

# Advanced School in Artificial Intelligence

## Introduction to Neural Networks

**Ing. Zese Riccardo**  
**[riccardo.zese@unife.it](mailto:riccardo.zese@unife.it)**

*Progetto di alta formazione in ambito tecnologico economico e culturale per una regione della conoscenza europea e attrattiva approvato e cofinanziato dalla Regione Emilia-Romagna con deliberazione di Giunta regionale n. 1625/2021*



**Università  
degli Studi  
di Ferrara**

## Outline

- Introduction to Python
- Introduction to Neural Networks
- Convolutional NN
- Recurrent NN
- Autoencoders and self supervised learning

## Sources

- These slides are taken from:

- Intel Nervana AI Academy Instructional Content

Intel Legal Notices and Disclaimers

This presentation is for informational purposes only. INTEL MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at [intel.com](https://intel.com).

This sample source code is released under the [Intel Sample Source Code License Agreement](#).

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

\*Other names and brands may be claimed as the property of others.

Copyright © 2017, Intel Corporation. All rights reserved.

- Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. MIT press, 2016.

[https://www.deeplearningbook.org/lecture\\_slides.html](https://www.deeplearningbook.org/lecture_slides.html)

- Chapter “Neural Networks” of “A Course in Machine Learning” by Hal Daume III.

<http://ciml.info/>

- Some parts from “CS231n: Convolutional Neural Networks for Visual Recognition”, Stanford University

<http://cs231n.stanford.edu/>

## Outline

- Introduction to Python
- Introduction to Neural Networks
- Convolutional NN
- Recurrent NN
- Autoencoders and self supervised learning

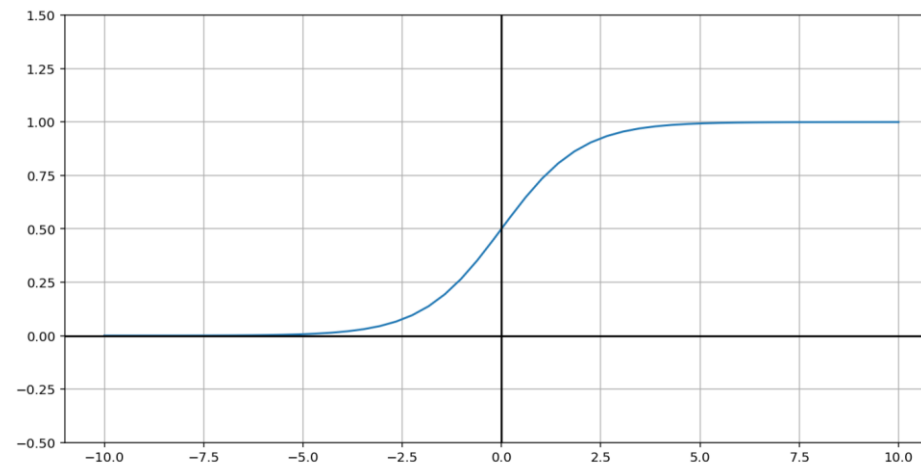


## Designing Neural networks

- We have seen a simple example of a feedforward network and how a NN computes the output given 1 input, in 1 single forward pass.
- Next, we address each of the design decisions needed to deploy a feedforward network.
  - First, training a feedforward network requires making many of the same design decisions as are necessary for a linear model: choosing the optimizer, the cost function, and the form of the output units.
  - Feedforward networks have introduced the concept of a hidden layer, and this requires us to choose the activation functions that will be used to compute the hidden layer values.
  - We must also design the architecture of the network, including how many layers the network should contain, how these layers should be connected to each other, and how many units should be in each layer.
- Learning in deep neural networks is performed by comparison between output and known truth, through a **loss function  $J$** .
  - According to it we adjust the weights by performing gradient descent.
  - Computing the gradient may be difficult.

