# Aprendizado Profundo (Deep Learning)

## Training Deep Networks

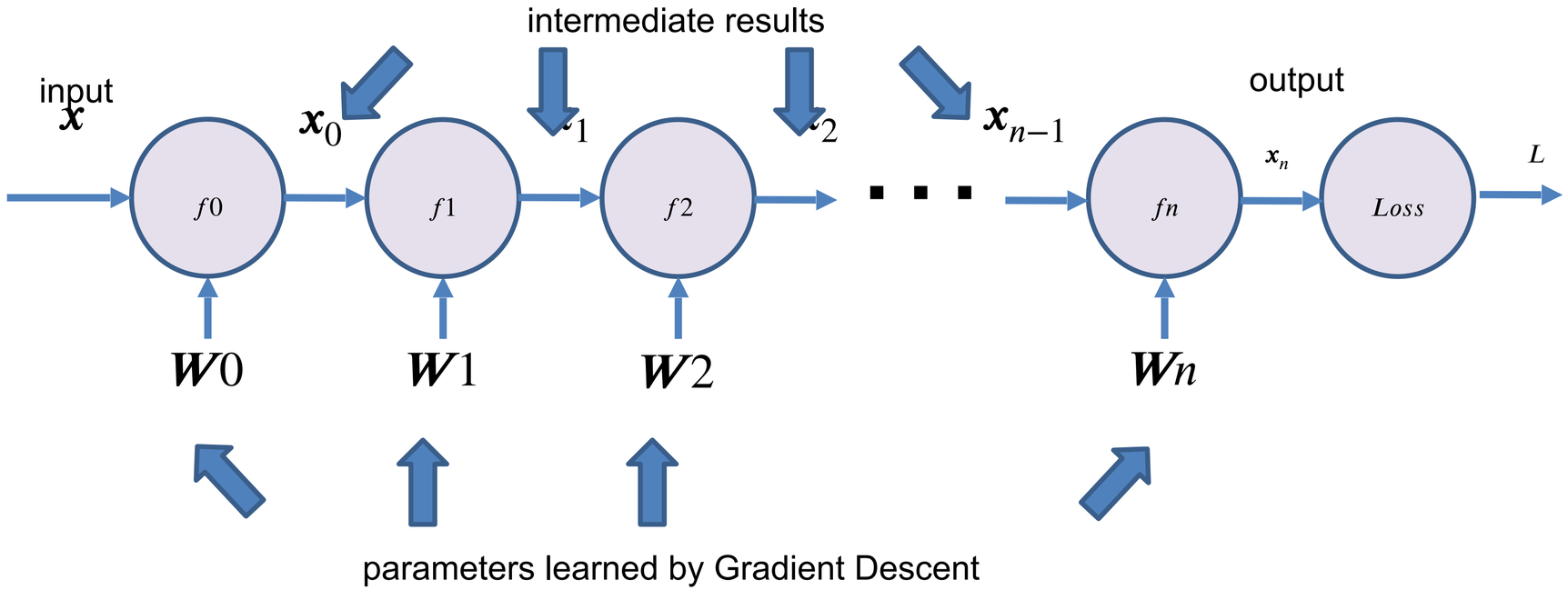
**Dario Oliveira (dario.oliveira@fgv.br)**

# Overview

1. Backpropagation
2. Activation Functions
3. Data Preprocessing
4. Weight Initialization
5. Batch Normalization

# Basic Neural Network (NN)

NN can implement very complex functions.



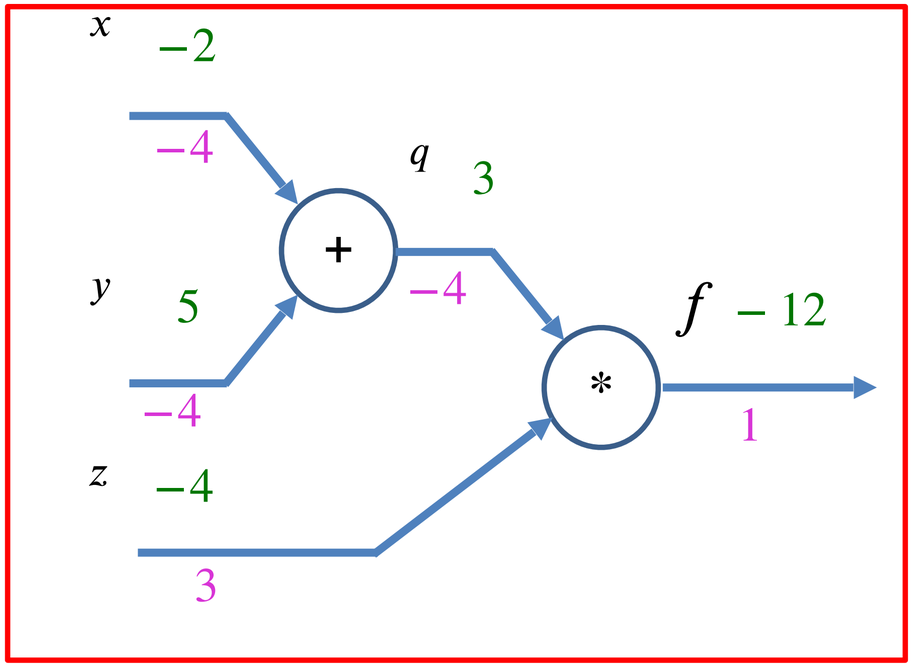

A NN can be very large → impractical to write down the gradient formula for all their parameters $\{W_i\}$.

# Computational Graphs

**Example 1:**

$f(x, y, z) = (x + y)z$

e.g., $x = -2, y = 5, z = -4$




$$q = (x + y) \rightarrow \frac{\partial q}{\partial x} = \frac{\partial q}{\partial y} = 1$$

$$f = qz \rightarrow \frac{\partial f}{\partial q} = z\,, \quad \frac{\partial f}{\partial z} = q$$

$\frac{\partial f}{\partial q} = z = -4$

$\frac{\partial f}{\partial z} = q = 3$

Compute:

$\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$ and $\frac{\partial f}{\partial z}$

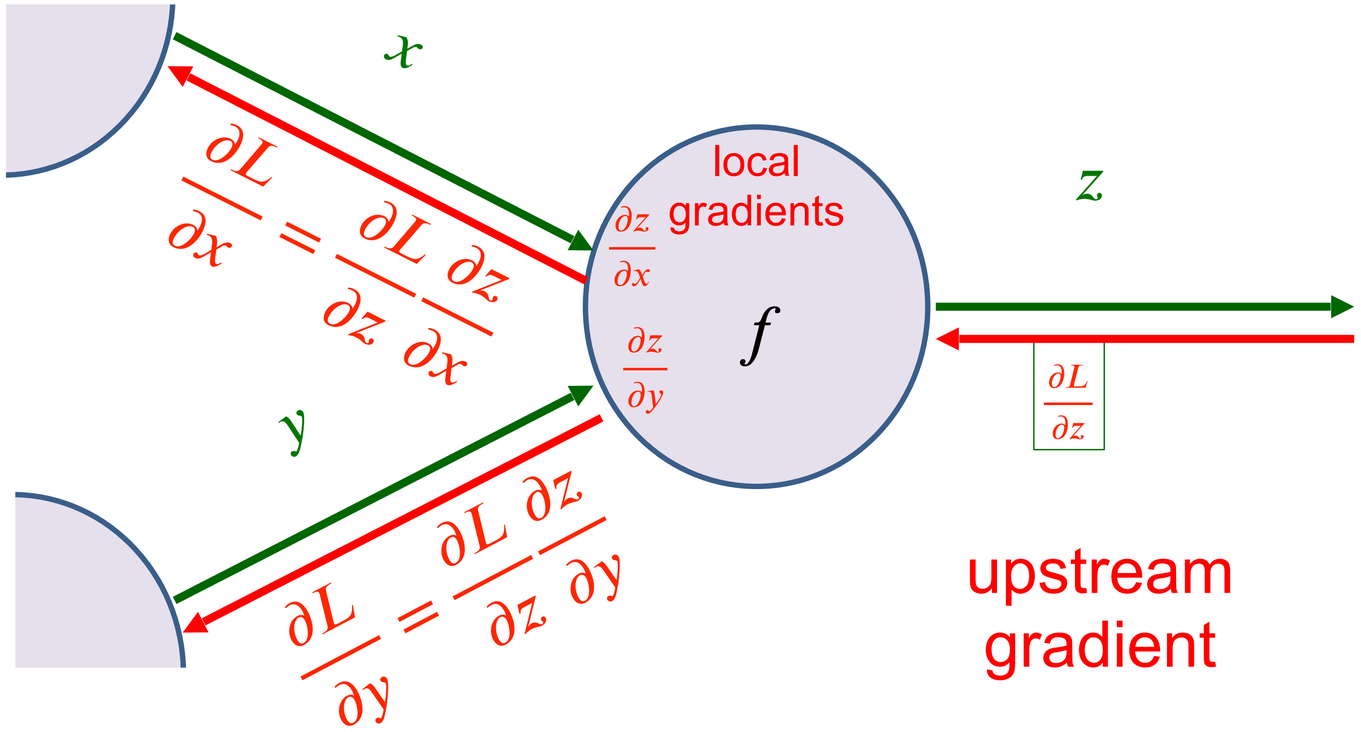
Now using the chain rule:

$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = z = -4$

$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = z = -4$

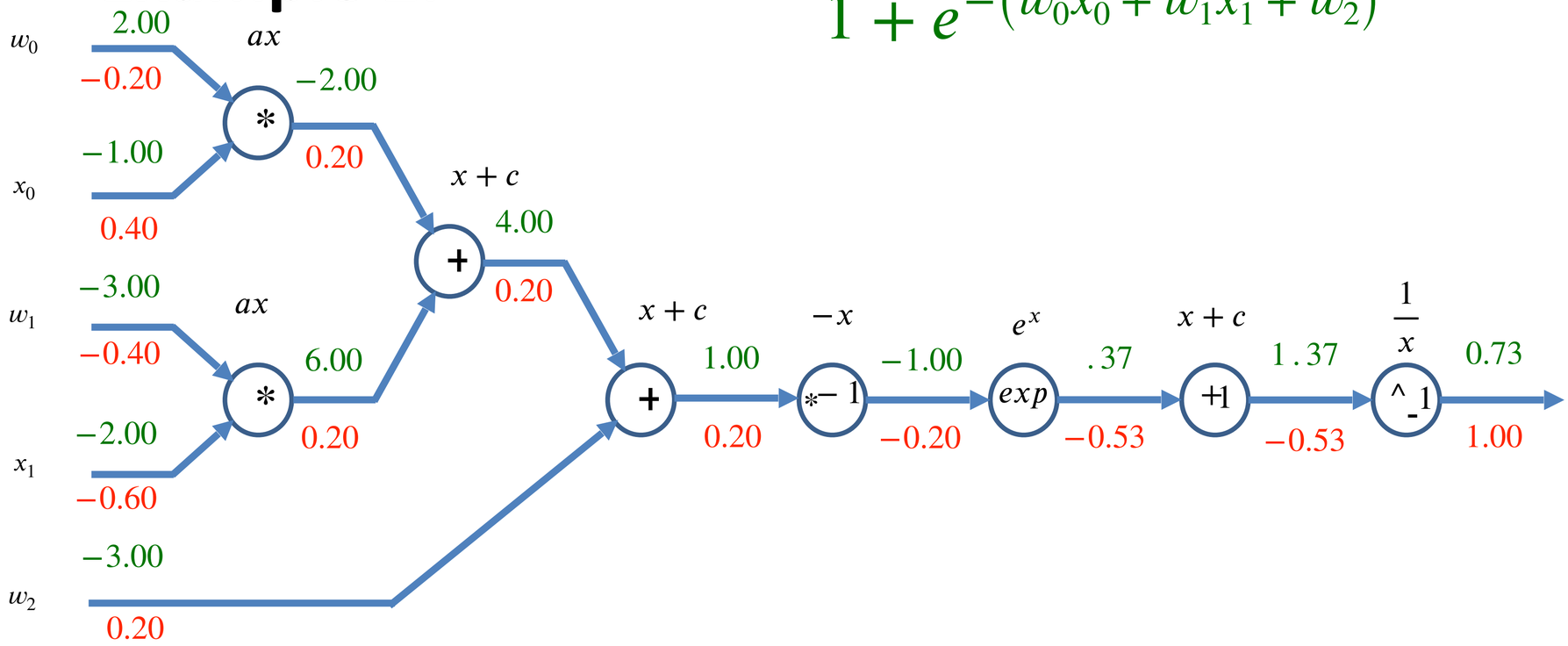
# Gradient is backpropagated

**Rule**: *local gradient* × *upstream gradient*



# Computational Graphs

**Example 2:** $\rightarrow f(w,x) = \dfrac{1}{1+e^{-(w_0x_0 + w_1x_1 + w_2)}}$




$*$ $\quad f(x) = ax \quad \rightarrow \dfrac{\partial f}{\partial x} = a$

$+$ $\quad f(x) = x + c \quad \rightarrow \dfrac{\partial f}{\partial x} = 1$

$exp$ $\quad f(x) = e^x \quad \rightarrow \dfrac{\partial f}{\partial x} = e^x$

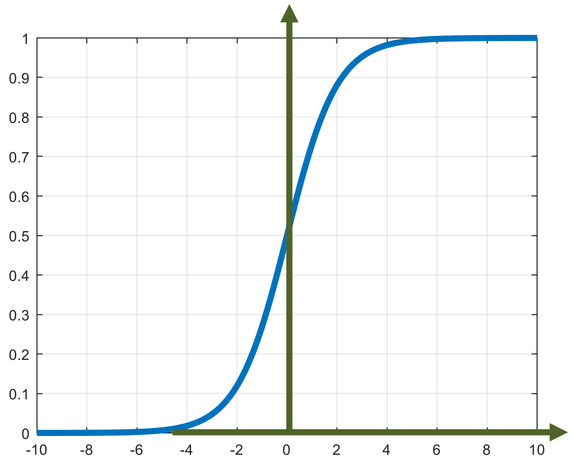
$^{-1}$ $\quad f(x) = \dfrac{1}{x} \rightarrow \dfrac{\partial f}{\partial x} = \dfrac{-1}{x^2}$

# Neuron complexity in CGs

$$f(w,x) = \frac{1}{1 + e^{-(w_2x_2 + w_1x_1 + w_0)}}$$

sigmoid function

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

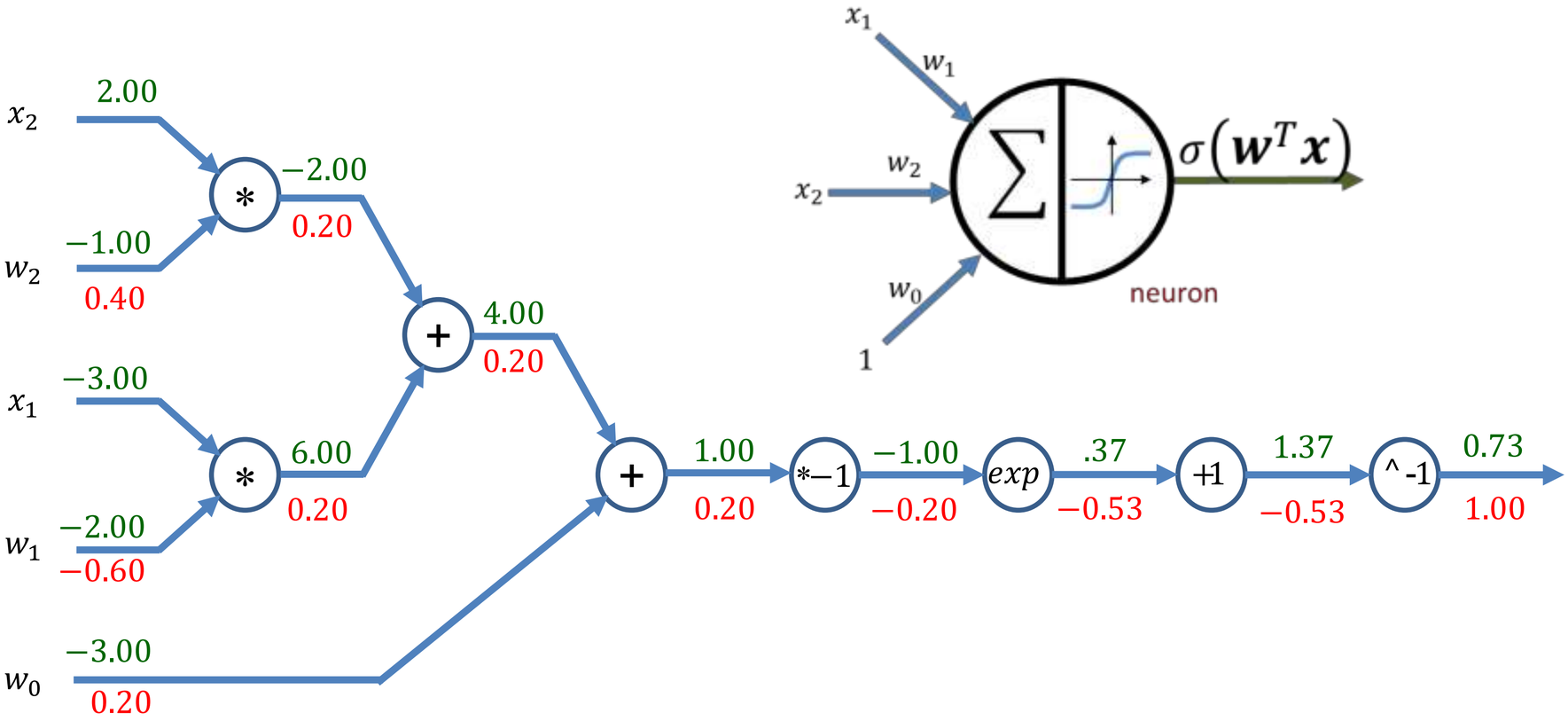


# Neuron complexity in CGs

$$f(w,x) = \frac{1}{1 + e^{-(w_2x_2 + w_1x_1 + w_0)}}$$

sigmoid function

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



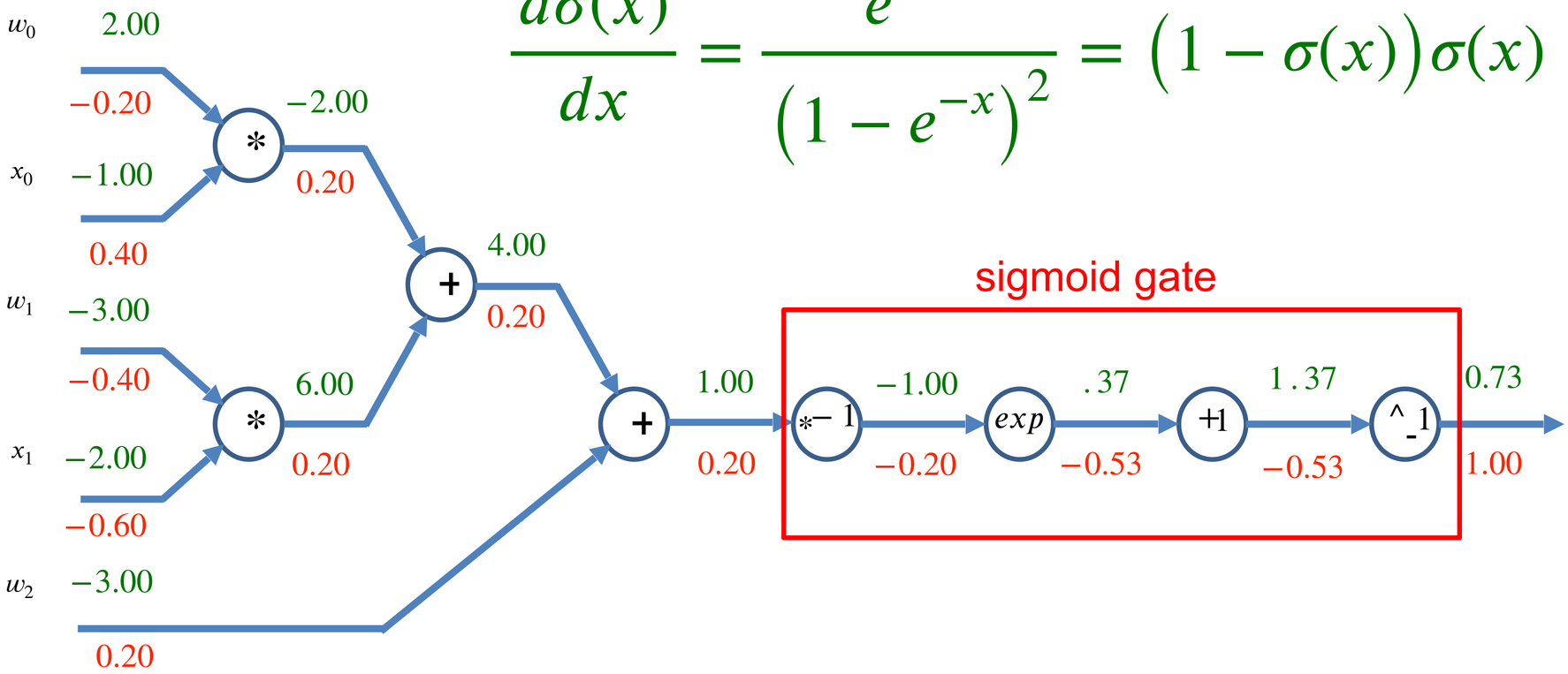
# Node complexity in CGs

$$f(w,x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$

sigmoid function

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

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{\left(1 - e^{-x}\right)^2} = \left(1 - \sigma(x)\right)\sigma(x)$$

