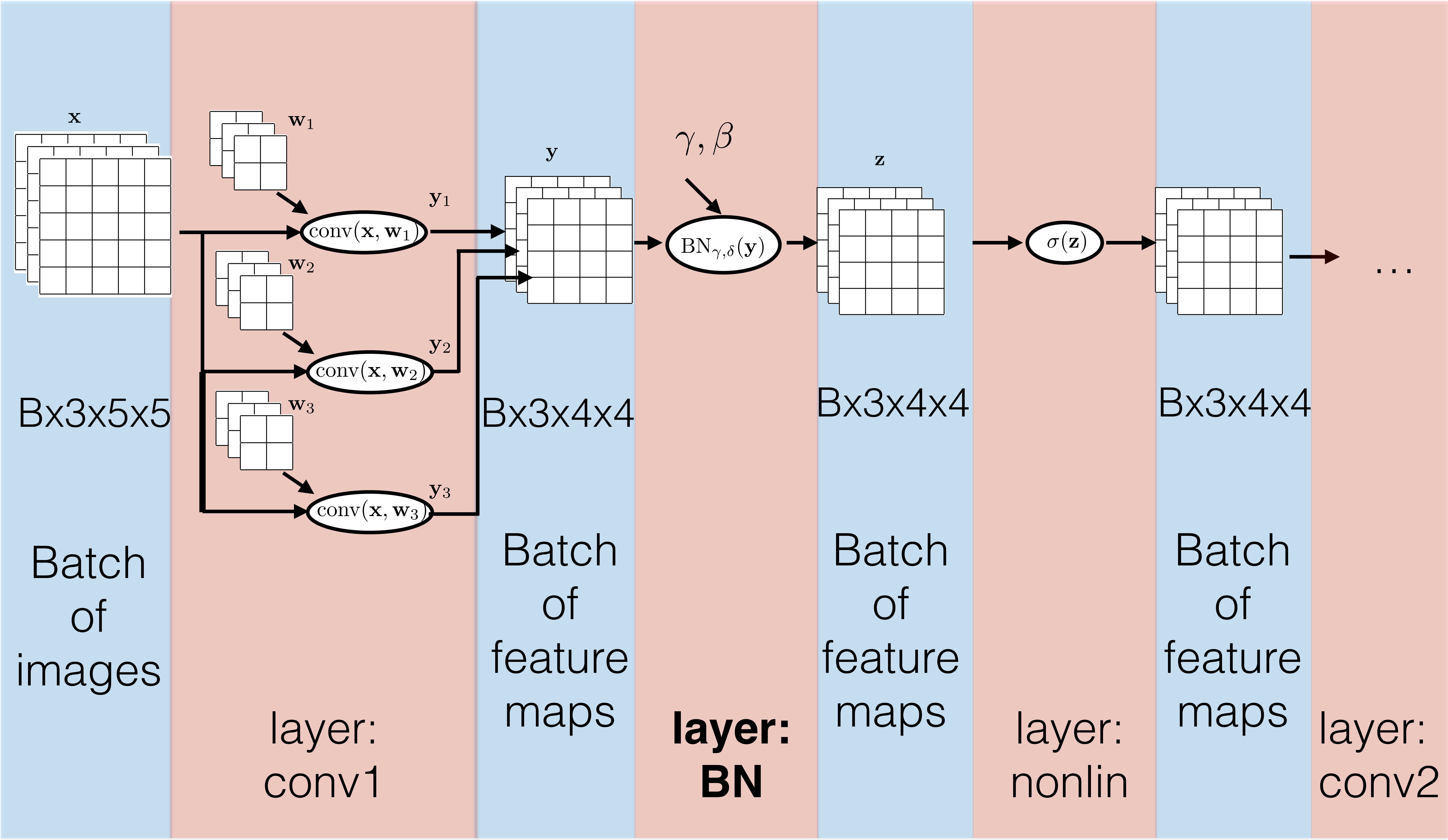


$$\frac{\partial y_i}{\partial x_i} = ?$$

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



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



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

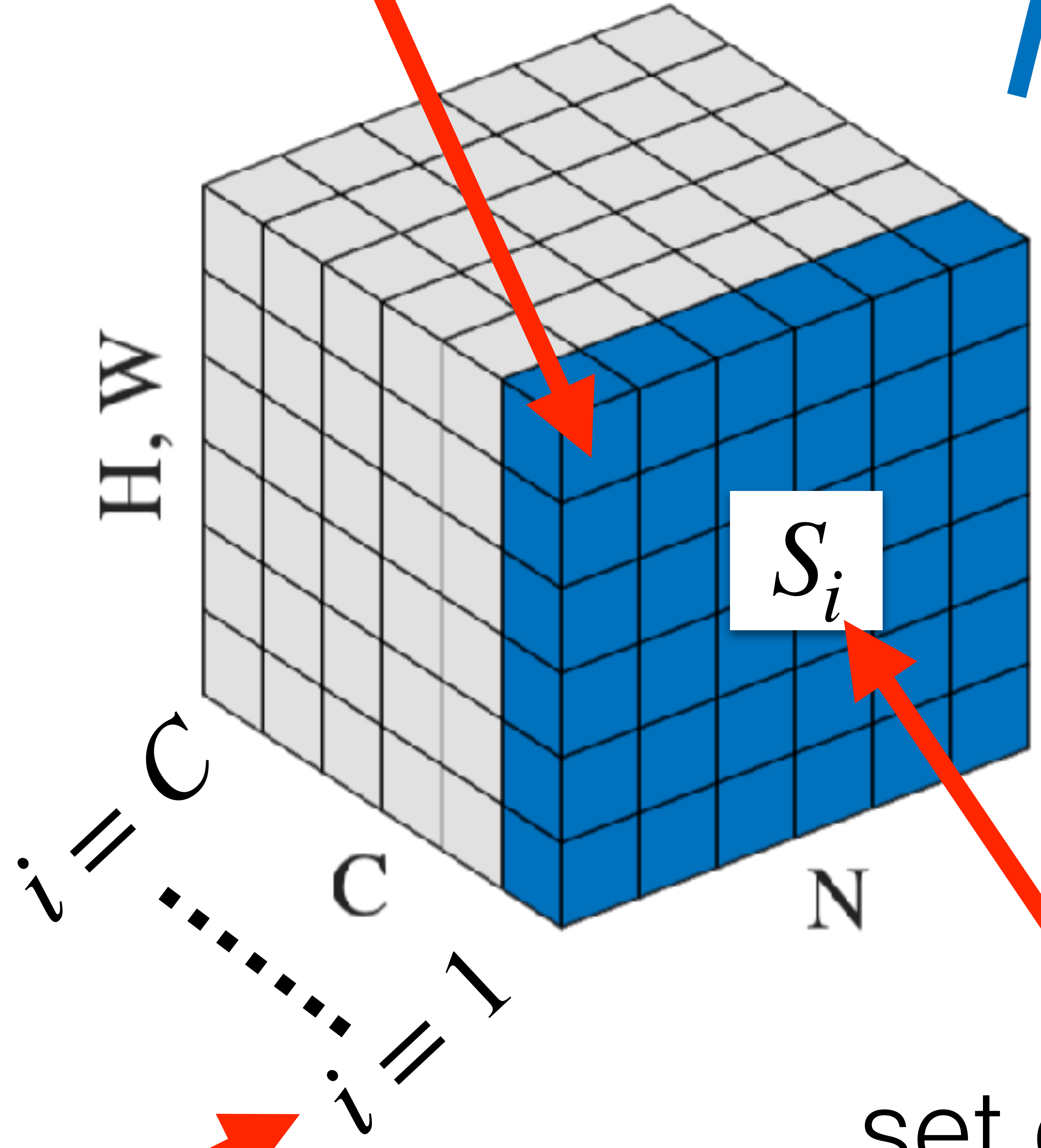
- BN improves beta-smoothness (i.e. Lipschitzness in loss and gradient) and predictivness.

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

Can you guess the drawback?