## Designing Neural networks

- Computing the gradient may be difficult, we need to chose activation functions wisely to maintain the computation of the gradient as much easy as possible while maintaining low values of loss function and good performance.
- Up to now we have seen the sigmoid activation function  $\sigma(z) = \frac{1}{1+e^{-z}}$



## Nice property of Sigmoid Function

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

$$\begin{aligned}\sigma'(z) &= \frac{0 - (-e^{-z})}{(1 + e^{-z})^2} = \frac{e^{-z}}{(1 + e^{-z})^2} = \frac{1 + e^{-z} - 1}{(1 + e^{-z})^2} = \\ &= \frac{1 + e^{-z}}{(1 + e^{-z})^2} - \frac{1}{(1 + e^{-z})^2} = \frac{1}{1 + e^{-z}} - \frac{1}{(1 + e^{-z})^2} = \\ &= \frac{1}{1 + e^{-z}} \left( 1 - \frac{1}{1 + e^{-z}} \right)\end{aligned}$$

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



## Designing Neural networks

- We will see other activation functions that are **linear** or **nearly-linear**.
- We will present the **back-propagation** algorithm and its modern generalizations, which can be used to efficiently compute these gradients.



## Loss functions

- **Classification:**

- Error or **0-1 loss**: the 0-1 loss on a particular example is 0 if it is correctly classified and 1 if it is not.
- **Probability Prediction** or density estimation: **negative log-likelihood** aka **Cross-Entropy** (or log-loss) between the training data and the model distribution.

$$J(\theta) = -\mathbb{E}_{x,y \sim \hat{p}_{data}} \log p_{model}(y|x)$$

$\theta$  contains weights  
and biases

$p_{model}$  is the probability  
distribution described by  
model of the designed network

## Loss functions

- Classification:
  - Probability Prediction or density estimation: negative log-likelihood aka Cross-Entropy (or log-loss) between the training data and the model distribution.

$$J(\theta) = -\mathbb{E}_{x,y \sim \hat{p}_{data}} \log p_{model}(y|x)$$

$$J(\theta) = \mathbb{E}_{x,y \sim \hat{p}_{data}} L(x, y, \theta) = \frac{1}{N} \sum_{i=1}^N L(x^{(i)}, y^{(i)}, \theta)$$



Where  $L(x, y, \theta) = -\log p(y|x; \theta) = -\log p_{model}(y|x)$

## Loss functions

- Regression:
  - Root Mean Squared Error (RMSE)

$$RMSE = \sqrt{\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2}$$

- Mean Absolute Deviation (MAD)

$$MAD = \frac{1}{N} \sum_{i=1}^N |y_i - \hat{y}_i|$$

## Output Units

- The choice of cost function is tightly coupled with the choice of output unit
- Most of the time, we simply use the cross-entropy between the data distribution and the model distribution.
- The choice of how to represent the output then determines the form of the cross-entropy function.

## Linear Unit for Regression

- One simple kind of output unit is called **linear units** (with  $h = f(x; \theta)$  the set of hidden features)

$$\hat{y} = W^T h + b$$

- No activation function is usually used.

## Sigmoid Units for Classification

- Many tasks require predicting the value of a binary variable  $y$ .
- Classification problems with two classes can be cast in this form.
- The network learns a distribution over a single binary random variable controlled by a single parameter  $\phi \in [0, 1]$ , which gives the probability of the random variable being equal to 1

$$P(x = 1) = \phi$$





## Sigmoid Units for Classification

- The neural net needs to predict only  $P(y = 1|x)$ .
- To model it we need a linear unit with threshold to obtain a valid value of probability (in  $[0,1]$ )

$$P(y = 1|x) = \max\{0, \min\{1, W^T h + b\}\}$$

- Difficult to train with gradient descent: outside the unit interval, the gradient of the output of the model with respect to its parameters would be 0.
  - A gradient of 0 is typically problematic because the learning algorithm no longer has a guide for how to improve the corresponding parameters.

## Sigmoid Units for Classification

- To solve this problem we can use Sigmoid output units combined with maximum likelihood
- The sigmoid output unit is  $\hat{y} = \sigma(W^T h + b)$  using the sigmoid function seen several slides ago.
- This can be split in two steps:
  1. Computing the linear layer output  $z = W^T h + b$
  2. Convert  $z$  into a probability with the sigmoid function

## Multiclass Classification with NNs

- For **binary classification** problems, we have a **final layer with a single node** and a **sigmoid activation**.
- This has many desirable properties
  - Gives an output strictly between 0 and 1
  - Can be interpreted as a probability
  - Derivative is “nice”
  - Analogous to logistic regression
- Is there a natural extension of this to a multiclass setting?