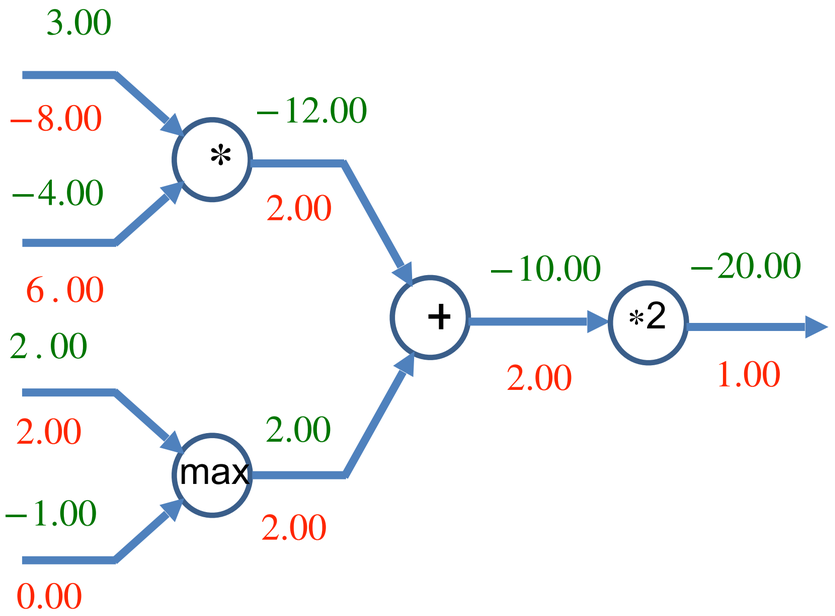


# Patterns in backward flow

**add** gate: gradient distributor

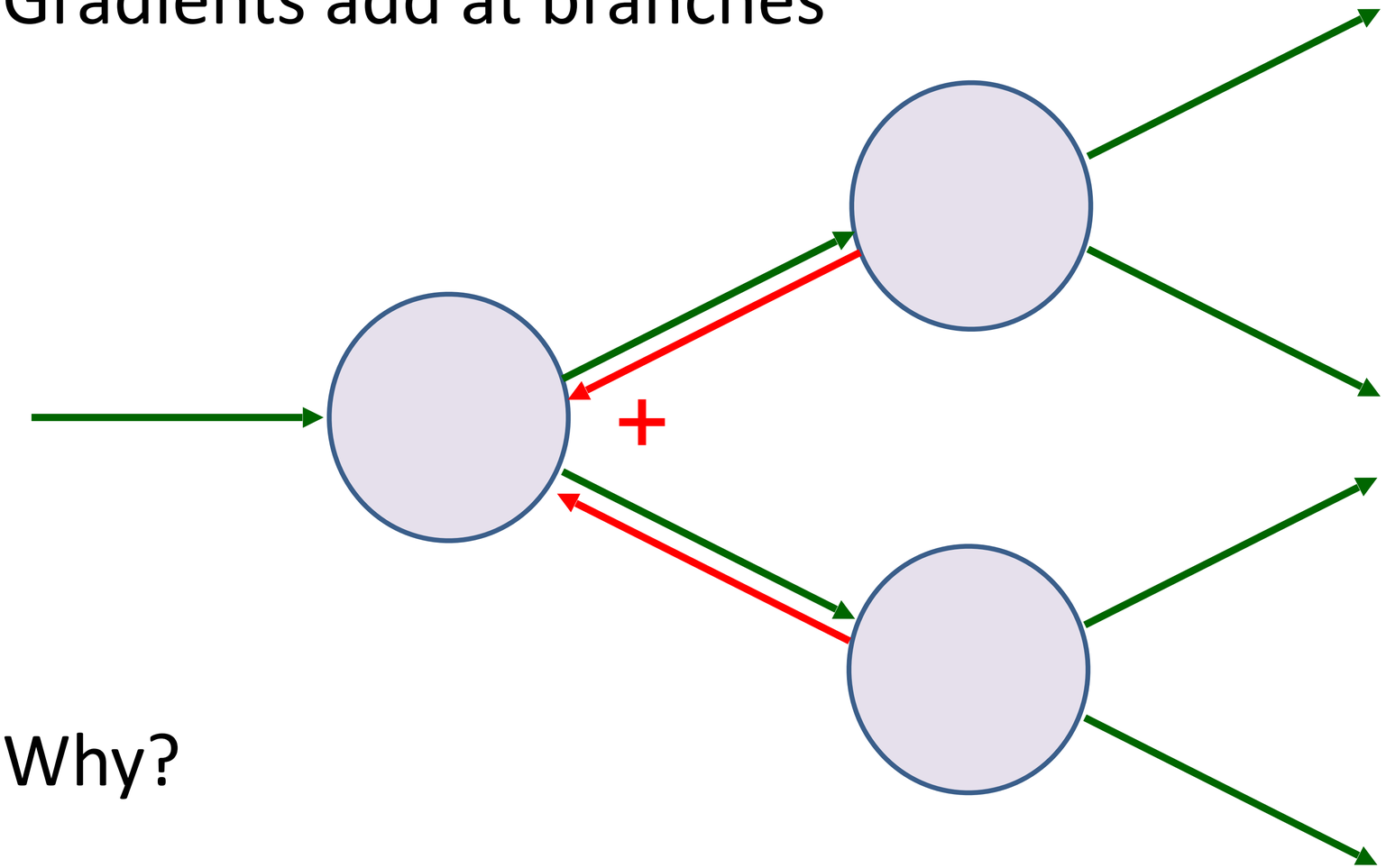
**max** gate: gradient router

**mul** gate: gradient switcher



# Patterns in backward flow

Gradients add at branches




Why?

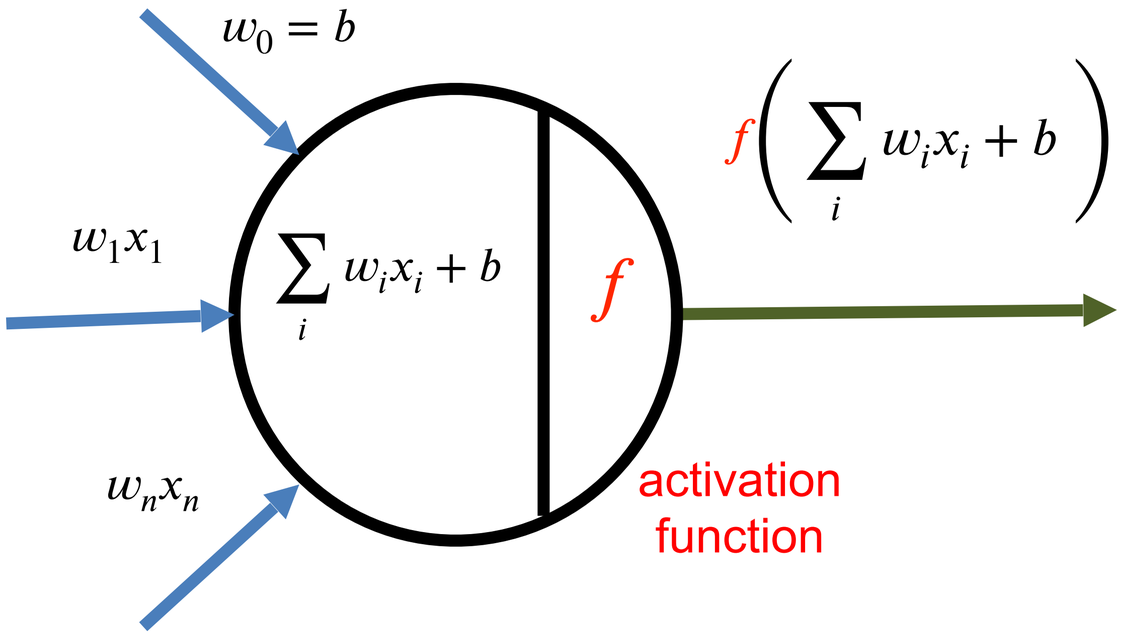
# Summary so far…

- Neural nets will be very large: impractical to write down gradient formula by hand for all parameters.
- **Backpropagation**: recursive application of the chain rule along a computational graph to compute the gradients of all inputs/parameters/intermediates.
- Code follows the graph structure, where the nodes implement the forward/ backward passes.
  - **Forward**: compute result of an operation and save any intermediates needed for gradient computation in memory.
  - **Backward**: apply the chain rule to compute the gradient of the loss function with respect to the inputs.
- Modular implementation (local × upstream gradient).

# Overview

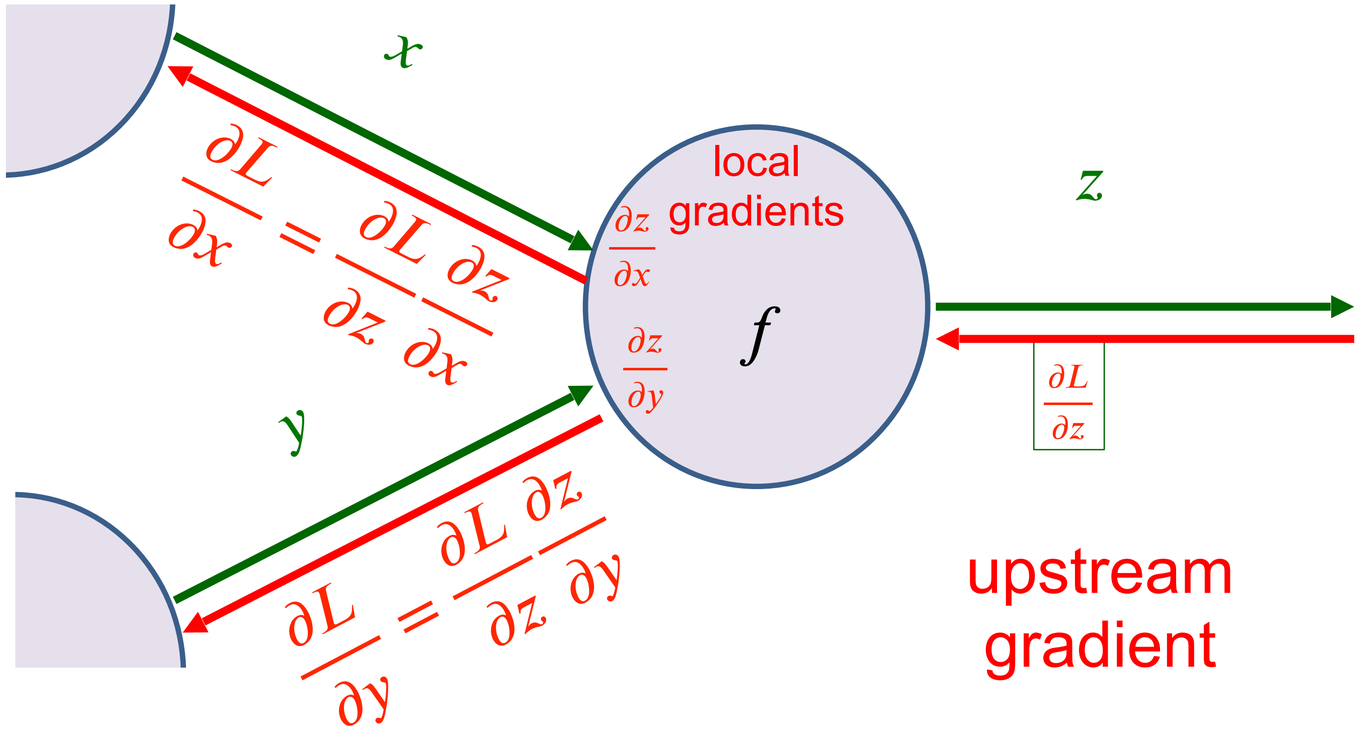
1. Backpropagation
2. Activation Functions
3. Data Preprocessing
4. Weight Initialization
5. Batch Normalization

# Activation Layer

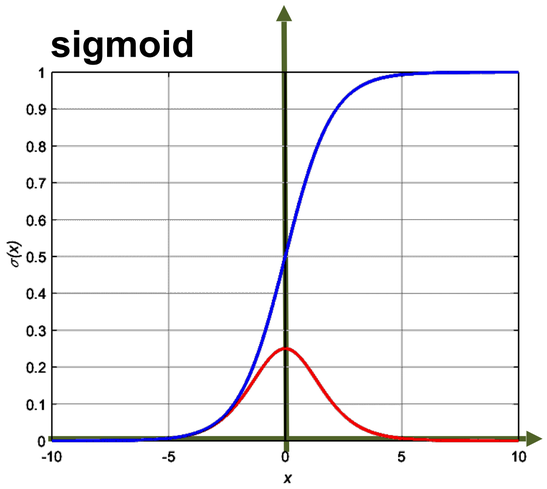


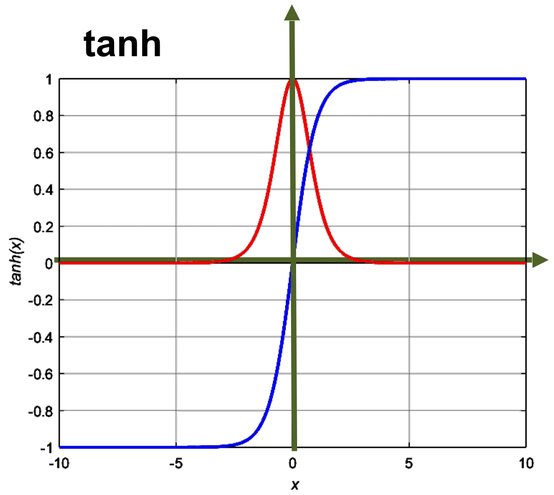
# Gradient is backpropagated

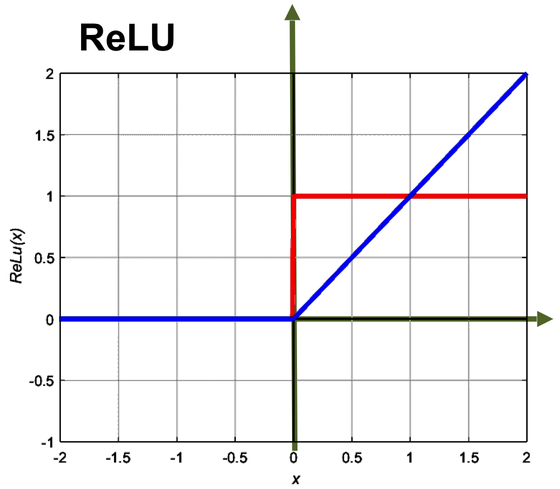
**Rule**: *local gradient* × *upstream gradient*

