# Backpropagation

A. M. Sadeghzadeh, Ph.D.

Sharif University of Technology
Computer Engineering Department (CE)
Data and Network Security Lab (DNSL)

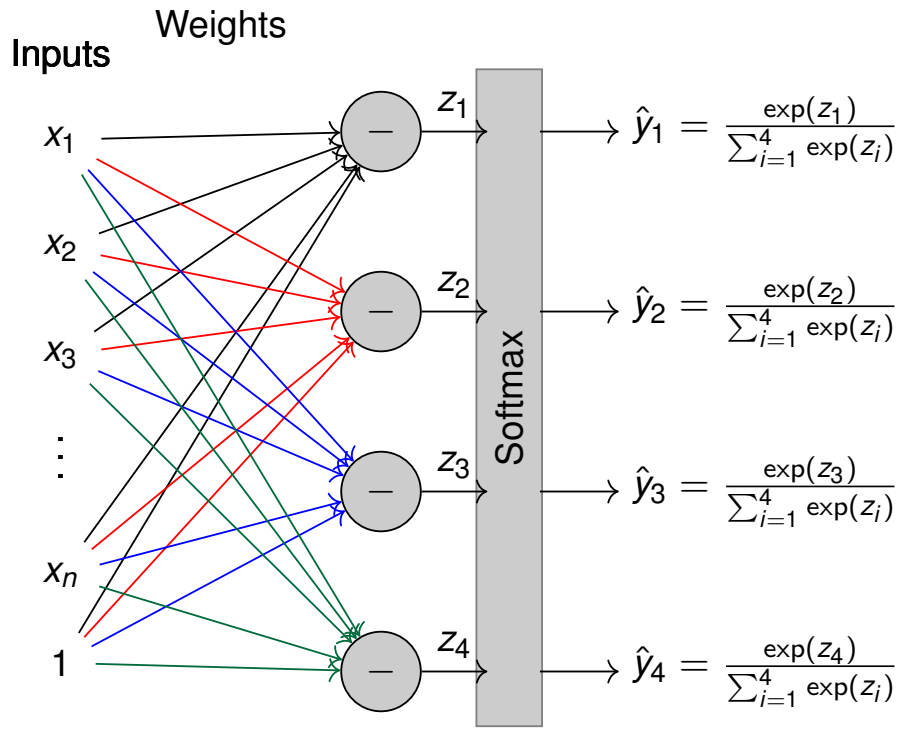

February 28, 2023

Most slides have been adapted from Bhiksha Raj, 11-785, CMU 2020, Justin Johnson, EECS 498-007, University of Michigan 2020, and Fei Fei Li, cs231n, Stanford 2017

# Today's agenda

# Recap

# Softmax classifier



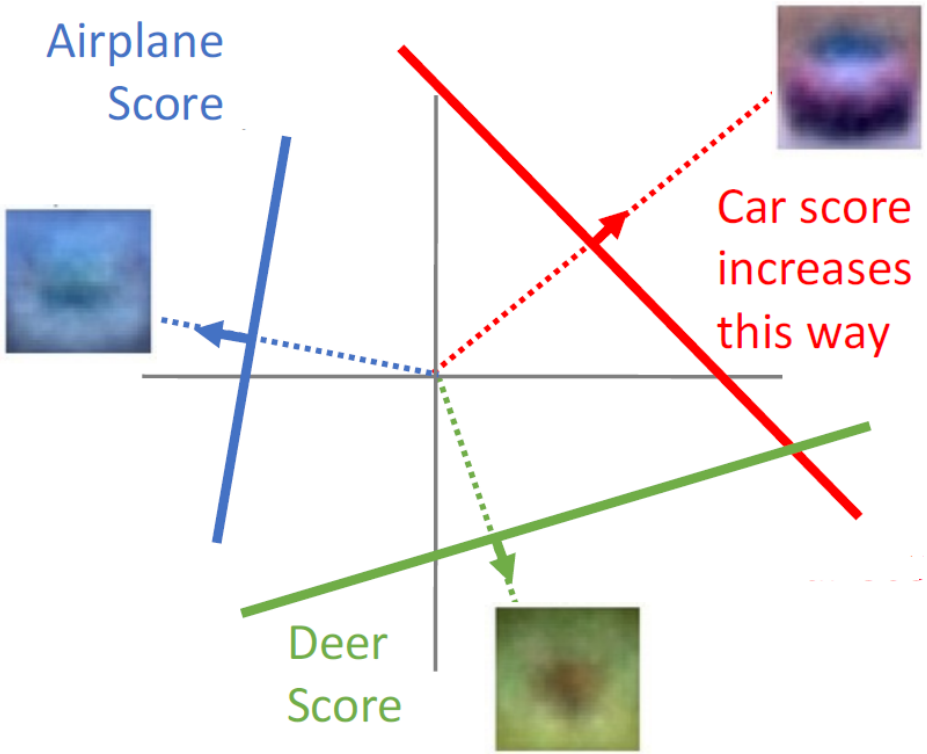

Softmax function:

$$P(\hat{y} = j|\boldsymbol{x}, W) = \frac{\exp(z_j)}{\sum_{i=1}^{K}\exp(z_i)} = \hat{y}_j$$

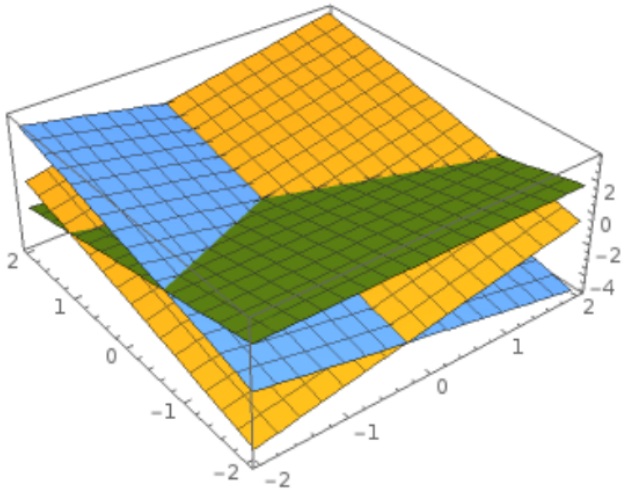
The output vector: $\hat{\boldsymbol{y}} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \\ \hat{y}_4 \end{bmatrix}$

The output of classifier: $\underset{j}{\text{argmax}}\ \hat{y}_j$

# Softmax classifier

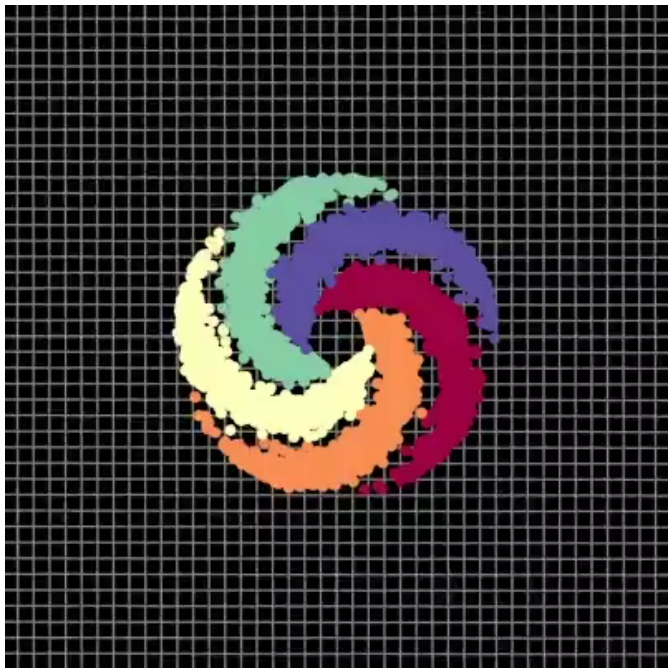



Hyperplanes carving up a high-dimensional space



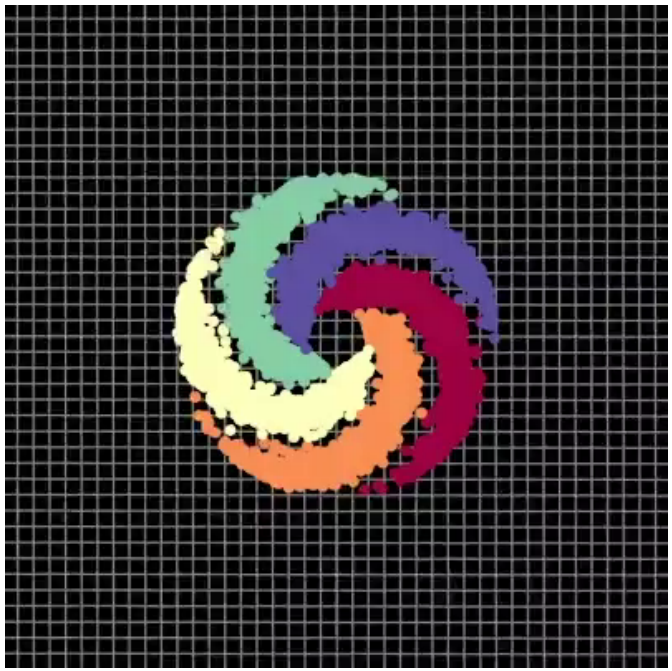

http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/

# Needs more layers

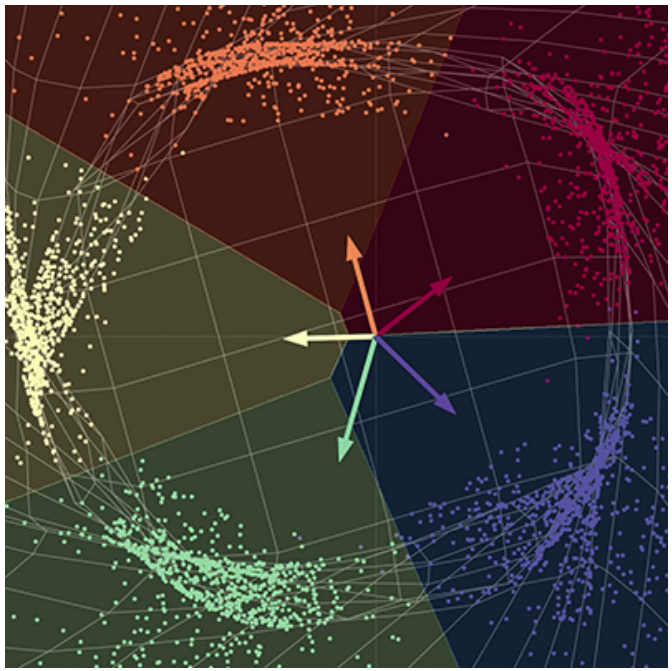


Input points, pre-network

# Needs more layers



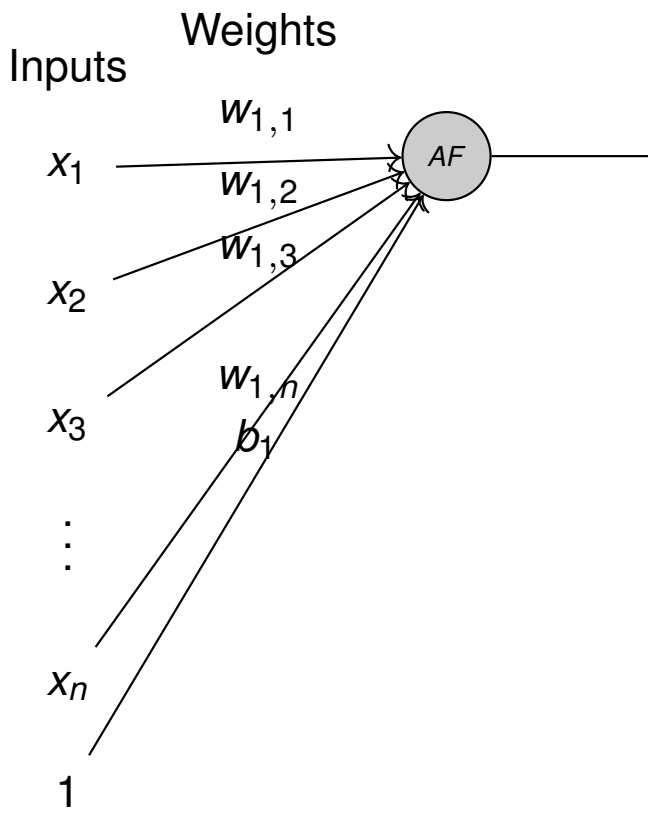
Input points, pre-network



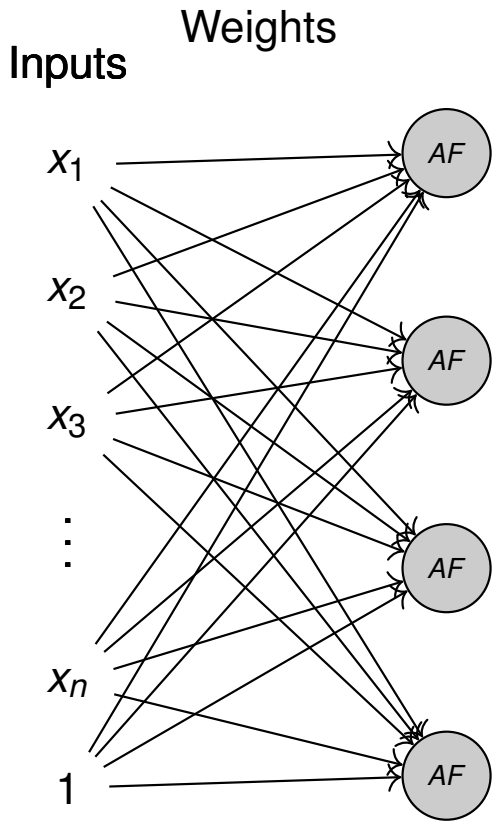
Output points, post-network

(Canziani, 2020)

# Fully connected neural networks

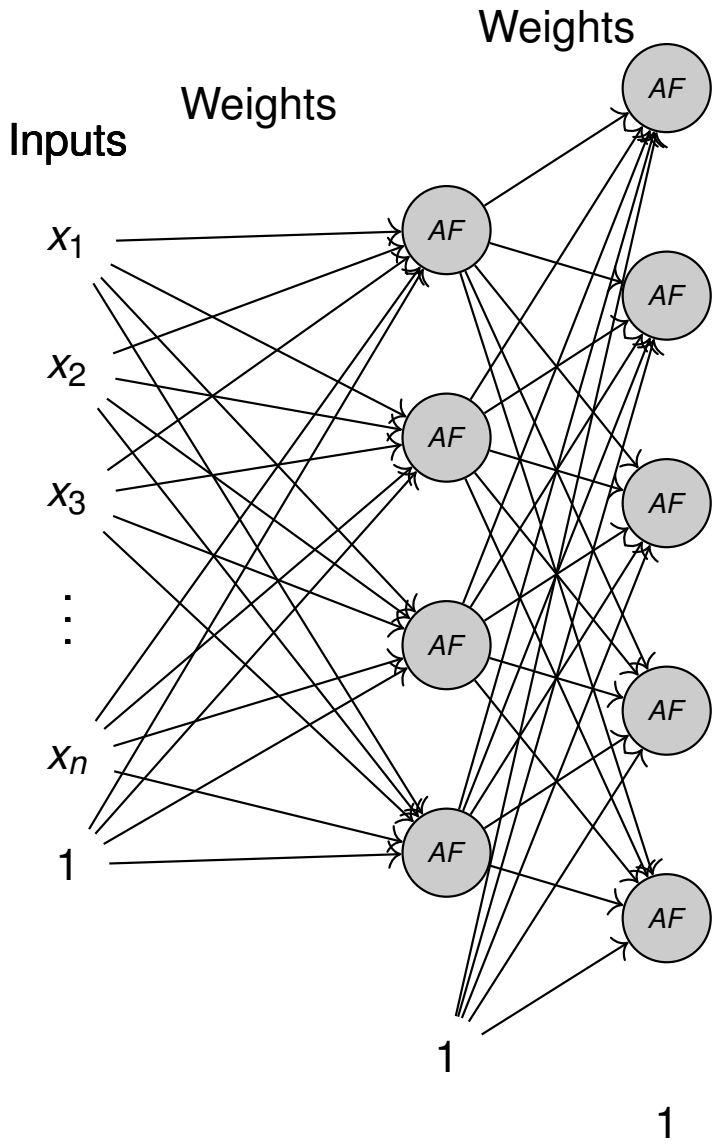


# Fully connected neural networks

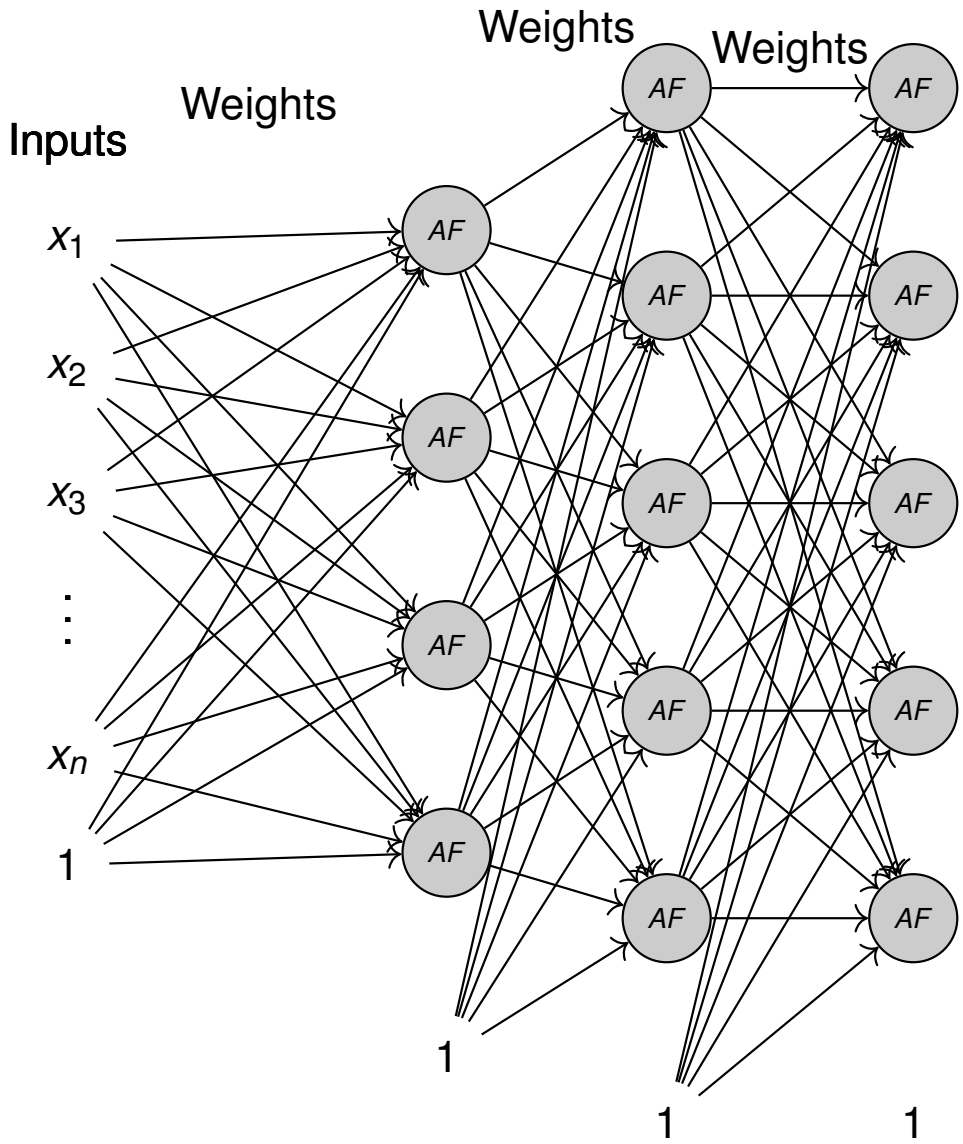



1

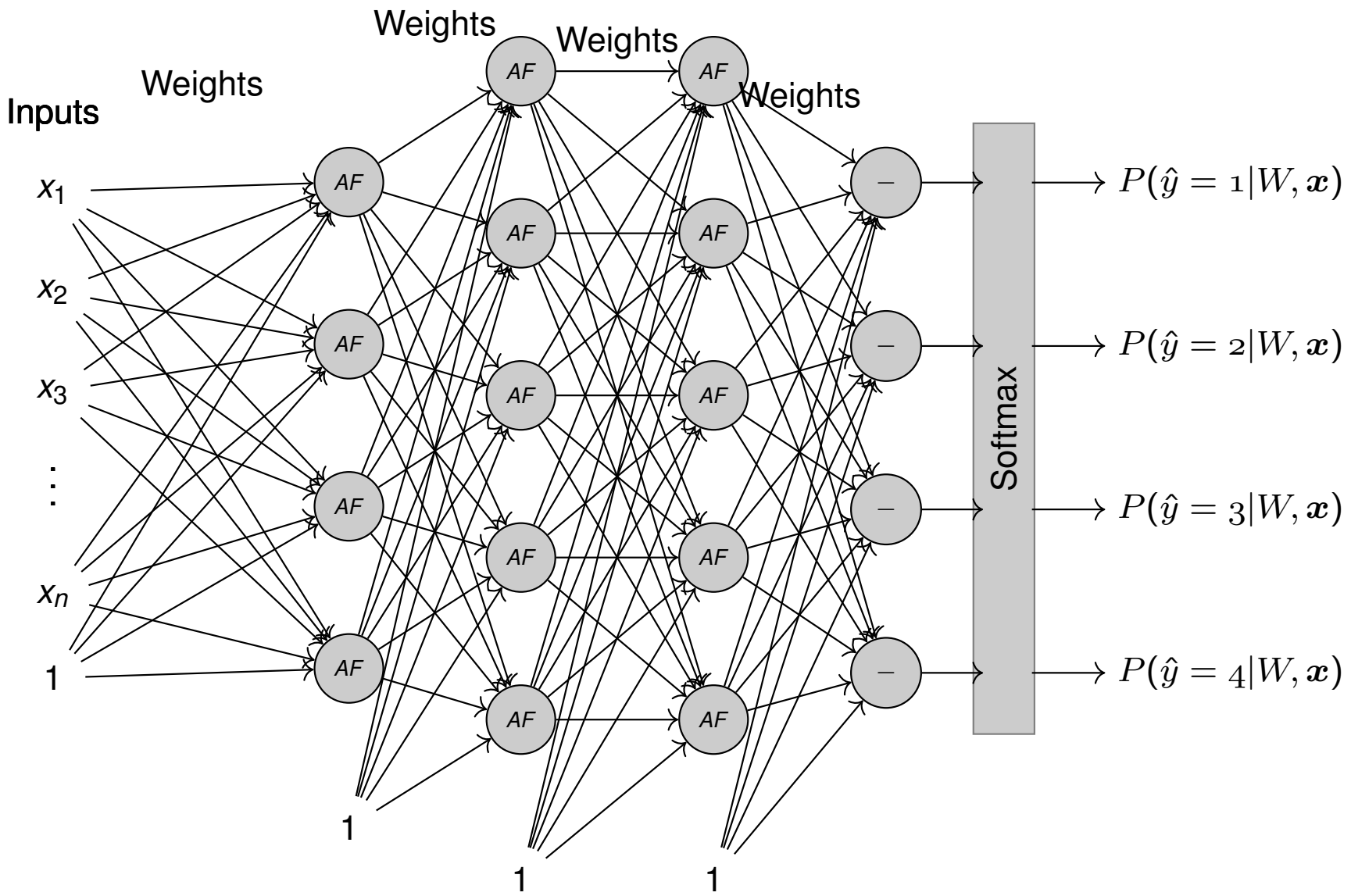
# Fully connected neural networks



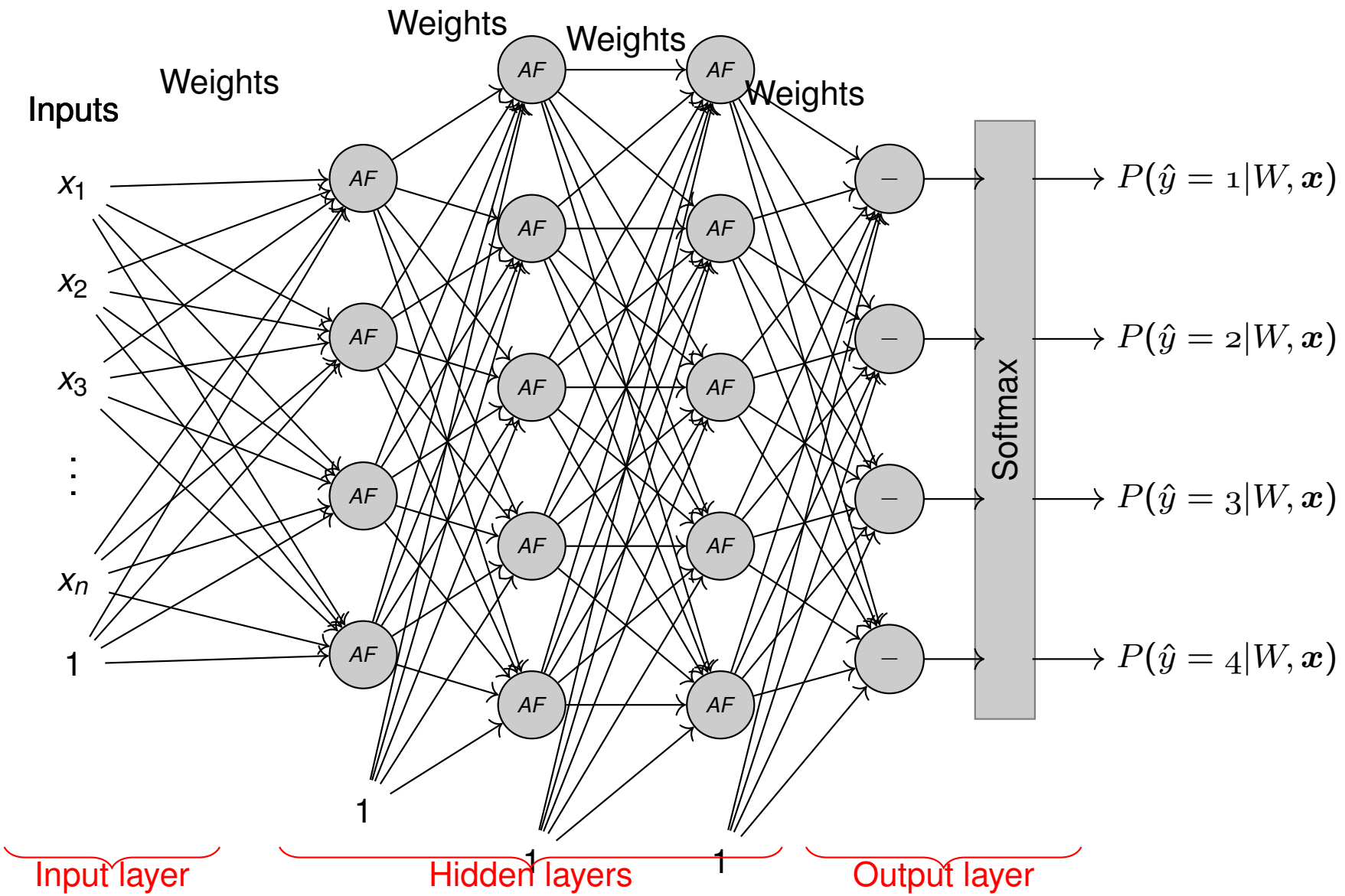
# Fully connected neural networks



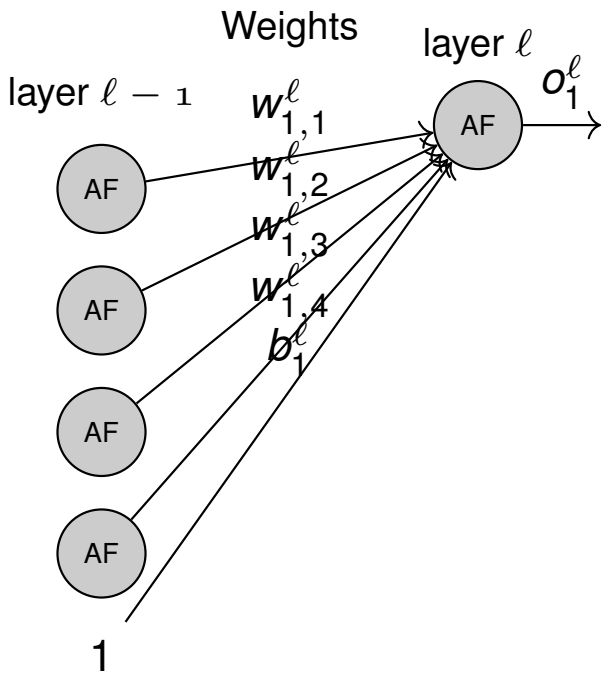
# Fully connected neural networks



# Fully connected neural networks



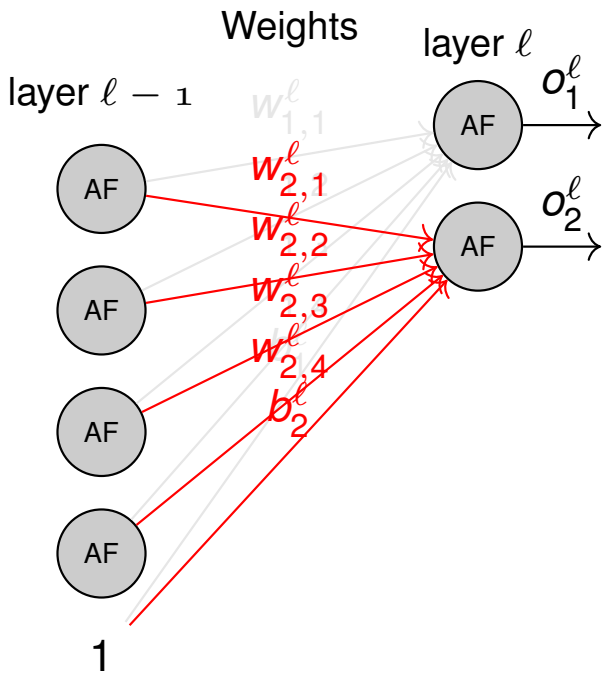
# Fully connected neural networks




$$z_1^\ell = \sum_i w_{1,i}^\ell o_i^{\ell-1} + b_1^\ell = \boldsymbol{w_1^\ell}^T \boldsymbol{o^{\ell-1}} + b_1^\ell, \quad o_1^\ell = AF(z_1^\ell)$$

$$z_1^\ell = \begin{bmatrix} w_{1,1}^\ell & w_{1,2}^\ell & w_{1,3}^\ell & w_{1,4}^\ell \end{bmatrix} \begin{bmatrix} o_1^{\ell-1} \\ o_2^{\ell-1} \\ o_3^{\ell-1} \\ o_4^{\ell-1} \end{bmatrix} + b_1^\ell$$