internal index within channel $\boldsymbol{i}$

k



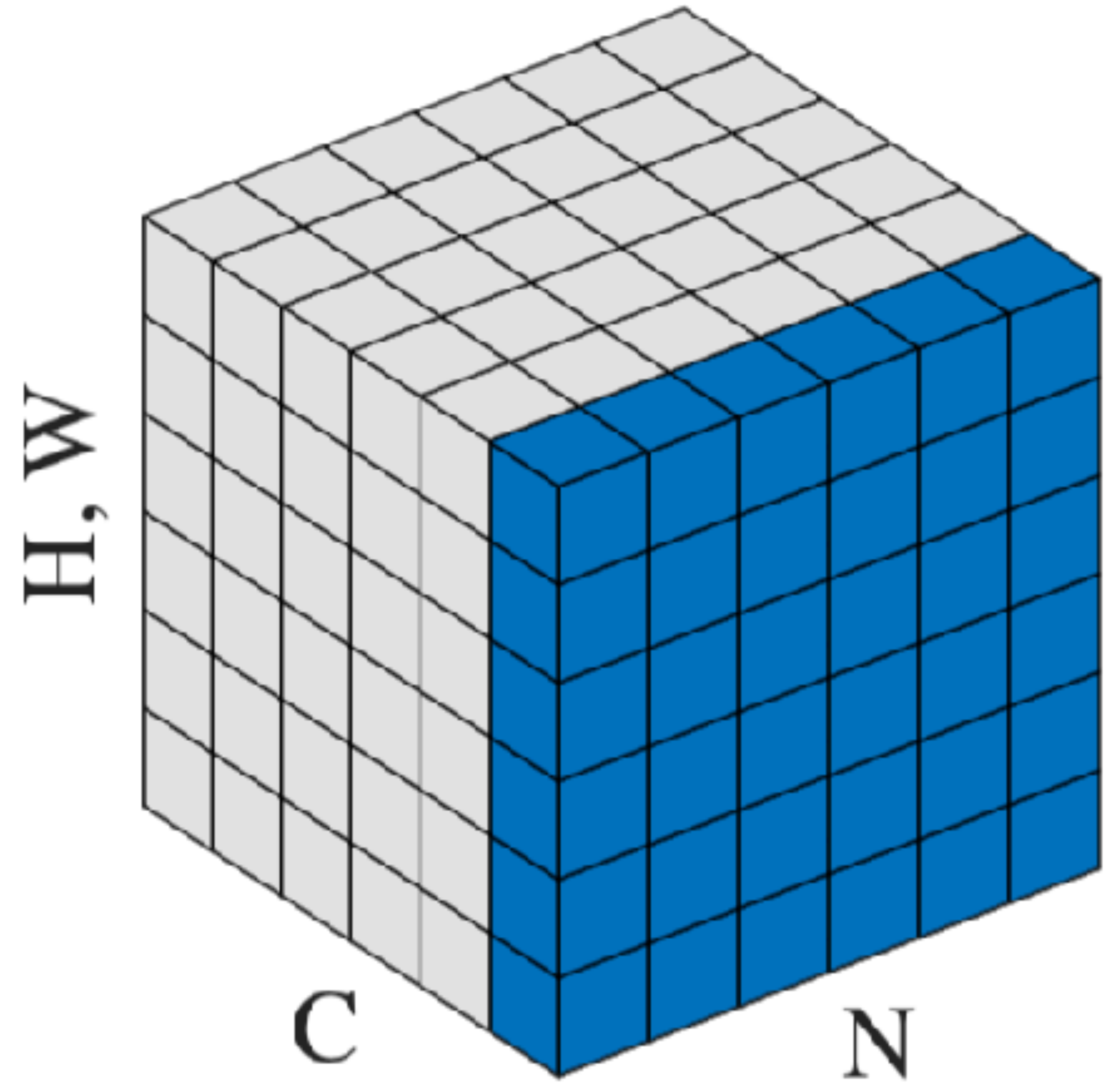
channel index

set of all element indexes within channel $\boldsymbol{i}$

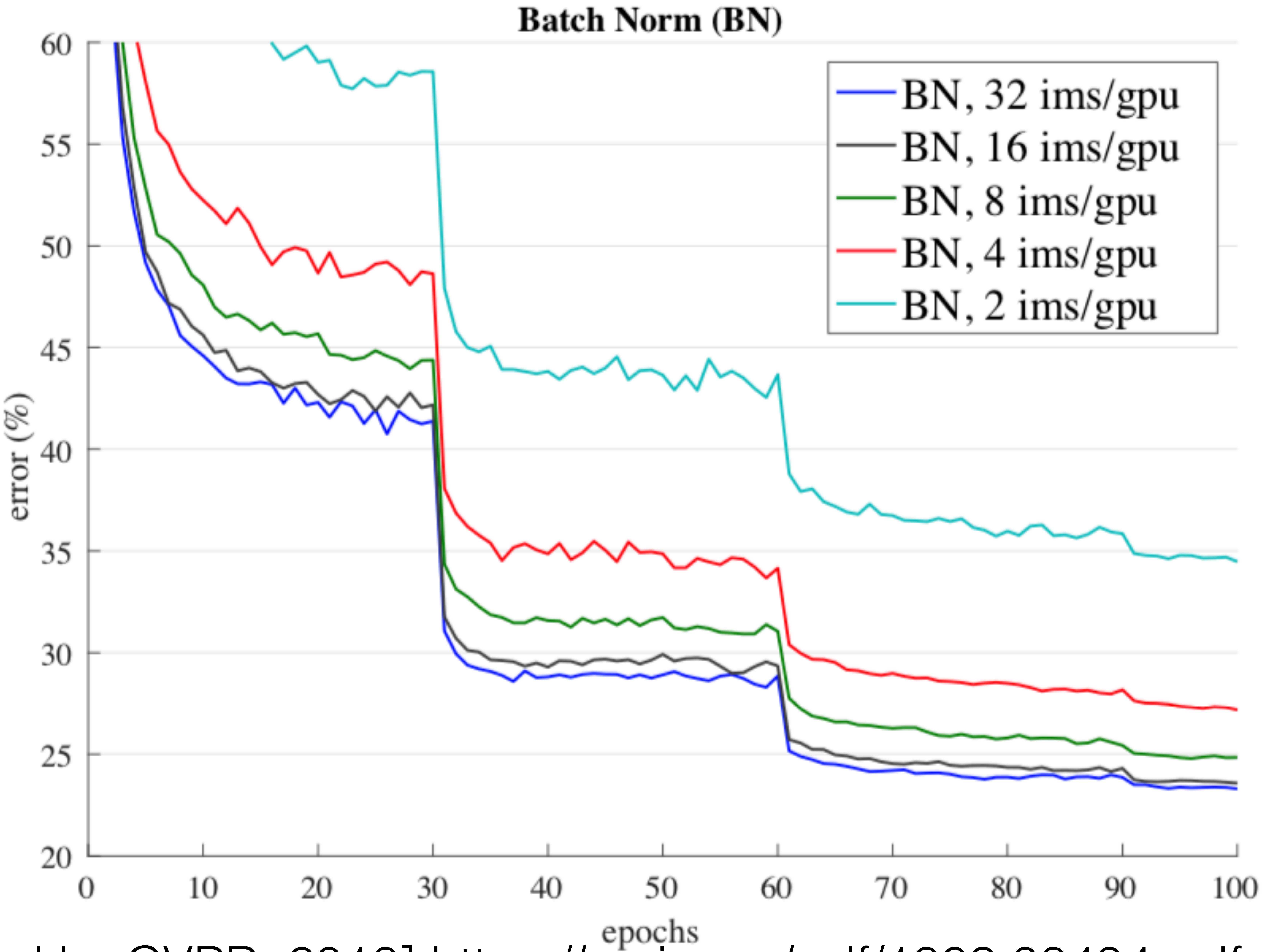
$$\mu_i = \frac{1}{m} \sum_{k \in S_i} \mathbf{x}_k \qquad \sigma_i = \sqrt{\frac{1}{m} \sum_{k \in S_i} (\mathbf{x}_k - \mu_i)^2 + \epsilon}$$

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

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



Batchnorm drawback: sensitivity to batch size



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

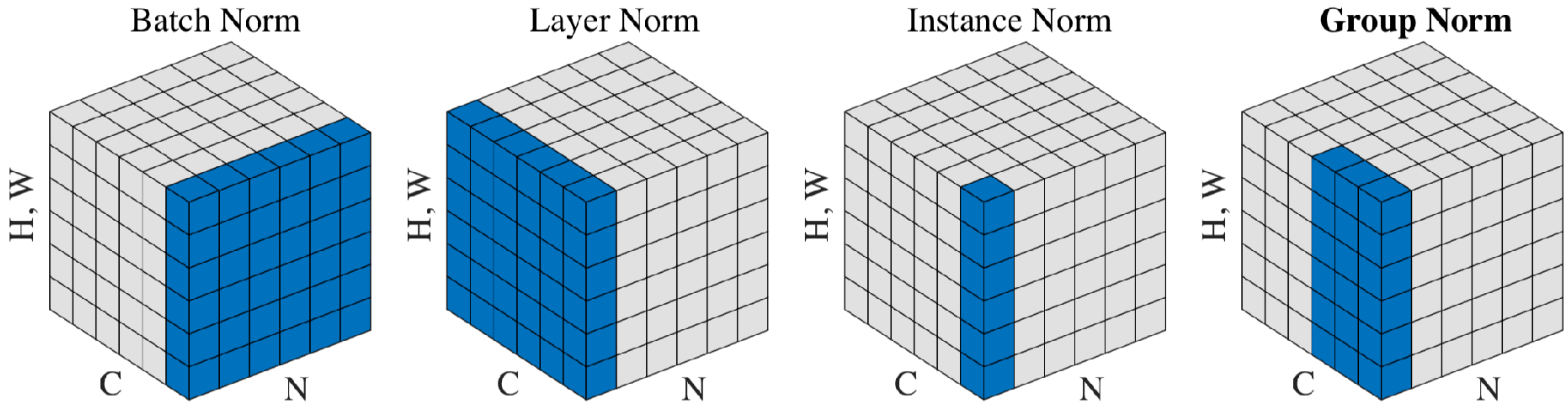
Batch Normalization - conclusions

- **Testing data** (no mini-batch available):
    - μ, σ estimated over the whole training set.
- **BN is reparametrization** of the original NN which has slightly higher expressive power.
- **Robust initialization:** many layers behave “as intended” around “normal” values.
- **Robust learning:** less sensitive to vanishing or exploding gradient (improves beta smoothness => faster learning).
- **BN is model regularizer:** one training example always normalized differently => small feature map jittering (dataset augmentation) => better generalization
- **Works well on classification** problems.
- **Not suitable for recurrent networks.** Different BN for each time-stamp => need to store statistics for each time-stamp.
- **Does not work on generative networks.** The reason is unclear.