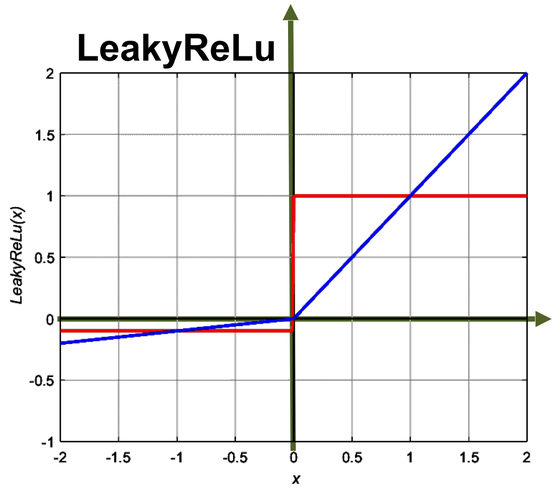


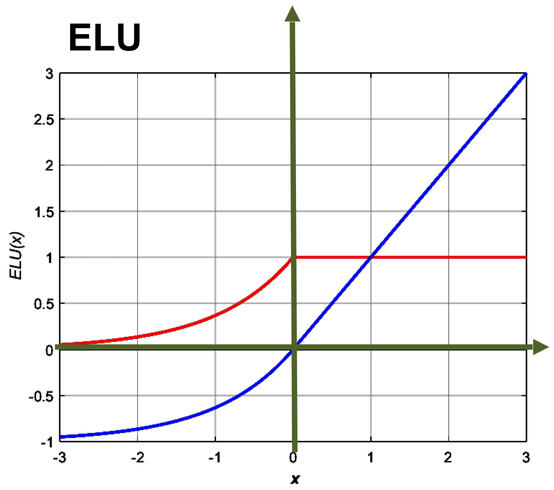
# Activation Functions



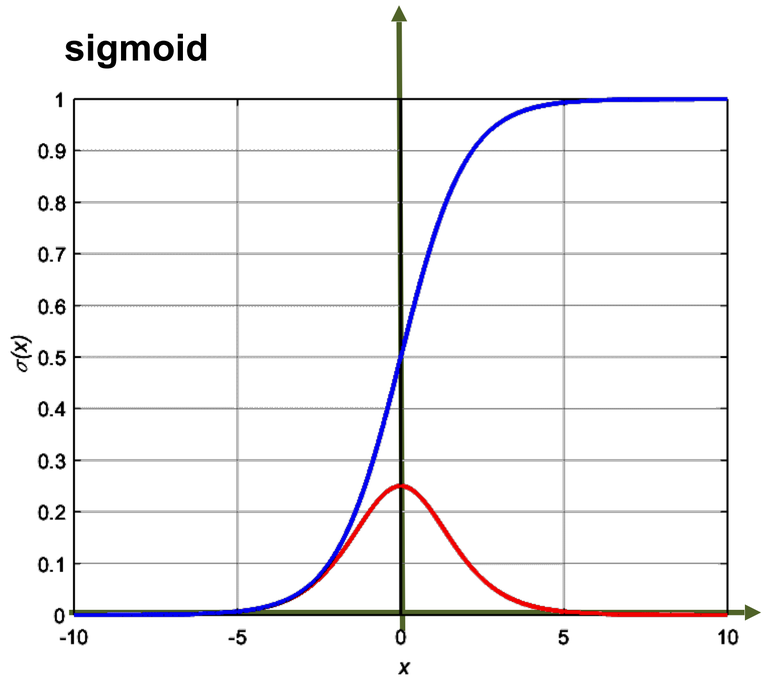



# Sigmoid

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

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

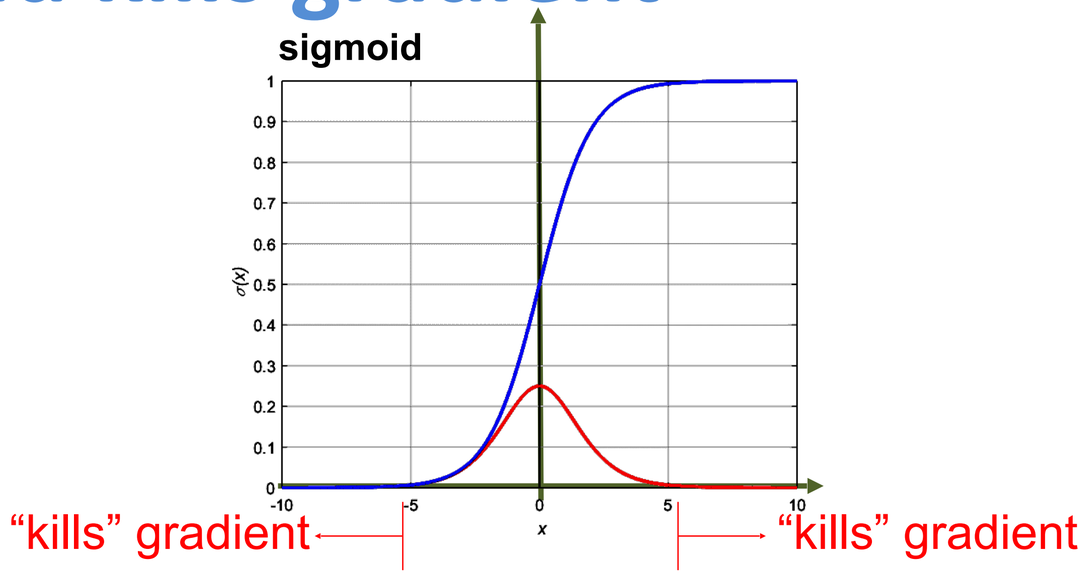


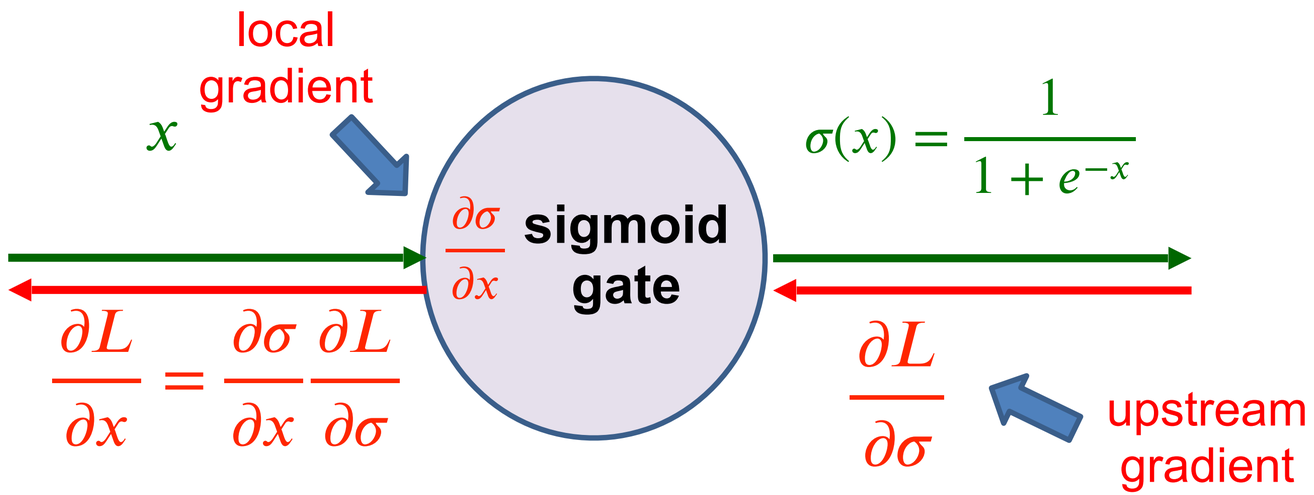

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron.

Three shortcomings:

1) Saturated neurons "kill" the gradients,
2) Sigmoid outputs are not zero-centered,
3) $exp()$ is a bit computation expensive.

# Sigmoid kills gradient

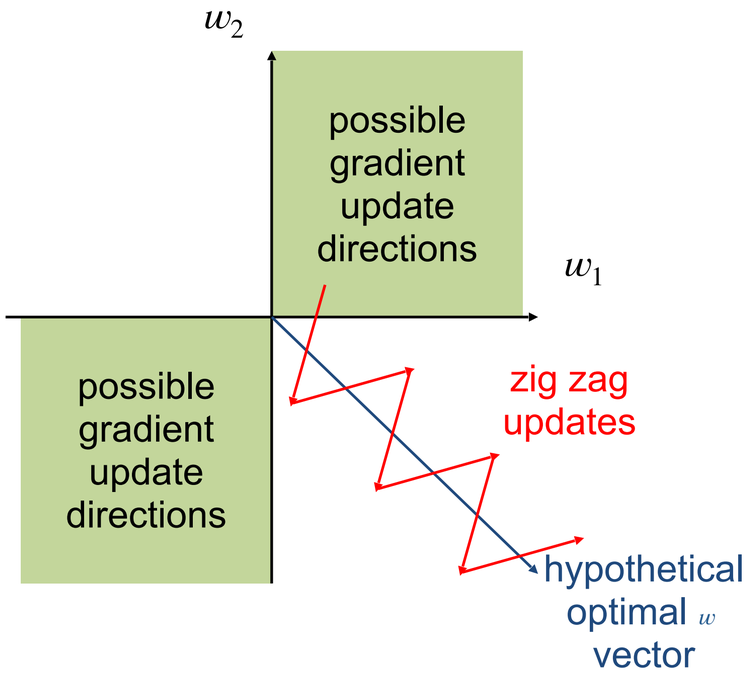


# Sigmoid output not zero-centered

What happens when the input ($x$) to a neuron is always positive?

$$f\left(\sum_i w_i x_i + b\right)$$



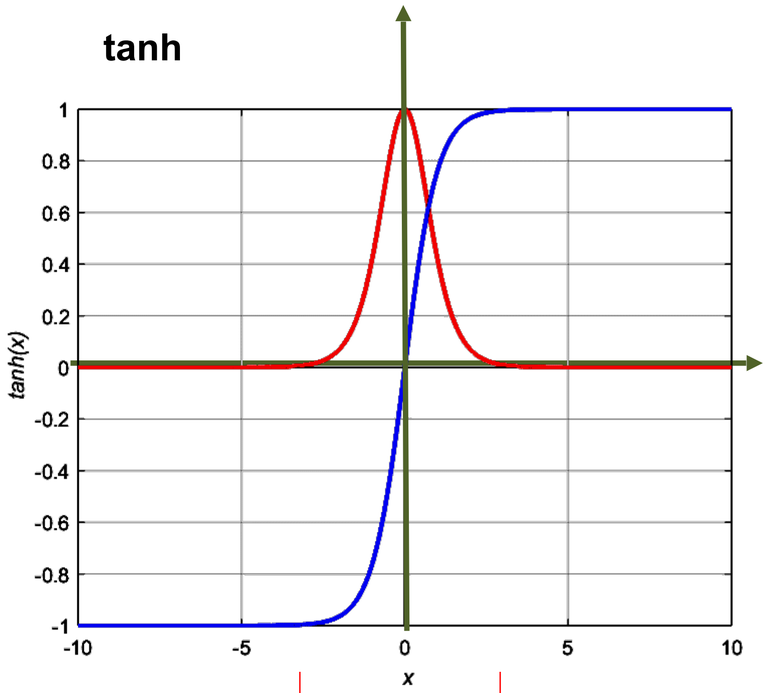

The gradients on $w$ are all positive (or all negative). 

Therefore, zero-mean data is highly desirable!

# Tanh

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

$$tanh'(x) = 1 - tanh(x)^2$$




"kills" gradient

"kills" gradient

- Squashes numbers to range [-1,1]
- Zero-centered.