# Fully connected neural networks

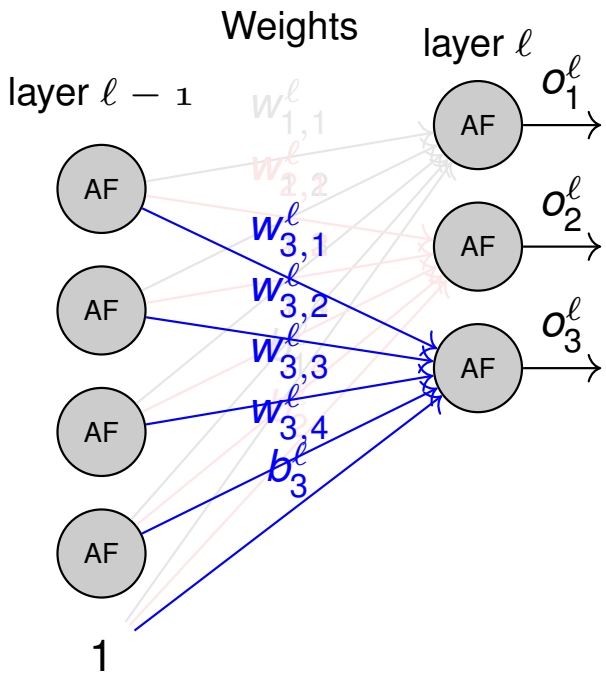



$$z_1^\ell = \sum_i w_{1,i}^\ell o_i^{\ell-1} + b_1^\ell = \boldsymbol{w_1^\ell}^T \boldsymbol{o^{\ell-1}} + b_1^\ell, \quad o_1^\ell = AF(z_1^\ell)$$

$$z_2^\ell = \sum_i w_{2,i}^\ell o_i^{\ell-1} + b_2^\ell = \boldsymbol{w_2^\ell}^T \boldsymbol{o^{\ell-1}} + b_2^\ell, \quad o_2^\ell = AF(z_2^\ell)$$

$$\begin{bmatrix} z_1^\ell \\ z_2^\ell \end{bmatrix} = \begin{bmatrix} w_{1,1}^\ell & w_{1,2}^\ell & w_{1,3}^\ell & w_{1,4}^\ell \\ w_{2,1}^\ell & w_{2,2}^\ell & w_{2,3}^\ell & w_{2,4}^\ell \end{bmatrix} \begin{bmatrix} o_1^{\ell-1} \\ o_2^{\ell-1} \\ o_3^{\ell-1} \\ o_4^{\ell-1} \end{bmatrix} + \begin{bmatrix} b_1^\ell \\ b_2^\ell \end{bmatrix}$$

# Fully connected neural networks



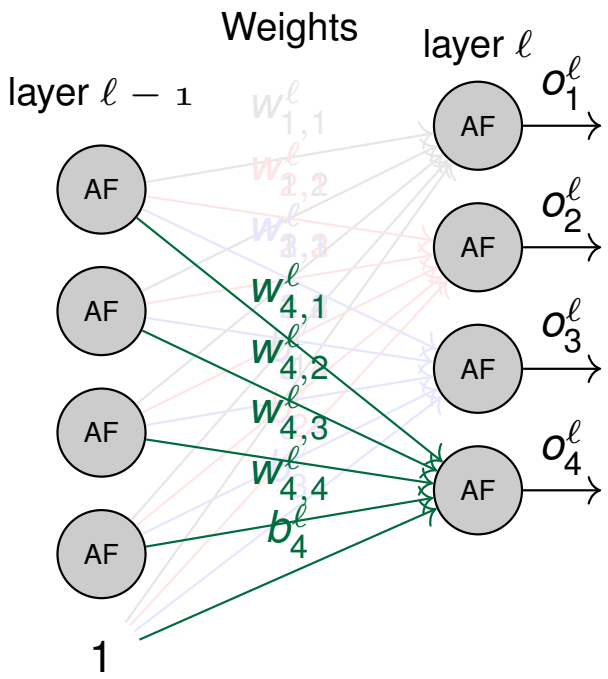

$$z_1^\ell = \sum_i w_{1,i}^\ell o_i^{\ell-1} + b_1^\ell = \boldsymbol{w_1^\ell}^T \boldsymbol{o^{\ell-1}} + b_1^\ell, \quad o_1^\ell = AF(z_1^\ell)$$

$$z_2^\ell = \sum_i w_{2,i}^\ell o_i^{\ell-1} + b_2^\ell = \boldsymbol{w_2^\ell}^T \boldsymbol{o^{\ell-1}} + b_2^\ell, \quad o_2^\ell = AF(z_2^\ell)$$

$$z_3^\ell = \sum_i w_{3,i}^\ell o_i^{\ell-1} + b_3^\ell = \boldsymbol{w_3^\ell}^T \boldsymbol{o^{\ell-1}} + b_3^\ell, \quad o_3^\ell = AF(z_3^\ell)$$

$$\begin{bmatrix} z_1^\ell \\ z_2^\ell \\ z_3^\ell \end{bmatrix} = \begin{bmatrix} w_{1,1}^\ell & w_{1,2}^\ell & w_{1,3}^\ell & w_{1,4}^\ell \\ w_{2,1}^\ell & w_{2,2}^\ell & w_{2,3}^\ell & w_{2,4}^\ell \\ w_{3,1}^\ell & w_{3,2}^\ell & w_{3,3}^\ell & w_{3,4}^\ell \end{bmatrix} \begin{bmatrix} o_1^{\ell-1} \\ o_2^{\ell-1} \\ o_3^{\ell-1} \\ o_4^{\ell-1} \end{bmatrix} + \begin{bmatrix} b_1^\ell \\ b_2^\ell \\ b_3^\ell \end{bmatrix}$$

# Fully connected neural networks




$$z_1^\ell = \sum_i w_{1,i}^\ell o_i^{\ell-1} + b_1^\ell = \boldsymbol{w_1^\ell}^T \boldsymbol{o^{\ell-1}} + b_1^\ell, \quad o_1^\ell = AF(z_1^\ell)$$

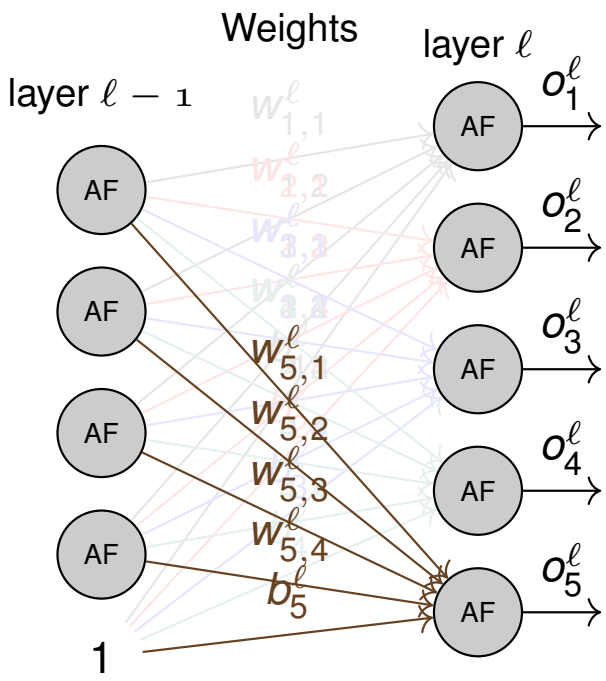
$$z_2^\ell = \sum_i w_{2,i}^\ell o_i^{\ell-1} + b_2^\ell = \boldsymbol{w_2^\ell}^T \boldsymbol{o^{\ell-1}} + b_2^\ell, \quad o_2^\ell = AF(z_2^\ell)$$

$$z_3^\ell = \sum_i w_{3,i}^\ell o_i^{\ell-1} + b_3^\ell = \boldsymbol{w_3^\ell}^T \boldsymbol{o^{\ell-1}} + b_3^\ell, \quad o_3^\ell = AF(z_3^\ell)$$

$$z_4^\ell = \sum_i w_{4,i}^\ell o_i^{\ell-1} + b_4^\ell = \boldsymbol{w_4^\ell}^T \boldsymbol{o^{\ell-1}} + b_4^\ell, \quad o_4^\ell = AF(z_4^\ell)$$

$$\begin{bmatrix} z_1^\ell \\ z_2^\ell \\ z_3^\ell \\ z_4^\ell \end{bmatrix} = \begin{bmatrix} w_{1,1}^\ell & w_{1,2}^\ell & w_{1,3}^\ell & w_{1,4}^\ell \\ w_{2,1}^\ell & w_{2,2}^\ell & w_{2,3}^\ell & w_{2,4}^\ell \\ w_{3,1}^\ell & w_{3,2}^\ell & w_{3,3}^\ell & w_{3,4}^\ell \\ w_{4,1}^\ell & w_{4,2}^\ell & w_{4,3}^\ell & w_{4,4}^\ell \end{bmatrix} \begin{bmatrix} o_1^{\ell-1} \\ o_2^{\ell-1} \\ o_3^{\ell-1} \\ o_4^{\ell-1} \end{bmatrix} + \begin{bmatrix} b_1^\ell \\ b_2^\ell \\ b_3^\ell \\ b_4^\ell \end{bmatrix}$$

# Fully connected neural networks




$$z_1^\ell = \sum_i w_{1,i}^\ell o_i^{\ell-1} + b_1^\ell = \boldsymbol{w_1^\ell}^T \boldsymbol{o^{\ell-1}} + b_1^\ell, \quad o_1^\ell = AF(z_1^\ell)$$

$$z_2^\ell = \sum_i w_{2,i}^\ell o_i^{\ell-1} + b_2^\ell = \boldsymbol{w_2^\ell}^T \boldsymbol{o^{\ell-1}} + b_2^\ell, \quad o_2^\ell = AF(z_2^\ell)$$

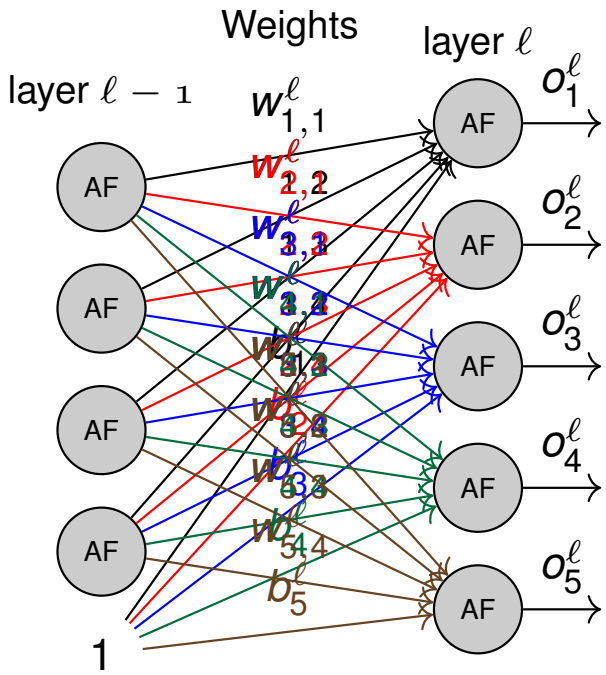
$$z_3^\ell = \sum_i w_{3,i}^\ell o_i^{\ell-1} + b_3^\ell = \boldsymbol{w_3^\ell}^T \boldsymbol{o^{\ell-1}} + b_3^\ell, \quad o_3^\ell = AF(z_3^\ell)$$

$$z_4^\ell = \sum_i w_{4,i}^\ell o_i^{\ell-1} + b_4^\ell = \boldsymbol{w_4^\ell}^T \boldsymbol{o^{\ell-1}} + b_4^\ell, \quad o_4^\ell = AF(z_4^\ell)$$

$$z_5^\ell = \sum_i w_{5,i}^\ell o_i^{\ell-1} + b_5^\ell = \boldsymbol{w_5^\ell}^T \boldsymbol{o^{\ell-1}} + b_5^\ell, \quad o_5^\ell = AF(z_5^\ell)$$

$$\begin{bmatrix} z_1^\ell \\ z_2^\ell \\ z_3^\ell \\ z_4^\ell \\ z_5^\ell \end{bmatrix} = \begin{bmatrix} w_{1,1}^\ell & w_{1,2}^\ell & w_{1,3}^\ell & w_{1,4}^\ell \\ w_{2,1}^\ell & w_{2,2}^\ell & w_{2,3}^\ell & w_{2,4}^\ell \\ w_{3,1}^\ell & w_{3,2}^\ell & w_{3,3}^\ell & w_{3,4}^\ell \\ w_{4,1}^\ell & w_{4,2}^\ell & w_{4,3}^\ell & w_{4,4}^\ell \\ w_{5,1}^\ell & w_{5,2}^\ell & w_{5,3}^\ell & w_{5,4}^\ell \end{bmatrix} \begin{bmatrix} o_1^{\ell-1} \\ o_2^{\ell-1} \\ o_3^{\ell-1} \\ o_4^{\ell-1} \end{bmatrix} + \begin{bmatrix} b_1^\ell \\ b_2^\ell \\ b_3^\ell \\ b_4^\ell \\ b_5^\ell \end{bmatrix}$$

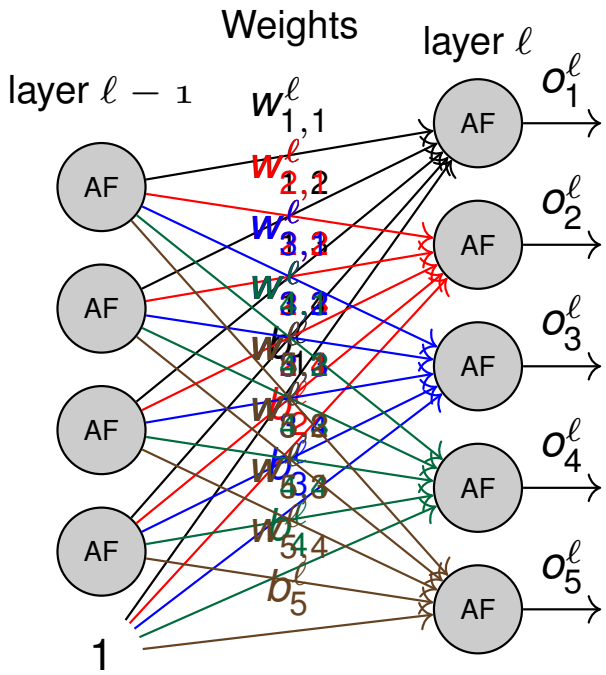
# Vector activation




$$\begin{bmatrix} z_1^\ell \\ z_2^\ell \\ z_3^\ell \\ z_4^\ell \\ z_5^\ell \end{bmatrix} = \begin{bmatrix} w_{1,1}^\ell & w_{1,2}^\ell & w_{1,3}^\ell & w_{1,4}^\ell \\ w_{2,1}^\ell & w_{2,2}^\ell & w_{2,3}^\ell & w_{2,4}^\ell \\ w_{3,1}^\ell & w_{3,2}^\ell & w_{3,3}^\ell & w_{3,4}^\ell \\ w_{4,1}^\ell & w_{4,2}^\ell & w_{4,3}^\ell & w_{4,4}^\ell \\ w_{5,1}^\ell & w_{5,2}^\ell & w_{5,3}^\ell & w_{5,4}^\ell \end{bmatrix} \begin{bmatrix} o_1^{\ell-1} \\ o_2^{\ell-1} \\ o_3^{\ell-1} \\ o_4^{\ell-1} \end{bmatrix} + \begin{bmatrix} b_1^\ell \\ b_2^\ell \\ b_3^\ell \\ b_4^\ell \\ b_5^\ell \end{bmatrix}$$

$$\begin{bmatrix} o_1^\ell \\ o_2^\ell \\ o_3^\ell \\ o_4^\ell \\ o_5^\ell \end{bmatrix} = \begin{bmatrix} AF(z_1^\ell) \\ AF(z_2^\ell) \\ AF(z_3^\ell) \\ AF(z_4^\ell) \\ AF(z_5^\ell) \end{bmatrix}$$

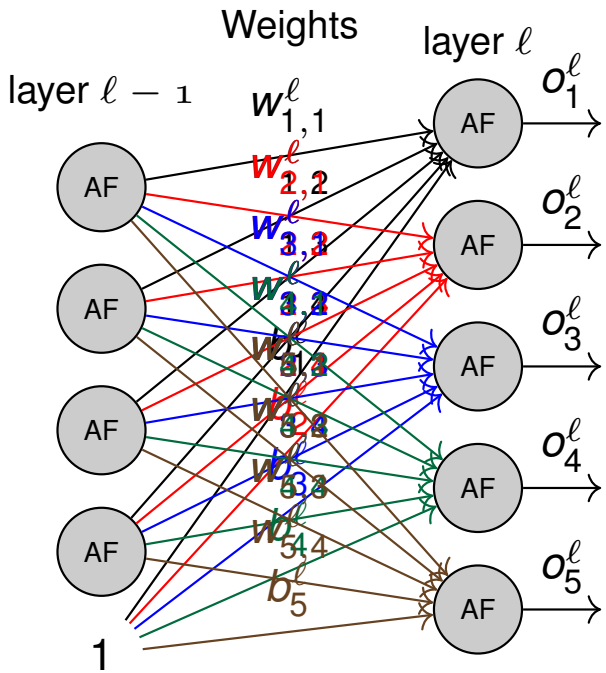
# Vector activation




$$\begin{bmatrix} z_1^\ell \\ z_2^\ell \\ z_3^\ell \\ z_4^\ell \\ z_5^\ell \end{bmatrix} = \begin{bmatrix} w_{1,1}^\ell & w_{1,2}^\ell & w_{1,3}^\ell & w_{1,4}^\ell \\ w_{2,1}^\ell & w_{2,2}^\ell & w_{2,3}^\ell & w_{2,4}^\ell \\ w_{3,1}^\ell & w_{3,2}^\ell & w_{3,3}^\ell & w_{3,4}^\ell \\ w_{4,1}^\ell & w_{4,2}^\ell & w_{4,3}^\ell & w_{4,4}^\ell \\ w_{5,1}^\ell & w_{5,2}^\ell & w_{5,3}^\ell & w_{5,4}^\ell \end{bmatrix} \begin{bmatrix} o_1^{\ell-1} \\ o_2^{\ell-1} \\ o_3^{\ell-1} \\ o_4^{\ell-1} \end{bmatrix} + \begin{bmatrix} b_1^\ell \\ b_2^\ell \\ b_3^\ell \\ b_4^\ell \\ b_5^\ell \end{bmatrix}$$

$$\begin{bmatrix} o_1^\ell \\ o_2^\ell \\ o_3^\ell \\ o_4^\ell \\ o_5^\ell \end{bmatrix} = \begin{bmatrix} AF(z_1^\ell) \\ AF(z_2^\ell) \\ AF(z_3^\ell) \\ AF(z_4^\ell) \\ AF(z_5^\ell) \end{bmatrix} = AF\left( \begin{bmatrix} z_1^\ell \\ z_2^\ell \\ z_3^\ell \\ z_4^\ell \\ z_5^\ell \end{bmatrix} \right)$$

# Vector activation

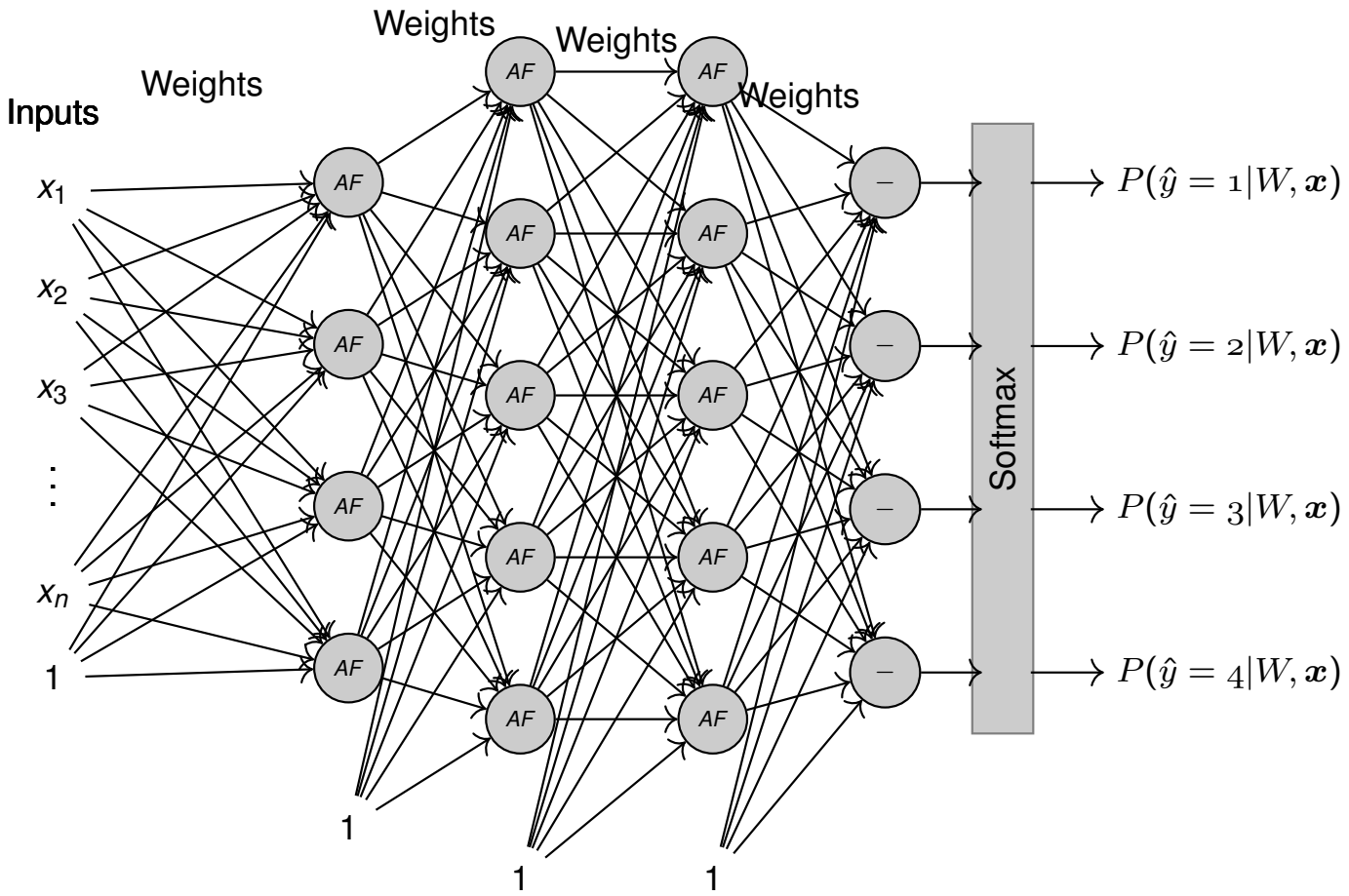



$$\begin{bmatrix} z^\ell_1 \\ z^\ell_2 \\ z^\ell_3 \\ z^\ell_4 \\ z^\ell_5 \end{bmatrix} = \begin{bmatrix} w^\ell_{1,1} & w^\ell_{1,2} & w^\ell_{1,3} & w^\ell_{1,4} \\ w^\ell_{2,1} & w^\ell_{2,2} & w^\ell_{2,3} & w^\ell_{2,4} \\ w^\ell_{3,1} & w^\ell_{3,2} & w^\ell_{3,3} & w^\ell_{3,4} \\ w^\ell_{4,1} & w^\ell_{4,2} & w^\ell_{4,3} & w^\ell_{4,4} \\ w^\ell_{5,1} & w^\ell_{5,2} & w^\ell_{5,3} & w^\ell_{5,4} \end{bmatrix} \begin{bmatrix} o^{\ell-1}_1 \\ o^{\ell-1}_2 \\ o^{\ell-1}_3 \\ o^{\ell-1}_4 \end{bmatrix} + \begin{bmatrix} b^\ell_1 \\ b^\ell_2 \\ b^\ell_3 \\ b^\ell_4 \\ b^\ell_5 \end{bmatrix}$$

$$\begin{bmatrix} o^\ell_1 \\ o^\ell_2 \\ o^\ell_3 \\ o^\ell_4 \\ o^\ell_5 \end{bmatrix} = \begin{bmatrix} AF(z^\ell_1) \\ AF(z^\ell_2) \\ AF(z^\ell_3) \\ AF(z^\ell_4) \\ AF(z^\ell_5) \end{bmatrix} = AF\left( \begin{bmatrix} z^\ell_1 \\ z^\ell_2 \\ z^\ell_3 \\ z^\ell_4 \\ z^\ell_5 \end{bmatrix} \right)$$

$$\boldsymbol{o}^\ell = AF(W^\ell \boldsymbol{o}^{\ell-1} + \boldsymbol{b}^\ell)$$

# Fully connected neural networks




$$\hat{\boldsymbol{y}} = Softmax(W^4(AF(W^3(AF(W^2(AF(W^1\boldsymbol{x} + \boldsymbol{b}^1)) + \boldsymbol{b}^2)) + \boldsymbol{b}^3)) + \boldsymbol{b}^4)$$

# Deep Neural Networks

# Activation functions

What happens if we build a neural network with no activation function?

$$\hat{\boldsymbol{y}} = Softmax(W^4(AF(W^3(AF(W^2(AF(W^1\boldsymbol{x} + \boldsymbol{b}^1)) + \boldsymbol{b}^2)) + \boldsymbol{b}^3)) + \boldsymbol{b}^4)$$

# Activation functions

What happens if we build a neural network with no activation function?

$$\hat{\boldsymbol{y}} = Softmax(W^4(AF(W^3(AF(W^2(AF(W^1\boldsymbol{x} + \boldsymbol{b}^1)) + \boldsymbol{b}^2)) + \boldsymbol{b}^3)) + \boldsymbol{b}^4)$$

$$\hat{\boldsymbol{y}} = Softmax(W^4(W^3(W^2(W^1\boldsymbol{x} + \boldsymbol{b}^1) + \boldsymbol{b}^2) + \boldsymbol{b}^3) + \boldsymbol{b}^4)$$

# Activation functions

What happens if we build a neural network with no activation function?

$$\hat{\boldsymbol{y}} = Softmax(W^4(AF(W^3(AF(W^2(AF(W^1\boldsymbol{x} + \boldsymbol{b}^1)) + \boldsymbol{b}^2)) + \boldsymbol{b}^3)) + \boldsymbol{b}^4)$$

$$\hat{\boldsymbol{y}} = Softmax(W^4(W^3(W^2(W^1\boldsymbol{x} + \boldsymbol{b}^1) + \boldsymbol{b}^2) + \boldsymbol{b}^3) + \boldsymbol{b}^4)$$

$$\hat{\boldsymbol{y}} = Softmax(W^4W^3W^2W^1\boldsymbol{x} + b')$$

# Activation functions

What happens if we build a neural network with no activation function?

$$\hat{\boldsymbol{y}} = Softmax(W^4(AF(W^3(AF(W^2(AF(W^1\boldsymbol{x} + \boldsymbol{b}^1)) + \boldsymbol{b}^2)) + \boldsymbol{b}^3)) + \boldsymbol{b}^4)$$

$$\hat{\boldsymbol{y}} = Softmax(W^4(W^3(W^2(W^1\boldsymbol{x} + \boldsymbol{b}^1) + \boldsymbol{b}^2) + \boldsymbol{b}^3) + \boldsymbol{b}^4)$$

$$\hat{\boldsymbol{y}} = Softmax(W^4W^3W^2W^1\boldsymbol{x} + b')$$

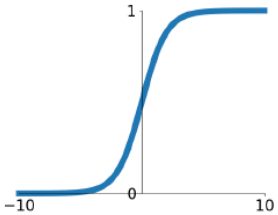
$$\hat{\boldsymbol{y}} = Softmax(W'\boldsymbol{x} + b')$$

**We end up with a softmax classifier!**

# Activation functions

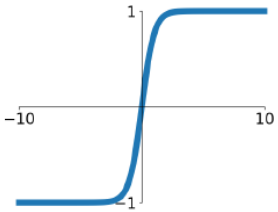
**Sigmoid**

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



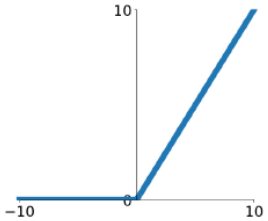

**tanh**

$$\tanh(x)$$



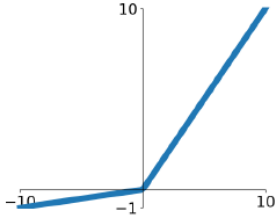

**ReLU**

$$\max(0, x)$$



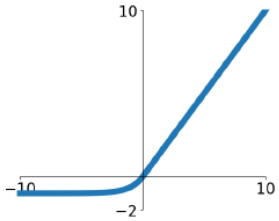

**Leaky ReLU**

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




**ELU**

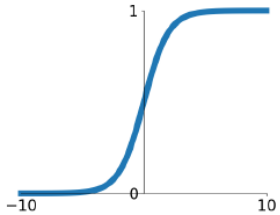
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