## Multiclass Classification with NNs

- For **binary classification** we want  $\hat{y} = P(y = 1|x)$  a single number.
- For **multiclass classification** we want  $\hat{y}$  to be a vector with  $\hat{y}_i = P(y = i|x)$ .
  - Each element of  $\hat{y}$  must be in  $[0,1]$  and they must sum up to 1



## Multiclass Classification with NNs

- A possibility is to use a one **hot encoding** for categories
  - Take a vector with length equal to the number of categories
  - Represent each category with one at a particular position (and zero everywhere else)

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Cat

$$\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Dog

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Toaster

## Multiclass Classification with NNs



Note that we should avoid to represent the classes using a single number such as cat = 1, dog = 2, toaster = 3. This approach implicitly defines ordinal relation between categories, while like in this case we don't want that (dog should not be "between" cat and toaster). Moreover, we don't want to represent any sort of similarity or affinity (a toaster is not "closer" or more similar to a dog than to a cat).

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Cat

$$\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Dog

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Toaster





## Multiclass Classification with NNs

- For multiclass classification problems, let the **final layer** be a **vector** with **length equal to the number of possible classes**.
- Extension of sigmoid to multiclass is the **softmax** function

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{k=1}^K e^{z_k}}$$

- Yields a vector with entries that are between 0 and 1, and sum to 1.
  - It can be interpreted as a probability distribution

## Multiclass Classification with NNs

- For multiclass classification problems, let the **final layer** be a **vector** with **length equal to the number of possible classes**.
- Extension of sigmoid to multiclass is the **softmax** function

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{k=1}^K e^{z_k}}$$

- Note that the exponentiation changes normalization
  - $z = [3, 1, 1]$  does not become  $[0.6, 0.2, 0.2]$  but rather  $[0.78, 0.11, 0.11]$



## Multiclass Classification with NNs

- As loss function use **categorical cross entropy** o **discrete cross entropy**
- This is just the log-loss function in disguise

$$C.E. = J(\theta) = - \sum_{i=1}^n y_i \log(\hat{y}_i)$$

- Note that typically  $y_i$  is either 0 or 1, so we have  $C.E. = -\log(\hat{y}_i)$  for the right answer.

## Multiclass Classification with NNs

- Cross entropy derivative has a nice property when used with softmax

$$J = - \sum_{i=1}^n y_i \log(\hat{y}_i)$$
$$\frac{\partial J}{\partial z_i} = \frac{\partial J}{\partial softmax} \cdot \frac{\partial softmax}{\partial z_i} = \hat{y}_i - y_i$$