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

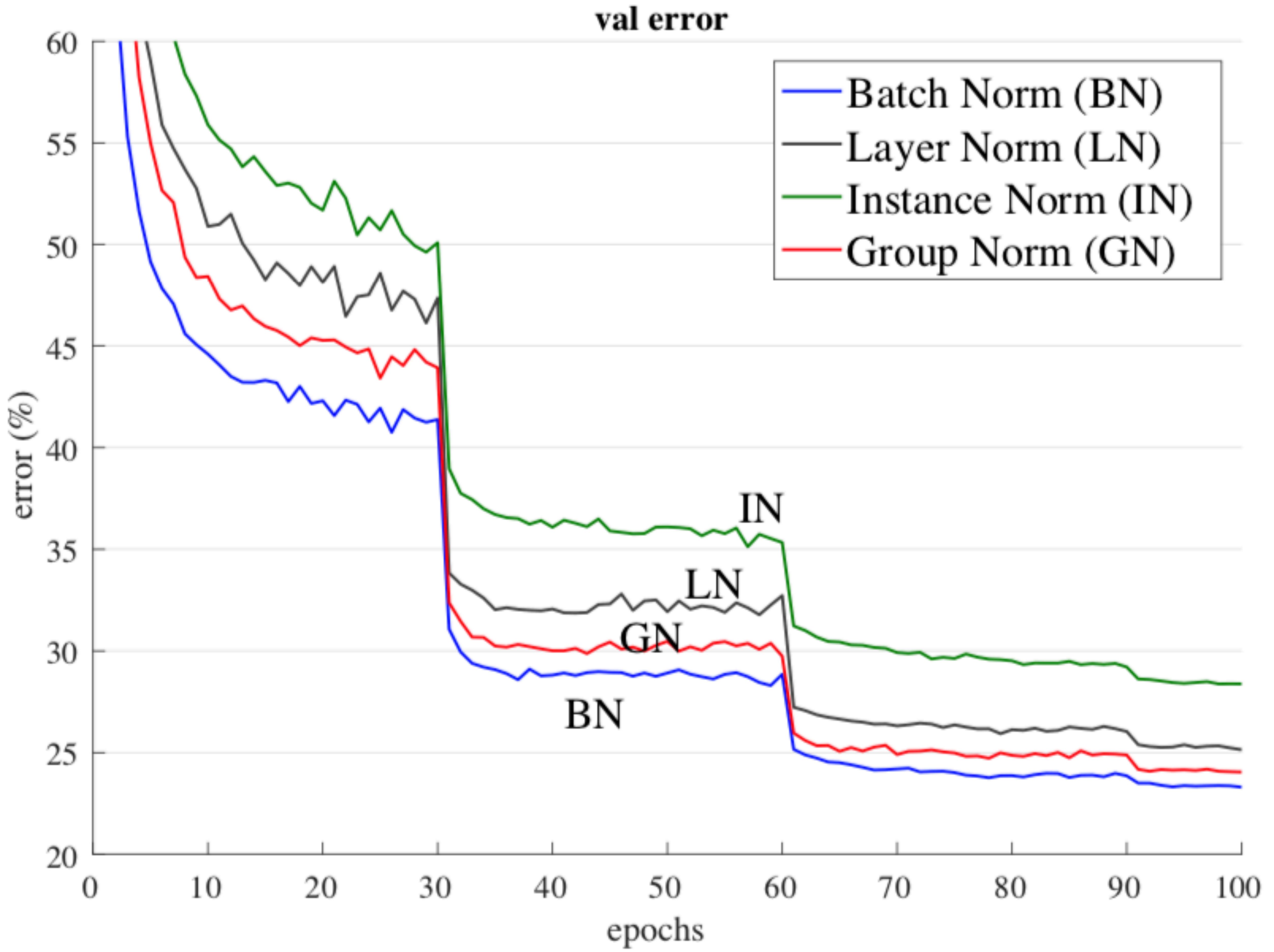


Classification RNN Style transfer

Classification task

- BN is best for classification tasks (including segmentation/detection)
- GN achieves similar performance