# Activation functions

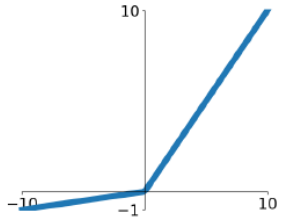
**Sigmoid**

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



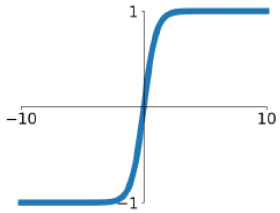

**Leaky ReLU**

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



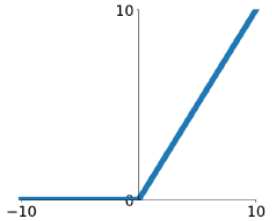

**tanh**

$$\tanh(x)$$



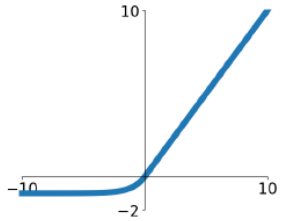

**ReLU**

$$\max(0, x)$$



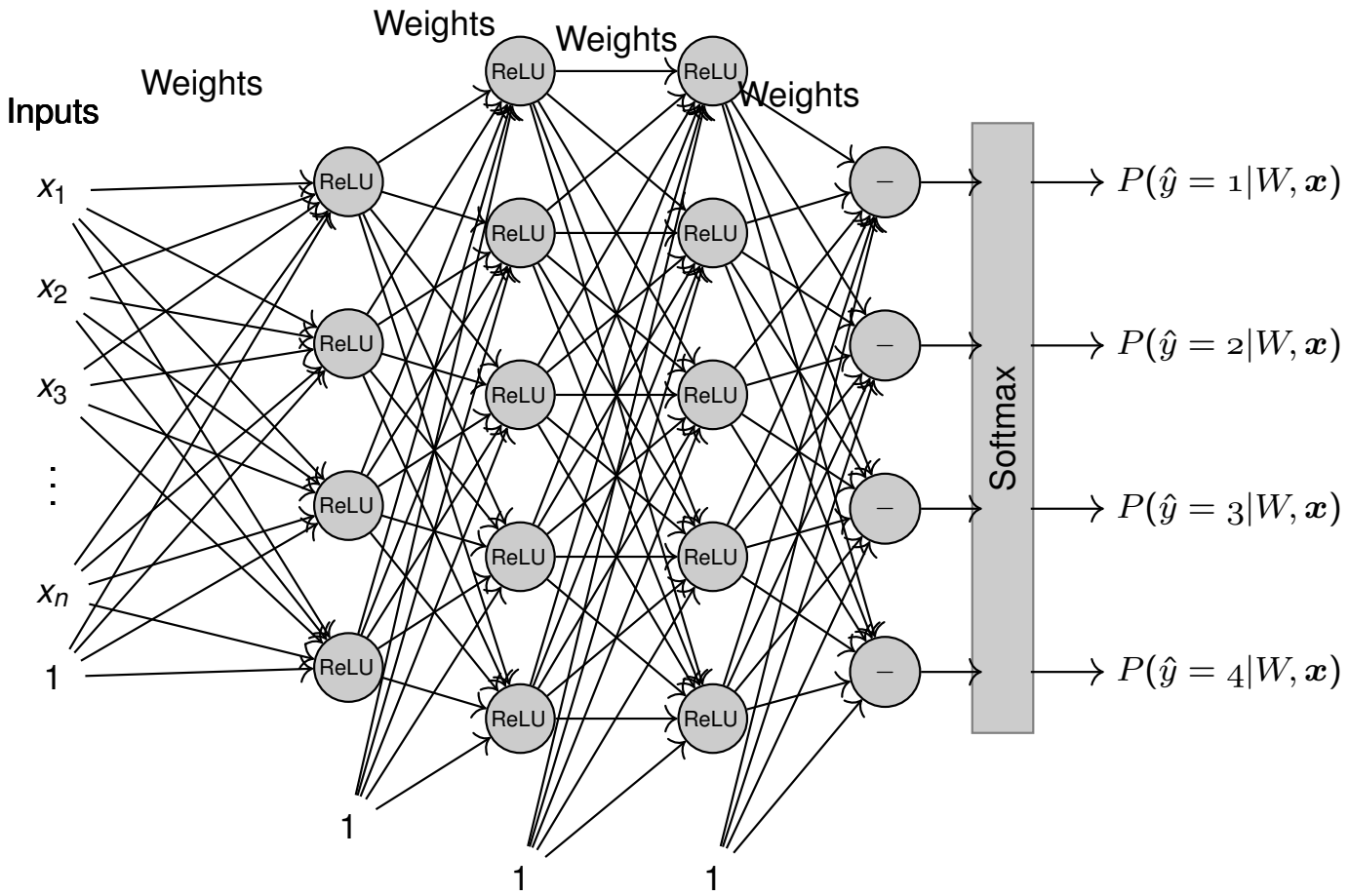

**ELU**

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




**ReLU is a good default choice for most problems**

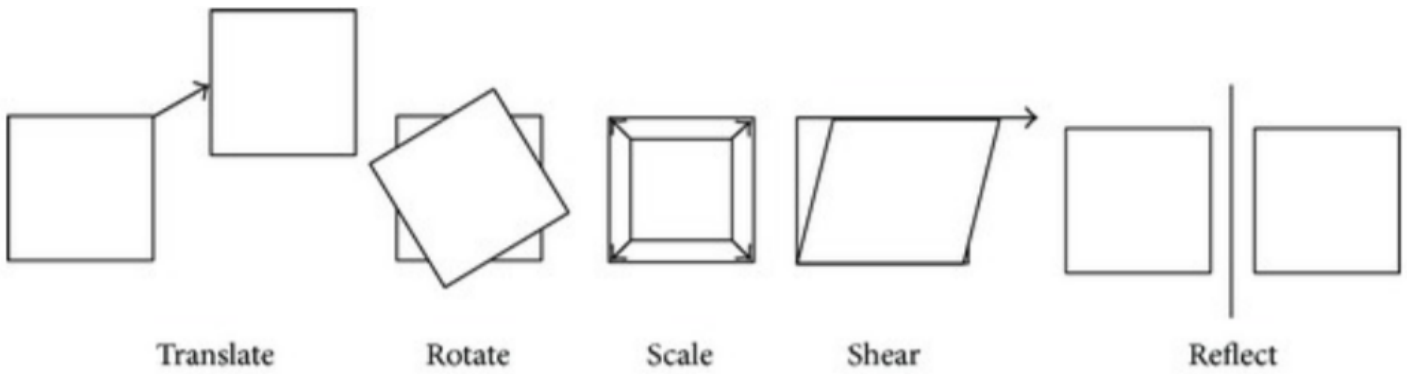
# ReLU activation function




$$\hat{\boldsymbol{y}} = Softmax(W^4(max(0, W^3(max(0, W^2(max(0, W^1\boldsymbol{x} + \boldsymbol{b}^1)) + \boldsymbol{b}^2)) + \boldsymbol{b}^3)) + \boldsymbol{b}^4)$$

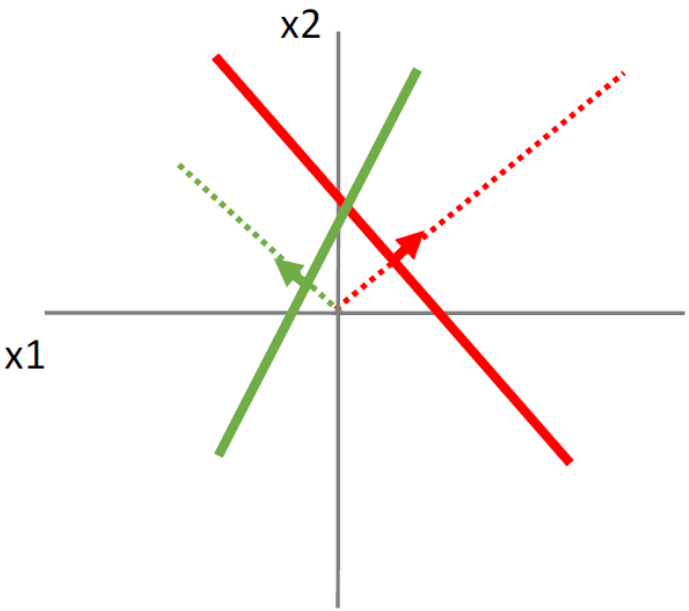
# Affine transformation

- Affine transformation
  - Translate
  - Rotate
  - Scale
  - Shear
  - Reflect
- Preserve parallel lines



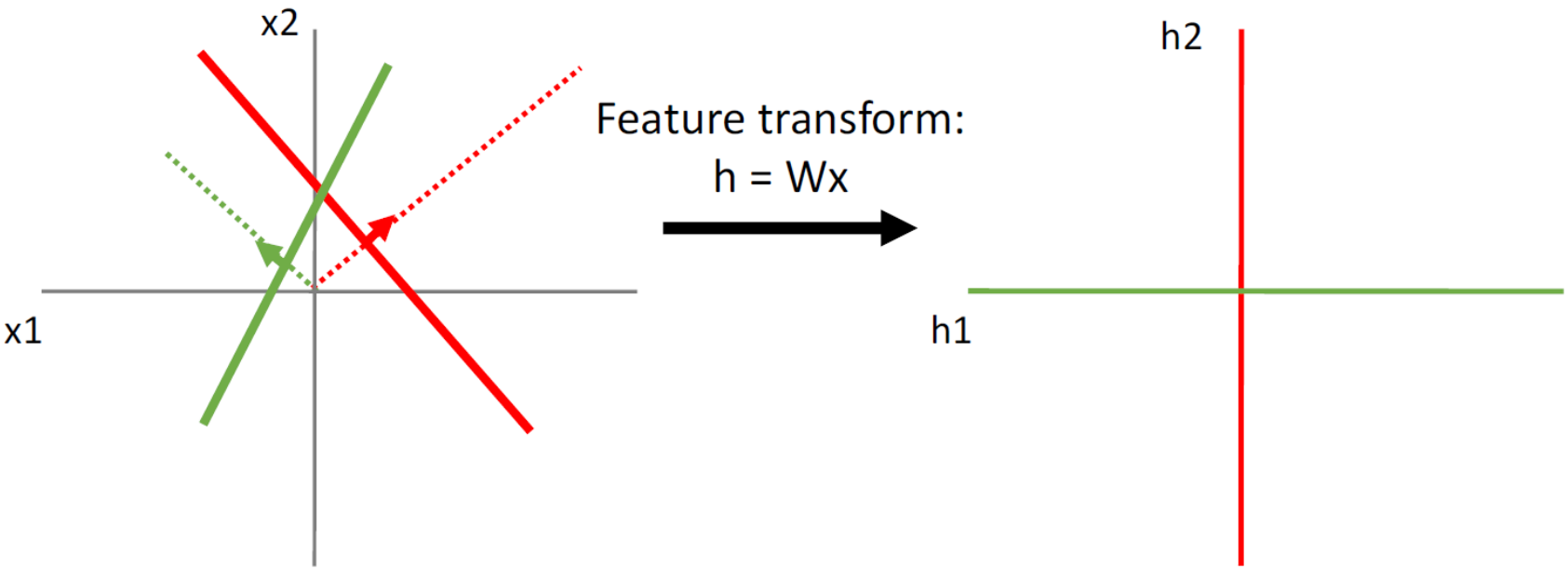
# Space warping

- Consider a linear transform: $\boldsymbol{h} = W\boldsymbol{x}$ Where $\boldsymbol{x}, \boldsymbol{h}$ are both 2-dimensional



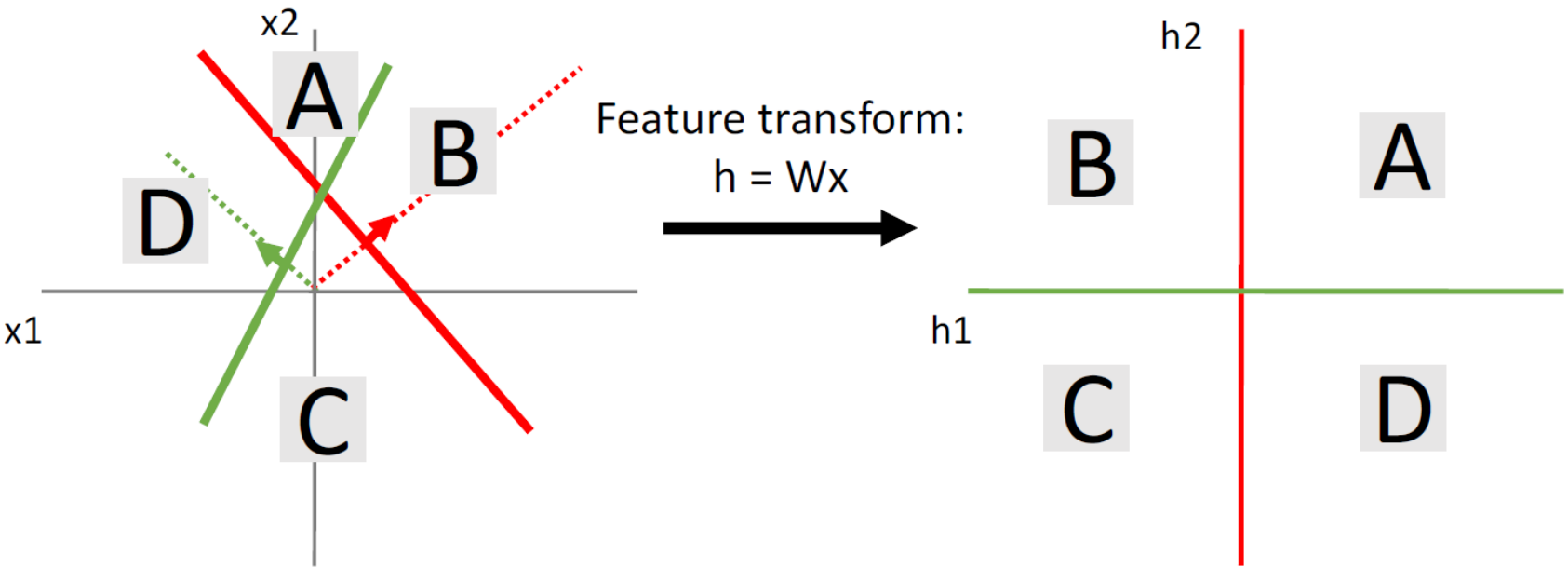
# Space warping

- Consider a linear transform: $\boldsymbol{h} = W\boldsymbol{x}$ Where $\boldsymbol{x}, \boldsymbol{h}$ are both 2-dimensional



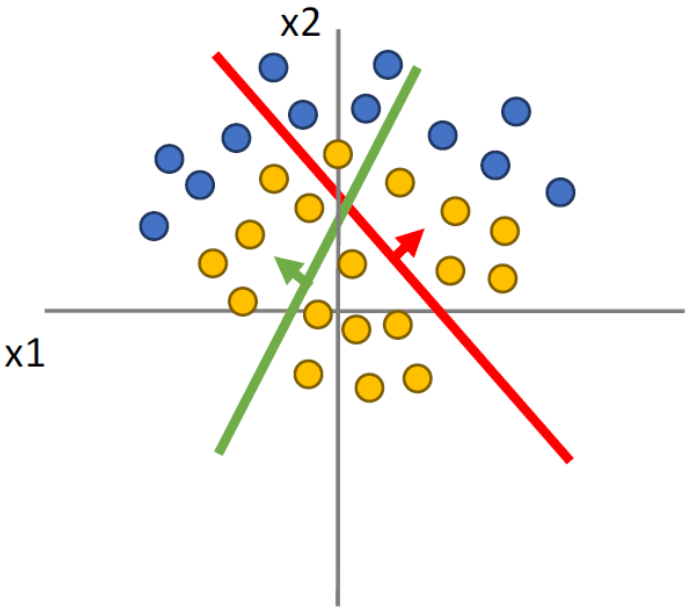
# Space warping

- Consider a linear transform: $\boldsymbol{h} = W\boldsymbol{x}$ Where $\boldsymbol{x}, \boldsymbol{h}$ are both 2-dimensional



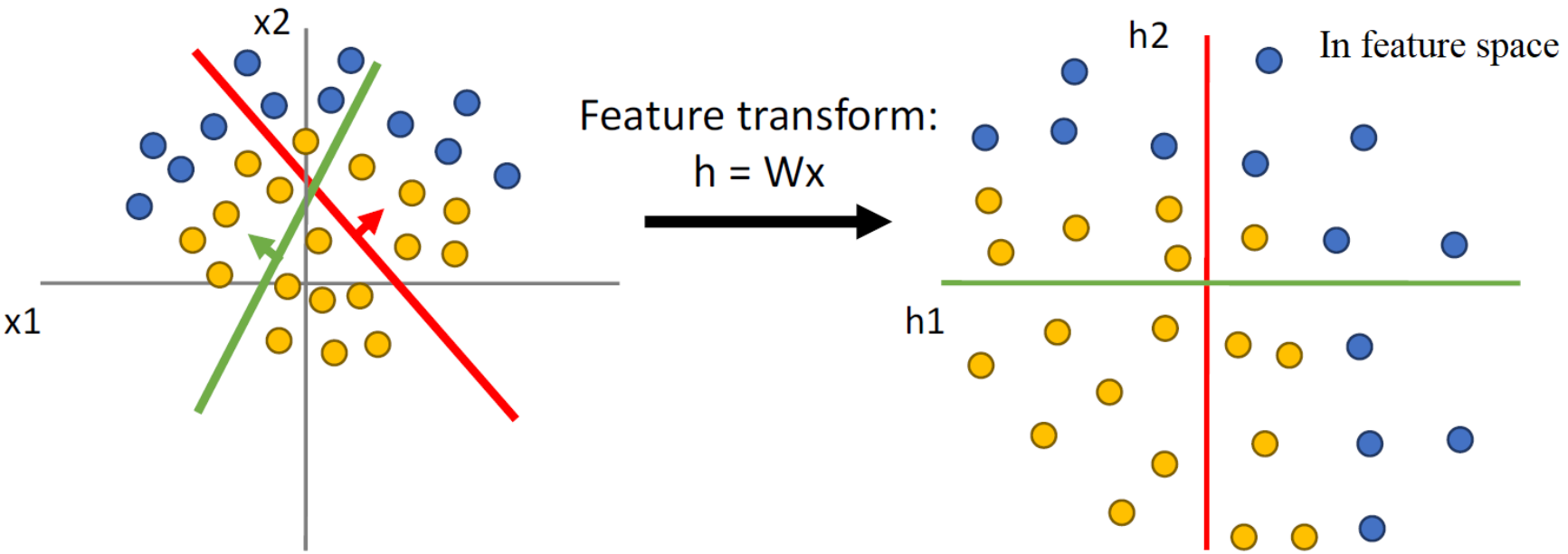
# Space warping

- Consider a linear transform: $\boldsymbol{h} = W\boldsymbol{x}$ Where $\boldsymbol{x}, \boldsymbol{h}$ are both 2-dimensional
- Points not linearly separable in original space



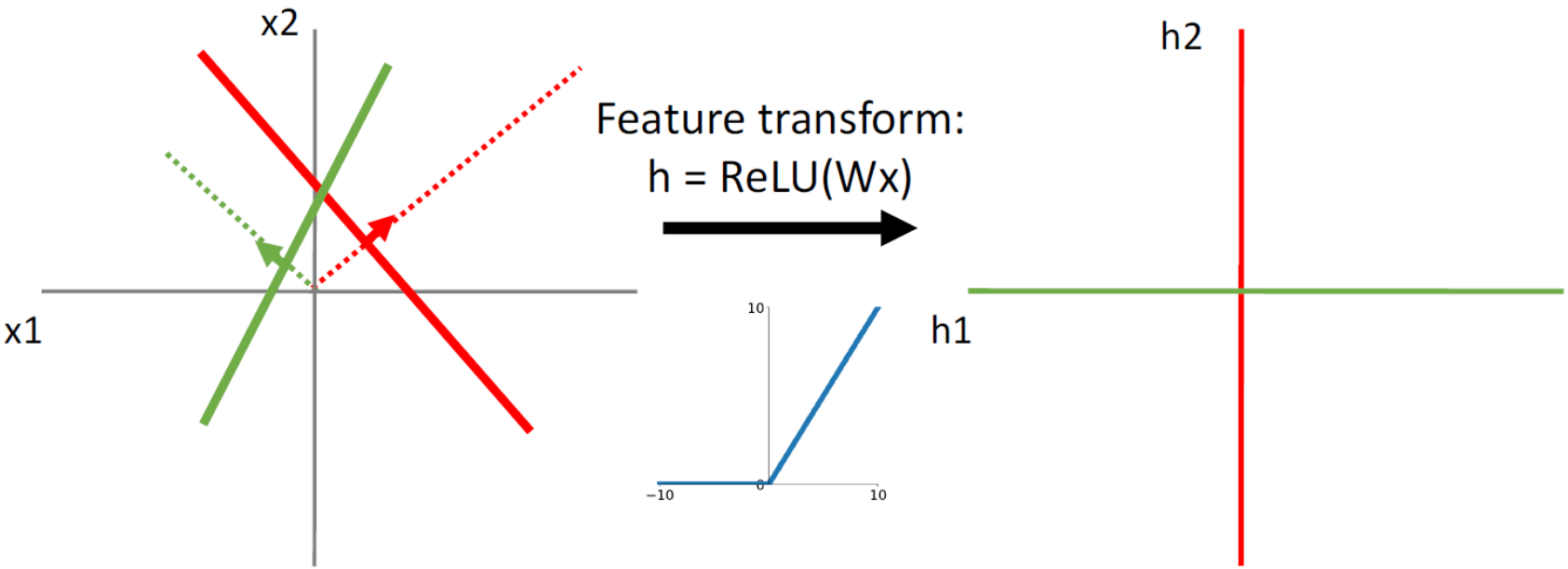
# Space warping

- Consider a linear transform: $\boldsymbol{h} = W\boldsymbol{x}$ Where $\boldsymbol{x}, \boldsymbol{h}$ are both 2-dimensional
- Points not linearly separable in original space
  - Not linearly separable in feature space



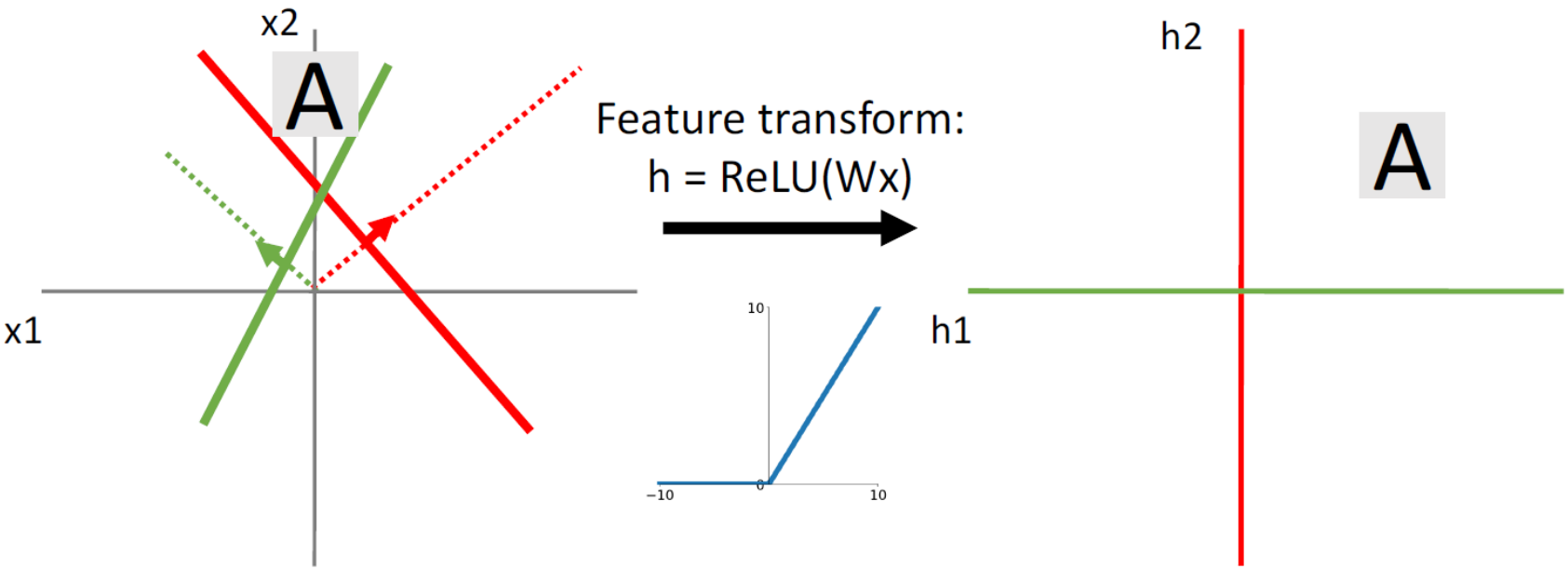
# Space warping

- Consider a neural net hidden layer: $\boldsymbol{h} = ReLU(W\boldsymbol{x}) = max(0, W\boldsymbol{x})$.
  - Where x, h are both 2-dimensional.



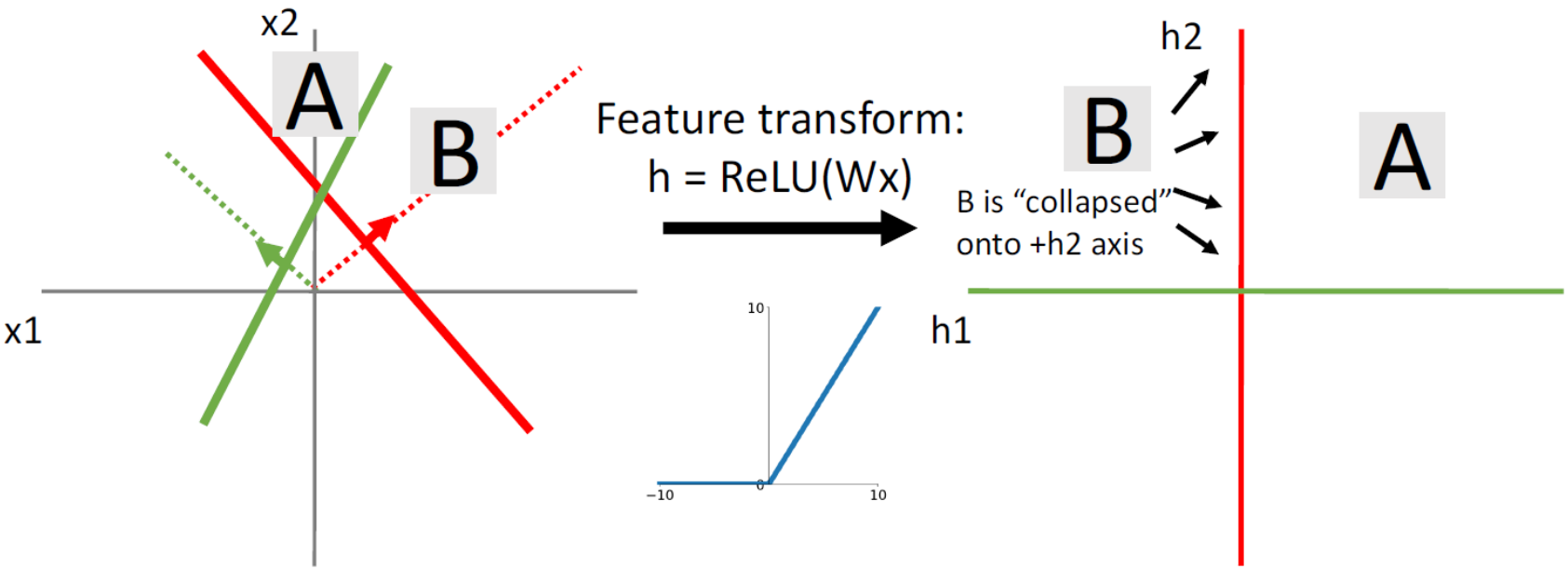
# Space warping

- Consider a neural net hidden layer: $\boldsymbol{h} = ReLU(W\boldsymbol{x}) = max(0, W\boldsymbol{x})$.
  - Where x, h are both 2-dimensional.



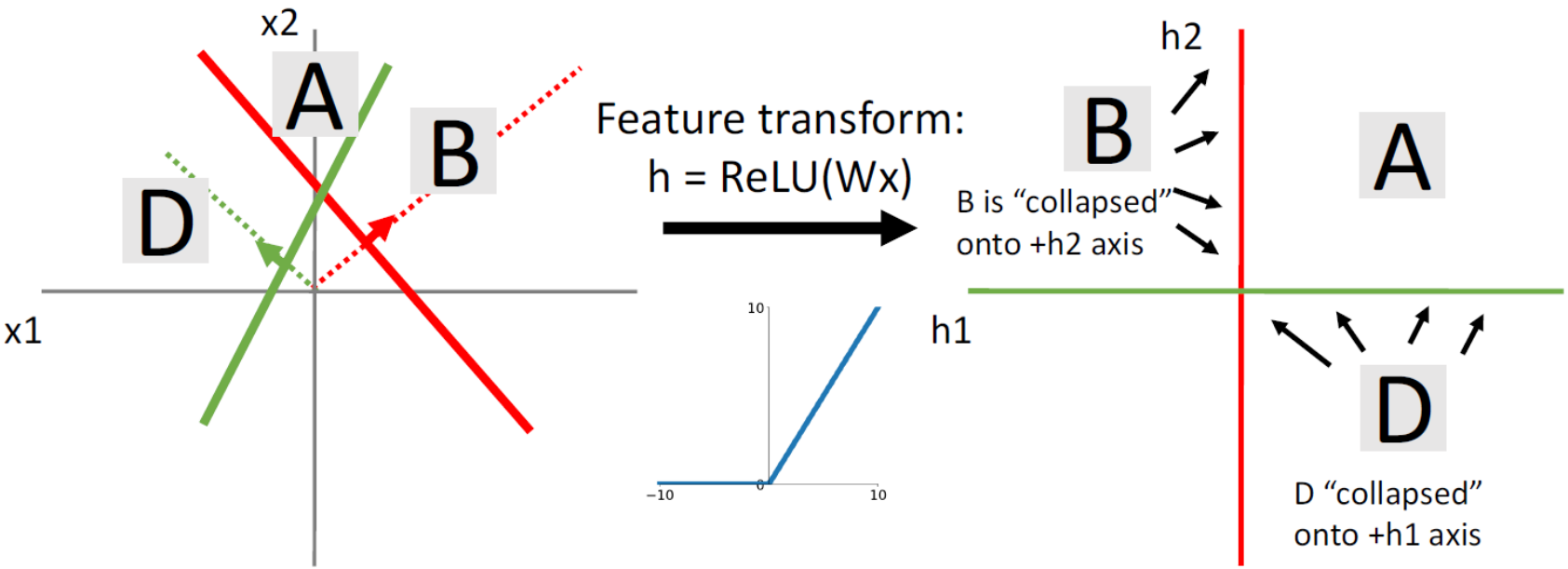
# Space warping

- Consider a neural net hidden layer: $\boldsymbol{h} = ReLU(W\boldsymbol{x}) = max(0, W\boldsymbol{x})$.
  - Where x, h are both 2-dimensional.