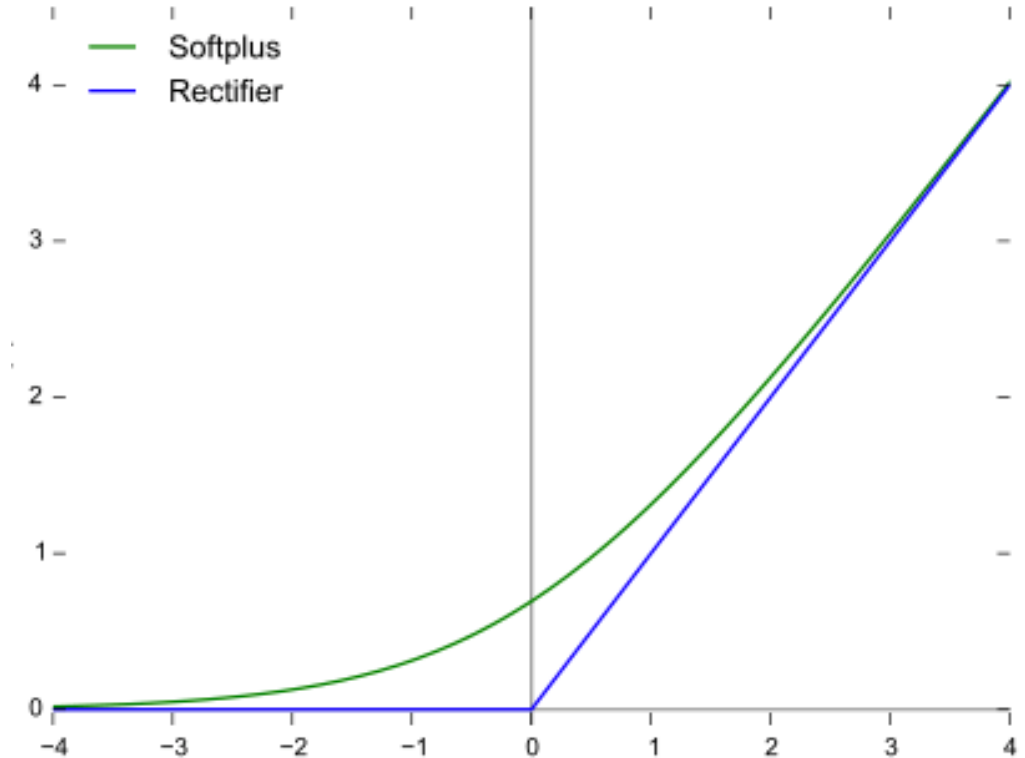
## Hidden Units

- The choice of the type of hidden unit to use in the hidden layers of the model is an issue specific of feedforward networks.
  - It is still under extensive research but does not yet have many definitive guiding theoretical principles.
- **Rectified linear units** are an excellent default choice of hidden unit.
- Many other types of hidden units are available

## Softplus

$$g(z) = \log(1 + e^z)$$

$$g'(z) = \frac{e^z}{e^z + 1} = \frac{1}{1 + e^{-z}}$$

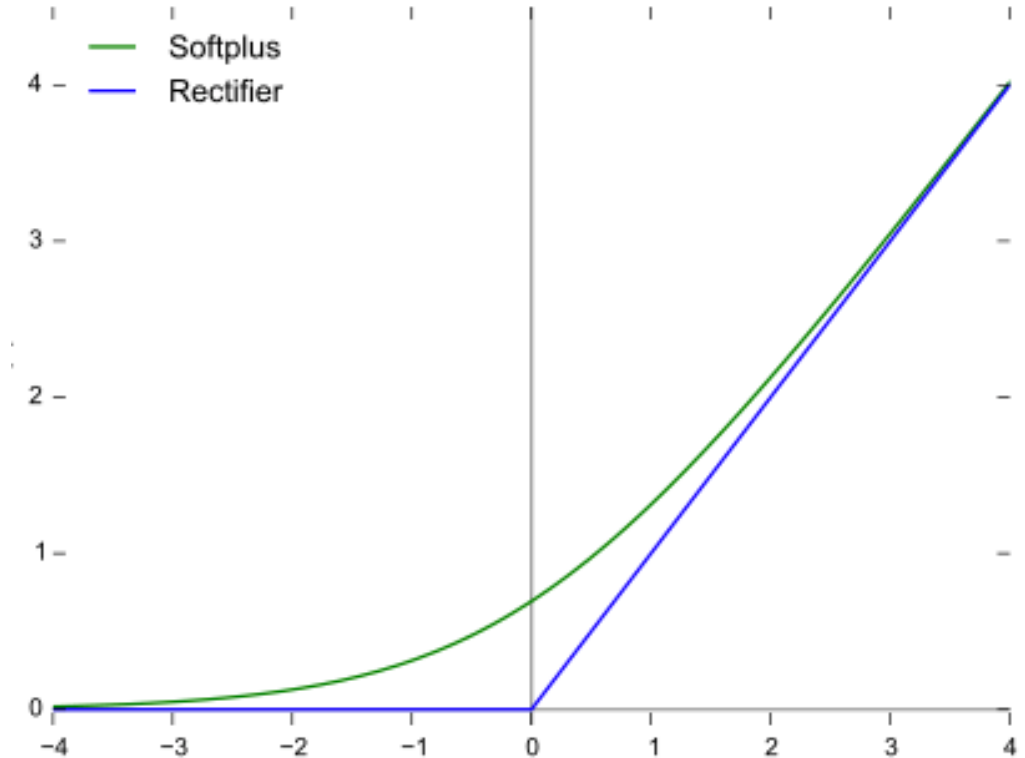





## Rectified Linear Units

- The **softplus function** is a smoothed or “softened” version of the function  $\max(0, z)$

$$\text{ReLU} = \max(0, z)$$



## Rectified Linear Units

- Note that, ReLU is not differentiable at  $z = 0$ . 
  - This may seem like this invalidates it for use with a gradient-based learning algorithm, however, neural network training algorithms do not usually arrive at a local minimum of the cost function, but instead merely reduce its value significantly.
  - Because we do not expect training to actually reach a point where the gradient is 0, it is acceptable for the minima of the cost function to correspond to points with undefined gradient.