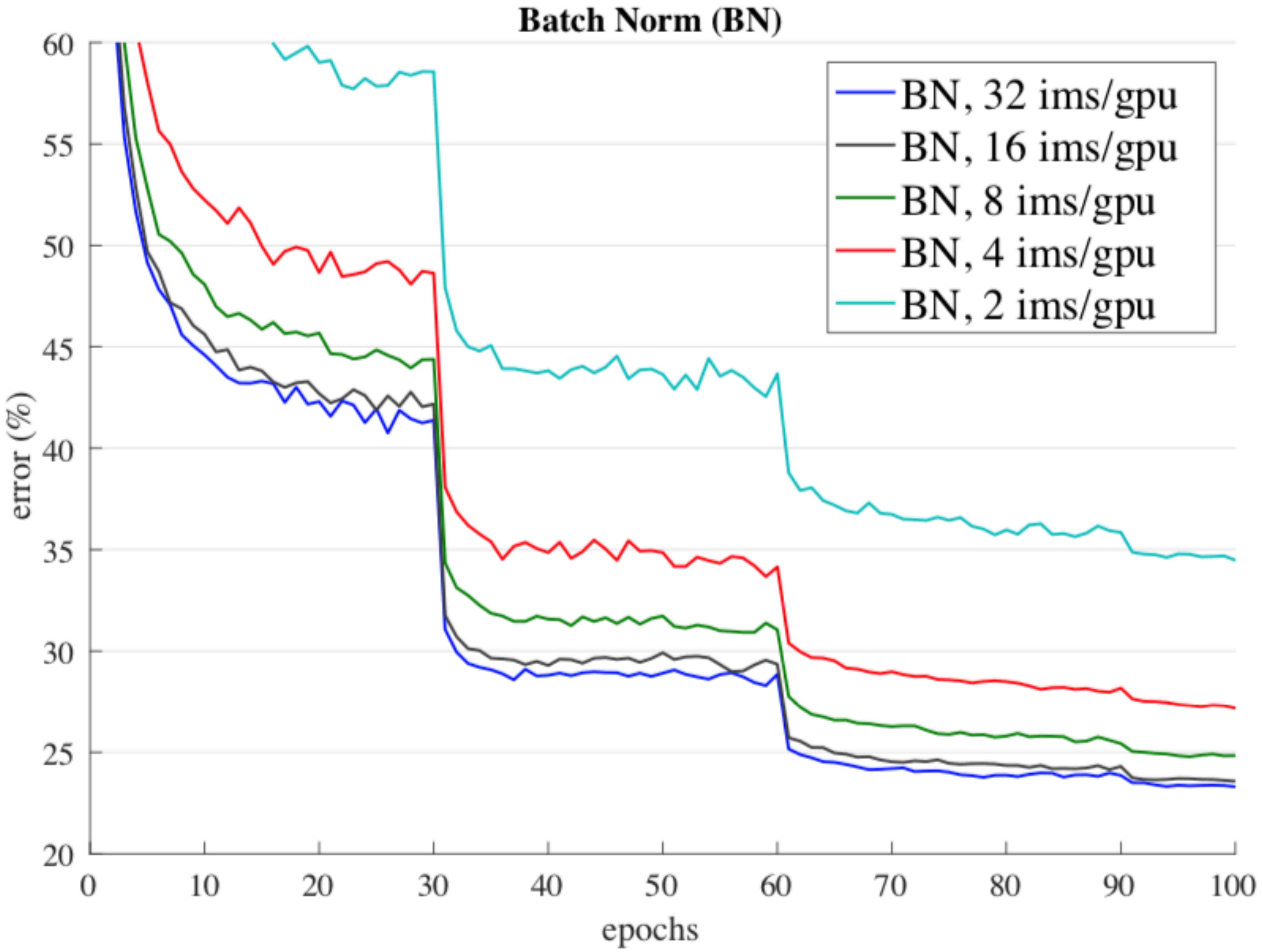
Sufficiently large mini-batch size = 32



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

Group Normalization - conclusions

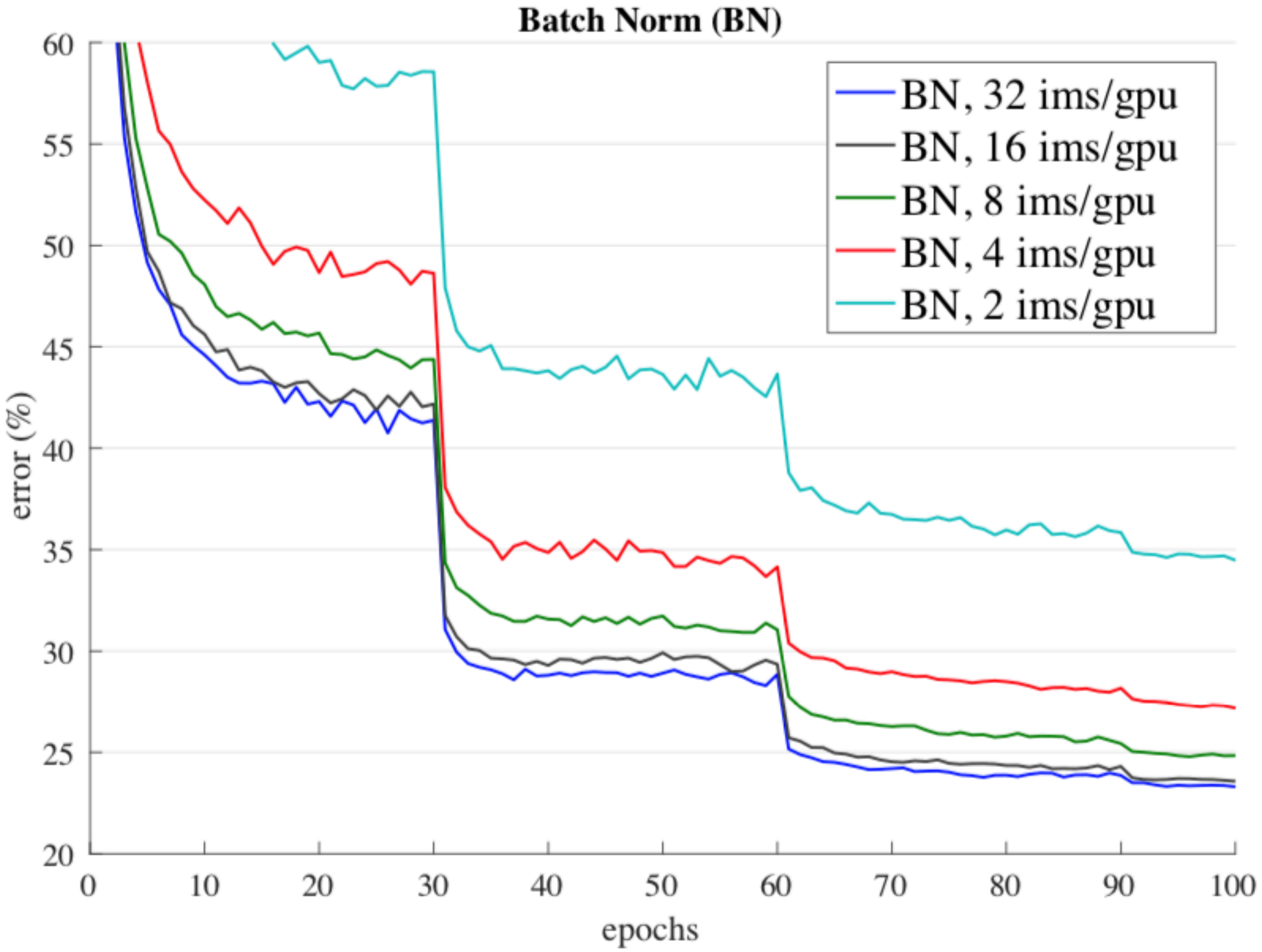
- BN is sensitive to mini-batch size.