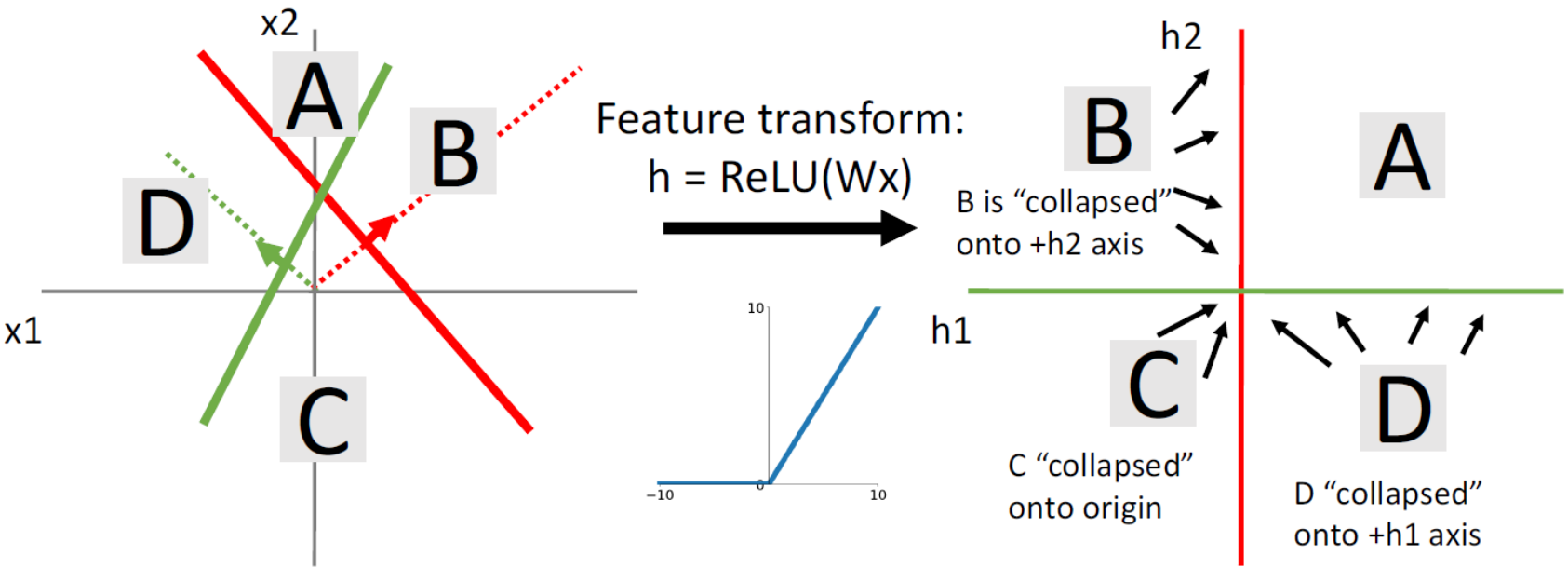
# Space warping

- Consider a neural net hidden layer: $\boldsymbol{h} = ReLU(W\boldsymbol{x}) = max(0, W\boldsymbol{x})$.
  - Where x, h are both 2-dimensional.



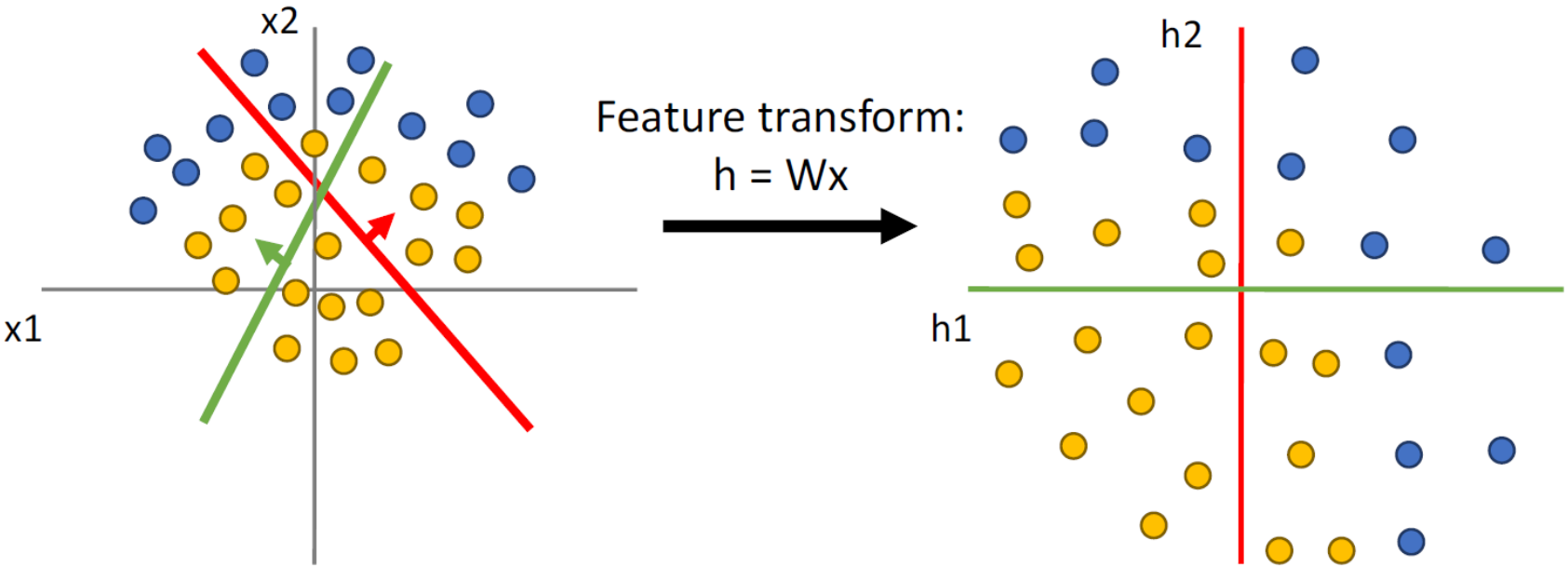
# Space warping

- Consider a neural net hidden layer: $\boldsymbol{h} = ReLU(W\boldsymbol{x}) = max(0, W\boldsymbol{x})$.
  - Where x, h are both 2-dimensional.



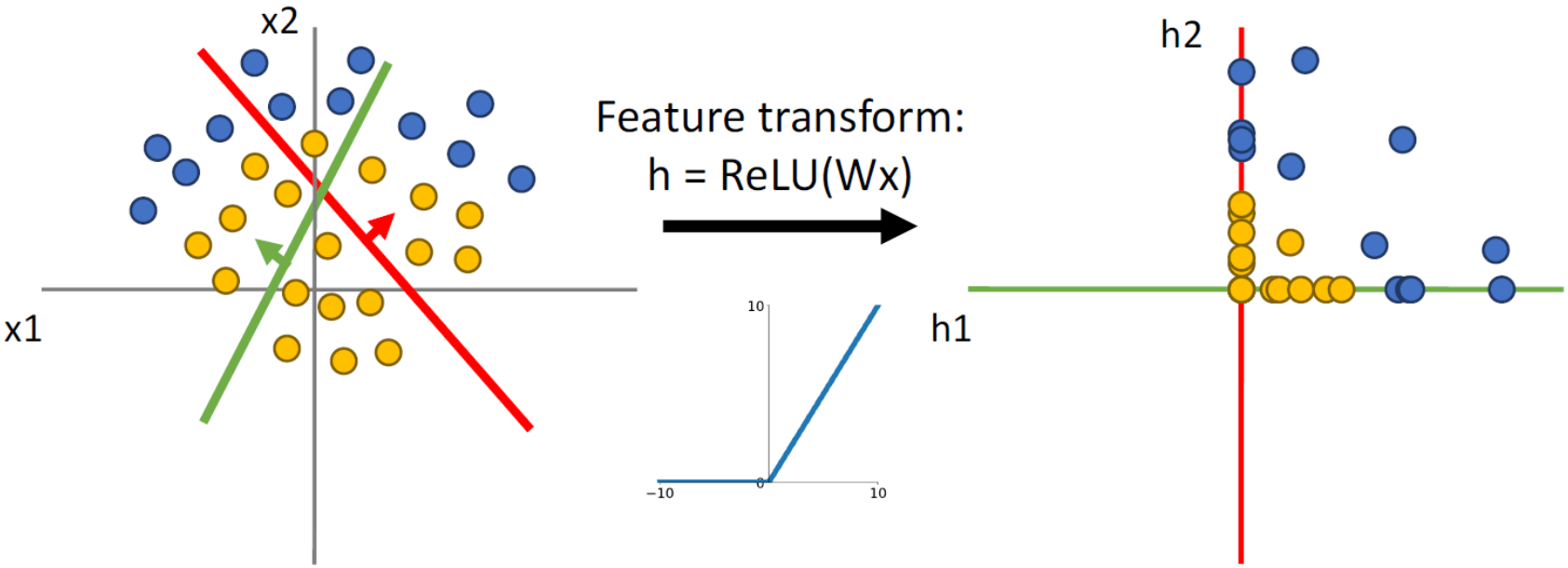
# Space warping

- Consider a neural net hidden layer: $\boldsymbol{h} = ReLU(W\boldsymbol{x}) = max(0, W\boldsymbol{x})$.
  - Where x, h are both 2-dimensional.
- Points not linearly separable in original space



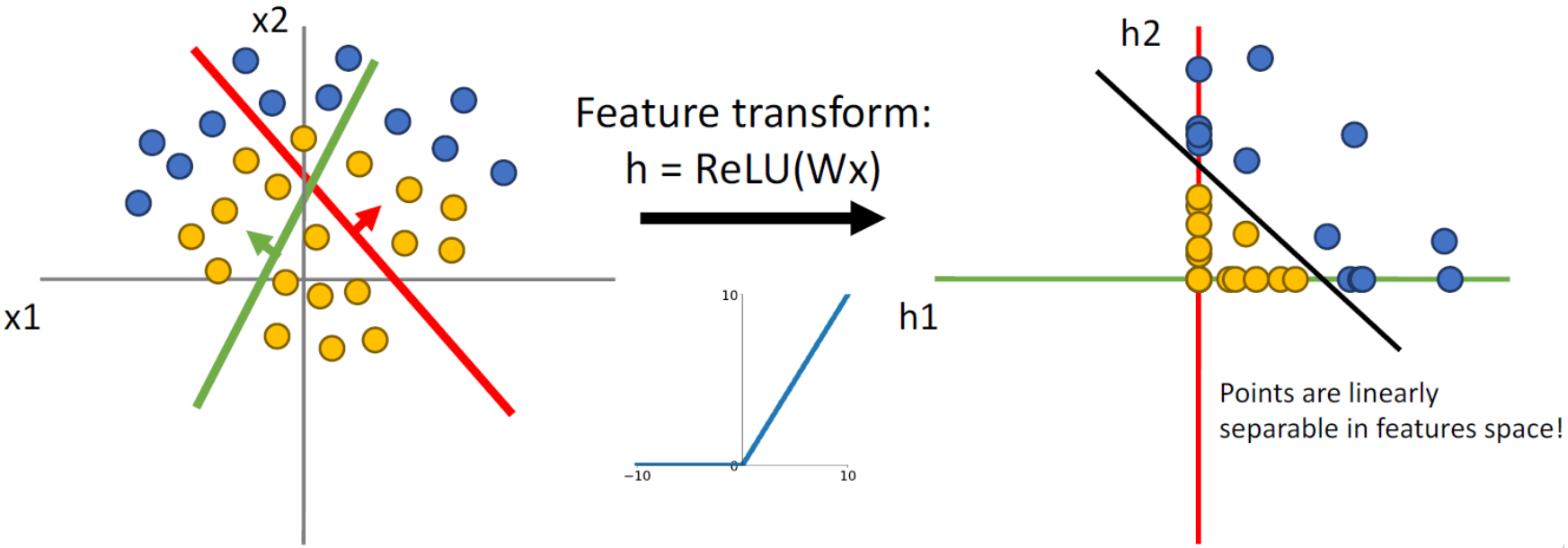
# Space warping

- Consider a neural net hidden layer: $\boldsymbol{h} = ReLU(W\boldsymbol{x}) = max(0, W\boldsymbol{x})$.
  - Where x, h are both 2-dimensional.
- Points not linearly separable in original space



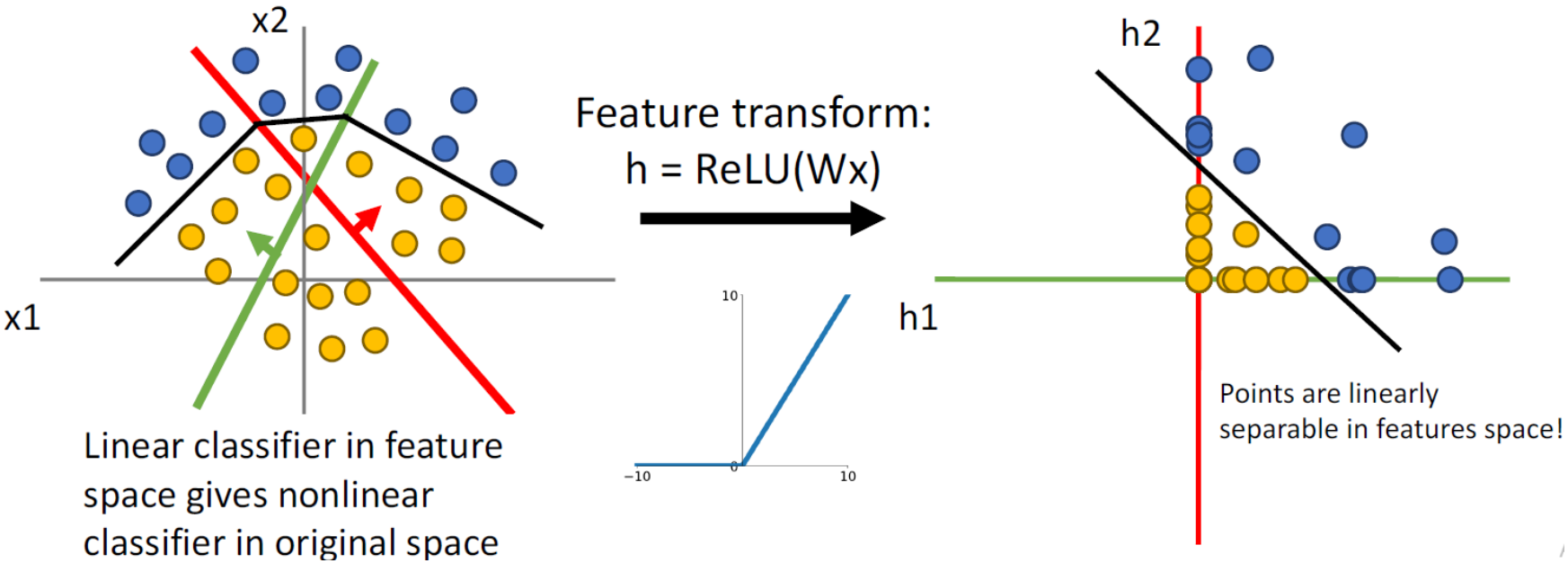
# Space warping

- Consider a neural net hidden layer: $\boldsymbol{h} = ReLU(W\boldsymbol{x}) = max(0, W\boldsymbol{x})$.
  - Where x, h are both 2-dimensional.
- Points not linearly separable in original space



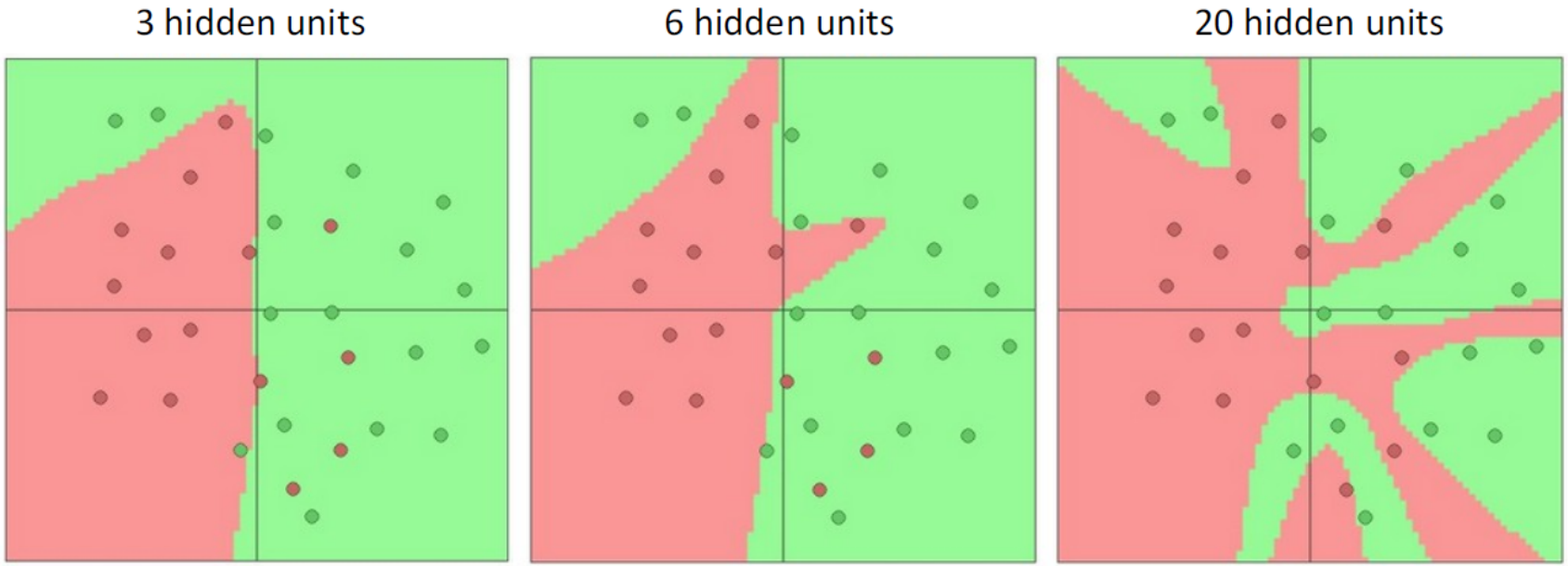
# Space warping

- Consider a neural net hidden layer: $\boldsymbol{h} = ReLU(W\boldsymbol{x}) = max(0, W\boldsymbol{x})$.
  - Where x, h are both 2-dimensional.
- Points not linearly separable in original space

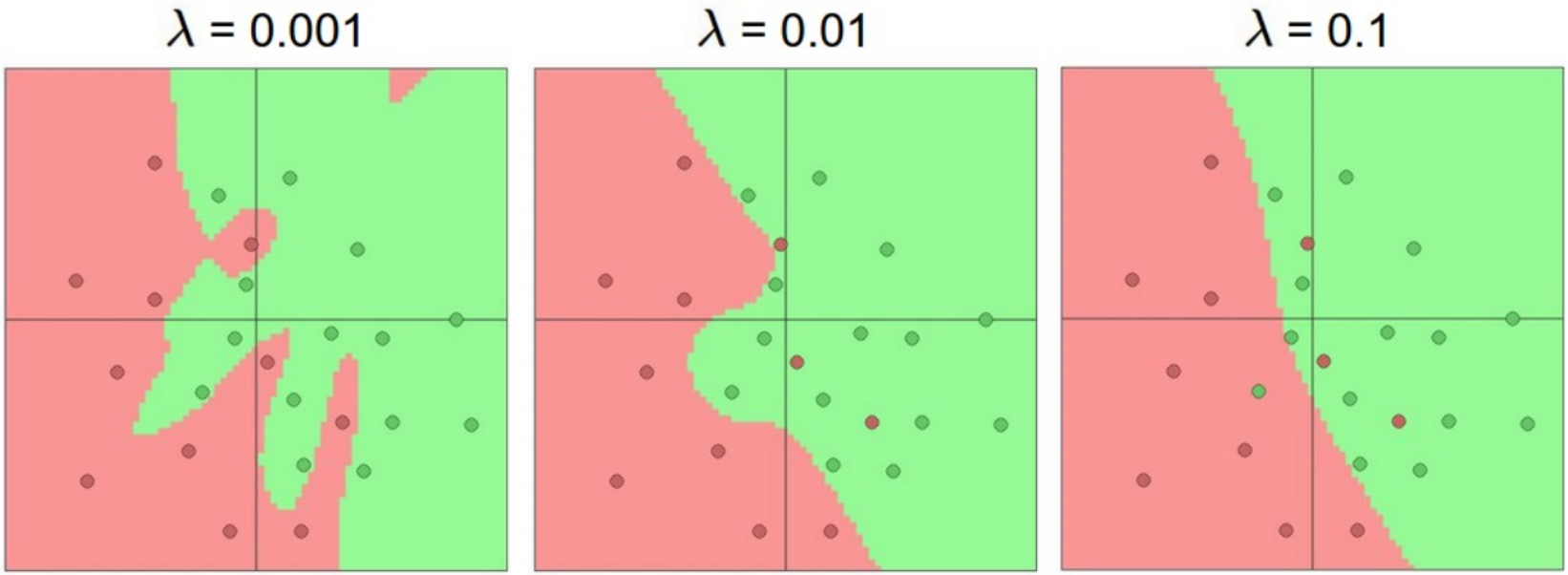


# Setting the number of layers and their sizes

**More hidden units = more capacity**

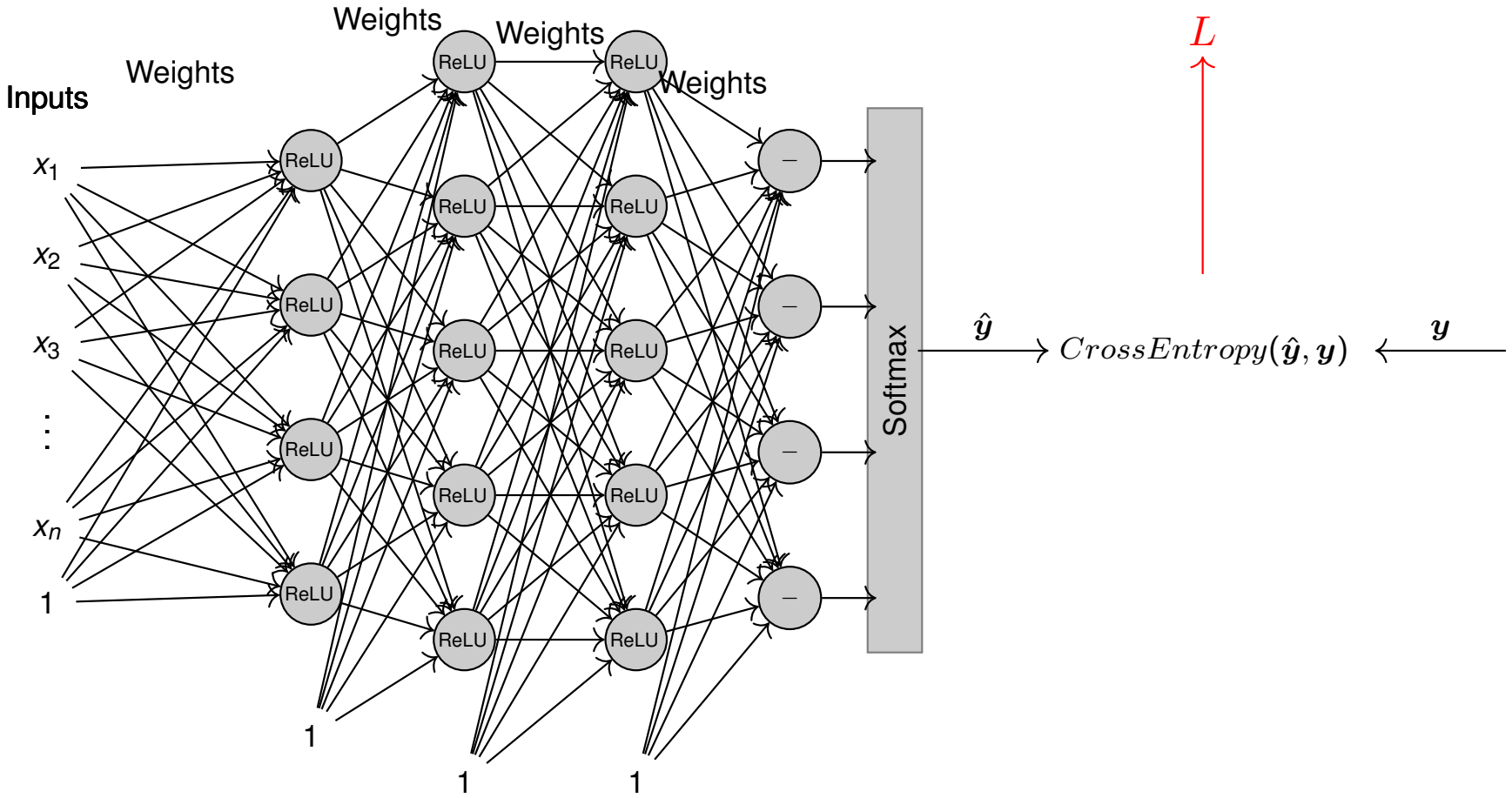


# Don’t regularize with size; instead use stronger L2




http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

# Fully connected neural network loss




$$\hat{y} = Softmax(W^4(max(0, W^3(max(0, W^2(max(0, W^1 x + b^1)) + b^2)) + b^3)) + b^4)$$

# Backpropagation

# Convex functions

- A function $f : X \subset \mathbb{R}^N \to \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0, 1]$,

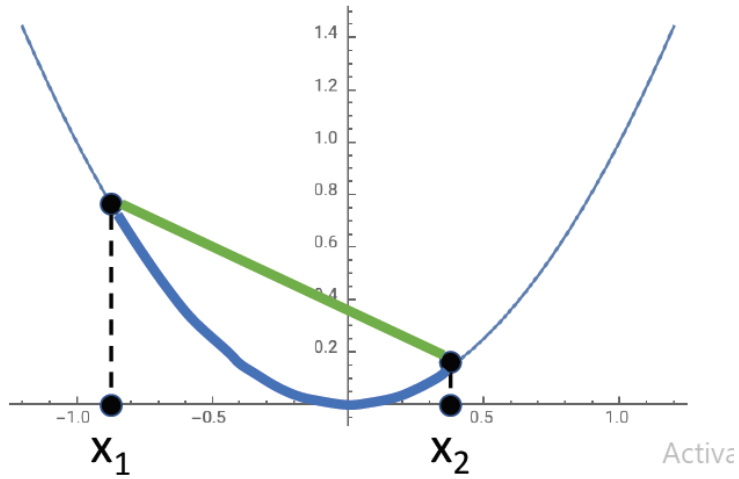
$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$

# Convex functions

- A function $f : X \subset \mathbb{R}^N \to \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0, 1]$,

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$

Example: $f(x) = x^2$ is convex:

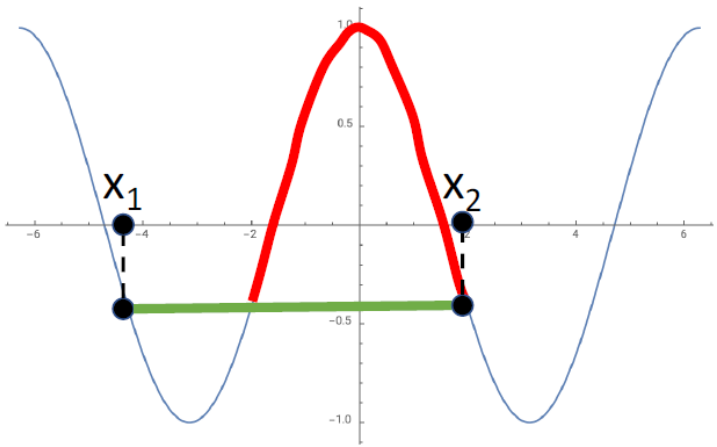


# Convex functions

- A function $f : X \subset \mathbb{R}^N \to \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0, 1]$,

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$

Example: $f(x) = \cos(x)$ is **not** convex:

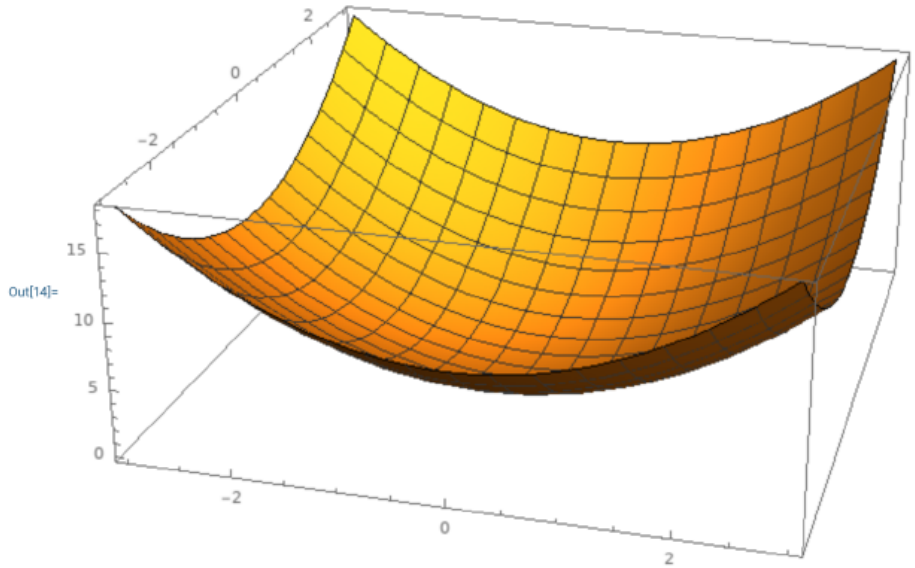


# Convex functions

- A function $f : X \subset \mathbb{R}^N \to \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0, 1]$,

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$

- **Intuition**: A convex function is a (multi-dimensional) bowl
- Generally speaking, convex functions are easy to optimize: can derive theoretical guarantees about converging to global minimum (Many technical details!).

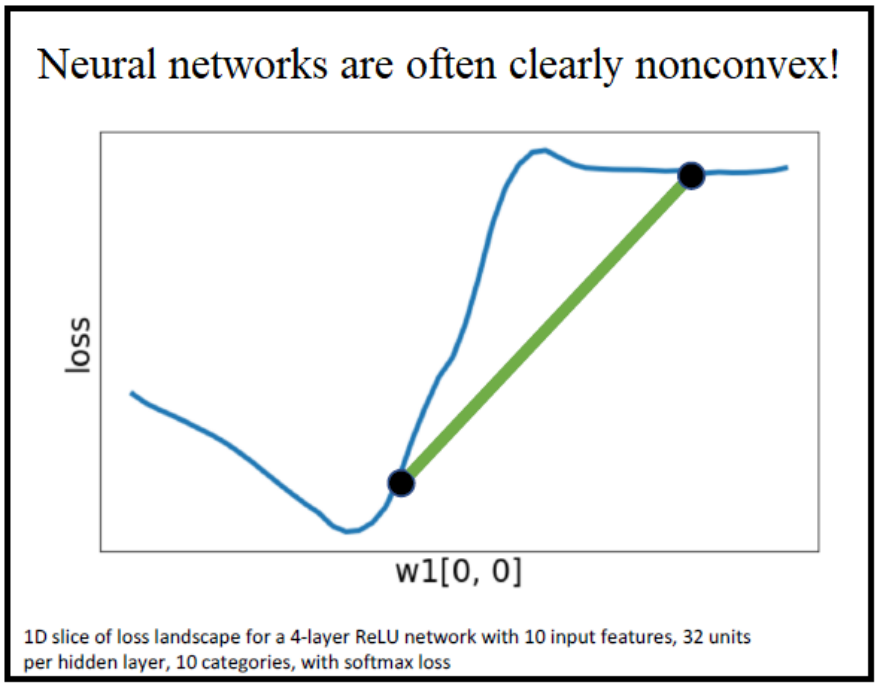


# Convex functions

- A function $f : X \subset \mathbb{R}^N \to \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0, 1]$,

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$

- **Intuition**: A convex function is a (multi-dimensional) bowl
- Generally speaking, convex functions are easy to optimize: can derive theoretical guarantees about converging to global minimum (Many technical details!).