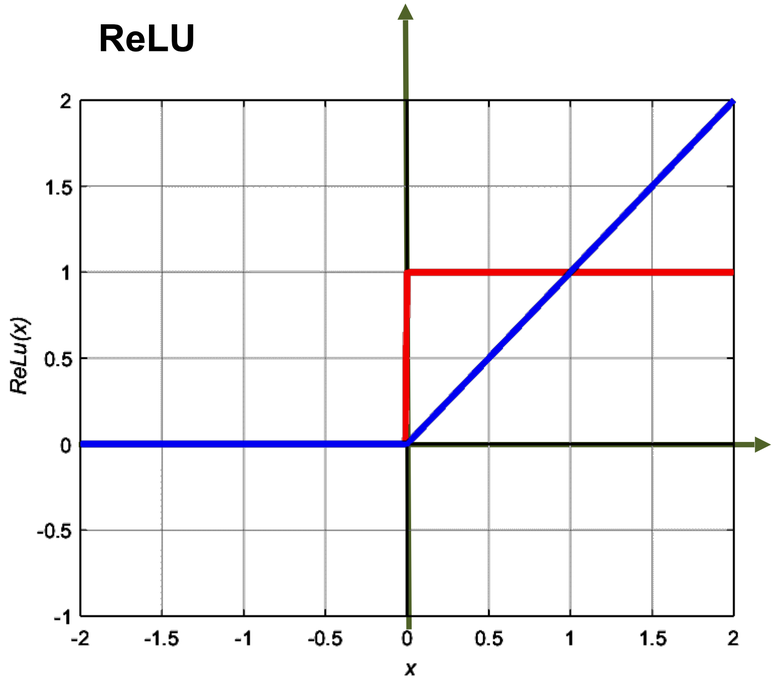
Shortcoming:

Still "kills" the gradients when saturated

# Rectified Linear Unit

$$ReLU(x) = max(0,x)$$

$$ReLU'(x) = \begin{cases} 0 \text{ for } & x < 0 \\ 1 \text{ for } & x \geq 0 \end{cases}$$

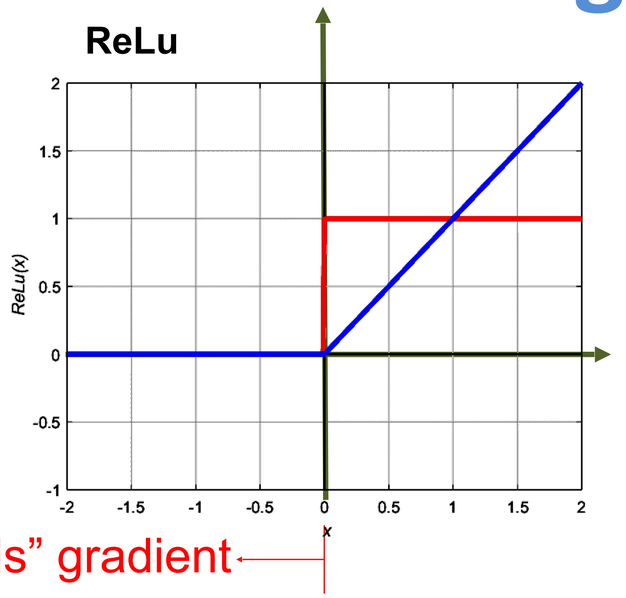


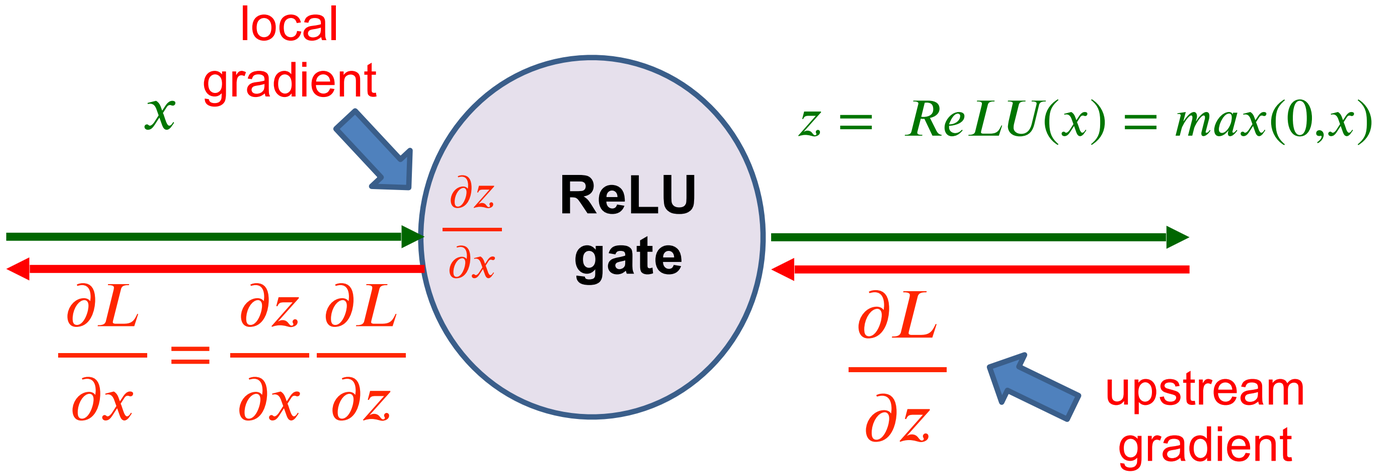

- Does not saturate in the positive region.
- Very computationally efficient
- Converges much faster than sigmoid/ tanh (≅ 6×)

Shortcoming:

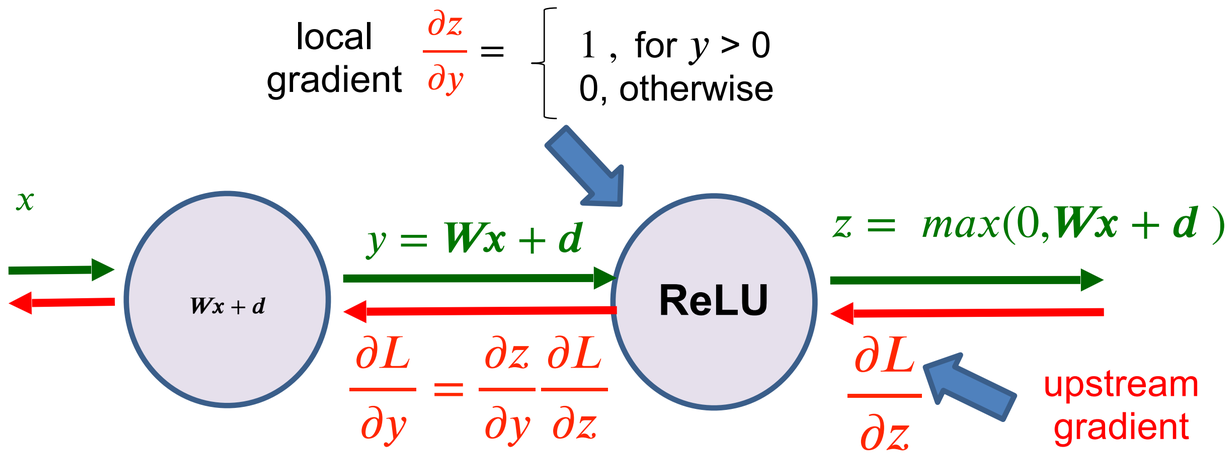
1) not zero-centered output
2) Kills gradient for negative input

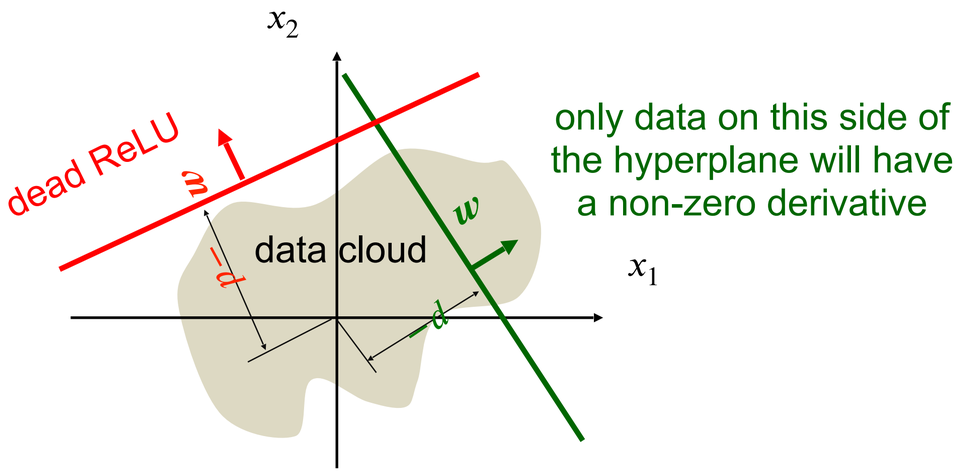
# ReLU kills gradient for negative input



# Dead ReLU

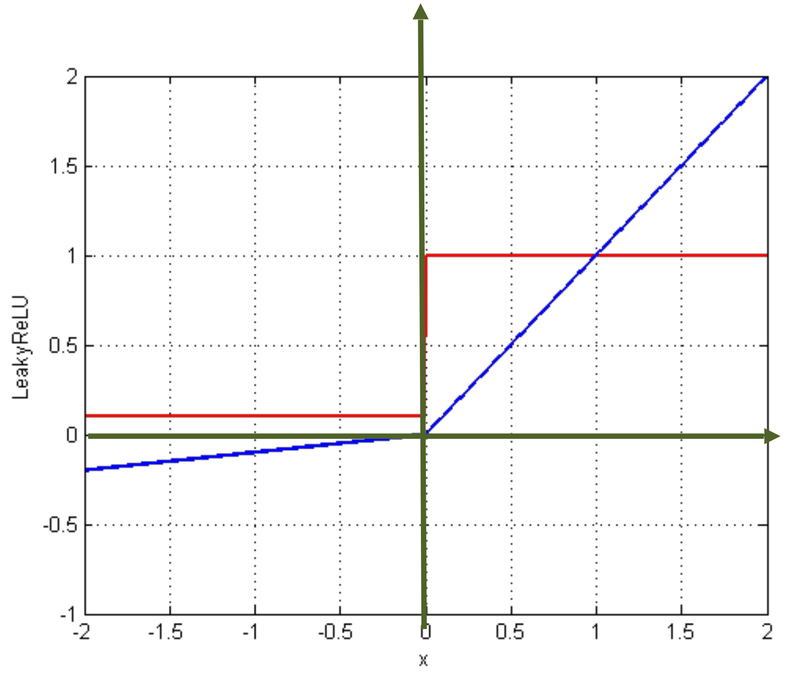

local gradient $\frac{\partial z}{\partial y} = \begin{cases} 1, \text{ for } y > 0 \\ 0, \text{ otherwise} \end{cases}$
$x$
$Wx + d$
$y = Wx + d$
ReLU
$z = max(0, Wx + d)$
$\frac{\partial L}{\partial y} = \frac{\partial z}{\partial y}\frac{\partial L}{\partial z}$
$\frac{\partial L}{\partial z}$
upstream gradient


$x_2$
dead ReLU
$w$
data cloud
$-d$
$w$
$-d$
$x_1$
only data on this side of the hyperplane will have a non-zero derivative

# LeakyReLU

$$LeakyReLU(x) = max(0.01x, x)$$

$$LeakyReLU'(x) = \begin{cases} 0.01 \text{ for } & x < 0 \\ 1 \text{ for } & x \geq 0 \end{cases}$$




- Never saturates.
- Computationally efficient
- Converges much faster than sigmoid/ tanh (≅ 6×)
- Never “dies”.
- Pushes the mean of activations closer to zero