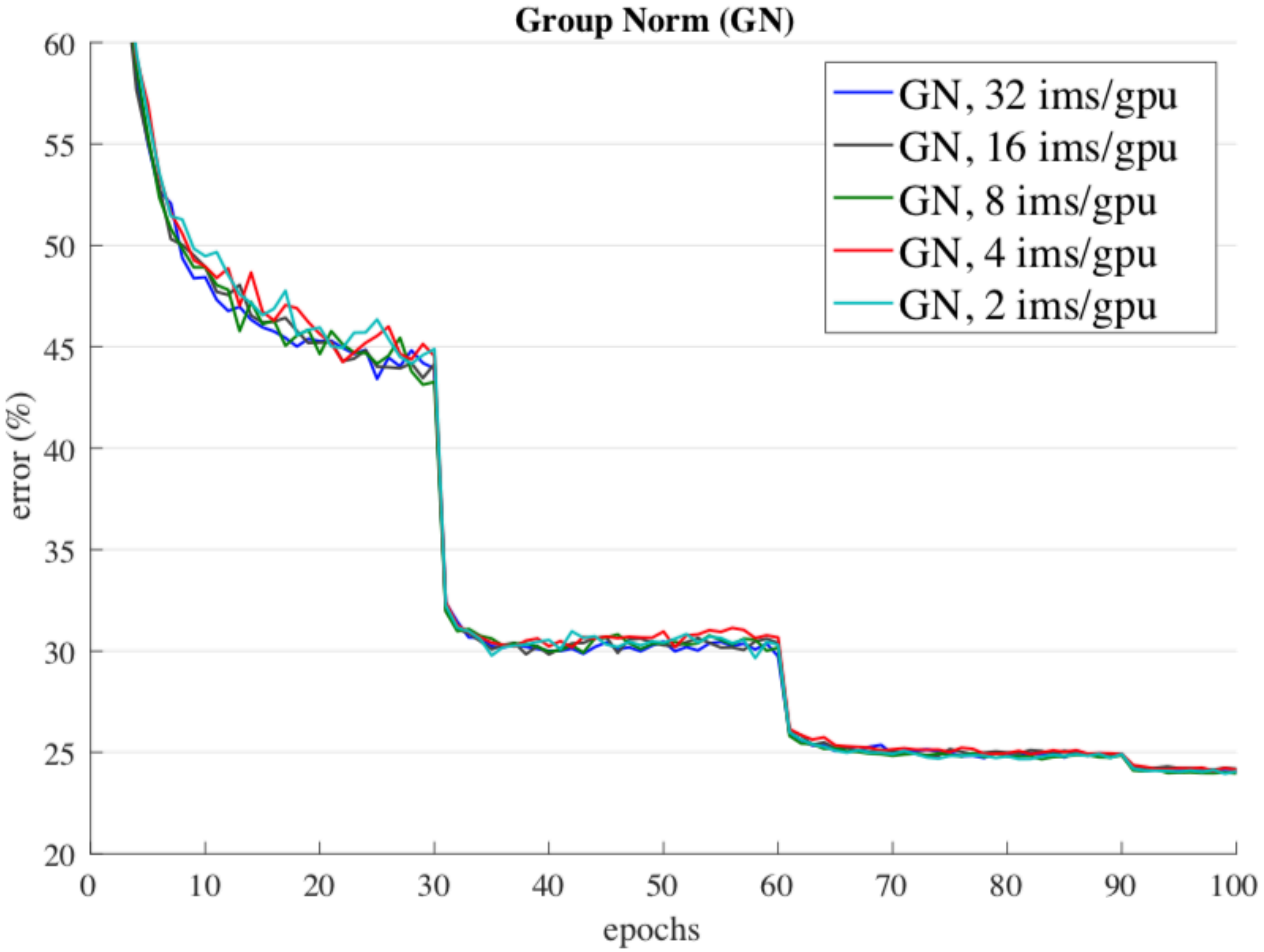


[Wu, He, CVPR, 2018] https://arxiv.org/pdf/1803.08494.pdf

Group Normalization - conclusions

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



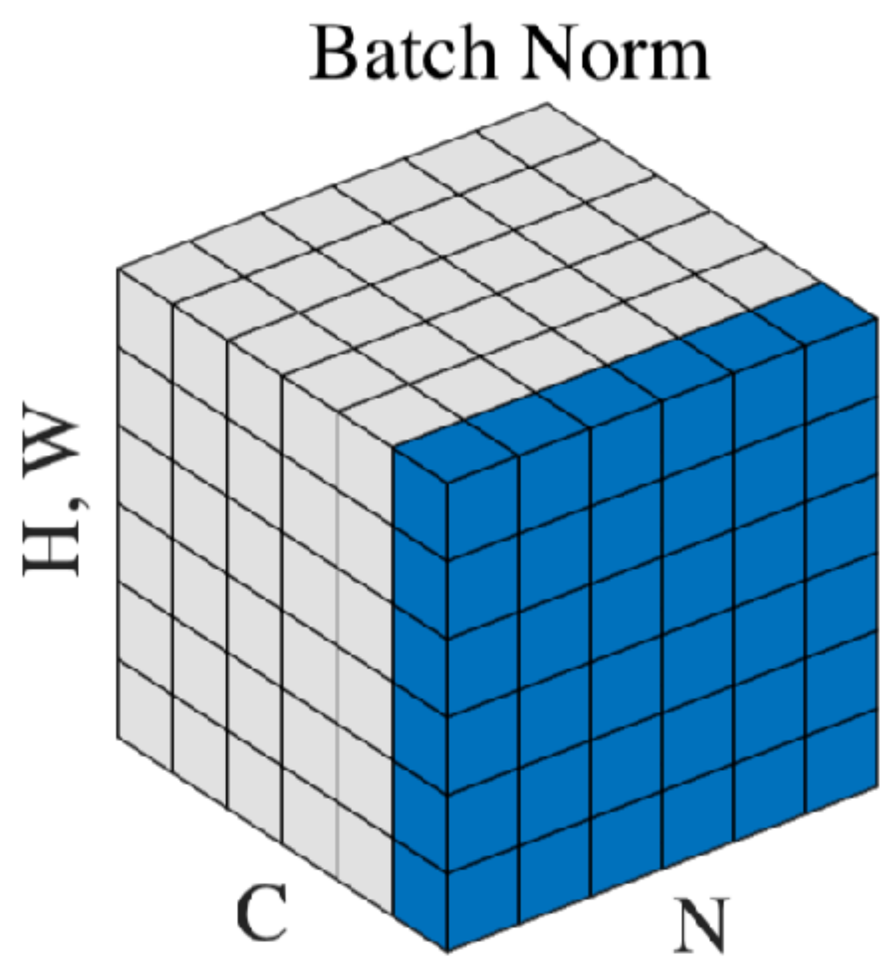


[Wu, He, CVPR, 2018] https://arxiv.org/pdf/1803.08494.pdf

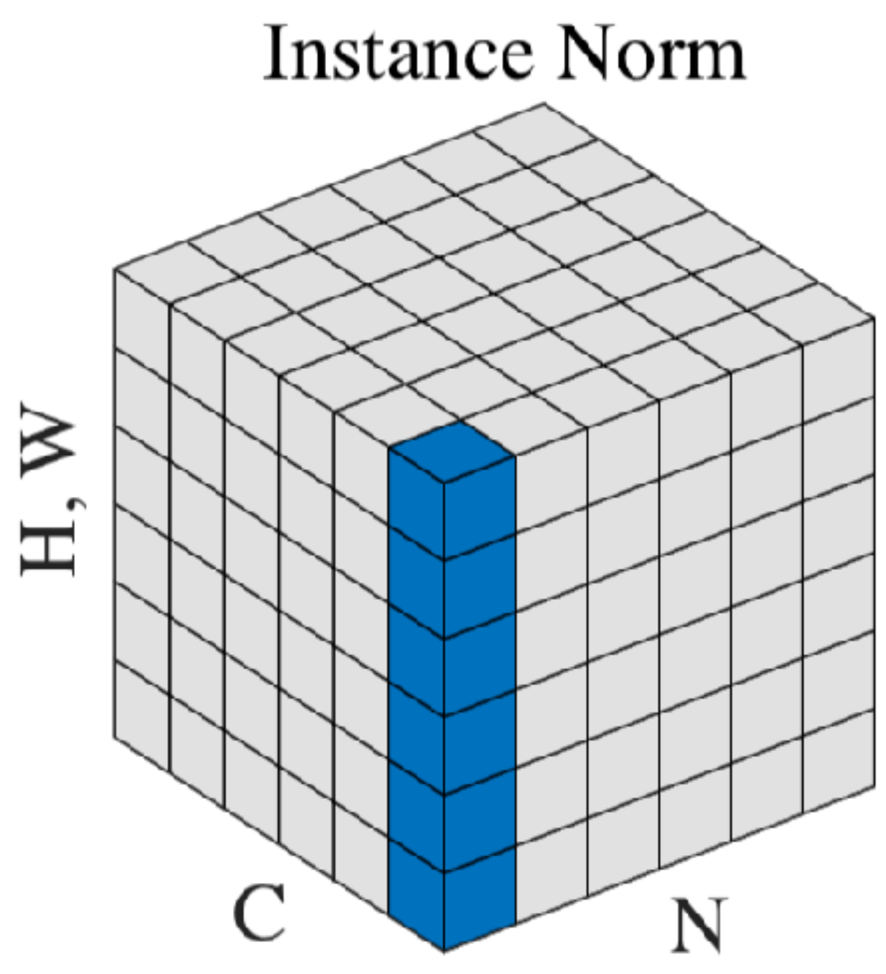
Batch-Instance normalization

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

What if we take best of both worlds?