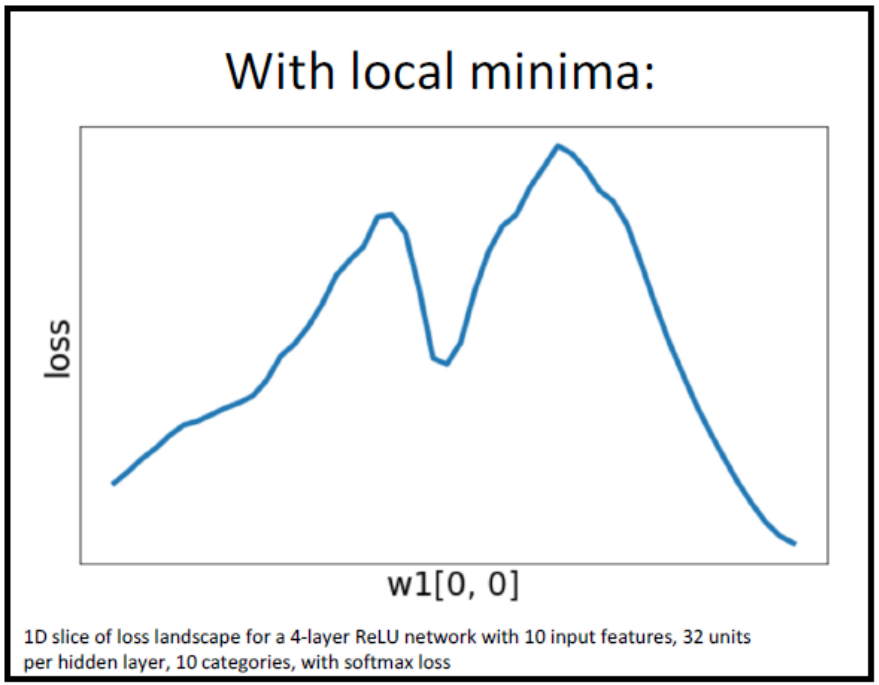


# Convex functions

- A function $f : X \subset \mathbb{R}^N \to \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0, 1]$,

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$

- **Intuition**: A convex function is a (multi-dimensional) bowl
- Generally speaking, convex functions are easy to optimize: can derive theoretical guarantees about converging to global minimum (Many technical details!).

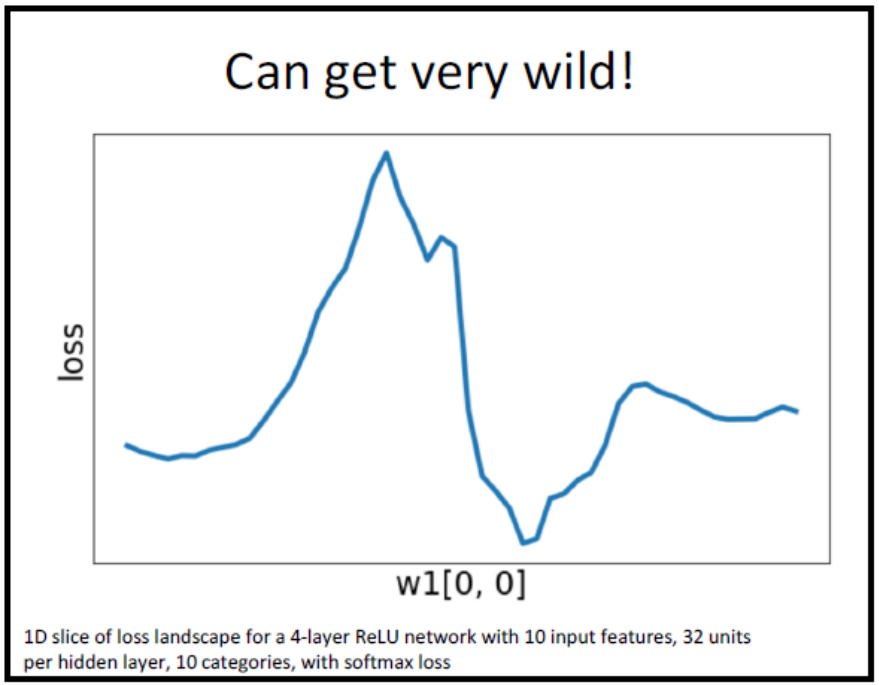


# Convex functions

- A function $f : X \subset \mathbb{R}^N \to \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0, 1]$,

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$

- **Intuition**: A convex function is a (multi-dimensional) bowl
- Generally speaking, convex functions are easy to optimize: can derive theoretical guarantees about converging to global minimum (Many technical details!).



1D slice of loss landscape for a 4-layer ReLU network with 10 input features, 32 units per hidden layer, 10 categories, with softmax loss

# Convex functions

- A function $f : X \subset \mathbb{R}^N \to \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0, 1]$,

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$

- **Intuition**: A convex function is a (multi-dimensional) bowl
- Generally speaking, convex functions are easy to optimize: can derive theoretical guarantees about converging to global minimum (Many technical details!).

Most neural networks need **nonconvex optimization**

- Few or no guarantees about convergence
- Empirically it seems to work anyway
- Active area of research

# Gradient descent

- The gradient descent method to find the minimum of a function iteratively

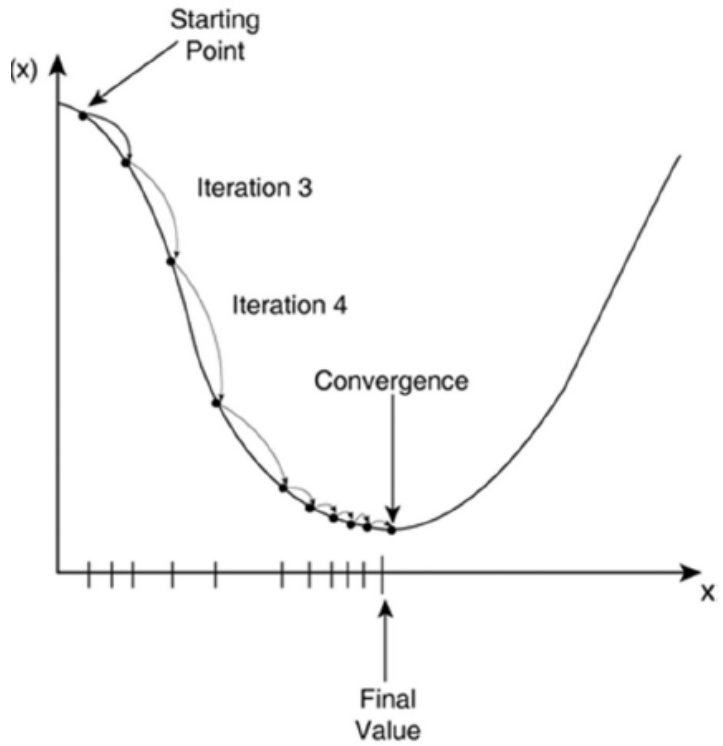
$$\boldsymbol{x}^{t+1} = \boldsymbol{x}^t - \eta \nabla_{\boldsymbol{x}} f(\boldsymbol{x}^t)$$

  - $\eta$ is the "step size" (Also called "learning rate")

- The gradient descent algorithm converges when the following criterion is satisfied.

$$|f(\boldsymbol{x}^{t+k}) - f(\boldsymbol{x}^t)| < \epsilon$$

  - $k$ is a hyperparameter.



# Training neural nets through gradient descent

- Total training loss

$$L(f(X, W), \boldsymbol{y}) = \frac{1}{N} \sum_i L_i(f(\boldsymbol{x}^{(i)}, W), y^{(i)})$$

- Gradient descent algorithm:
  - Initialize all weights and biases $w_{ij}^{(\ell)}$
    - Using the extended notation: the bias is also a weight
  - Do:
    - For every layer $\ell$ for all $i$, $j$ update:

$$w_{ij}^{(\ell)} = w_{ij}^{(\ell)} - \eta \frac{\partial L}{\partial w_{ij}^{(\ell)}}$$

  - Until loss has converged

# The derivative

- Total training Loss:

$$L(f(X,W),\boldsymbol{y}) = \frac{1}{N}\sum_i L_i(f(\boldsymbol{x}^{(i)},W),y^{(i)})$$

- Computing the derivative

$$\frac{\partial L(f(X,W),\boldsymbol{y})}{\partial w_{ij}^{(\ell)}} = \frac{1}{N}\sum_i \frac{\partial L_i(f(\boldsymbol{x}^{(i)},W),y^{(i)})}{\partial w_{ij}^{(\ell)}}$$

- So we must first figure out how to compute the derivative of divergences of individual training inputs

# Training by gradient descent

- Initialize all weights $w_{ij}^{(\ell)}$
- Do
  - For all $i, j, \ell$, initialize $\frac{\partial L}{\partial w_{ij}^{(\ell)}} = 0$
  - For all $n = 1 : N$
    - For every $\ell$ for all $i, j$:

      Compute $\frac{\partial L_n}{\partial w_{ij}^{(\ell)}}$

      $$\frac{\partial L}{\partial w_{ij}^{(\ell)}} += \frac{1}{N} \frac{\partial L_n}{\partial w_{ij}^{(\ell)}}$$
  - For every $\ell$ for all $i, j$:
    $$w_{ij}^{(\ell)} = w_{ij}^{(\ell)} - \eta \frac{\partial L}{w_{ij}^{(\ell)}}$$
- Until loss has converged

# Calculus refresher: chain rule

For any nested function $y = f(g(x))$

$$\frac{dy}{dx} = \frac{df}{dg(x)} \frac{dg(x)}{dx}$$

Check - we can confirm that : $\Delta y = \frac{dy}{dx} \Delta x$

$z = g(x) \Rightarrow \Delta z = \frac{dg(x)}{dx} \Delta x$

$y = f(z) \Rightarrow \Delta y = \frac{df}{dz} \Delta z = \frac{df}{dg(x)} \frac{dg(x)}{dx} \Delta x$ ✓

# Calculus refresher: distributed chain rule

$$y = f\left(g_1(x), g_2(x), \ldots, g_M(x)\right)$$

$$\frac{dy}{dx} = \frac{\partial f}{\partial g_1(x)}\frac{dg_1(x)}{dx} + \frac{\partial f}{\partial g_2(x)}\frac{dg_2(x)}{dx} + \cdots + \frac{\partial f}{\partial g_M(x)}\frac{dg_M(x)}{dx}$$

Check: $\Delta y = \frac{dy}{dx}\Delta x$ Let $z_i = g_i(x)$

$$\Delta y = \frac{\partial f}{\partial z_1}\Delta z_1 + \frac{\partial f}{\partial z_2}\Delta z_2 + \cdots + \frac{\partial f}{\partial z_M}\Delta z_M$$

$$\Delta y = \frac{\partial f}{\partial z_1}\frac{dz_1}{dx}\Delta x + \frac{\partial f}{\partial z_2}\frac{dz_2}{dx}\Delta x + \cdots + \frac{\partial f}{\partial z_M}\frac{dz_M}{dx}\Delta x$$

$$\Delta y = \left(\frac{\partial f}{\partial g_1(x)}\frac{dg_1(x)}{dx} + \frac{\partial f}{\partial g_2(x)}\frac{dg_2(x)}{dx} + \cdots + \frac{\partial f}{\partial g_M(x)}\frac{dg_M(x)}{dx}\right)\Delta x$$

# Calculus refresher: distributed chain rule

$$y = f\left(g_1(x), g_2(x), \dots, g_M(x)\right)$$

$$\frac{dy}{dx} = \frac{\partial f}{\partial g_1(x)}\frac{dg_1(x)}{dx} + \frac{\partial f}{\partial g_2(x)}\frac{dg_2(x)}{dx} + \cdots + \frac{\partial f}{\partial g_M(x)}\frac{dg_M(x)}{dx}$$

Check: $\Delta y = \frac{dy}{dx}\Delta x$

$$\Delta y = \frac{\partial f}{\partial g_1(x)}\Delta g_1(x) + \frac{\partial f}{\partial g_2(x)}\Delta g_2(x) + \cdots + \frac{\partial f}{\partial g_M(x)}\Delta g_M(x)$$

$$\Delta y = \frac{\partial f}{\partial g_1(x)}\frac{dg_1(x)}{dx}\Delta x + \frac{\partial f}{\partial g_2(x)}\frac{dg_2(x)}{dx}\Delta x + \cdots + \frac{\partial f}{\partial g_M(x)}\frac{dg_M(x)}{dx}\Delta x$$

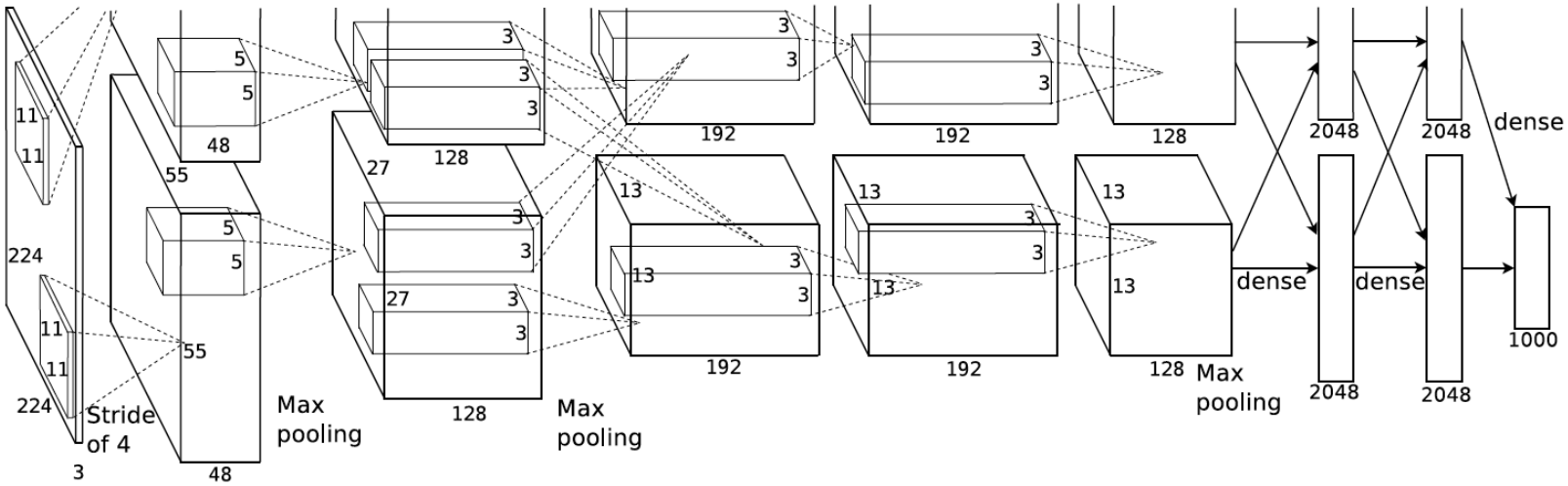
$$\Delta y = \left(\frac{\partial f}{\partial g_1(x)}\frac{dg_1(x)}{dx} + \frac{\partial f}{\partial g_2(x)}\frac{dg_2(x)}{dx} + \cdots + \frac{\partial f}{\partial g_M(x)}\frac{dg_M(x)}{dx}\right)\Delta x$$

# Idea: derive $\frac{\partial L}{\partial W}$ on paper

- Problems
  - Very tedious: Lots of matrix calculus, need lots of paper
  - What if we want to change loss? Need to re-derive from scratch. Not modular!
  - Not feasible for very complex models!

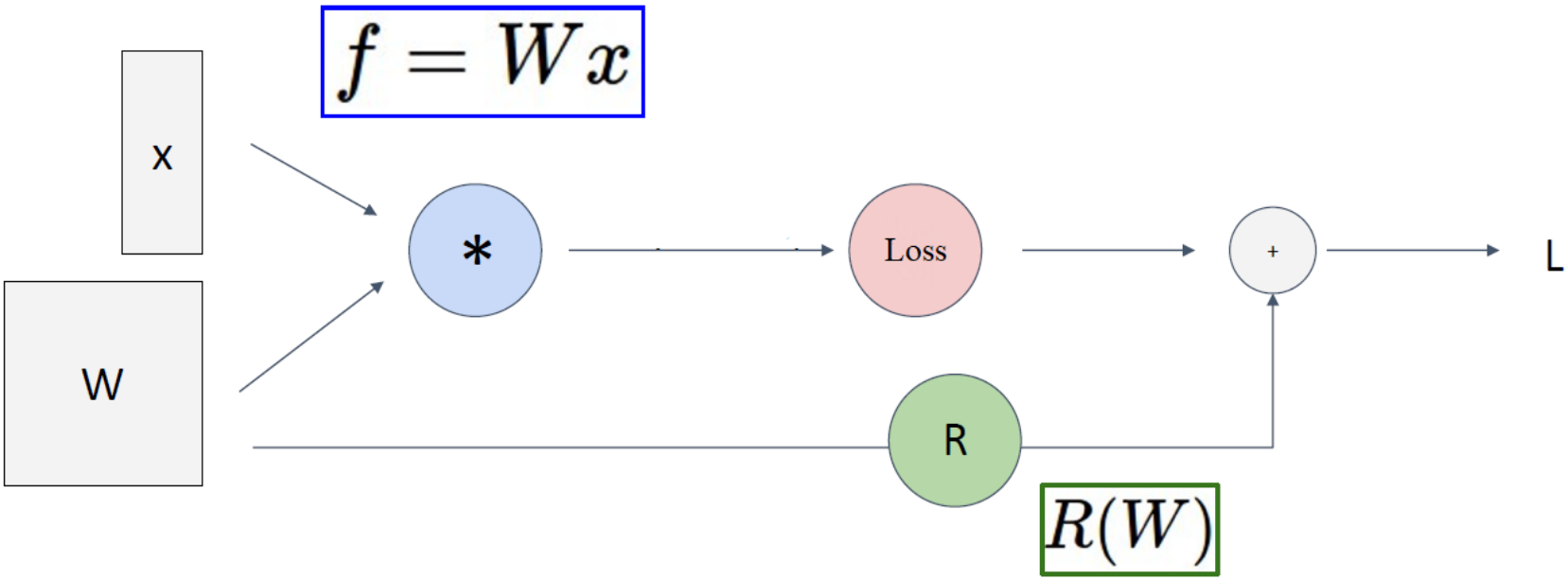
# Idea: derive $\frac{\partial L}{\partial W}$ on paper

- Problems
  - Very tedious: Lots of matrix calculus, need lots of paper
  - What if we want to change loss? Need to re-derive from scratch. Not modular!
  - Not feasible for very complex models!



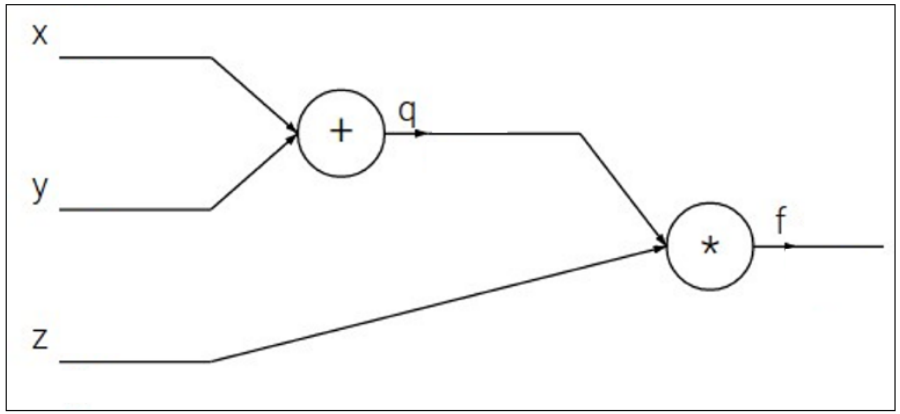

AlexNet has about 660K units, 61M parameters,

# Better Idea: Computational Graphs



# Backpropagation: simple example

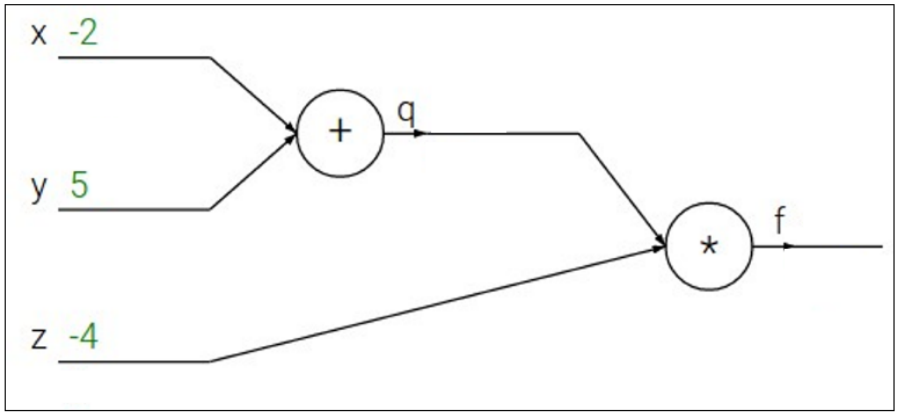
$$f(x, y, z) = (x + y)z$$



# Backpropagation: simple example

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

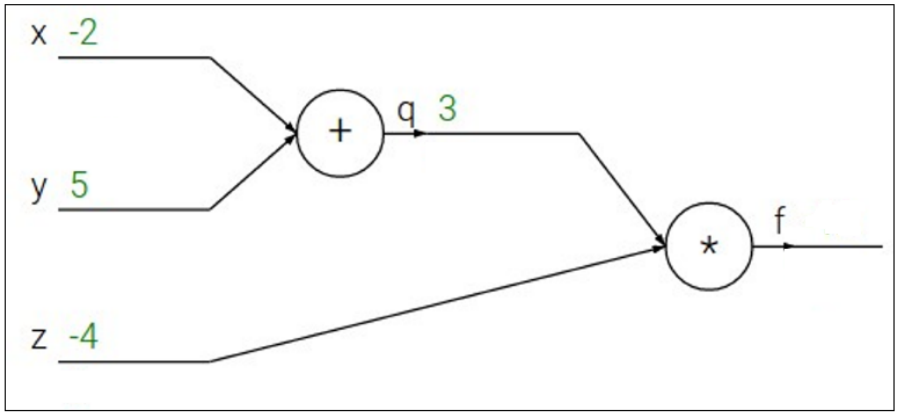
e.g. x = -2, y = 5, z = -4



# Backpropagation: simple example

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

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




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

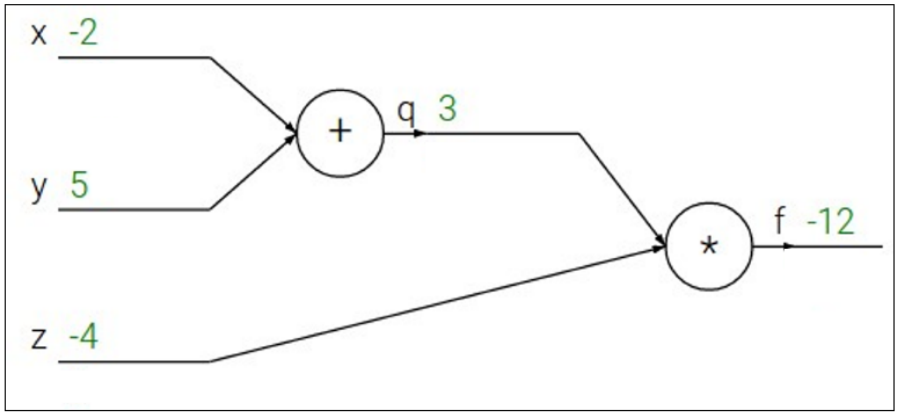
# Backpropagation: simple example

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

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

**1. Forward pass**: Compute outputs

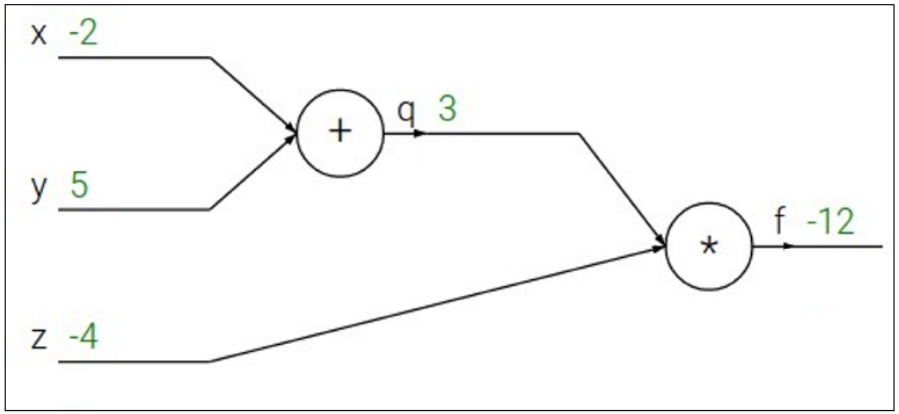
$$q = x + y \qquad f = qz$$



# Backpropagation: simple example

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

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




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

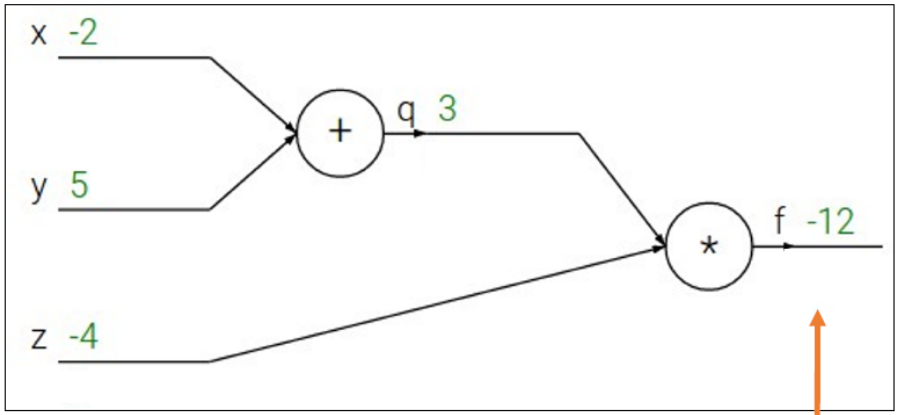
**2. Backward pass**: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

# Backpropagation: simple example

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

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




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

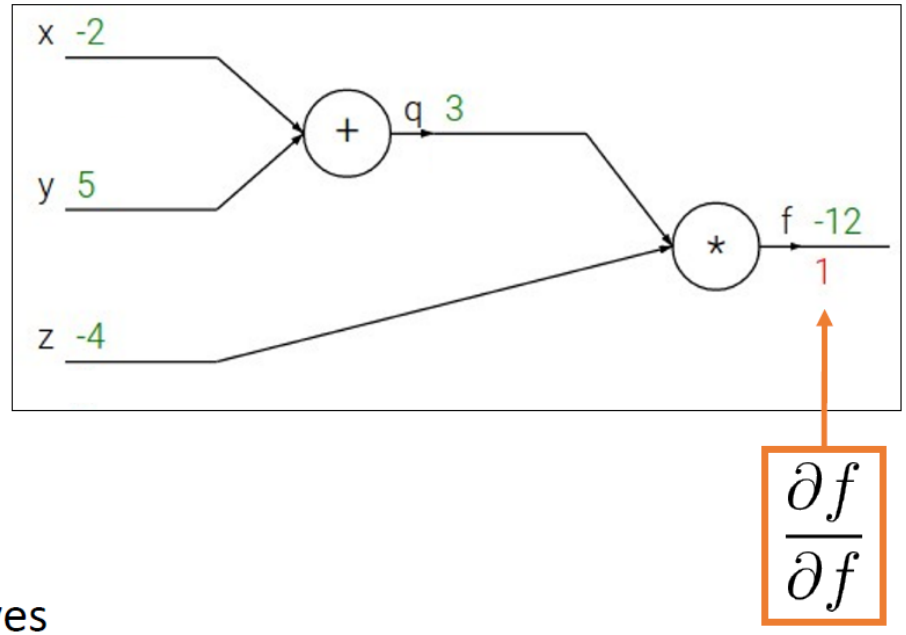
**2. Backward pass**: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

# Backpropagation: simple example

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

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




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

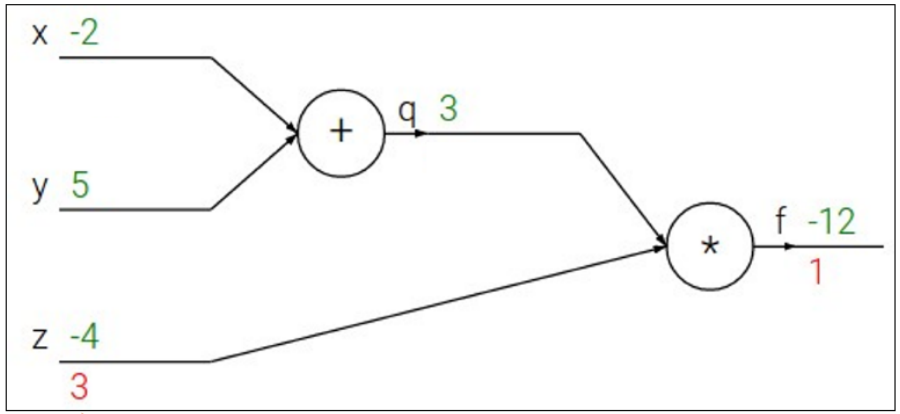
**2. Backward pass**: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

# Backpropagation: simple example

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

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




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

**2. Backward pass**: Compute derivatives

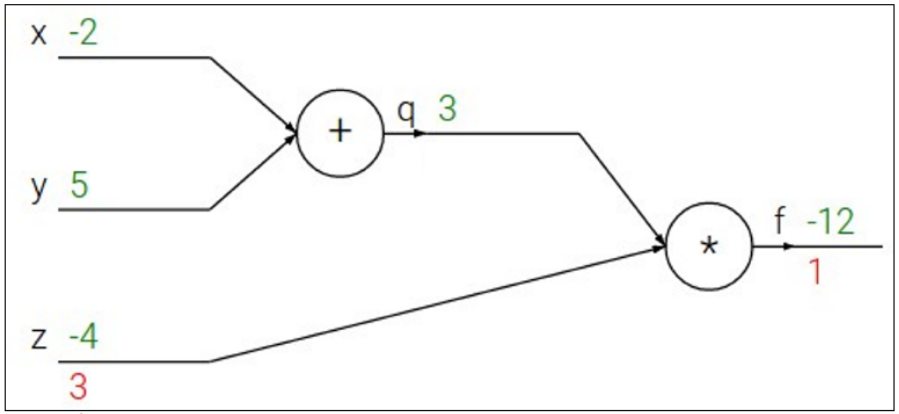
$$\frac{\partial f}{\partial z}$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