## Beyond ReLU

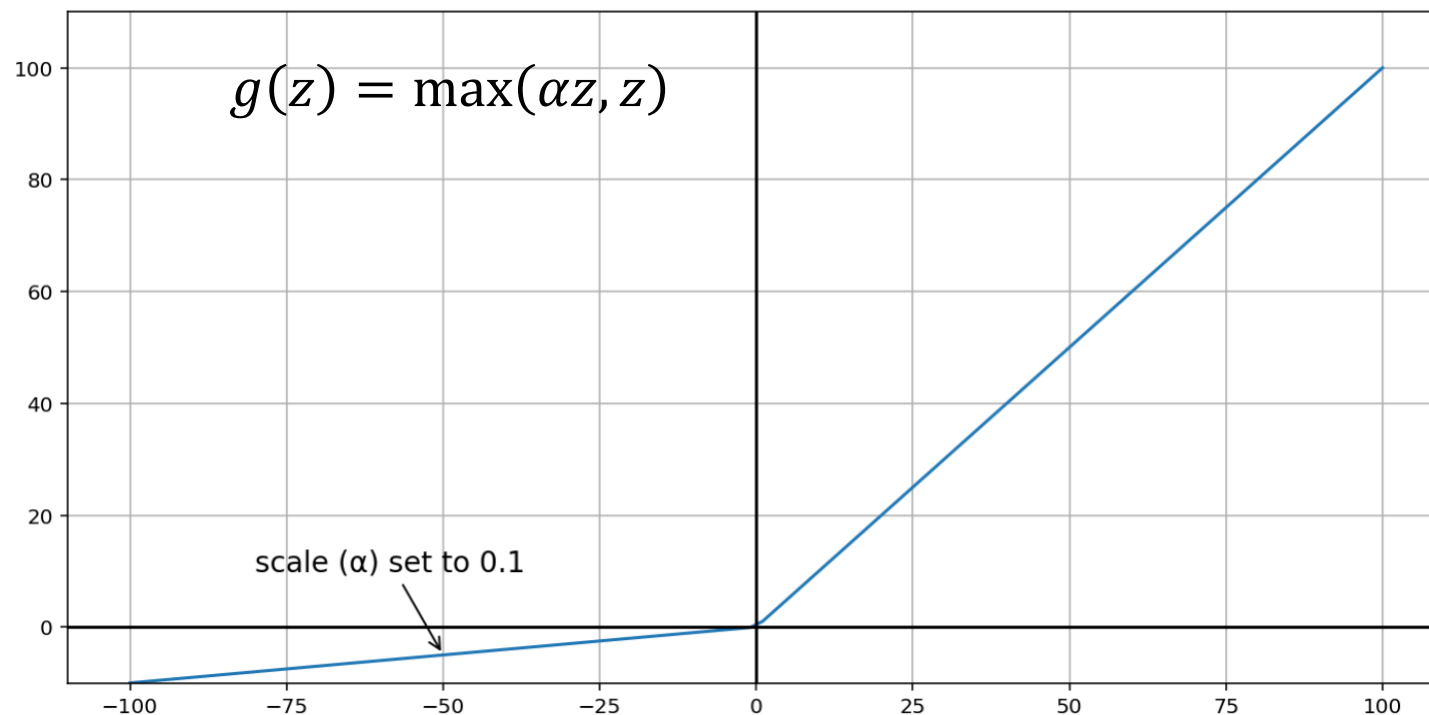
- ReLU generalizations uses a non-zero slope  $\alpha$  when  $z < 0$

$$g(z, \alpha) = \max(0, z) + \alpha \min(0, z)$$

- **Absolute value rectification:**  $\alpha = -1$ .
  - Simulates  $g(z) = |z|$
  - Used in object recognition from images