**Parametric Rectifier (PReLU):**

$$f(x) = max(\alpha x, x)$$

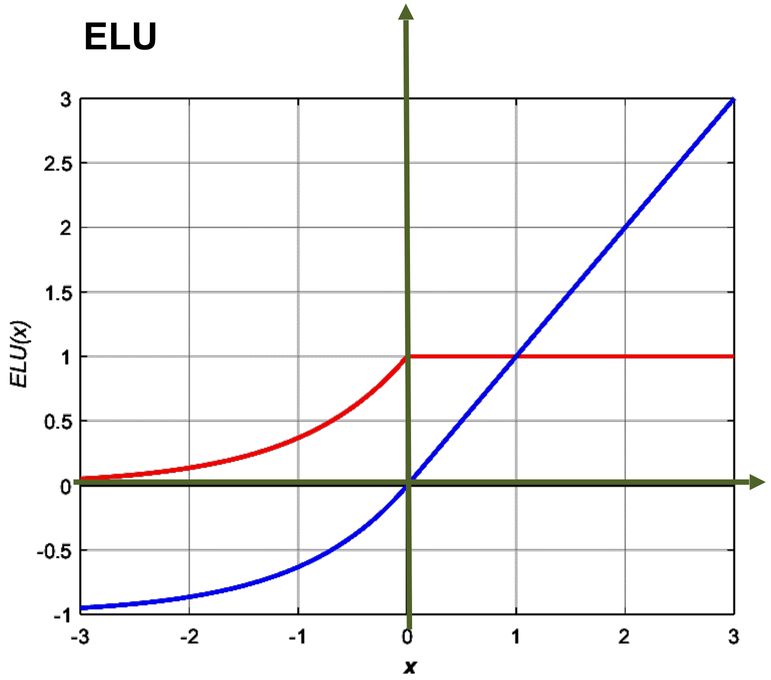
Shortcoming:

not zero-centered output

# ELU

$$ELU(x) = \begin{cases} \alpha\left(e^{x} - 1\right) & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$$

$$ELU'(x) = \begin{cases} ELU(x) + \alpha & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$$



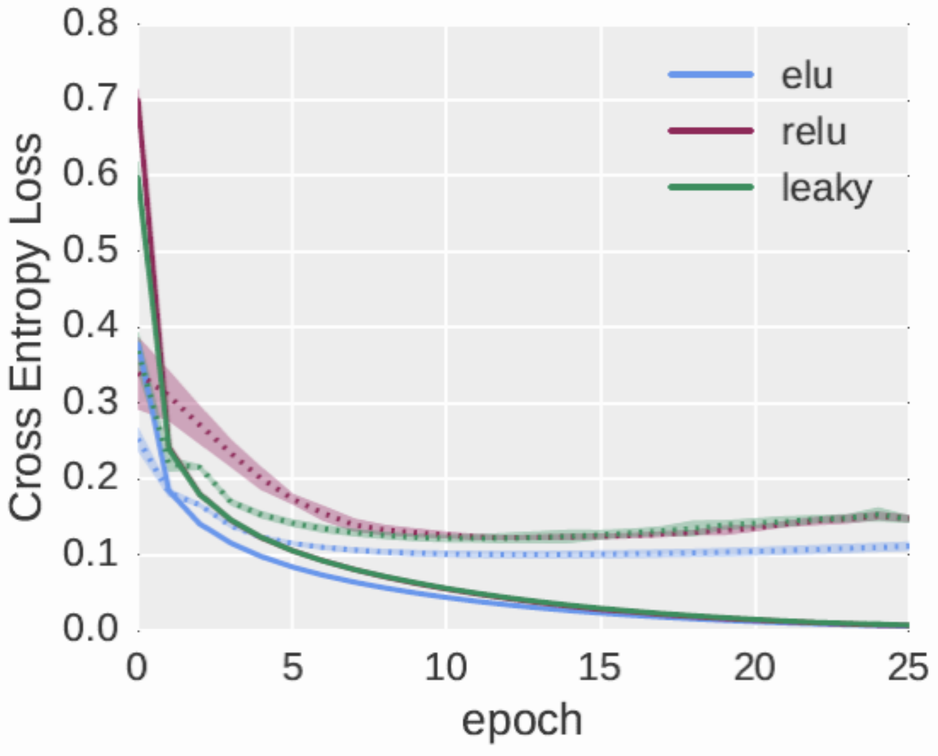

- All pros of LeakyReLU.
- but saturates at smaller values .
- Therefore, ELU is more robust to noise.

Shortcoming:

not zero-centered output

Clevert et al., 2016, available at https://arxiv.org/pdf/1511.07289.pdf

# Comparing activation functions



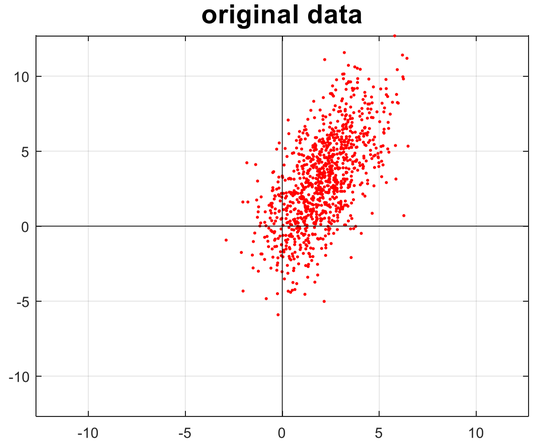

Evaluation at MNIST: training set (straight line) and validation set (dotted line)

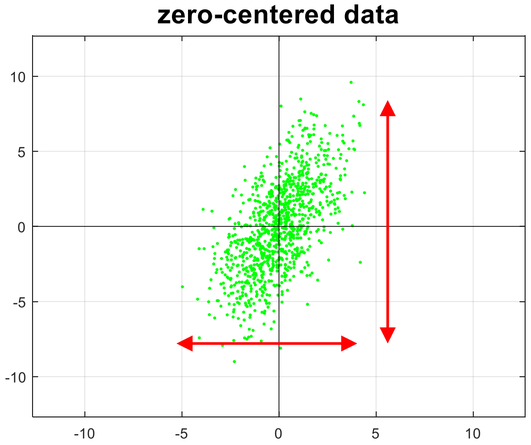
Figures from Clevert et al., 2016, available at https://arxiv.org/pdf/1511.07289.pdf

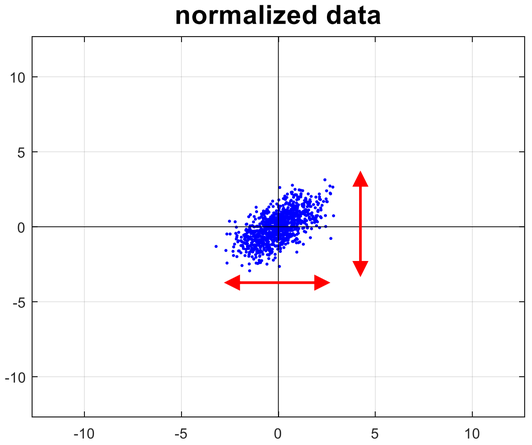
# Overview

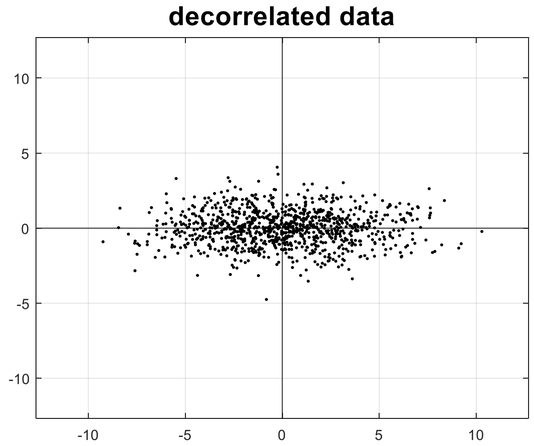
1. Backpropagation
2. Activation Functions
3. Data Preprocessing
4. Weight Initialization
5. Batch Normalization

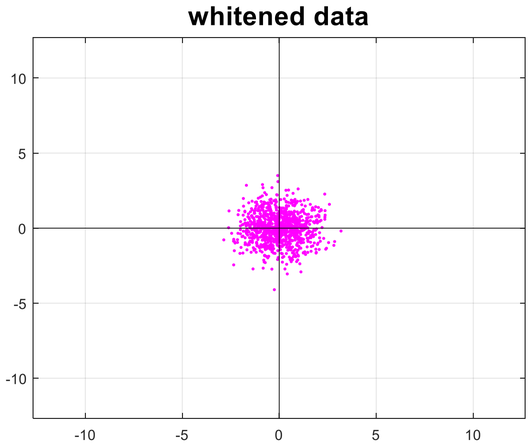
# Preprocessing Operations




**Usual practices (for images):**

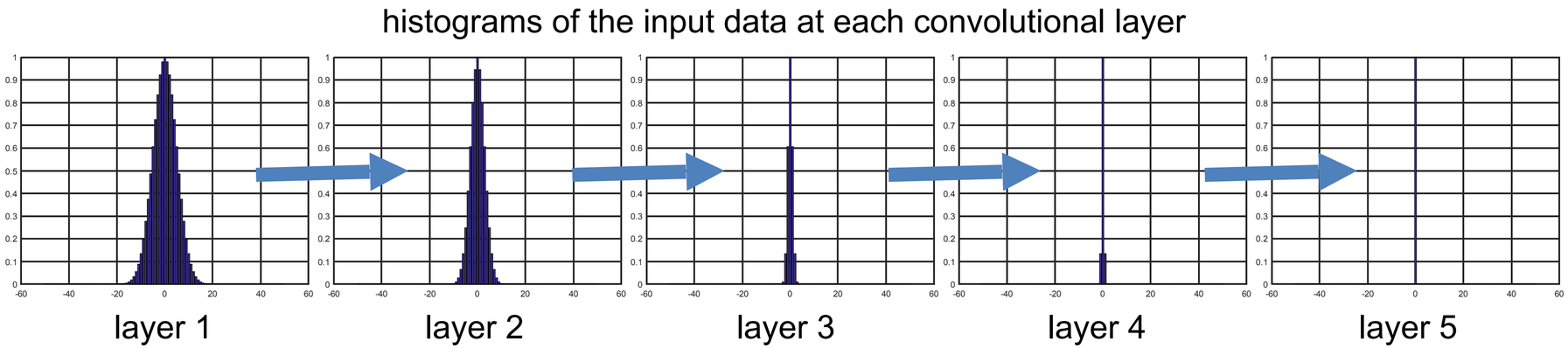
- Zero center.
- Subtract the mean image (e.g. AlexNet).
- Subtract per-channel mean (e.g. VGGNet).
- Normalize from -1 to 1.

# Overview

1. Backpropagation
2. Activation Functions
3. Data Preprocessing
4. Weight Initialization
5. Batch Normalization

# Small initial Weights

As the data flows forward, the input ($x$) at each layer tends to concentrate around zero.




Recall that the local gradient $Wx$

$$\frac{\partial Wx}{\partial W} = x$$

As the gradient back props, it tends to vanish → no update.

# Large initial Weights

Assume a *tanh* or a *sigmoid* activation function.

The input to the activation functions are in the saturated region, where the gradient is small.

As the gradient propagates back, they tend to vanish → no update.