# Backpropagation: simple example

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

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




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

**2. Backward pass**: Compute derivatives

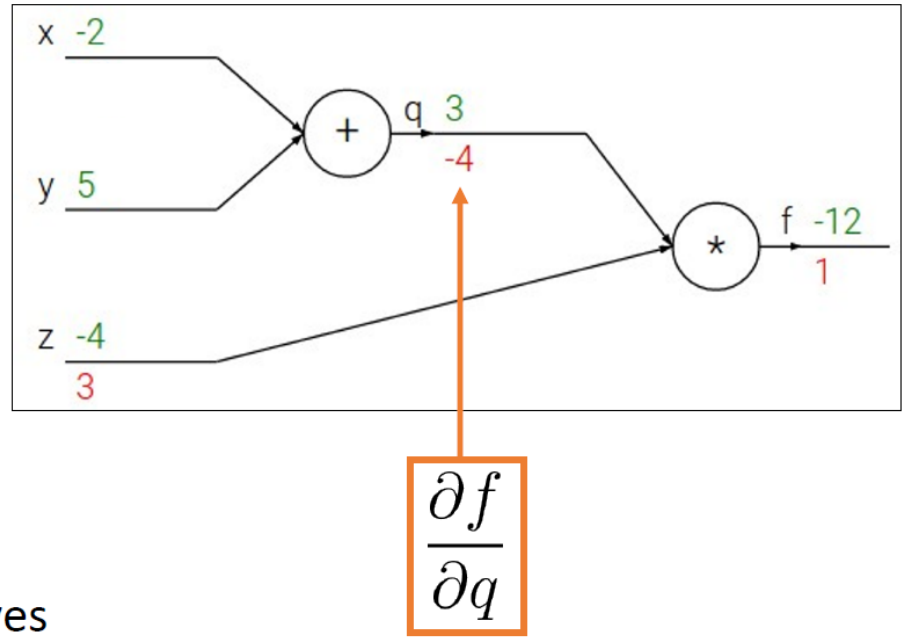
$$\frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

# Backpropagation: simple example

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

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




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

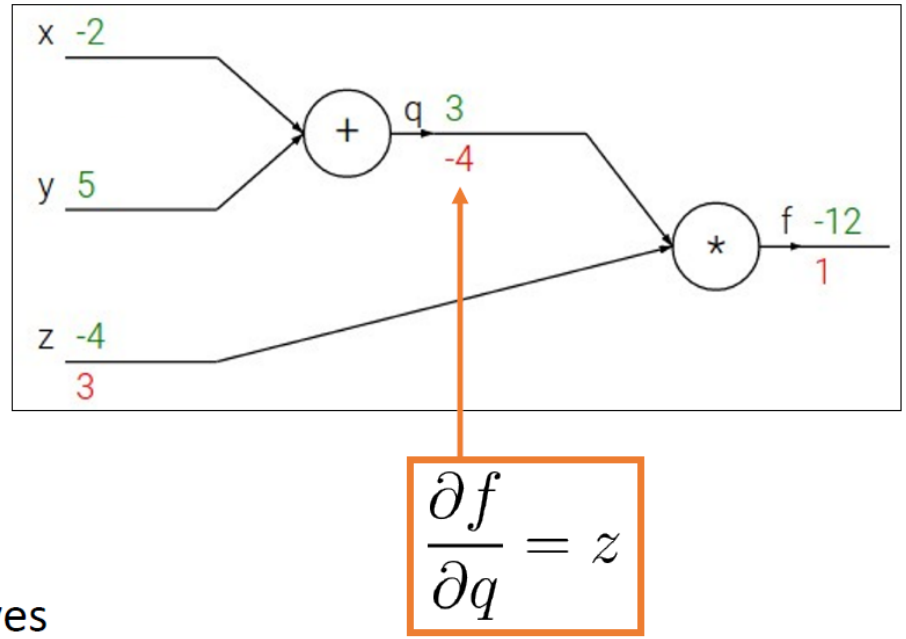
**2. Backward pass**: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

# Backpropagation: simple example

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

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




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

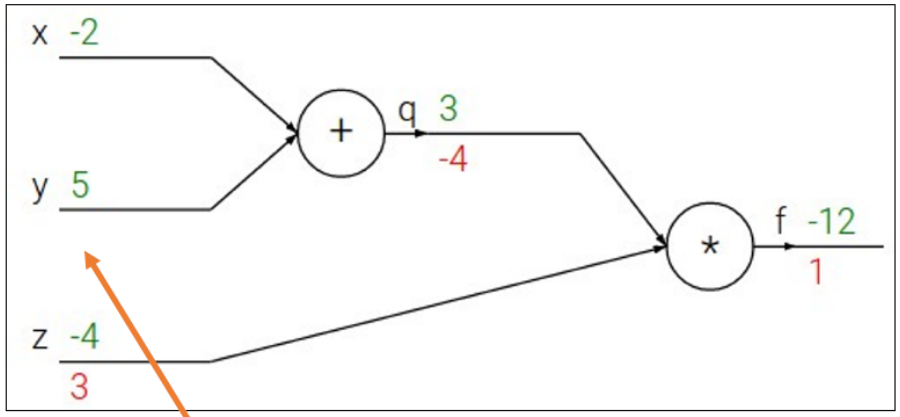
**2. Backward pass**: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

# Backpropagation: simple example

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

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




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

**2. Backward pass**: Compute derivatives

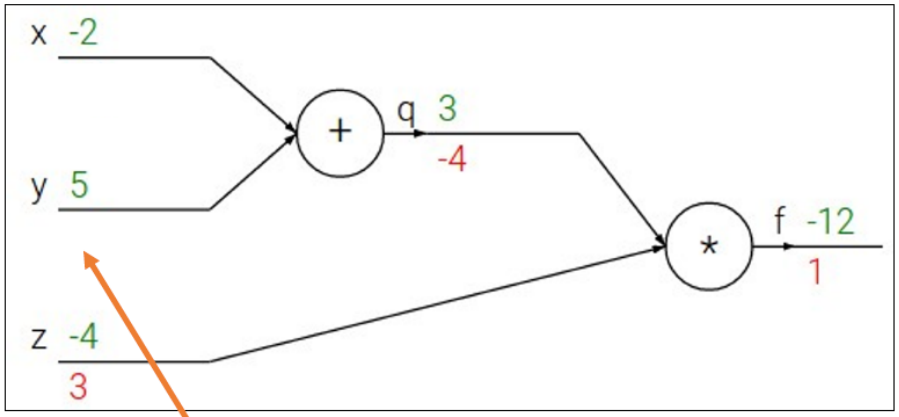
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

$\frac{\partial f}{\partial y}$

# Backpropagation: simple example

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

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




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

**2. Backward pass**: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

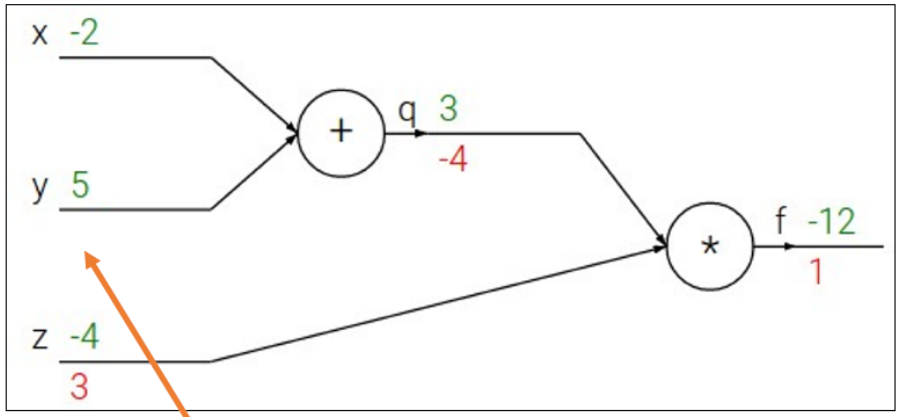
**Chain Rule**

$$\frac{\partial f}{\partial y} = \frac{\partial q}{\partial y}\frac{\partial f}{\partial q}$$

# Backpropagation: simple example

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

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




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

**2. Backward pass**: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

**Chain Rule**

$$\frac{\partial f}{\partial y} = \frac{\partial q}{\partial y}\frac{\partial f}{\partial q}$$

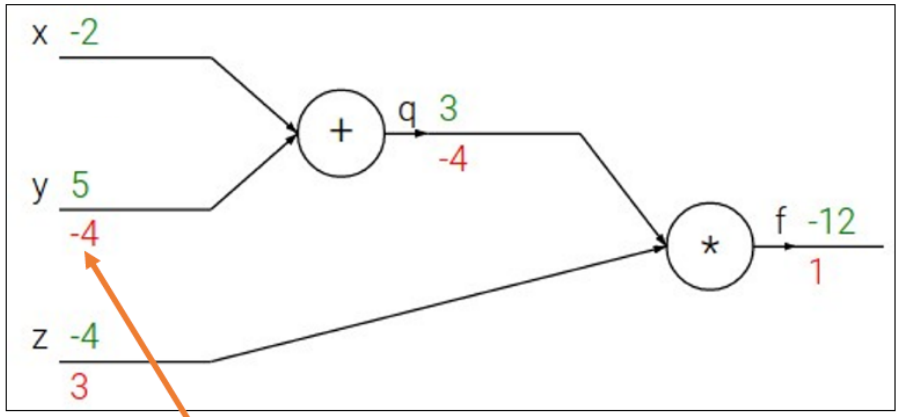
$$\frac{\partial q}{\partial y} = 1$$

**Downstream Gradient** **Local Gradient** **Upstream Gradient**

# Backpropagation: simple example

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

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




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

**2. Backward pass**: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

**Chain Rule**

$$\frac{\partial f}{\partial y} = \frac{\partial q}{\partial y}\frac{\partial f}{\partial q}$$

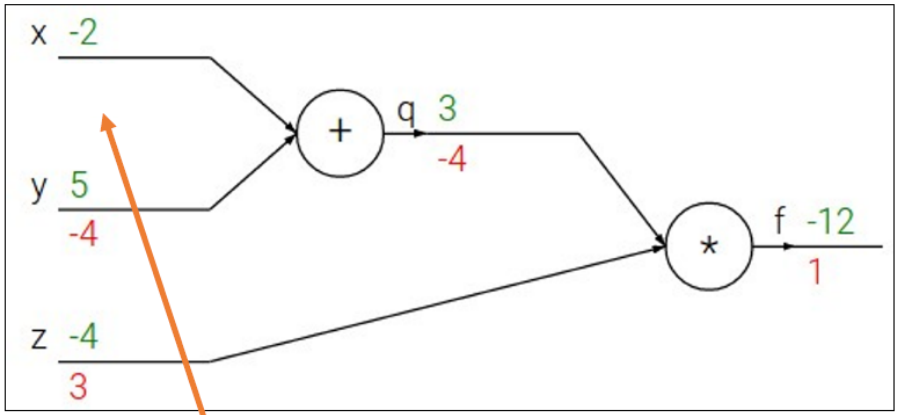
$$\frac{\partial q}{\partial y} = 1$$

**Downstream Gradient** **Local Gradient** **Upstream Gradient**

# Backpropagation: simple example

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

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




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

**2. Backward pass**: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

**Chain Rule**

$$\frac{\partial f}{\partial x} = \frac{\partial q}{\partial x}\frac{\partial f}{\partial q}$$

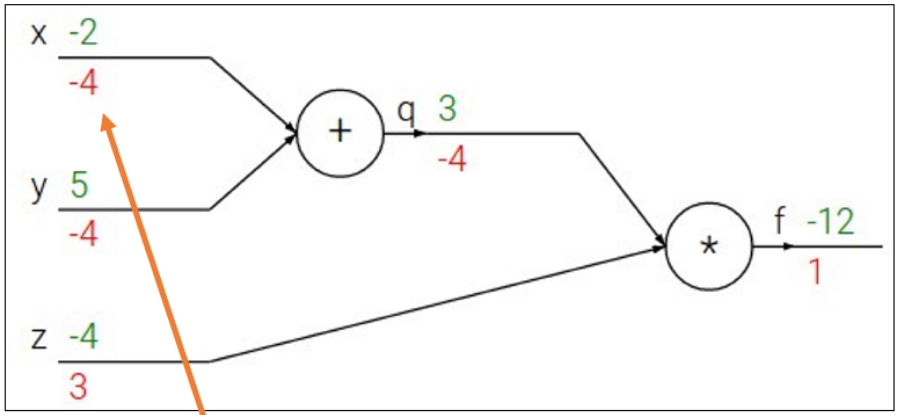
$$\frac{\partial q}{\partial x} = 1$$

**Downstream Gradient** **Local Gradient** **Upstream Gradient**

# Backpropagation: simple example

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

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




**1. Forward pass**: Compute outputs

$$q = x + y \qquad f = qz$$

**2. Backward pass**: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

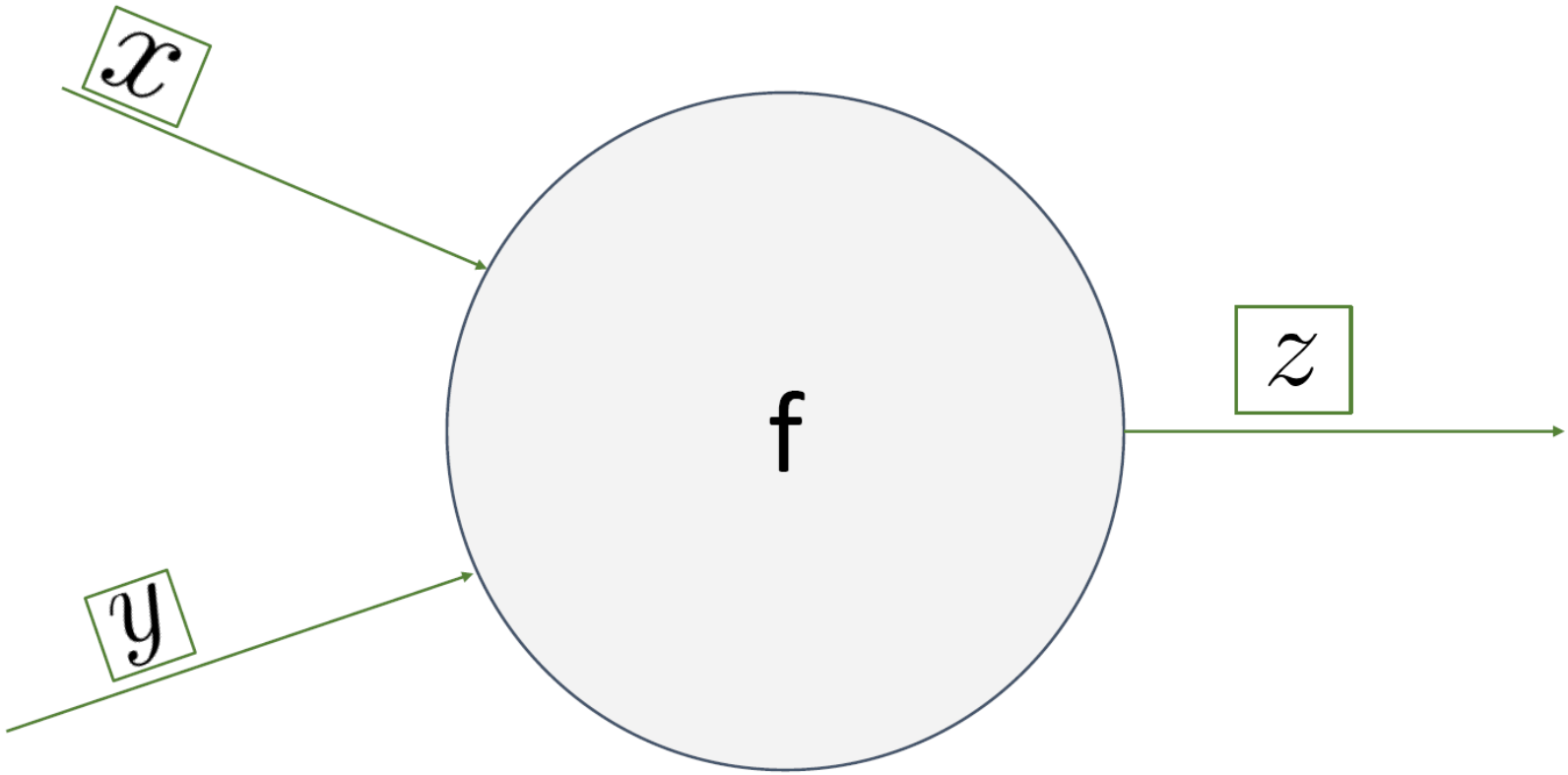
**Chain Rule**

$$\frac{\partial f}{\partial x} = \frac{\partial q}{\partial x}\frac{\partial f}{\partial q}$$