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



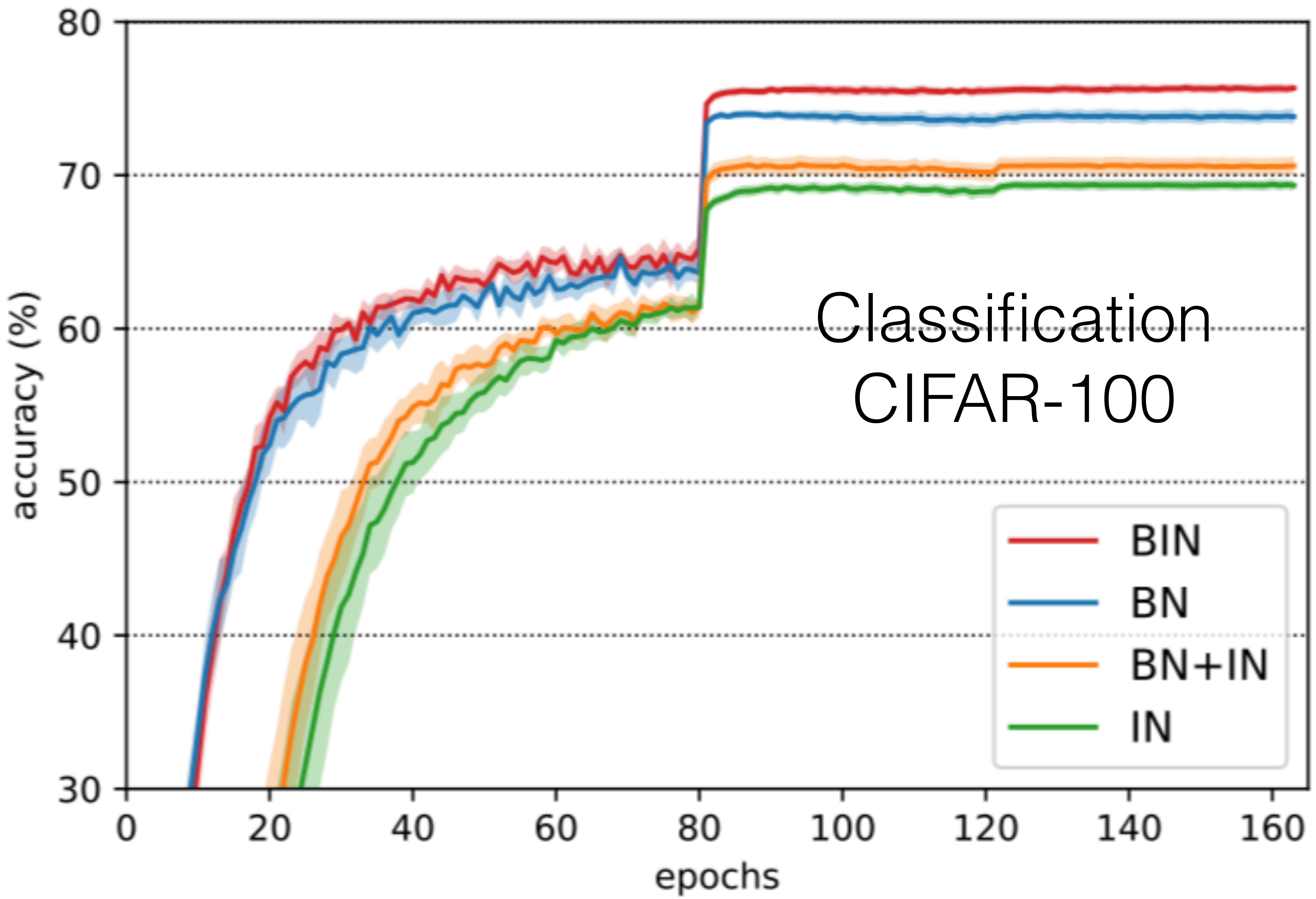
$\hat{x}^{(IN)}$

Batch-Instance normalization

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

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

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

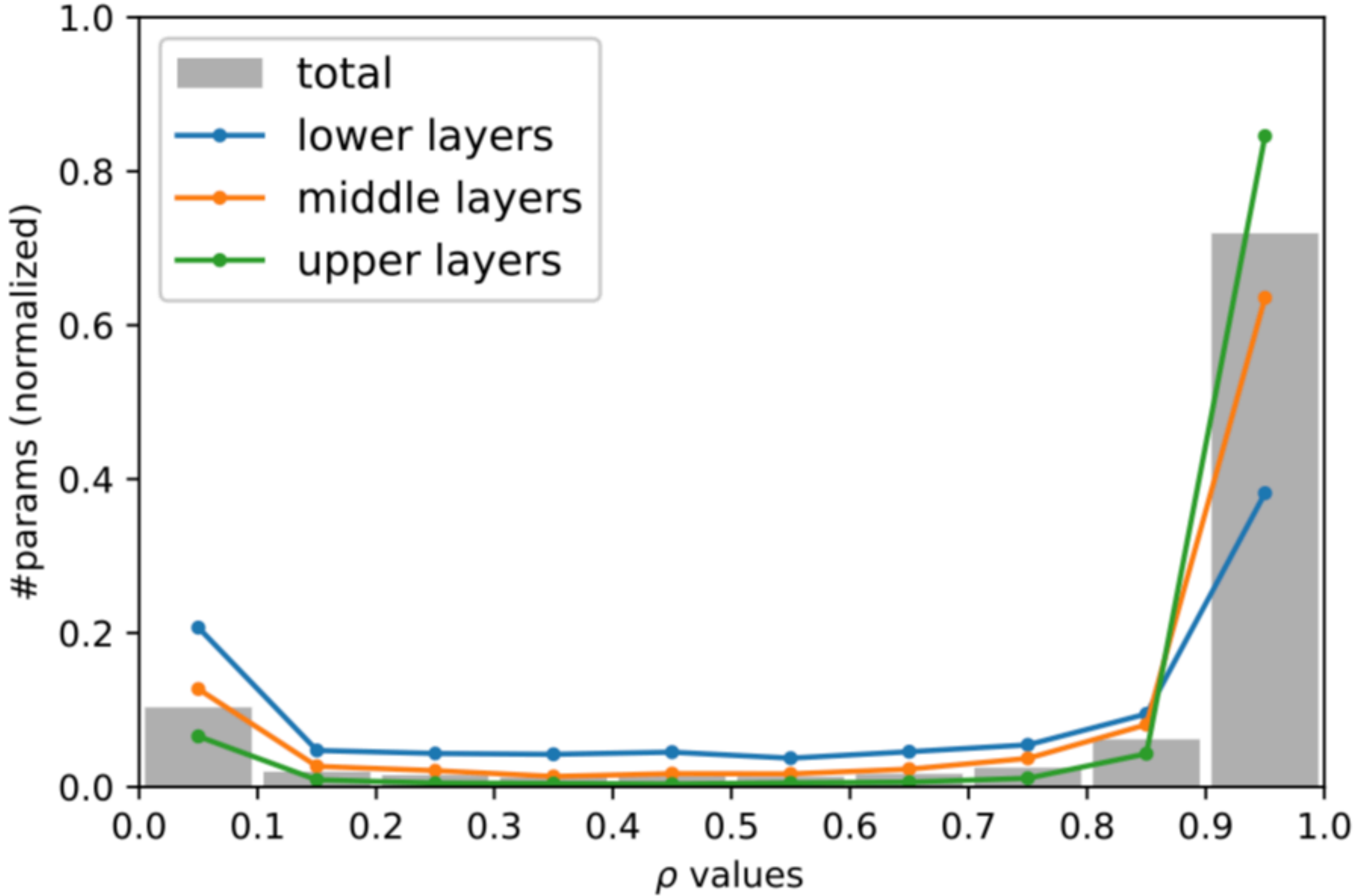


Batch-Instance normalization

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

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

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



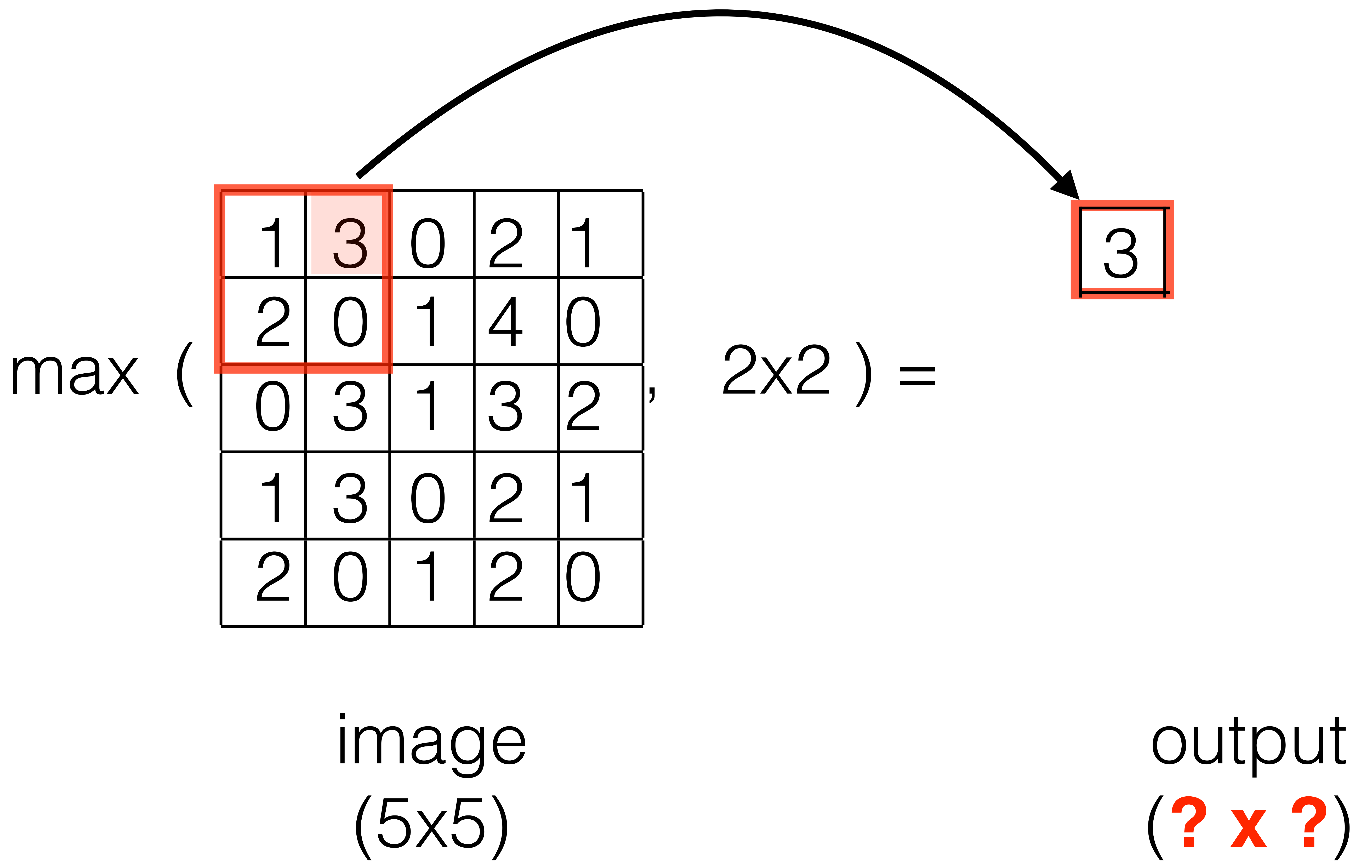
Normalization layers - Summary

- 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

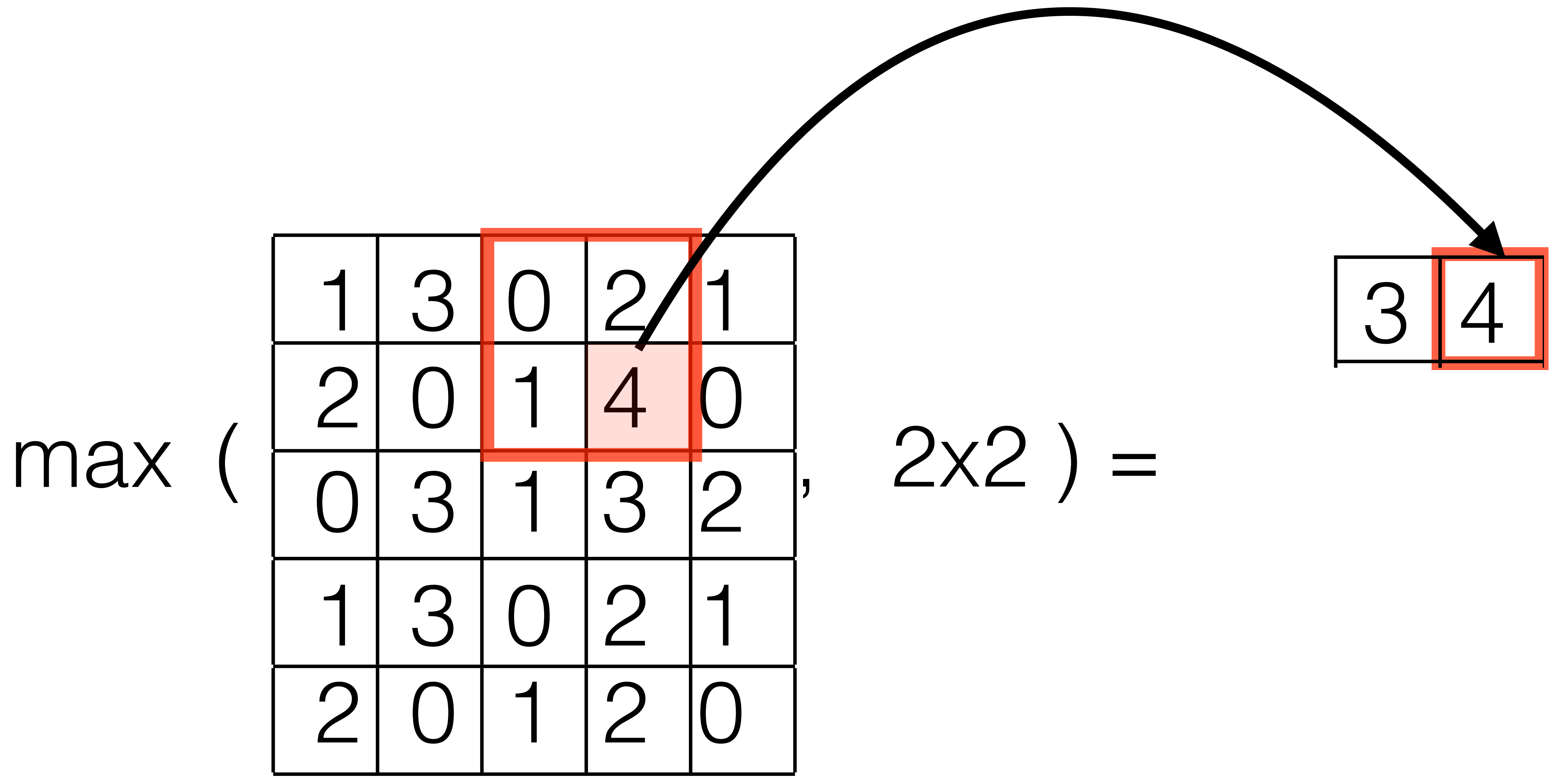
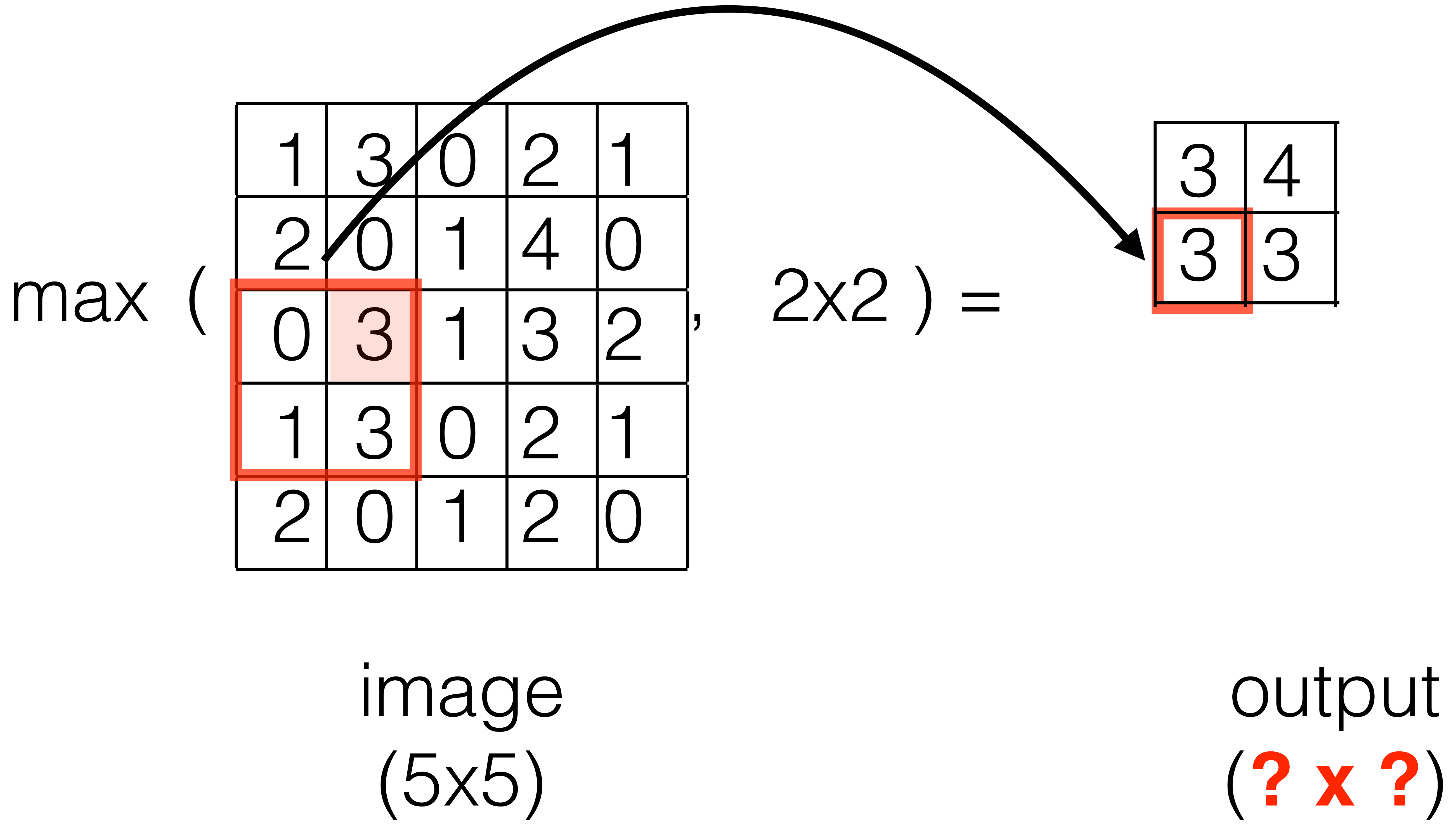