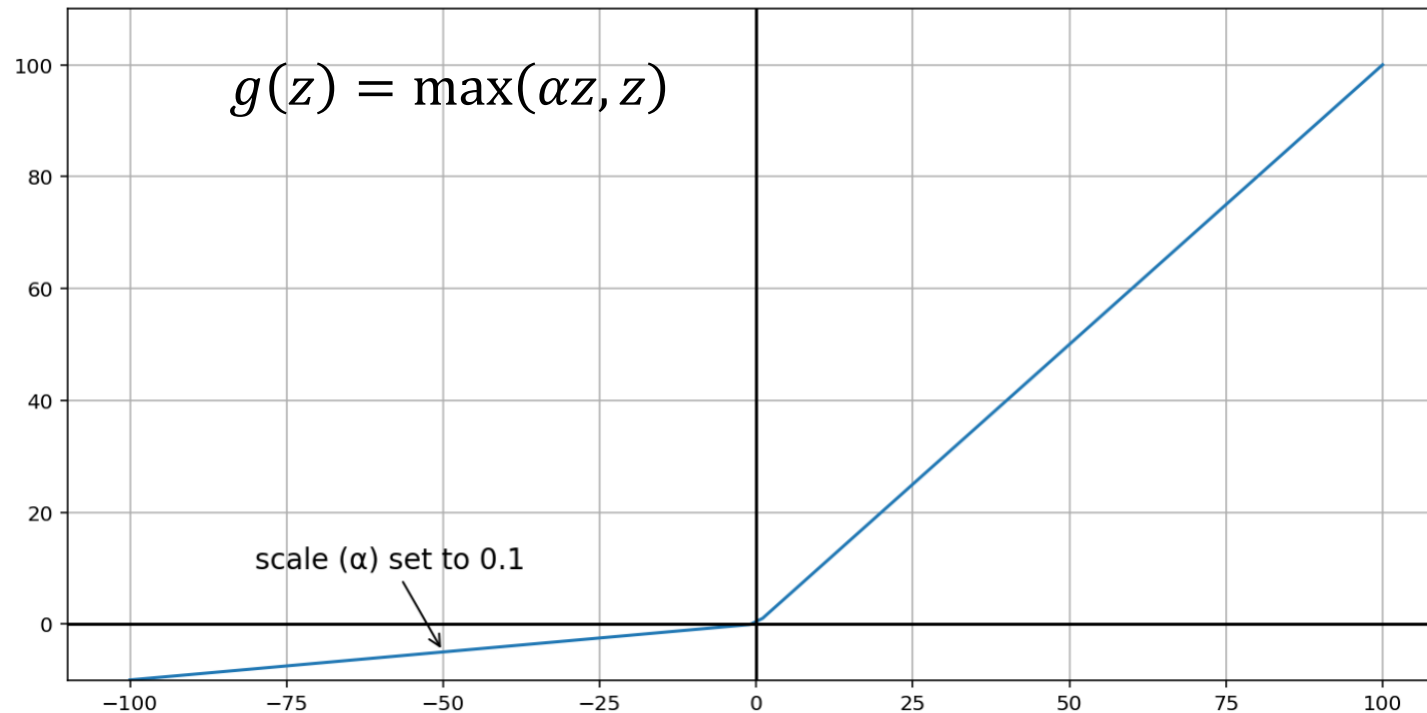
## Beyond ReLU

- **Leaky ReLU** o **LReLU**: having a zero gradient means something will never change. In LReLU  $\alpha$  is set to a small value like 0.01.



## Beyond ReLU

- **Parametric ReLU o PReLU:**  $\alpha$  is a learnable parameter.