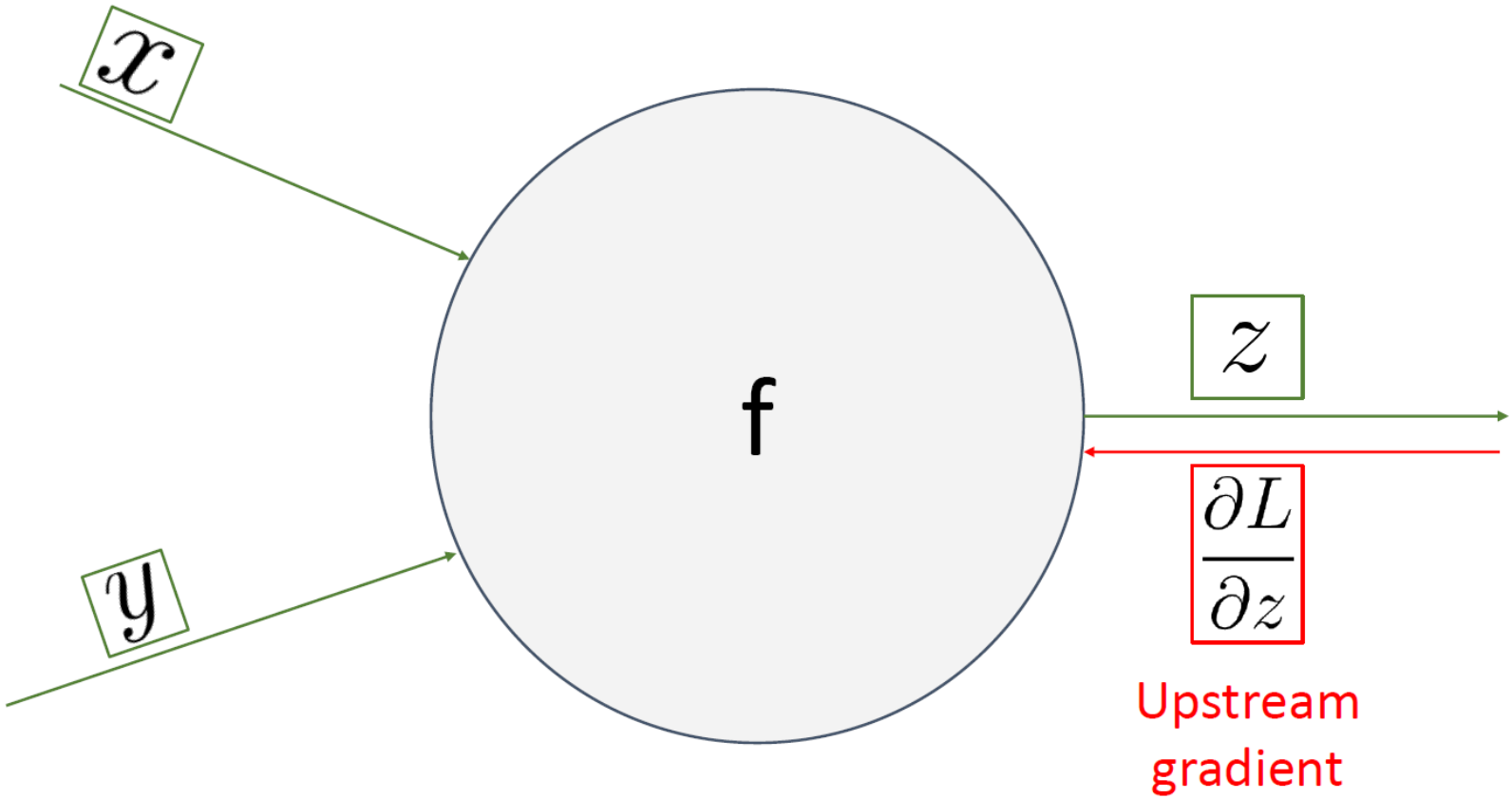
$$\frac{\partial q}{\partial x} = 1$$

**Downstream Gradient** **Local Gradient** **Upstream Gradient**

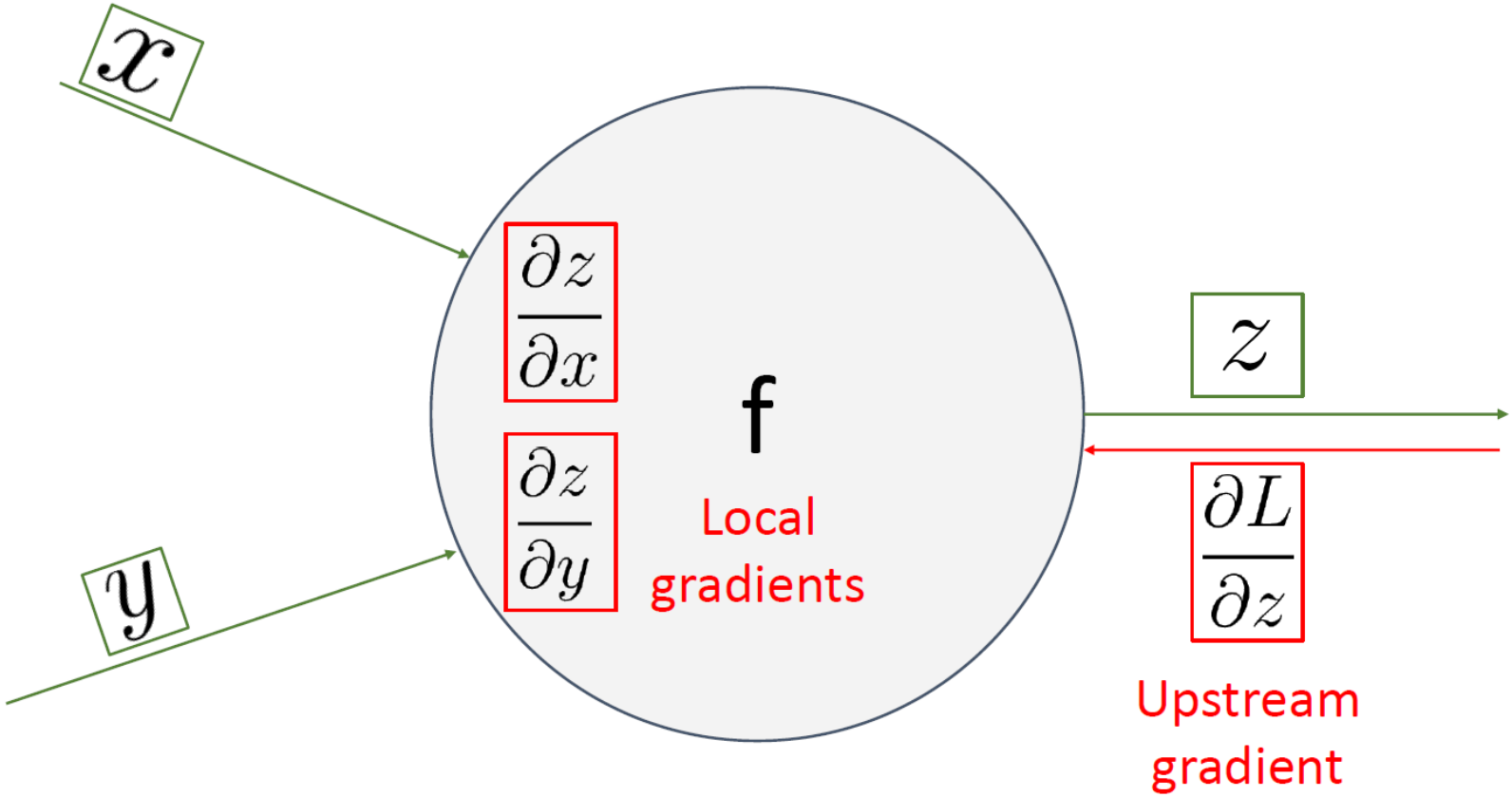
# Up/Down stream gradients



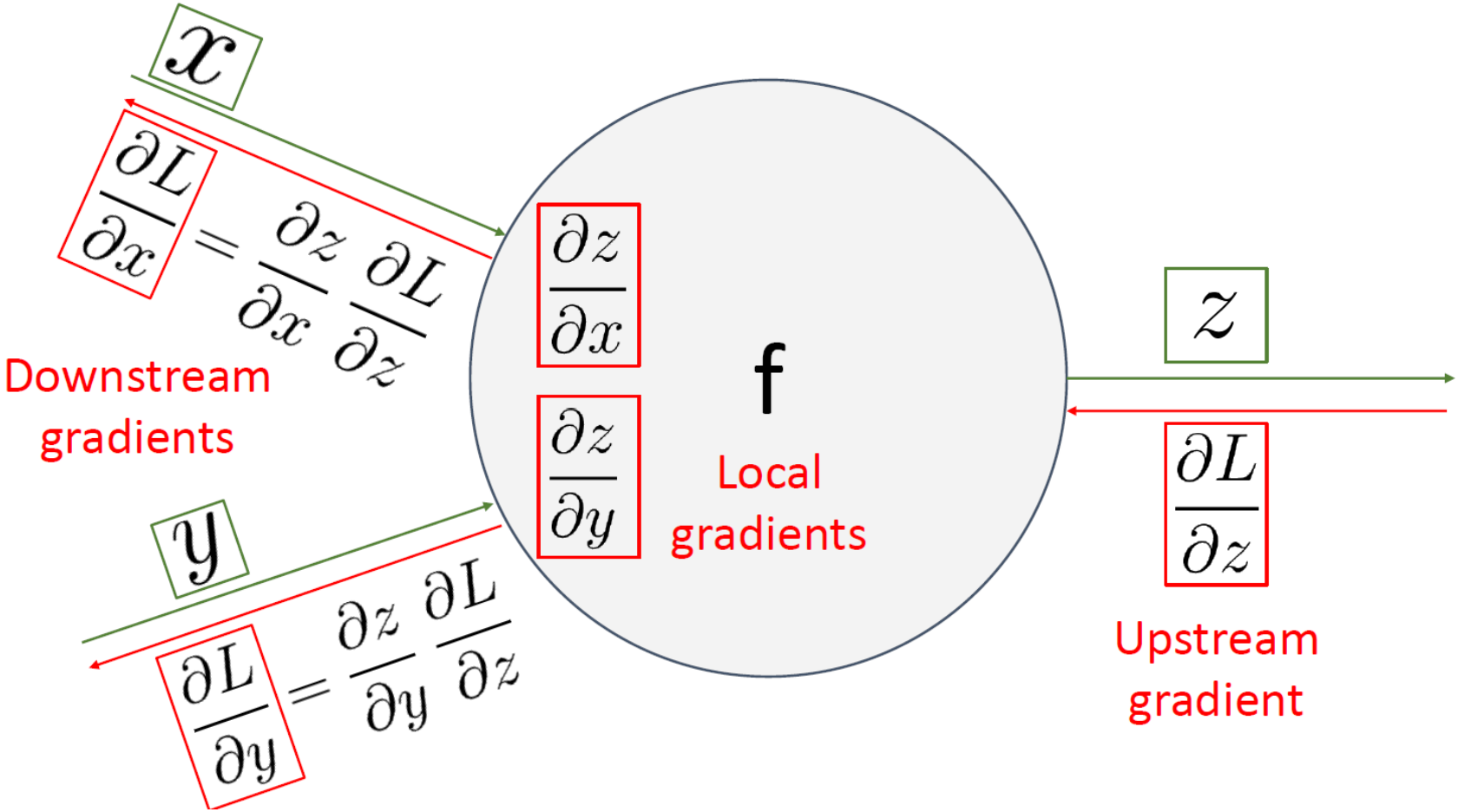
# Up/Down stream gradients



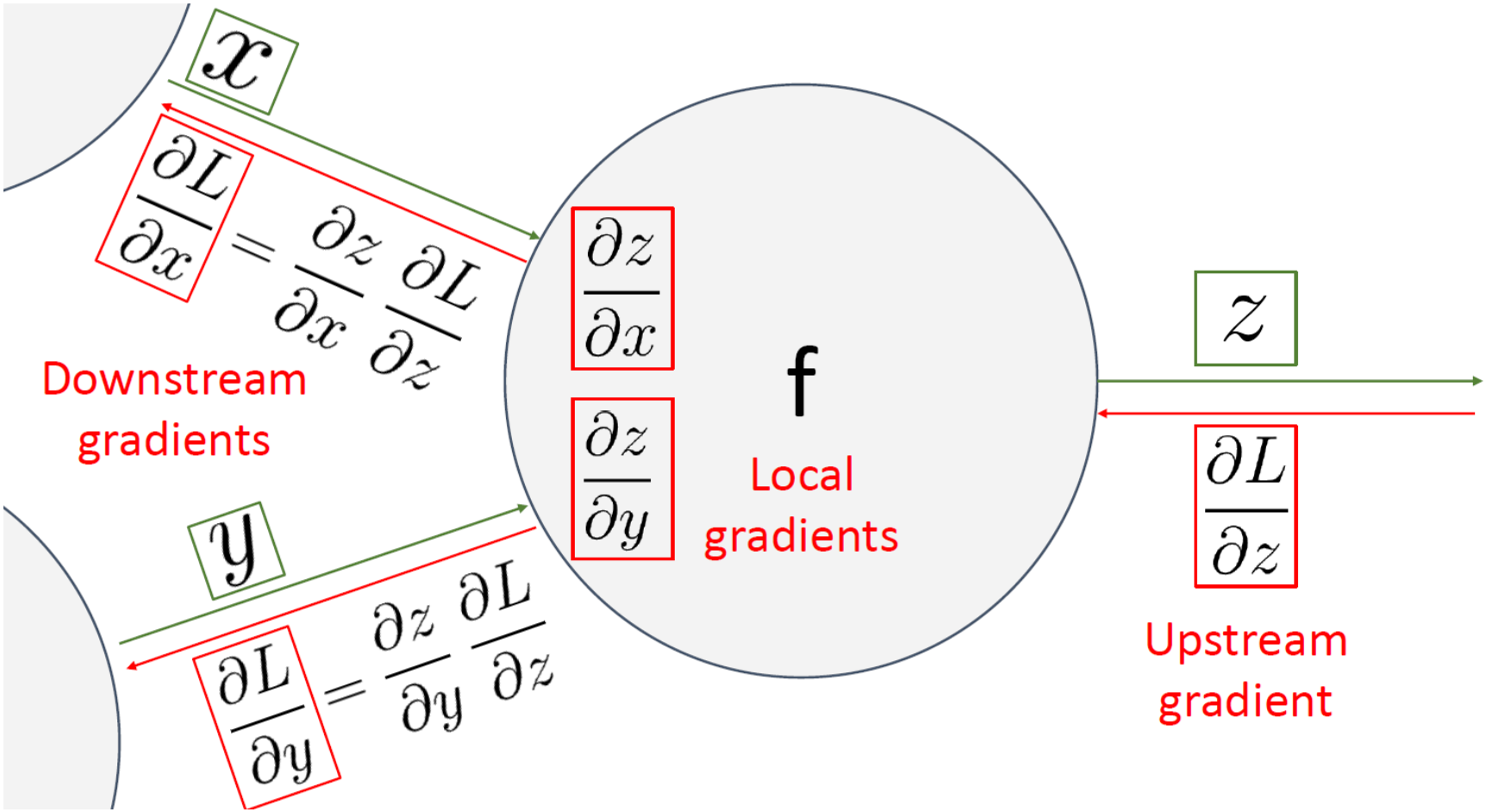
# Up/Down stream gradients



# Up/Down stream gradients

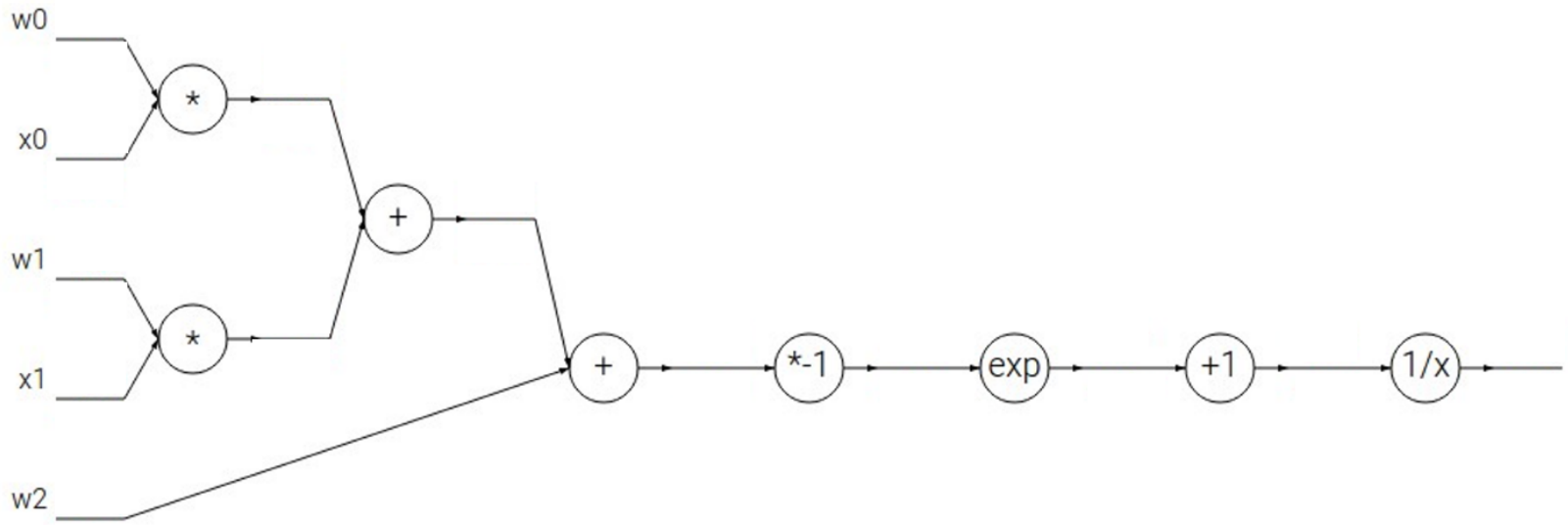


# Up/Down stream gradients



# Another Example

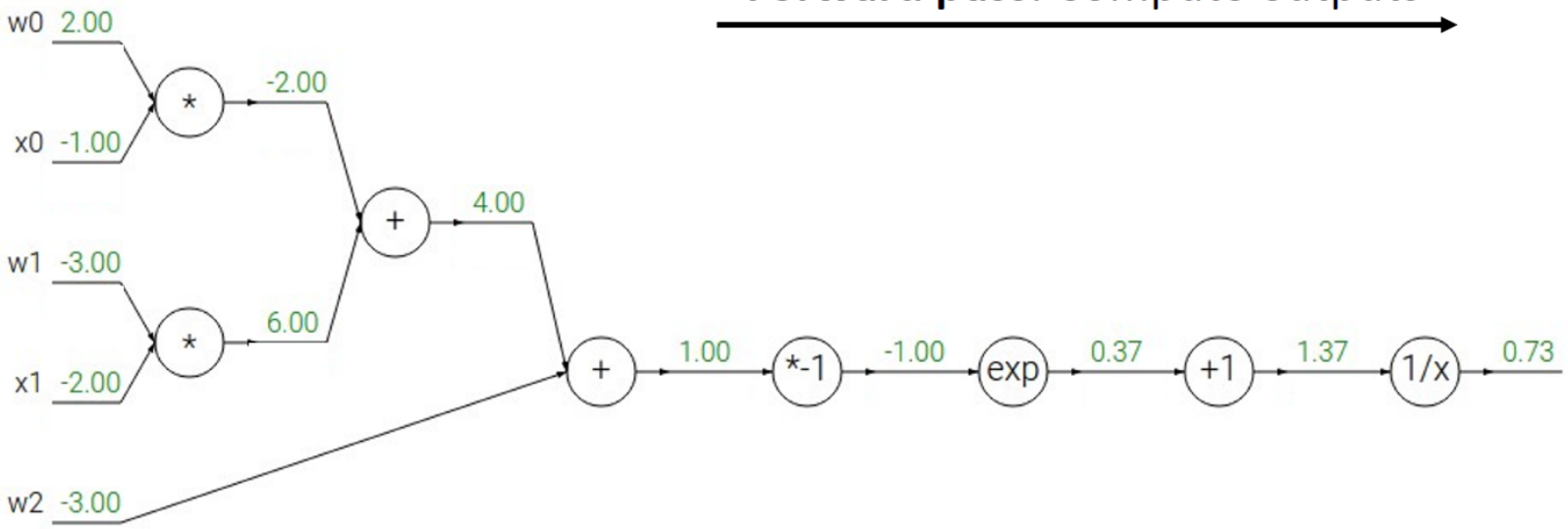
$$f(x, w) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



# Another Example

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

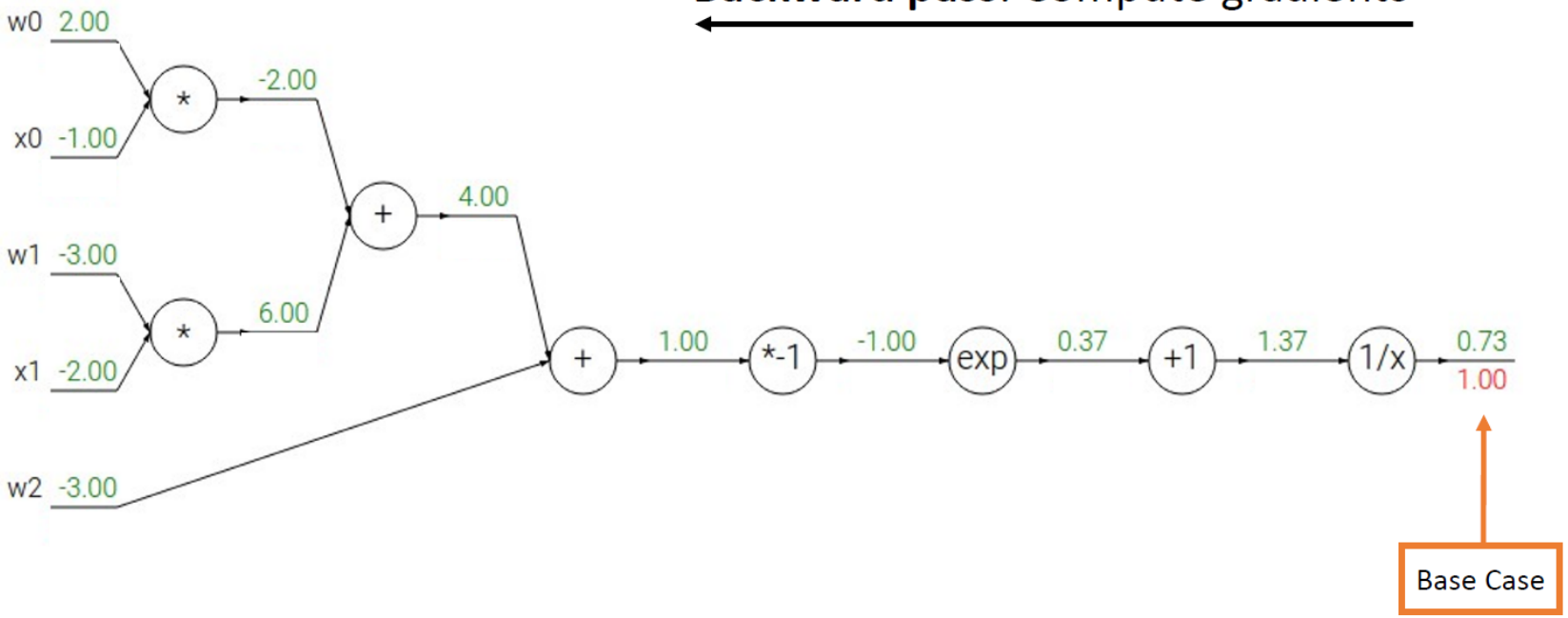
**Forward pass**: Compute outputs



# Another Example

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

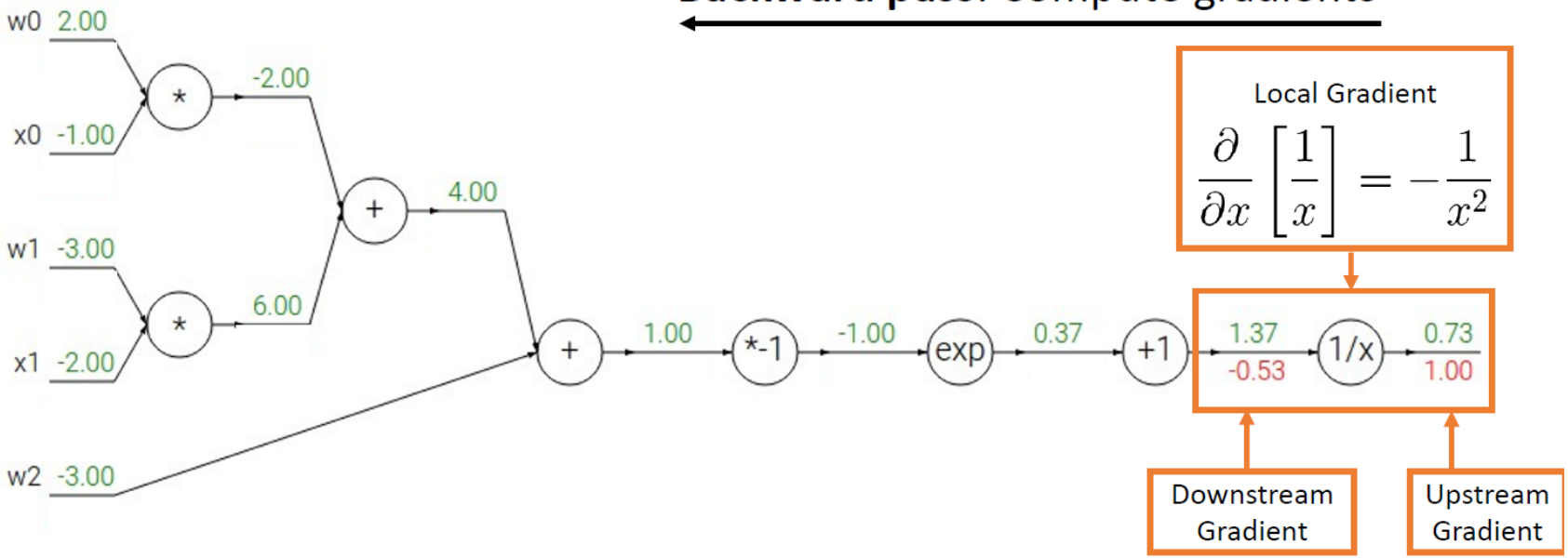
**Backward pass**: Compute gradients



# Another Example

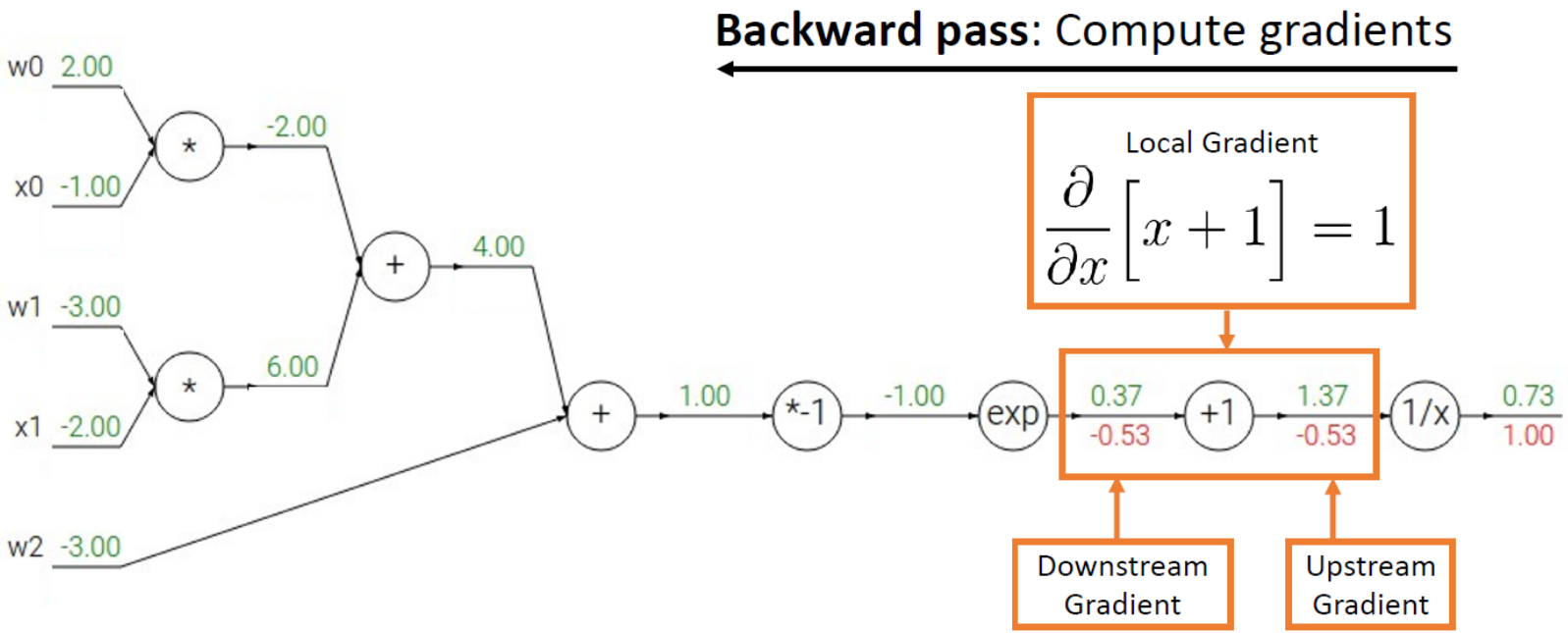
$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

**Backward pass**: Compute gradients



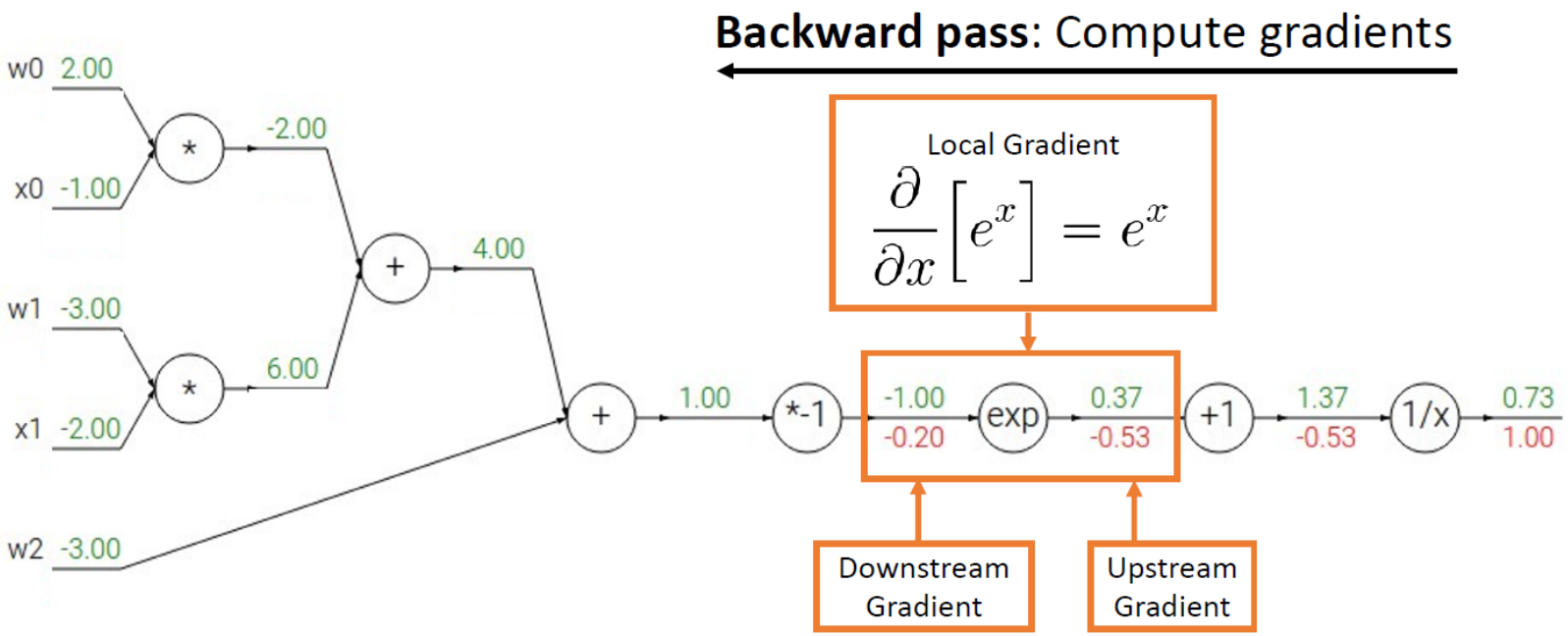
# Another Example

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



# Another Example

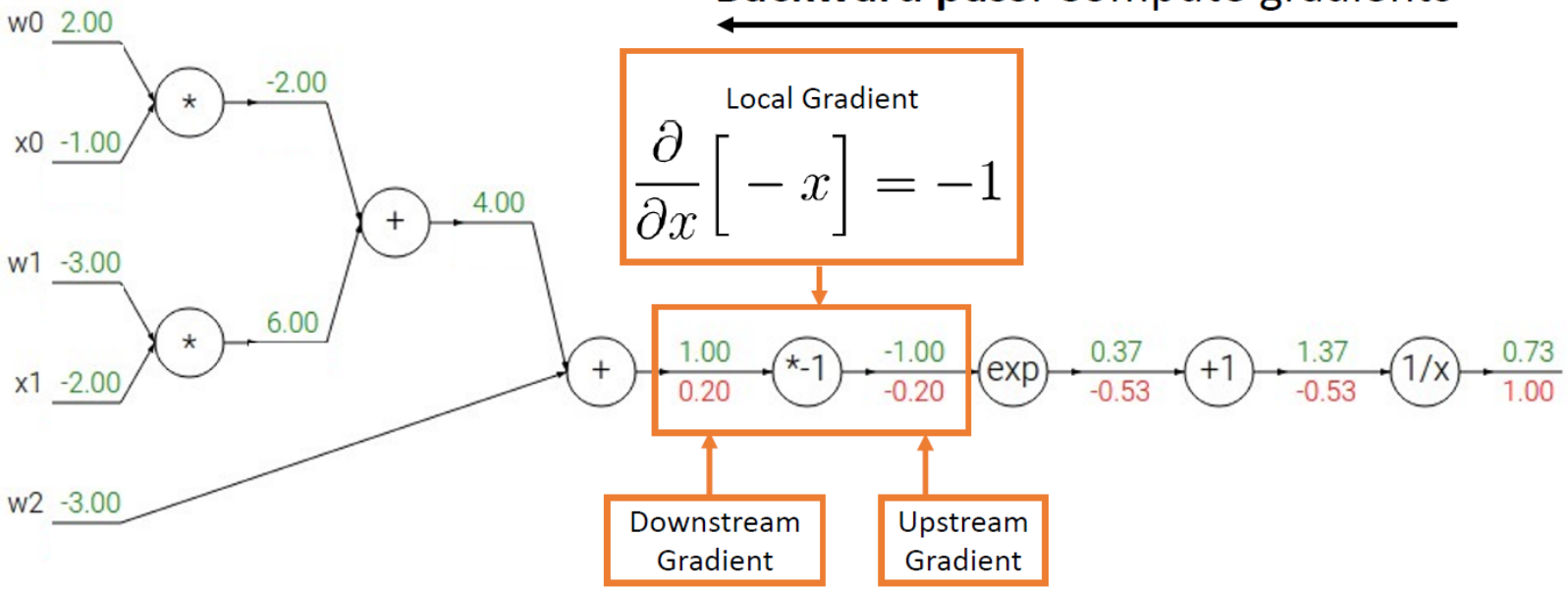
$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



# Another Example

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

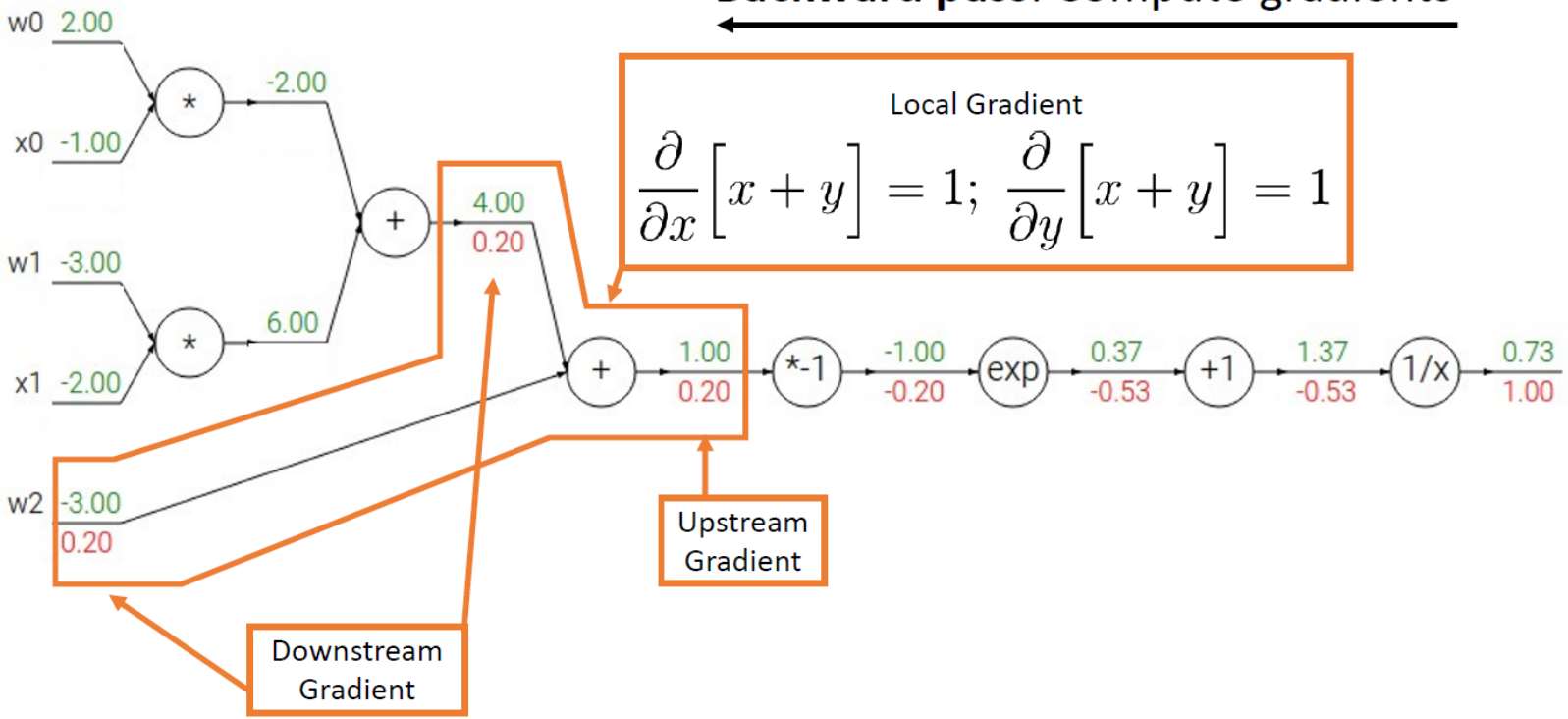
**Backward pass**: Compute gradients



# Another Example

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

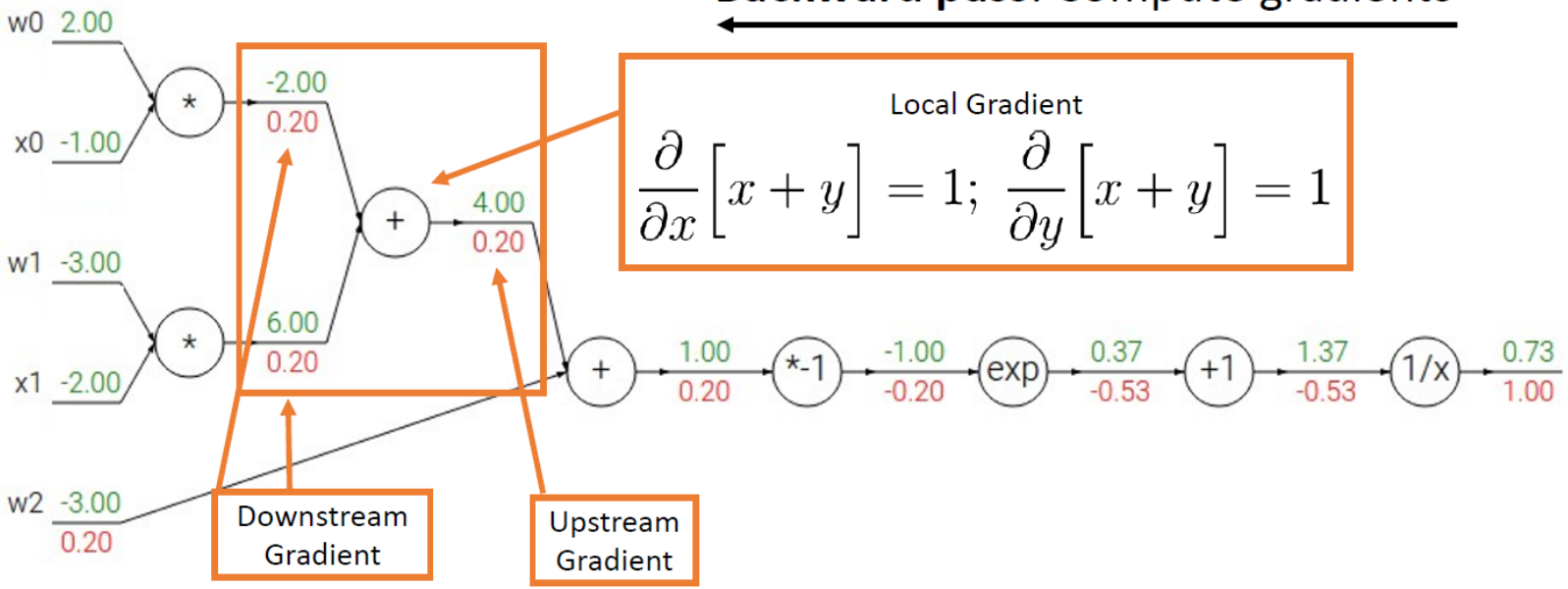
**Backward pass**: Compute gradients



# Another Example

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

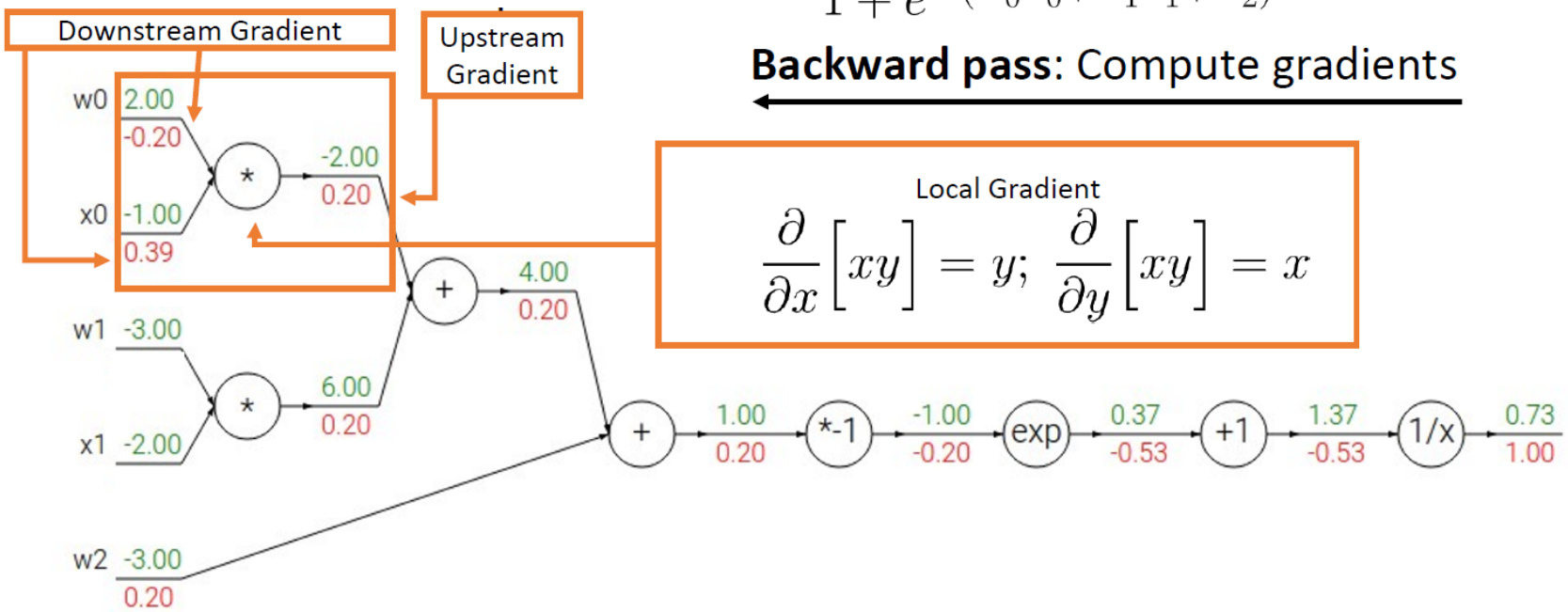
**Backward pass**: Compute gradients



# Another Example

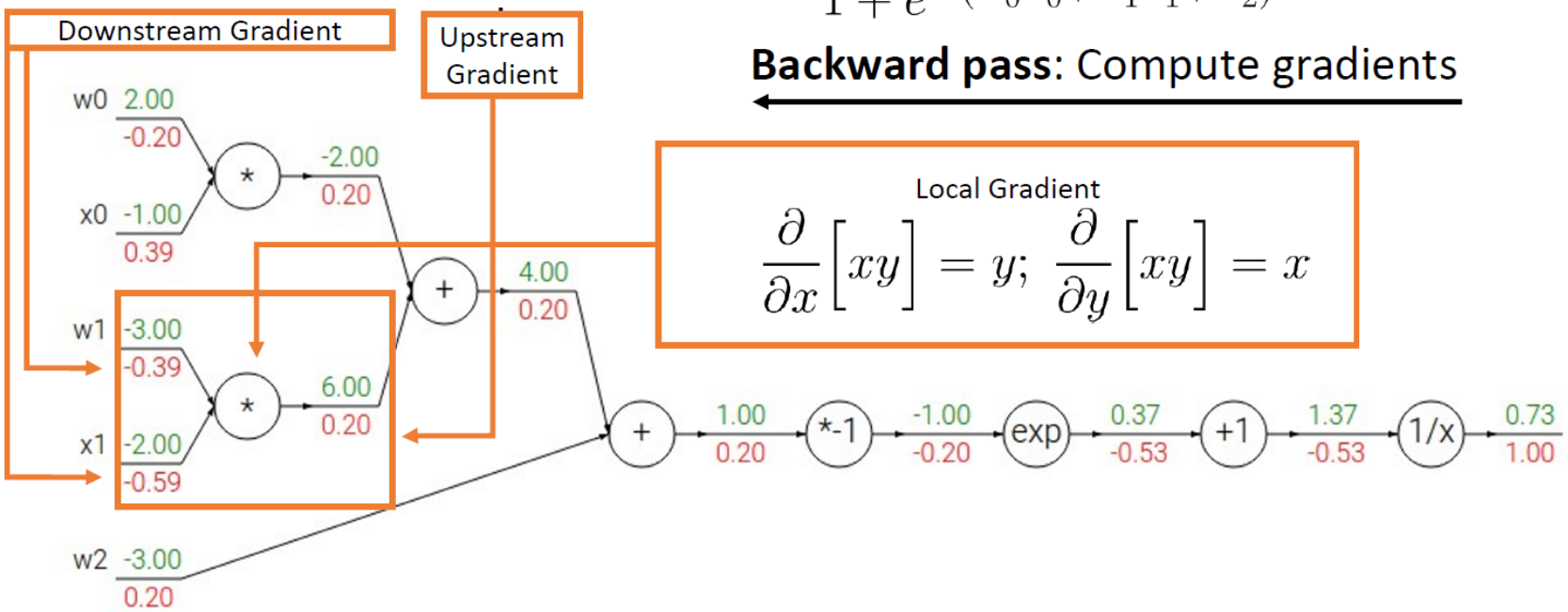
$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