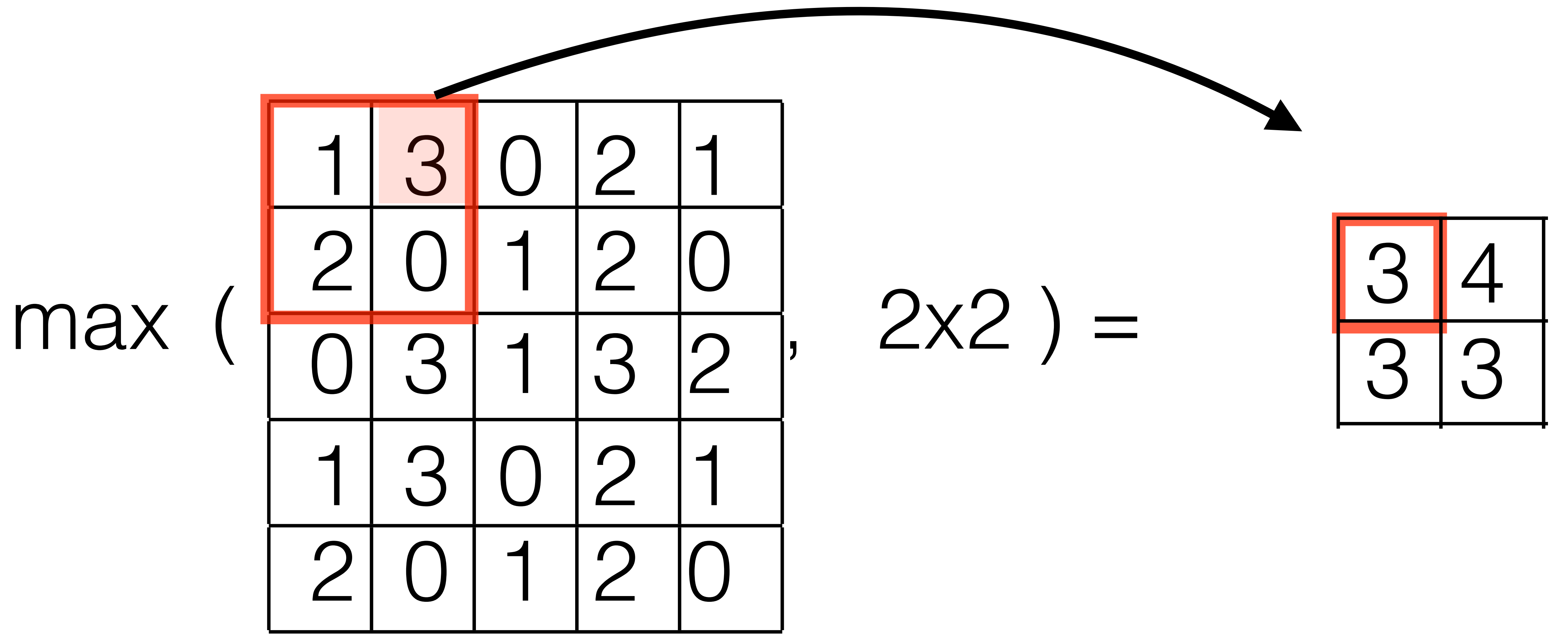
image
(5x5)

output
(**? x ?**)