## Beyond ReLU

- **Smooth ReLU o SReLU:**

$$g(z) = \log(1 + e^z)$$

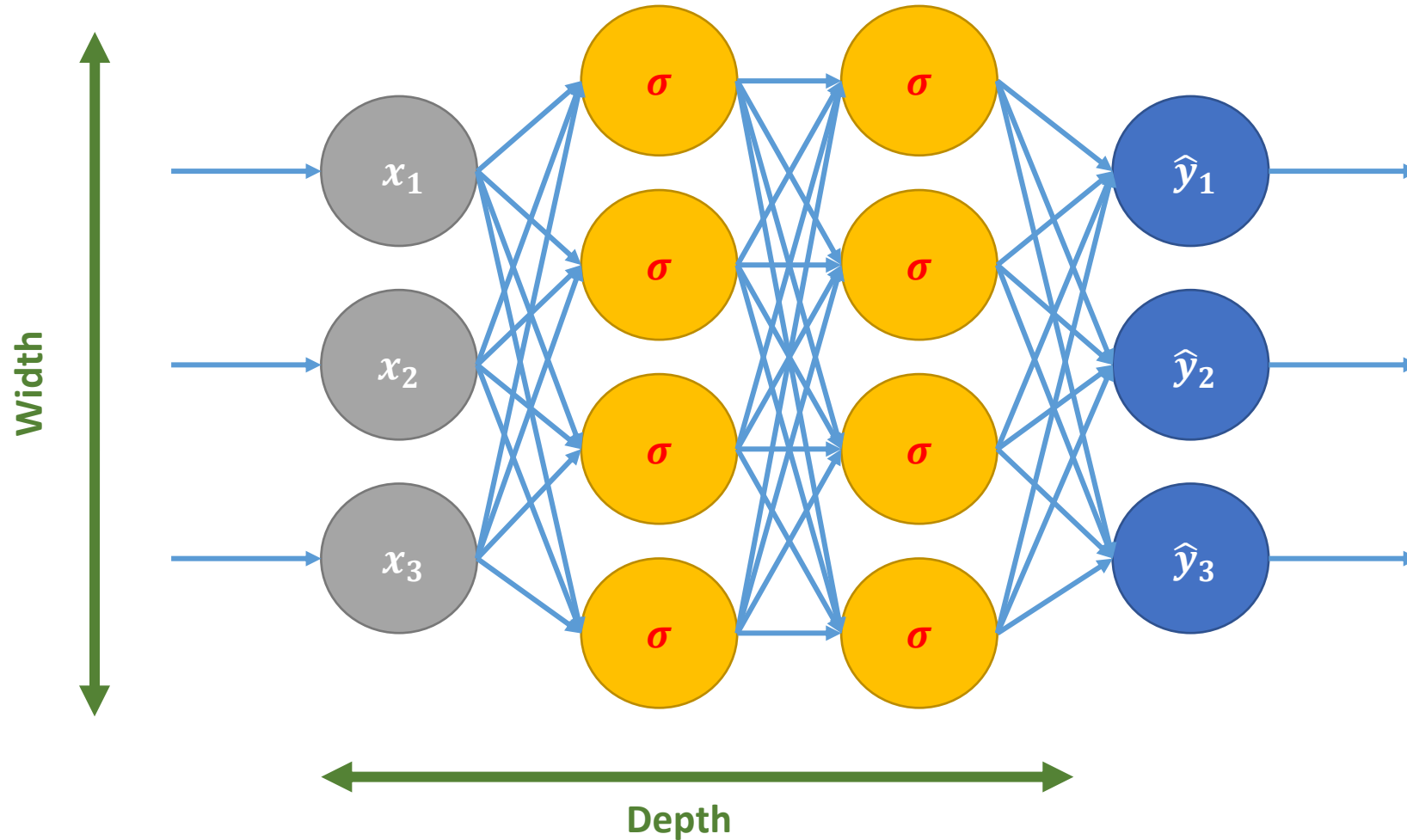
- **Exponential Linear Unit o ELU:**

$$g(z) = \begin{cases} z & z \geq 0 \\ \alpha(e^z - 1) & \text{otherwise} \end{cases}$$

- **Scaled Exponential Linear Unit o SELU:**

$$g(z) = \begin{cases} \lambda z & z \geq 0 \\ \lambda \alpha(e^z - 1) & \text{otherwise} \end{cases}$$