**Backward pass**: Compute gradients



# Another Example

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



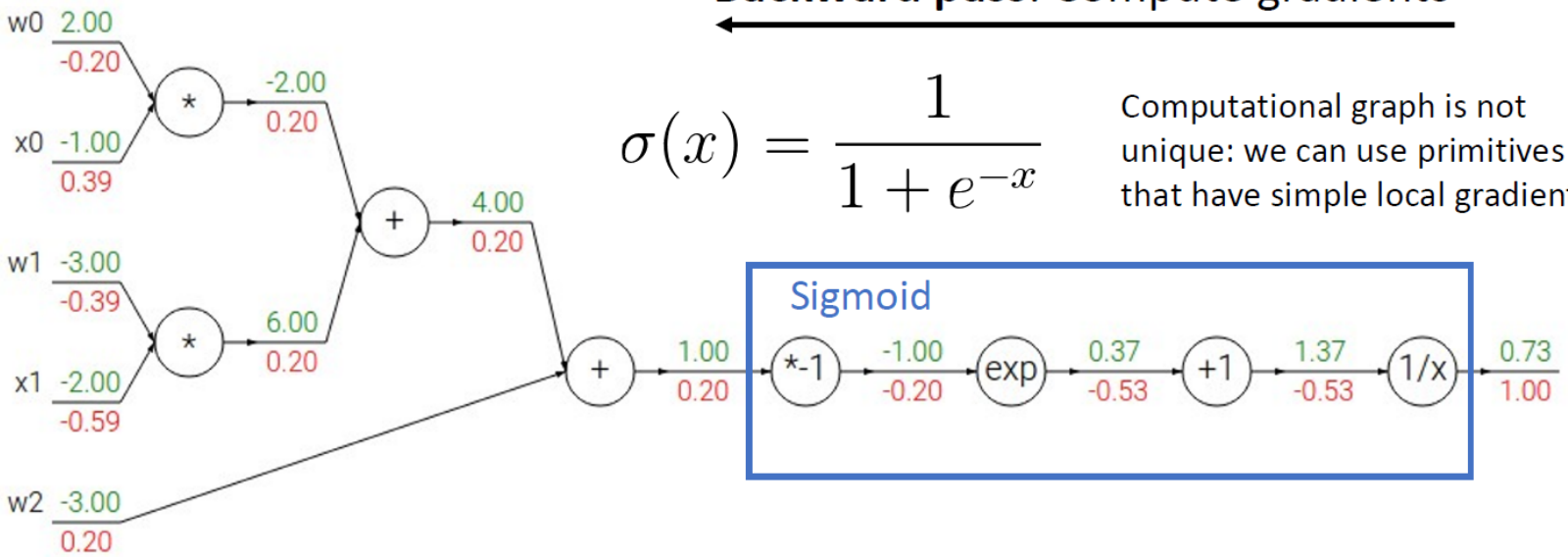
# Another Example

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}} = \sigma(w_0x_0 + w_1x_1 + w_2)$$

**Backward pass**: Compute gradients

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

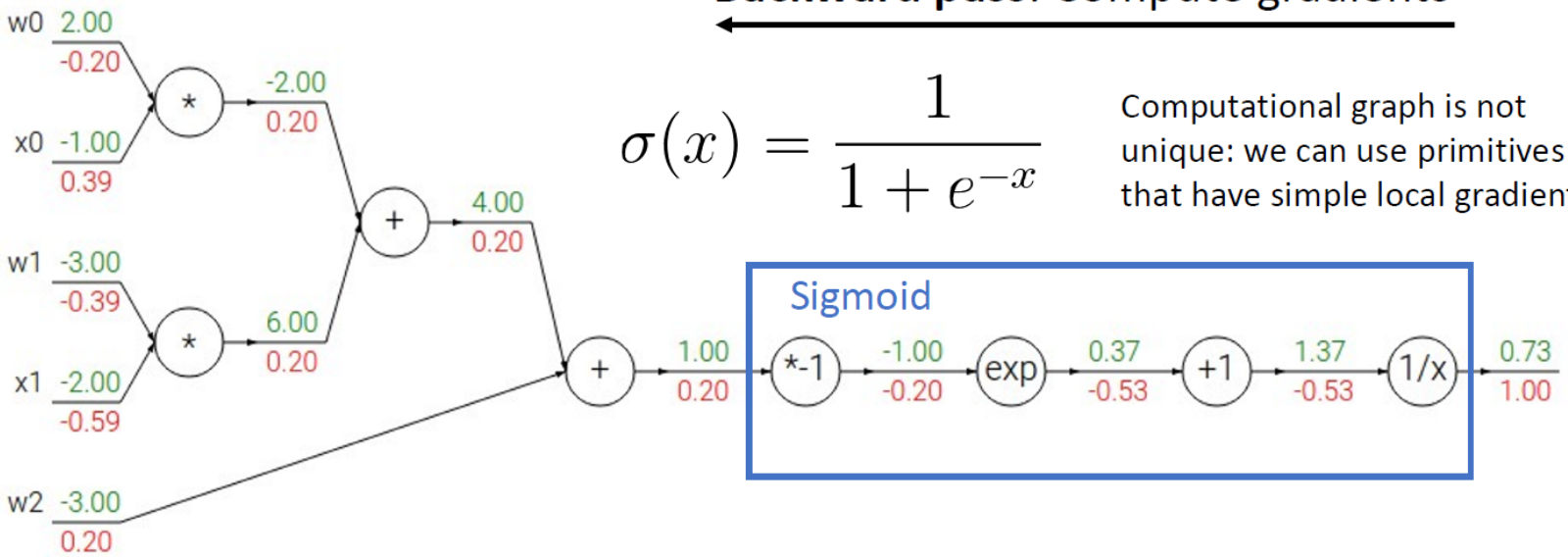
Computational graph is not unique: we can use primitives that have simple local gradients



# Another Example

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}} = \sigma(w_0x_0 + w_1x_1 + w_2)$$

**Backward pass**: Compute gradients




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

Computational graph is not unique: we can use primitives that have simple local gradients

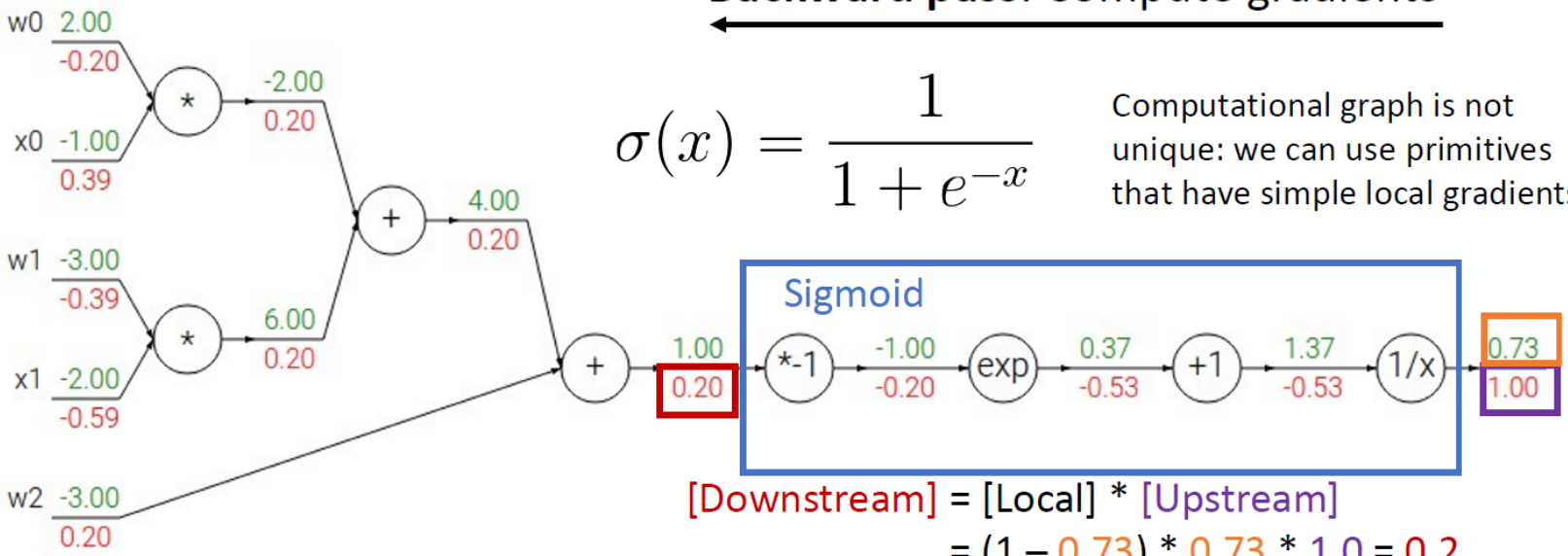
Sigmoid local gradient:

$$\frac{\partial}{\partial x}\left[\sigma(x)\right] = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\sigma(x)$$

# Another Example

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}} = \sigma(w_0x_0 + w_1x_1 + w_2)$$

**Backward pass**: Compute gradients




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

Computational graph is not unique: we can use primitives that have simple local gradients

[Downstream] = [Local] * [Upstream]

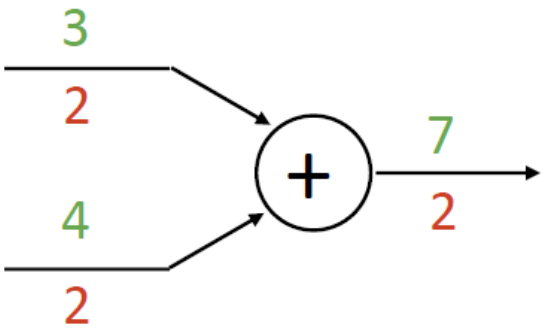
= (1 – 0.73) * 0.73 * 1.0 = 0.2

Sigmoid local gradient:

$$\frac{\partial}{\partial x}\left[\sigma(x)\right] = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\sigma(x)$$

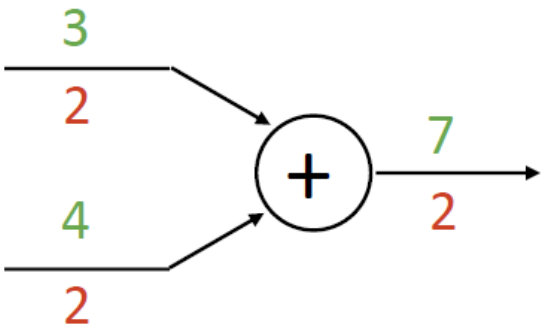
# Patterns in Gradient Flow

**add** gate: gradient distributor

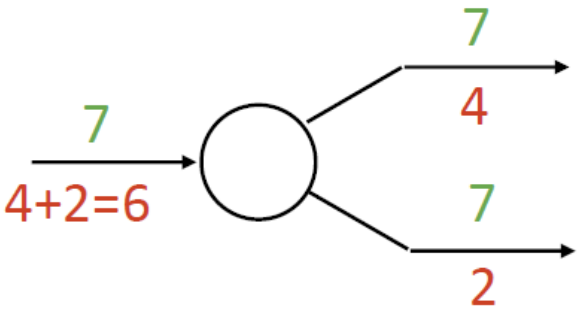


# Patterns in Gradient Flow

**add** gate: gradient distributor

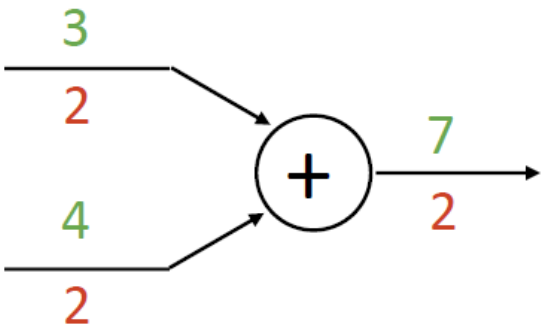



**copy** gate: gradient adder

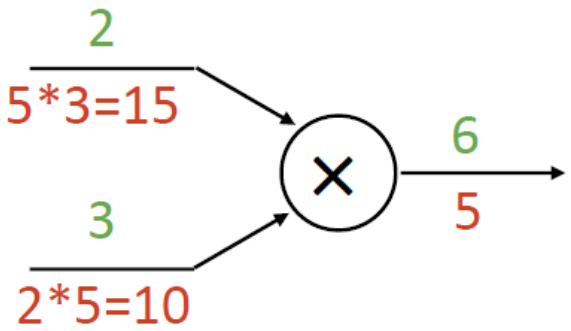


# Patterns in Gradient Flow

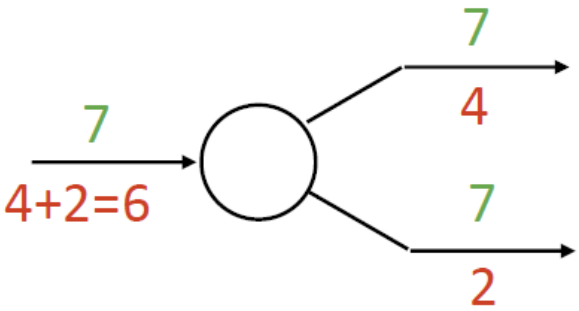
**add** gate: gradient distributor




**mul** gate: "swap multiplier"

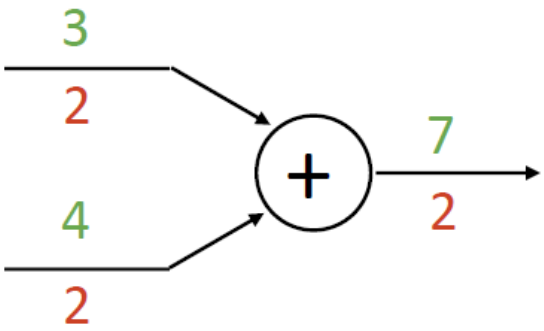



**copy** gate: gradient adder

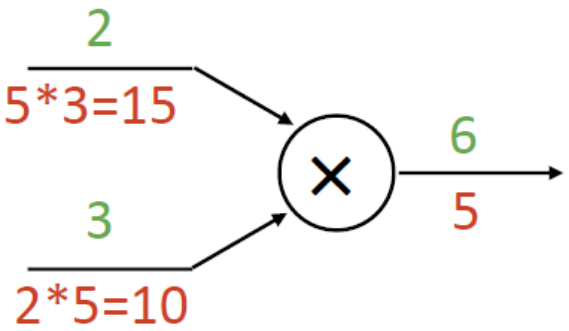


# Patterns in Gradient Flow

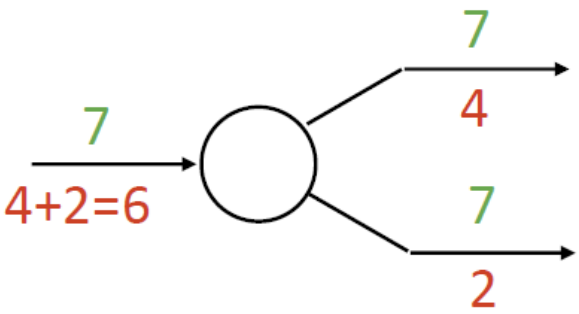
**add** gate: gradient distributor



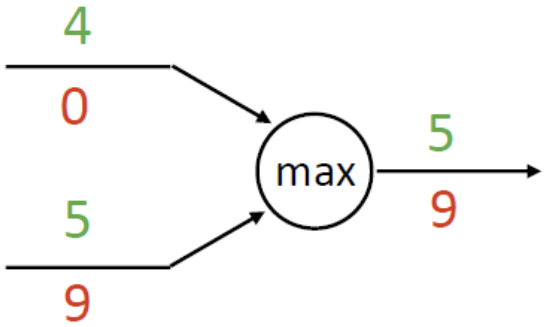

**mul** gate: "swap multiplier"




**copy** gate: gradient adder

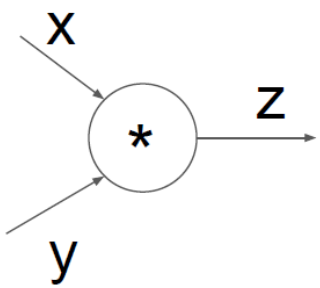



**max** gate: gradient router



# Modularized implementation: forward / backward API

- Actual PyTorch code




(x,y,z are scalars)

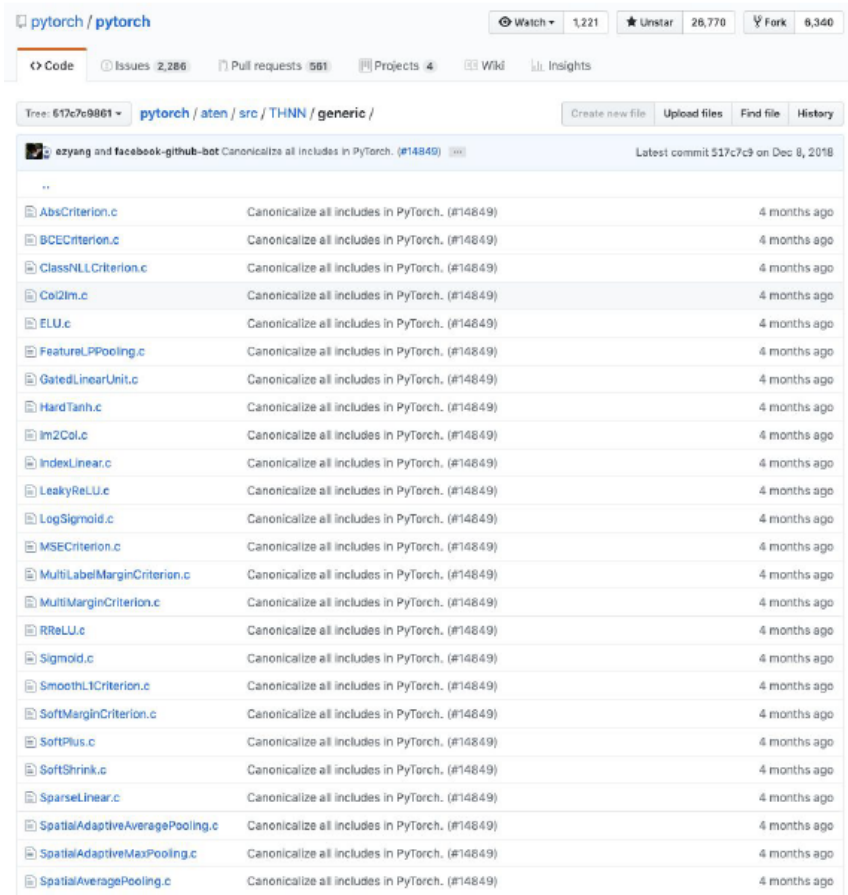
```
class Multiply(torch.autograd.Function):
  @staticmethod
  def forward(ctx, x, y):
    ctx.save_for_backward(x, y)
    z = x * y
    return z
  @staticmethod
  def backward(ctx, grad_z):
    x, y = ctx.saved_tensors
    grad_x = y * grad_z   # dz/dx * dL/dz
    grad_y = x * grad_z   # dz/dy * dL/dz
    return grad_x, grad_y
```

Need to stash some values for use in backward

Upstream gradient

Multiply upstream and local gradients

# Example: PyTorch operators



pytorch / pytorch

Watch 1,221 | Unstar 26,770 | Fork 8,340

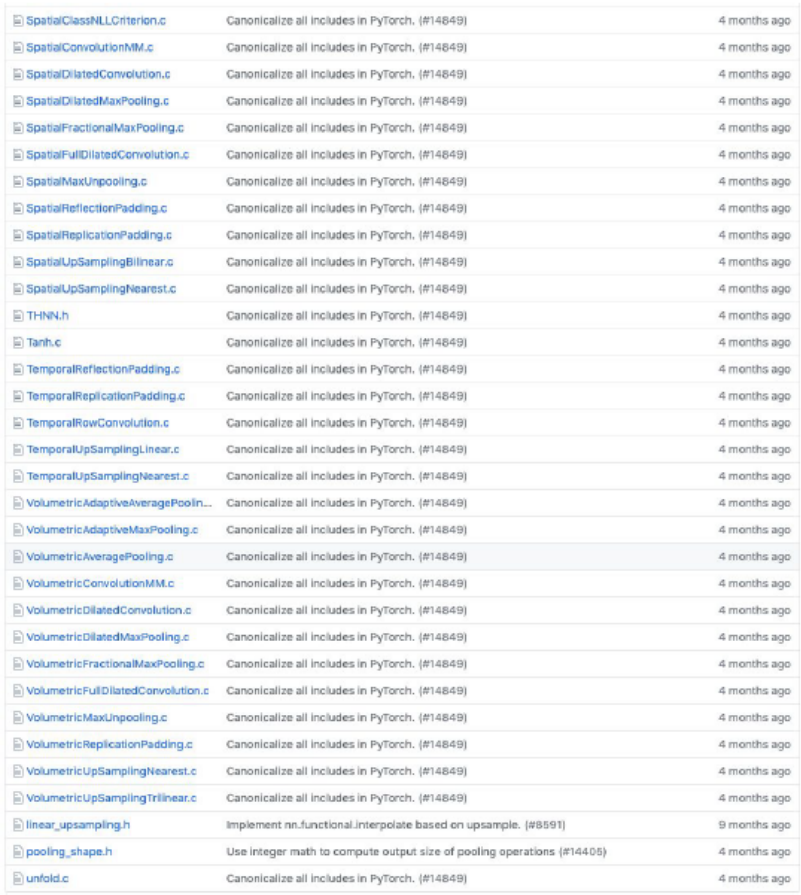
Code | Issues 2,286 | Pull requests 561 | Projects 4 | Wiki | Insights

Tree: 617c7c9861 ▾ | pytorch / aten / src / THNN / generic / | Create new file | Upload files | Find file | History

ezyang and facebook-github-bot Canonicalize all includes in PyTorch. (#14849) ... Latest commit 517c7c9 on Dec 8, 2018

..

| | | |
|---|---|---|
| AbsCriterion.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| BCECriterion.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| ClassNLLCriterion.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| Col2Im.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| ELU.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| FeatureLPPooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| GatedLinearUnit.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| HardTanh.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| Im2Col.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| IndexLinear.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| LeakyReLU.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| LogSigmoid.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| MSECriterion.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| MultiLabelMarginCriterion.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| MultiMarginCriterion.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| RReLU.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| Sigmoid.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SmoothL1Criterion.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SoftMarginCriterion.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SoftPlus.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SoftShrink.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SparseLinear.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialAdaptiveAveragePooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialAdaptiveMaxPooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialAveragePooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |



| | | |
|---|---|---|
| SpatialClassNLLCriterion.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialConvolutionMM.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialDilatedConvolution.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialDilatedMaxPooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialFractionalMaxPooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialFullDilatedConvolution.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialMaxUnpooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialReflectionPadding.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialReplicationPadding.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialUpSamplingBilinear.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| SpatialUpSamplingNearest.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| THNN.h | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| Tanh.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| TemporalReflectionPadding.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| TemporalReplicationPadding.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| TemporalRowConvolution.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| TemporalUpSamplingLinear.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| TemporalUpSamplingNearest.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricAdaptiveAveragePoolin... | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricAdaptiveMaxPooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricAveragePooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricConvolutionMM.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricDilatedConvolution.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricDilatedMaxPooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricFractionalMaxPooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricFullDilatedConvolution.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricMaxUnpooling.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricReplicationPadding.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricUpSamplingNearest.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| VolumetricUpSamplingTrilinear.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |
| linear_upsampling.h | Implement nn.functional.interpolate based on upsample. (#8591) | 9 months ago |
| pooling_shape.h | Use integer math to compute output size of pooling operations (#14405) | 4 months ago |
| unfold.c | Canonicalize all includes in PyTorch. (#14849) | 4 months ago |

# PyTorch sigmoid layer

```
#ifndef TH_GENERIC_FILE
#define TH_GENERIC_FILE "THNN/generic/Sigmoid.c"
#else

void THNN_(Sigmoid_updateOutput)(
          THNNState *state,
          THTensor *input,
          THTensor *output)
{
  THTensor_(sigmoid)(output, input);
}

void THNN_(Sigmoid_updateGradInput)(
          THNNState *state,
          THTensor *gradOutput,
          THTensor *gradInput,
          THTensor *output)
{
  THNN_CHECK_NELEMENT(output, gradOutput);
  THTensor_(resizeAs)(gradInput, output);
  TH_TENSOR_APPLY3(scalar_t, gradInput, scalar_t, gradOutput, scalar_t, output,
    scalar_t z = *output_data;
    *gradInput_data = *gradOutput_data * (1. - z) * z;
  );
}

#endif
```

Forward

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

Source

# PyTorch sigmoid layer

```
#ifndef TH_GENERIC_FILE
#define TH_GENERIC_FILE "THNN/generic/Sigmoid.c"
#else

void THNN_(Sigmoid_updateOutput)(
          THNNState *state,
          THTensor *input,
          THTensor *output)
{
  THTensor_(sigmoid)(output, input);
}

void THNN_(Sigmoid_updateGradInput)(
          THNNState *state,
          THTensor *gradOutput,
          THTensor *gradInput,
          THTensor *output)
{
  THNN_CHECK_NELEMENT(output, gradOutput);
  THTensor_(resizeAs)(gradInput, output);
  TH_TENSOR_APPLY3(scalar_t, gradInput, scalar_t, gradOutput, scalar_t, output,
    scalar_t z = *output_data;
    *gradInput_data = *gradOutput_data * (1. - z) * z;
  );
}

#endif
```

Forward

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

```
static void sigmoid_kernel(TensorIterator& iter) {
  AT_DISPATCH_FLOATING_TYPES(iter.dtype(), "sigmoid_cpu", [&]() {
    unary_kernel_vec(
        iter,
        [=](scalar_t a) -> scalar_t { return (1 / (1 + std::exp((-a)))); },
        [=](Vec256<scalar_t> a) {
          a = Vec256<scalar_t>((scalar_t)(0)) - a;
          a = a.exp();
          a = Vec256<scalar_t>((scalar_t)(1)) + a;
          a = a.reciprocal();
          return a;
        });
  });
}
```

Forward actually defined elsewhere...

```
return (1 / (1 + std::exp((-a))));
```

Source

# PyTorch sigmoid layer

```
#ifndef TH_GENERIC_FILE
#define TH_GENERIC_FILE "THNN/generic/Sigmoid.c"
#else

void THNN_(Sigmoid_updateOutput)(
          THNNState *state,
          THTensor *input,
          THTensor *output)
{
  THTensor_(sigmoid)(output, input);
}

void THNN_(Sigmoid_updateGradInput)(
          THNNState *state,
          THTensor *gradOutput,
          THTensor *gradInput,
          THTensor *output)
{
  THNN_CHECK_NELEMENT(output, gradOutput);
  THTensor_(resizeAs)(gradInput, output);
  TH_TENSOR_APPLY3(scalar_t, gradInput, scalar_t, gradOutput, scalar_t, output,
    scalar_t z = *output_data;
    *gradInput_data = *gradOutput_data * (1. - z) * z;
  );
}

#endif
```

Forward

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

```
static void sigmoid_kernel(TensorIterator& iter) {
  AT_DISPATCH_FLOATING_TYPES(iter.dtype(), "sigmoid_cpu", [&]() {
    unary_kernel_vec(
        iter,
        [=](scalar_t a) -> scalar_t { return (1 / (1 + std::exp((-a)))); },
        [=](Vec256<scalar_t> a) {
          a = Vec256<scalar_t>((scalar_t)(0)) - a;
          a = a.exp();
          a = Vec256<scalar_t>((scalar_t)(1)) + a;
          a = a.reciprocal();
          return a;
        });
  });
}
```

Forward actually defined elsewhere...

Backward

$$(1-\sigma(x))\,\sigma(x)$$

Source

# References


- Deep Learning, Ian Goodfellow, MIT Press, Ch.6, 7 ,and 8