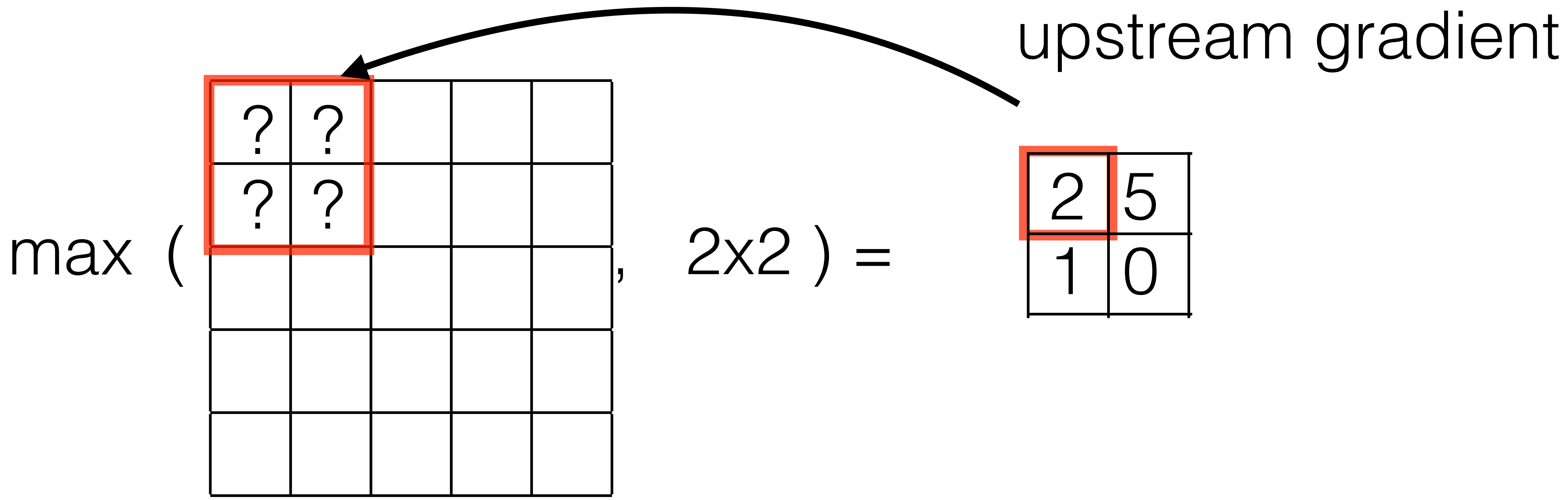
Max-pooling


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

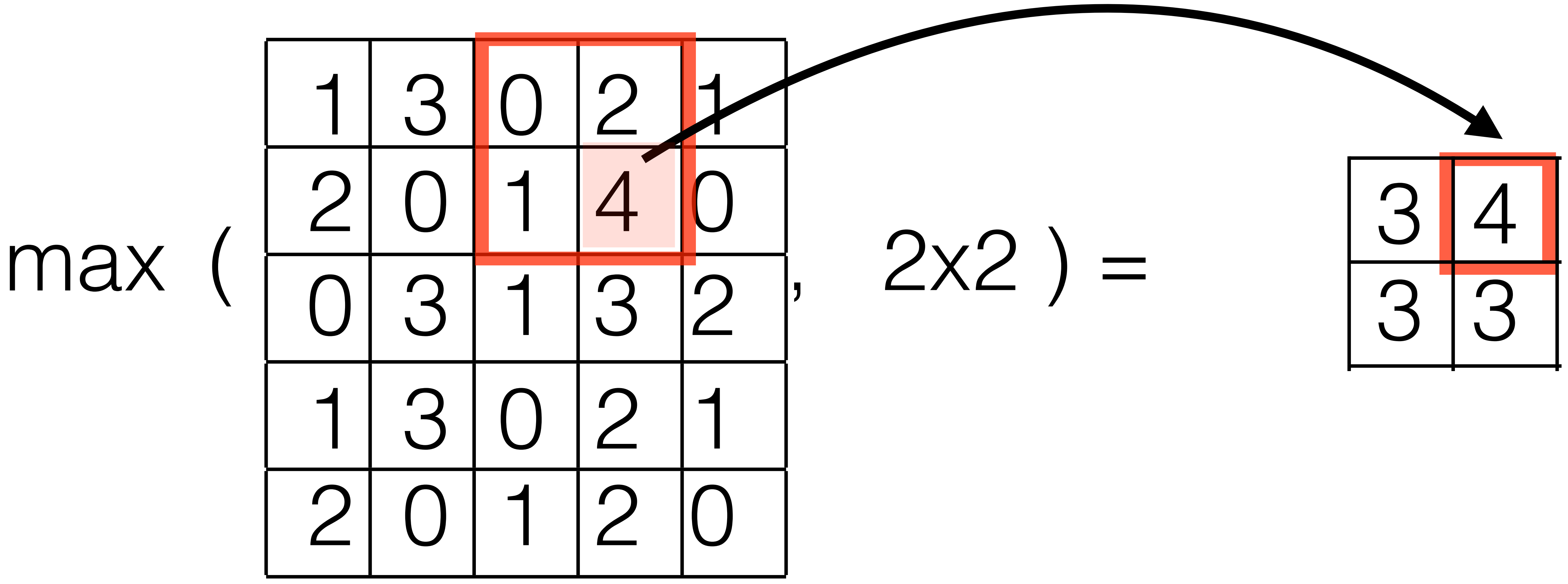
$$\max\left(\begin{bmatrix} 1 & 3 & 0 & 2 & 1 \\ 2 & 0 & 1 & 2 & 0 \\ 0 & 3 & 1 & 3 & 2 \\ 1 & 3 & 0 & 2 & 1 \\ 2 & 0 & 1 & 2 & 0 \end{bmatrix},\ 2\text{x}2\right) = \begin{bmatrix} 3 & 4 \\ 3 & 3 \end{bmatrix}$$

Max-pooling Backprop

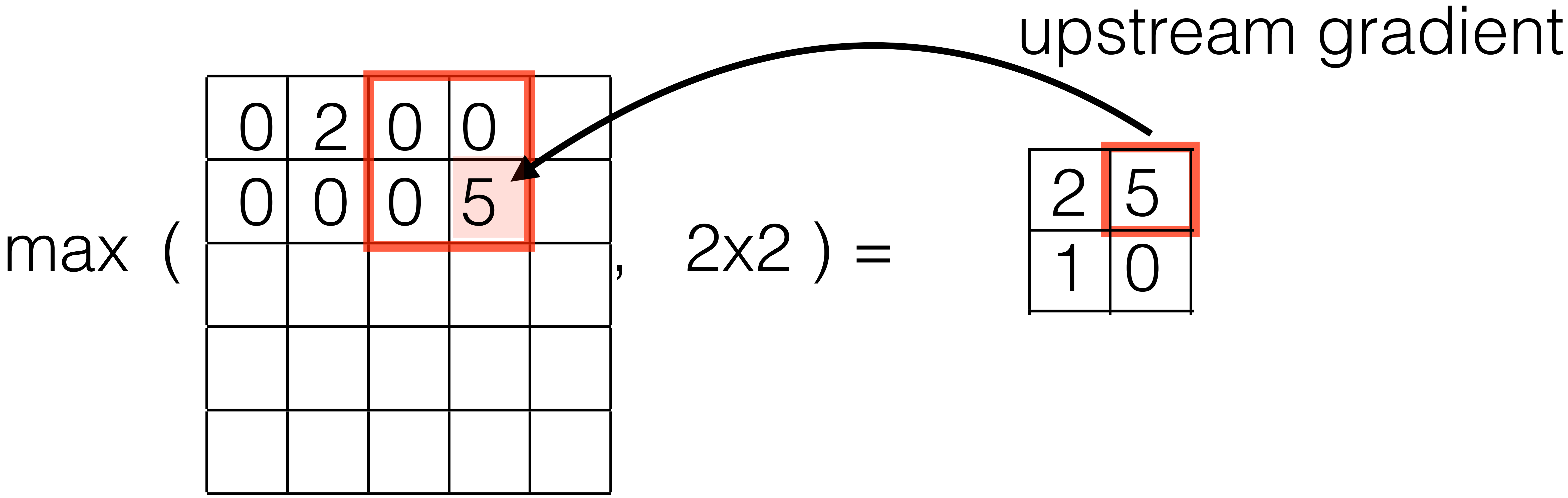
upstream gradient

$$\max\left(\begin{bmatrix} 0 & 2 & & & \\ 0 & 0 & & & \\ & & & & \\ & & & & \\ & & & & \end{bmatrix},\ 2\text{x}2\right) = \begin{bmatrix} 2 & 5 \\ 1 & 0 \end{bmatrix}$$

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 and spatial insensitivity
- You can live without it (larger conv stride and/or rate achieve similar behaviour)
- 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

PyTorch:

Regression:

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

`nn.MSELoss()`

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

`nn.L1Loss()`

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

`nn.SmoothL1Loss()`

Classification (cross entropy):

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

input logits: `torch.nn.NLLLoss`

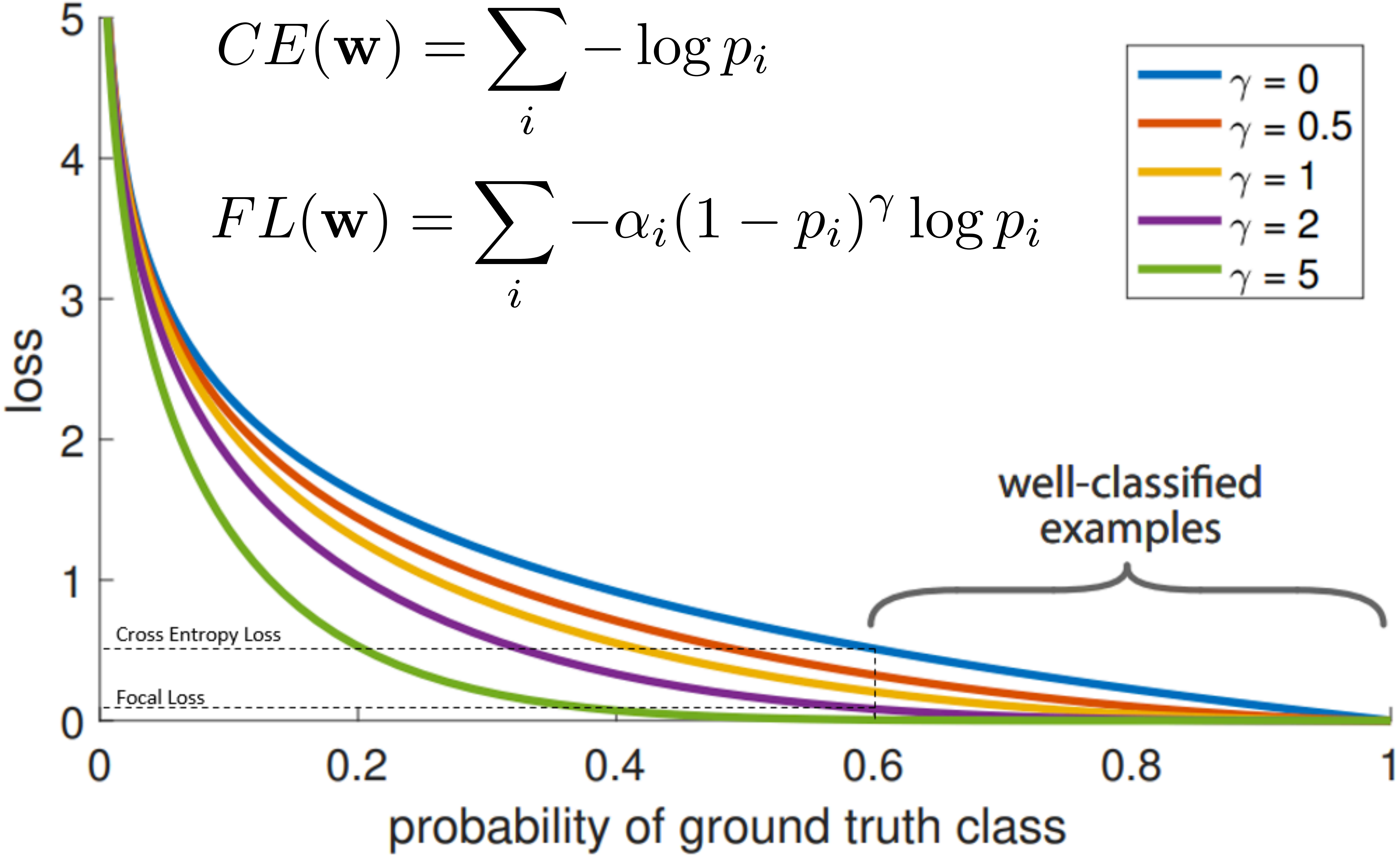
input probs: `torch.nn.CrossEntropyLoss`

Loss functions

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

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

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



Loss functions: Ranking loss

PyTorch: `torch.nn.MarginRankingLoss()`

Trn data triplets:

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

Interpretation:

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

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

Loss construction:

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

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