# Optimal Weight Initialization

## Still an active research topic:

recommended

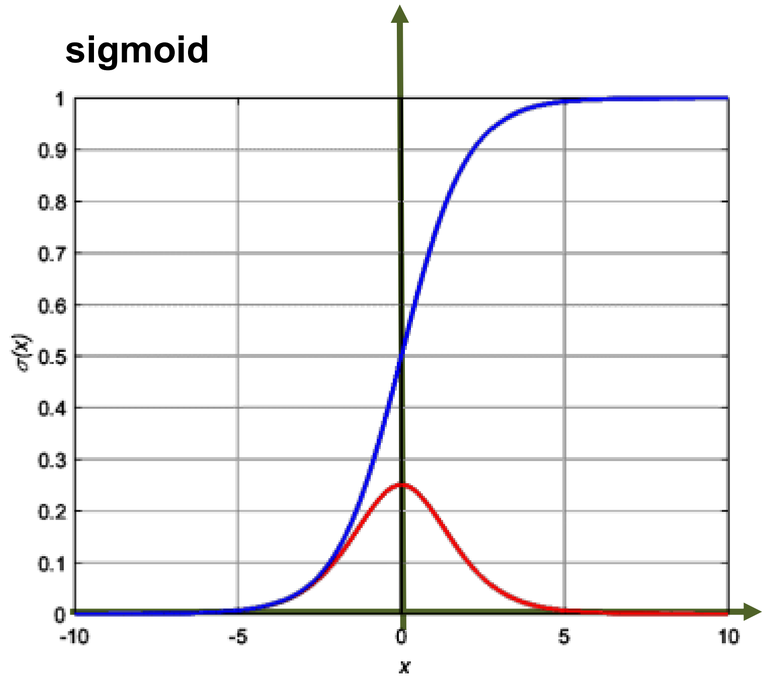
- *Understanding the difficulty of training deep feedforward neural networks* by Glorot and Bengio, 2010
- *Exact solutions to the nonlinear dynamics of learning in deep linear neural networks* by Saxe et al, 2013
- *Random walk initialization for training very deep feedforward networks* by Sussillo and Abbott, 2014
- *Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification* by He et al., 2015
- *Data-dependent Initializations of Convolutional Neural Networks* by Krähenbühl et al., 2015
- *All you need is a good init,* Mishkin and Matas, 2015

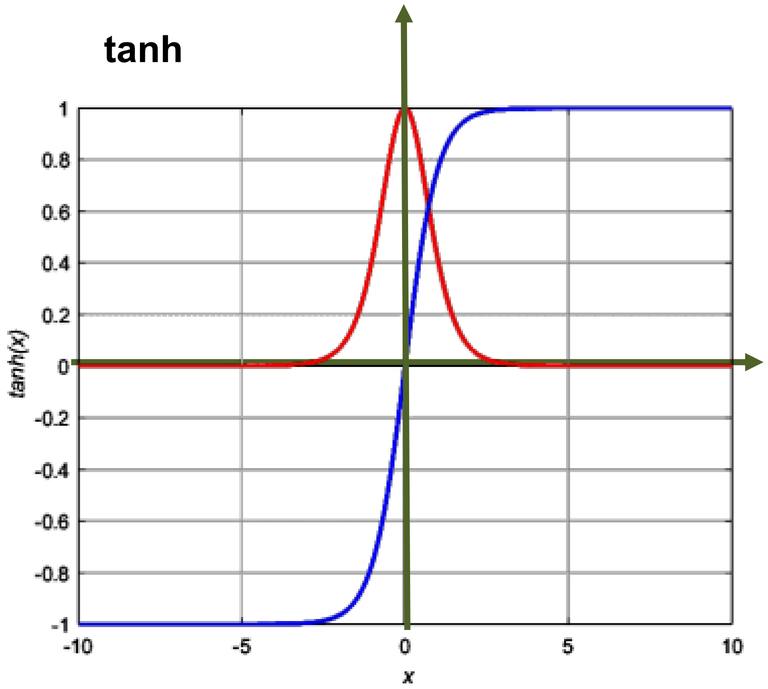
# Overview

1. Backpropagation
2. Activation Functions
3. Data Preprocessing
4. Weight Initialization
5. Batch Normalization

# Batch Normalization

- The nonlinearity functions that lie at the heart of neural networks do their most interesting work around the neighborhood of zero, plus or minus one or two.



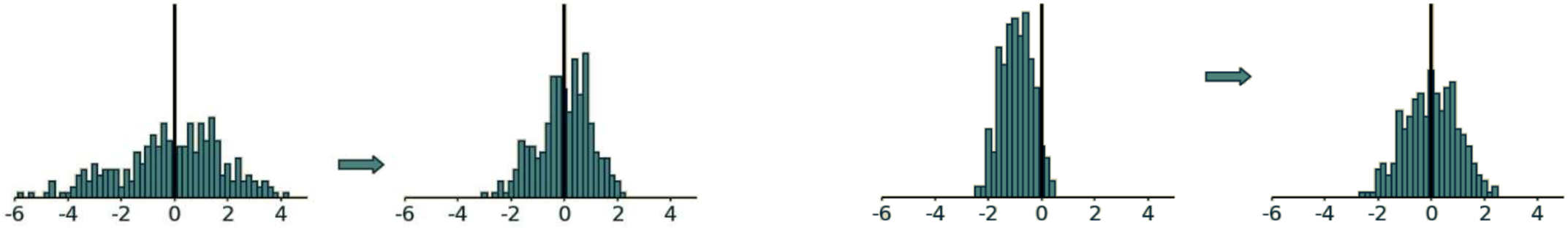
# Batch Normalization

- The nonlinearity functions that lie at the heart of neural networks do their most interesting work around the neighborhood of zero, plus or minus one or two.
- If a node's activity distribution climbs too far away from that, it can escape the reach of the backpropagation training signal (enter a gradient plateau).
- Batch Normalization tends to **smooth out the loss function**: allows for more aggressive training rates, and shorter training runs!

# Making the activations Gaussian

- Take a batch of feature values at some layer.
- To make each feature dimension $(d)$ look like a unit Gaussian, apply:

$$\hat{x}^{(d)} = \frac{x^{(d)} - \mathbb{E}\left[x^{(d)}\right]}{\sqrt{Var\left[x^{(d)}\right]}}$$



# Making the activations Gaussian

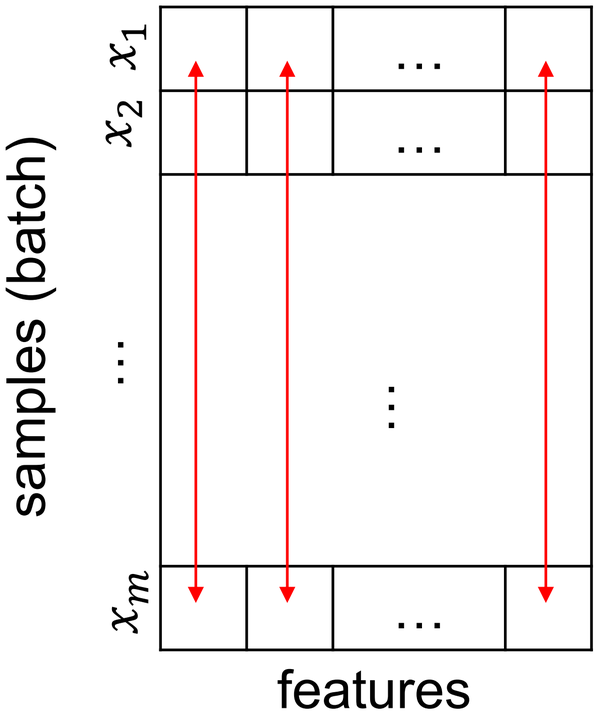
- Take a batch of features values at some layer.
- To make each feature dimension $(d)$ look like a unit Gaussian, apply:

$$\hat{x}^{(d)} = \frac{x^{(d)} - \mathbb{E}\left[x^{(d)}\right]}{\sqrt{Var\left[x^{(d)}\right]}}$$

- Important remark: BN won't work properly if the batch size is small!
- How small?

# Making the activations Gaussian

- In practice, we take the empirical mean and variance.
- For $x$ over a mini-batch $\mathcal{B} = \{x_{1\ldots m}\}$ :




$\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}^2$ computed for each dimension independently

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \varepsilon}}$$

constant added for numerical stability

# Recovering the identity mapping

- Normalize:

$$\hat{x}^{(d)} = \frac{x^{(d)} - \mathbb{E}\left[x^{(d)}\right]}{\sqrt{Var\left[x^{(d)}\right]}}$$

- Then allow the network to undo it, if "wished":

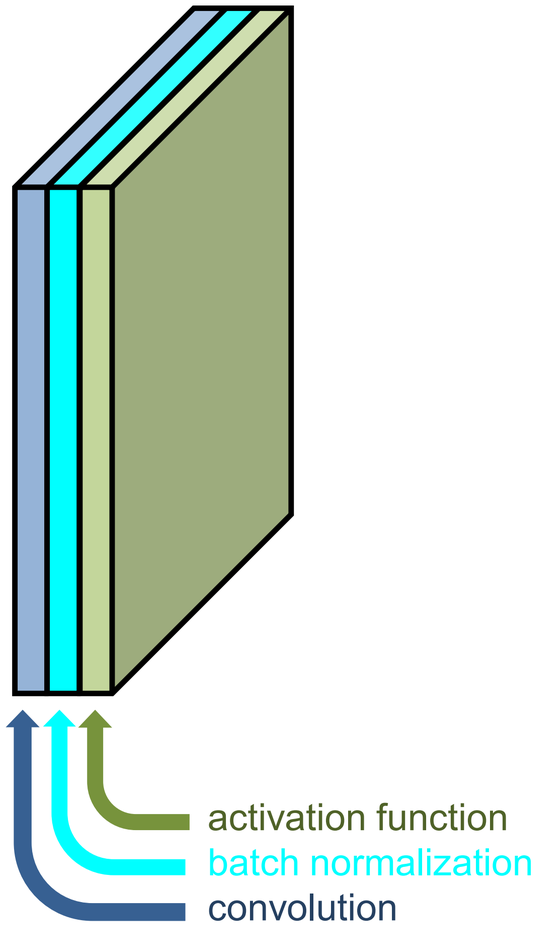
$$y^{(d)} = \gamma^{(d)}\hat{x}^{(d)} + \beta^{(d)}$$

learnable parameters

- Notice that the network can learn to recover the identity mapping:

$$\gamma^{(d)} = \sqrt{Var\left[x^{(d)}\right]} \qquad \beta^{(d)} = \mathbb{E}\left[x^{(d)}\right]$$

# CNN layer with Batch Normalization




Batch normalization is inserted prior to the nonlinearity:

$$y^{(d)} = \gamma^{(d)} \hat{x}^{(d)} + \beta^{(d)}$$

# Batch Normalization Transform

**Input:** Values of $x$ over a mini-batch $\mathcal{B} = \{x_{1\ldots m}\}$;

Parameters to be learned: $\gamma, \beta$

**Input:**

$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i$ // mini-batch mean

$\sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$ // mini-batch variance

$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \varepsilon}}$ // normalize

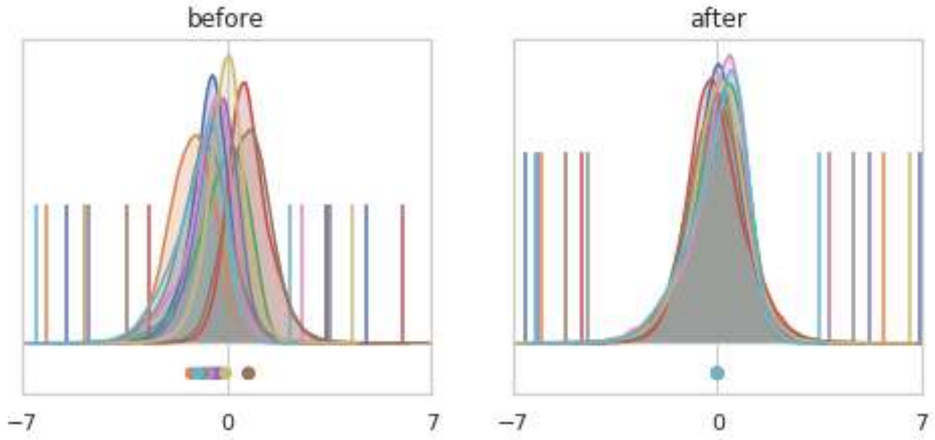
$y_i = \gamma\,\hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i)$ // scale and shift

- Improves gradient flow through the network.
- Allows higher learning rates.
- Reduces the dependence on the initialization.