## Shallow vs Deep Networks



## Shallow vs Deep Networks

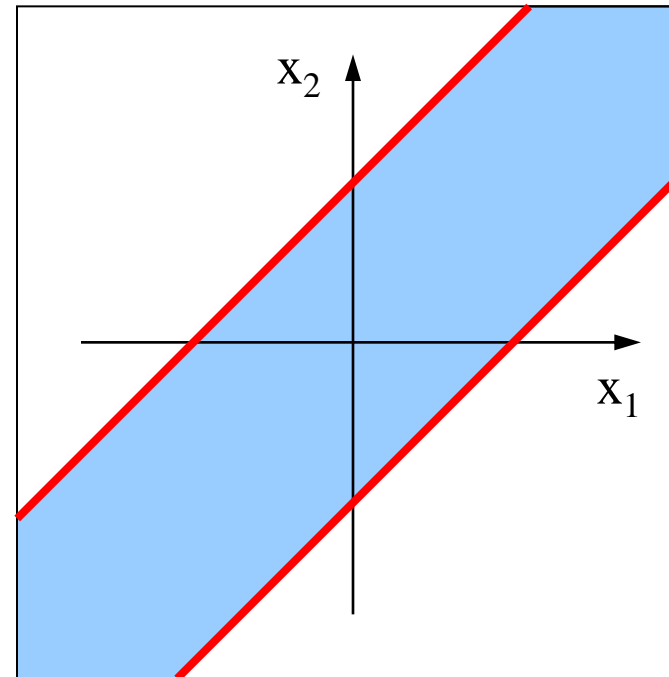
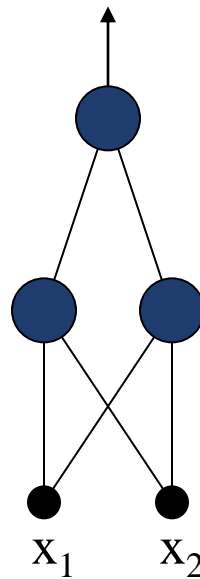
- One hidden layer is enough to represent (not learn) an approximation of any function to an arbitrary degree of accuracy
- So why deeper?
  - Shallow net may need (exponentially) more width
  - Shallow net may overfit more





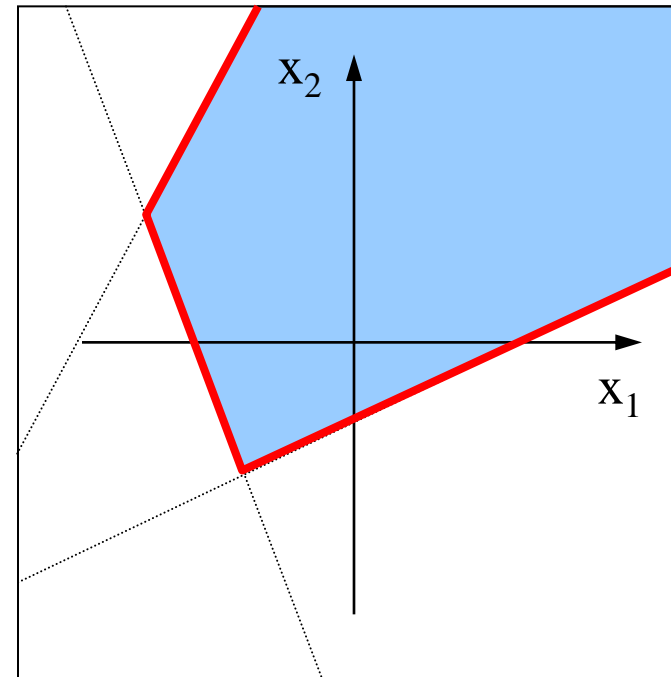
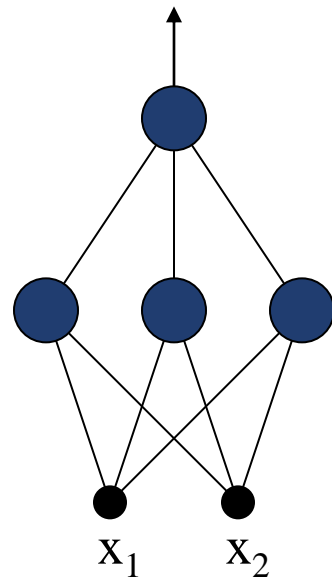
## Network with three layers

- A network is able to locate a number of convex areas  $\leq$  number of neurons in the hidden layers



## Network with three layers

- A network is able to locate a number of convex areas  $\leq$  number of neurons in the hidden layers



## Network with three layers

- A network is able to locate a number of convex areas  $\leq$  number of neurons in the hidden layers