Ioffe, S. and Szegedy, C., 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv [Available Online] at https://arxiv.org/abs/1502.03167

# Visual example

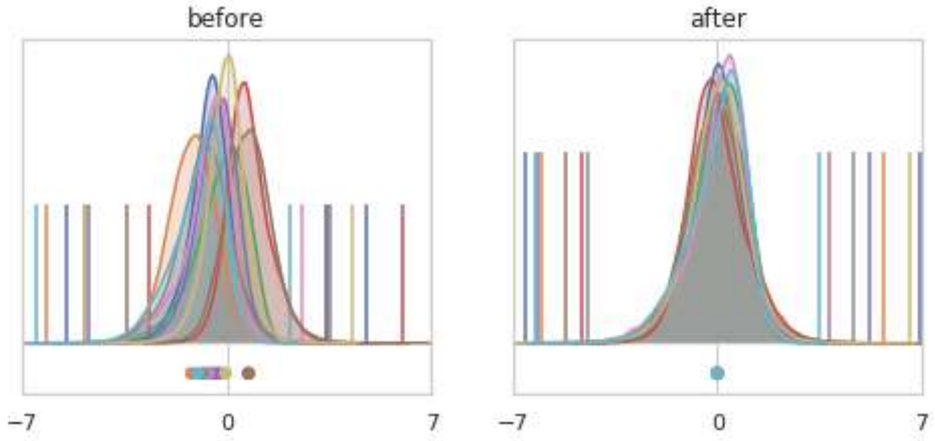
- Simple network with 50 fully connected layers + ReLU.
- Network weights initialized randomly.
- Histograms of activation values across the batch, before and after a batch norm layer.




https://colab.research.google.com/github/davidcpage/cifar10-fast/blob/master/batch_norm_post.ipynb#scrollTo=It8ETvigF3yb

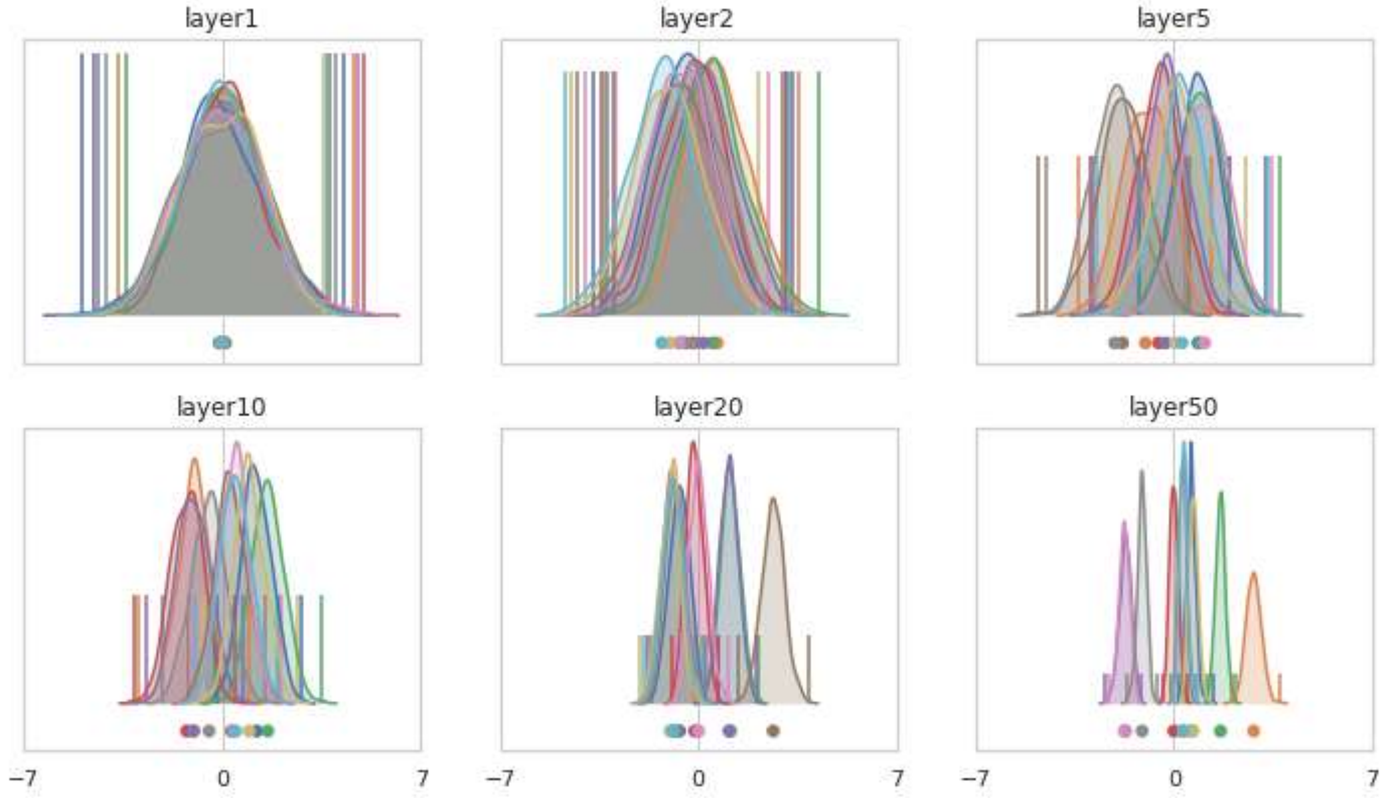
# Visual example

- Different neuron activations represented by different colors.
- Mean activation values are shown underneath.
- Minimum and maximum values are indicated by the vertical ticks.



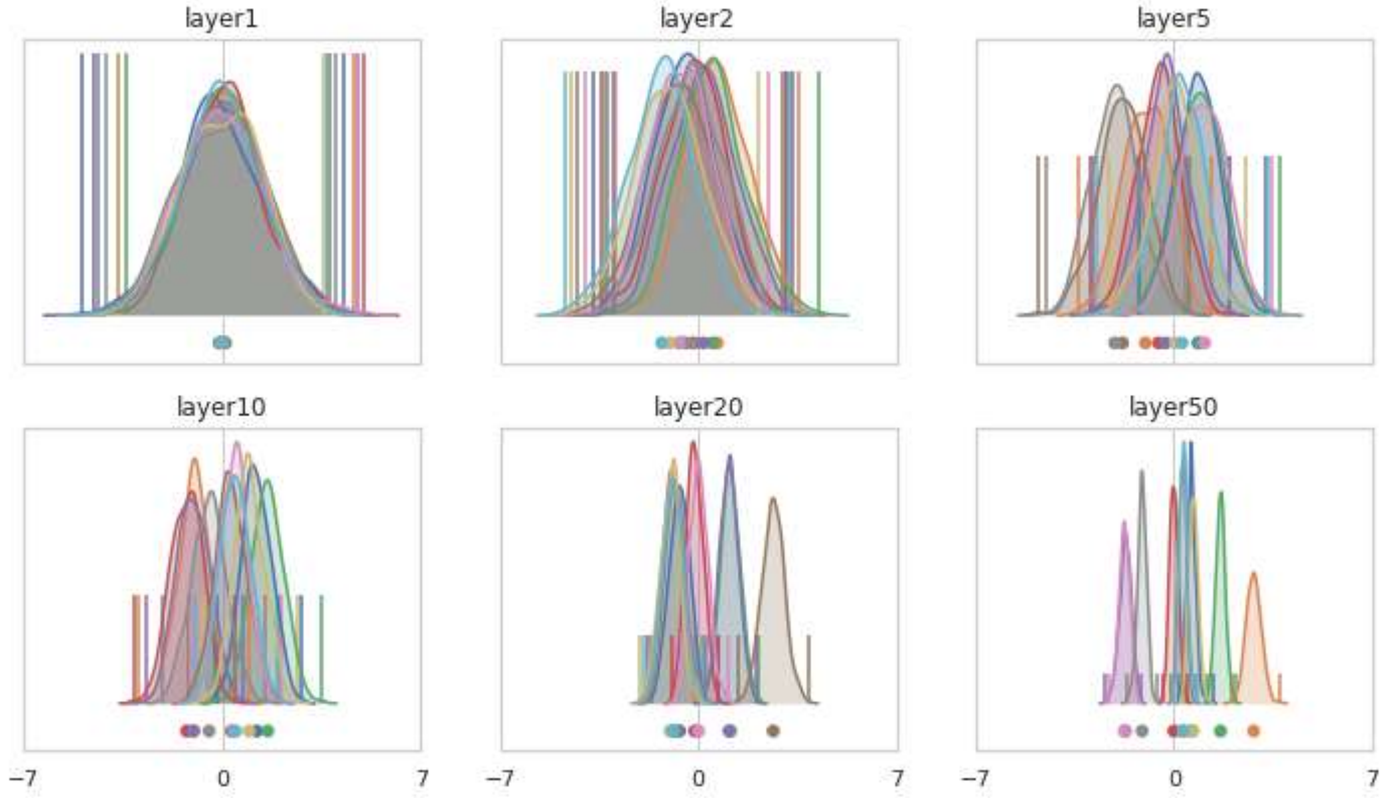
# Visual example

Without BN, the network learns a constant function…



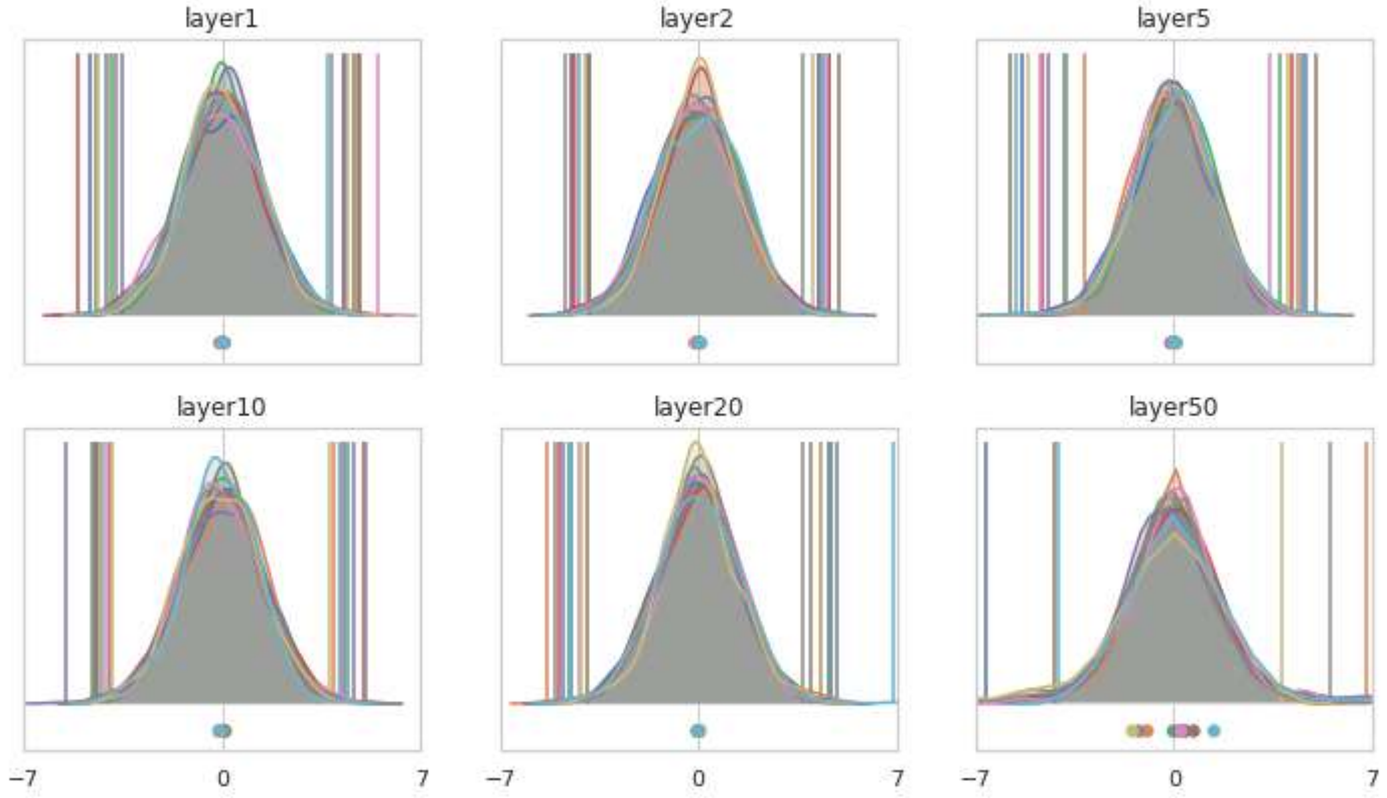
# Visual example

If we plug a softmax classifier after layer 50, the orange neuron will (almost) always win!



# Visual example

After Batch Normalization…



# Next Class

# Neural Networks
# Optimization and Generalization

See you next class!