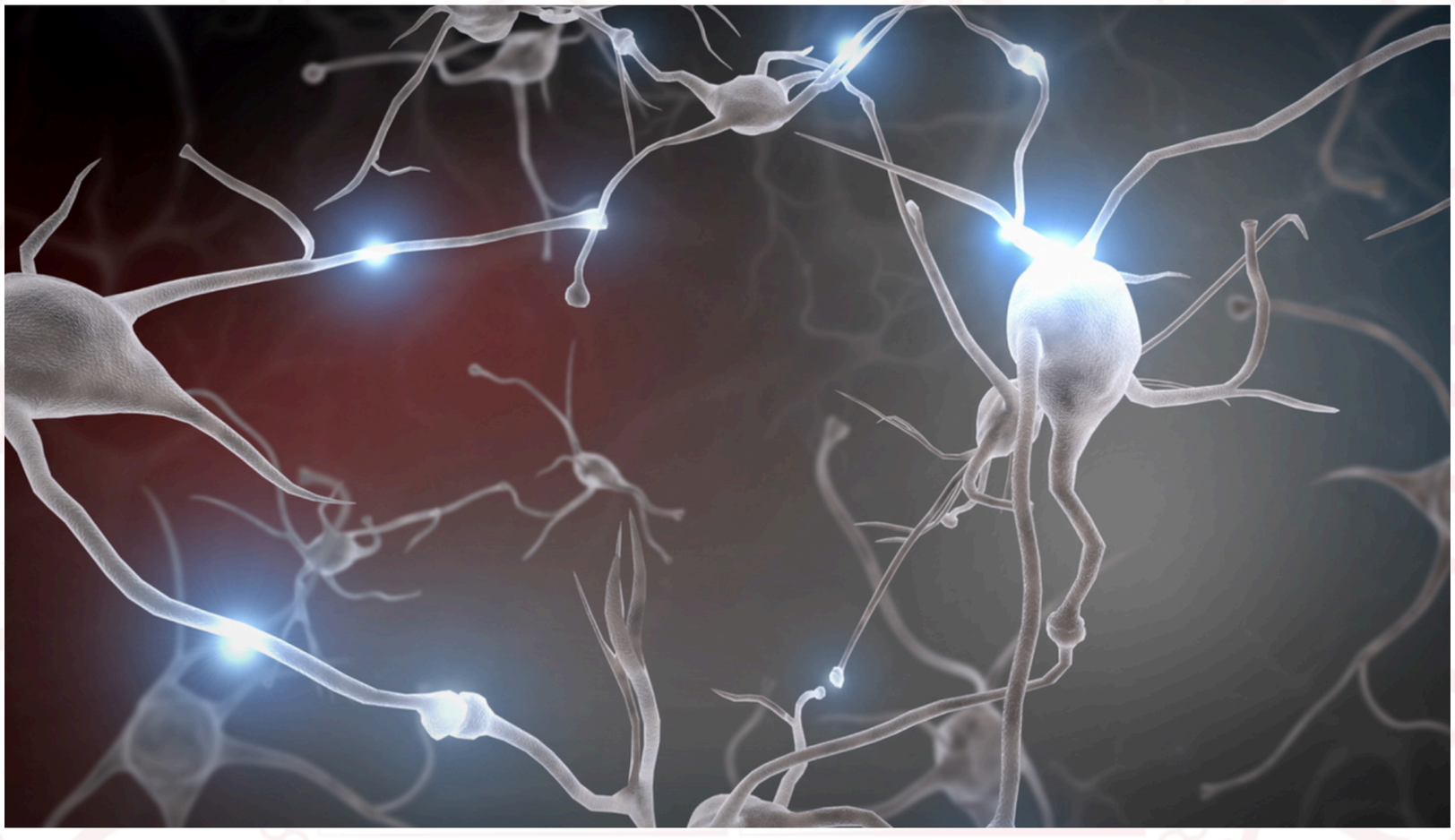
Deep Learning

LM Computer Science, Data Science, Cybersecurity

2nd semester - 6 CFU

Luca Pasa, Nicolò Navarin & Alessandro Sperduti

Neural Networks



Types of problems in AI

- In the early days of artificial intelligence solved Problems
  - Intellectually difficult for human beings
  - Straightforward for computers
  - E.g. play chess
- True challenge
  - Solve problems hard to describe formally
  - Recognizing spoken words, recognize objects images...
- Recognize object/Speech seems easier than play chess, but…
  - Requires an immense amount of knowledge about the world
  - Much of this knowledge is subjective and intuitive

Why Deep?

- How humans solve this kind of problems?
  - Experience
  - Knowledge
  - Relation to simpler concepts
    - Learn complicated concepts by build building them out simpler ones

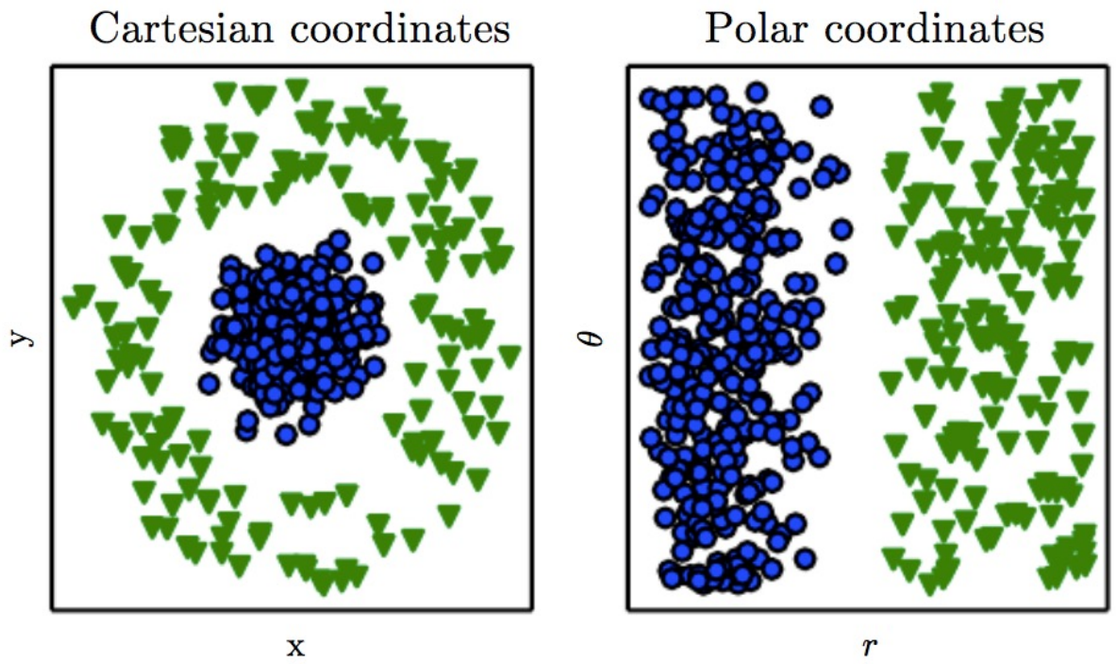
“If we draw a graph showing how complex concepts are built on top of each other, the graph is deep, with many layers.

For this reason, we call this approach to AI

deep learning”

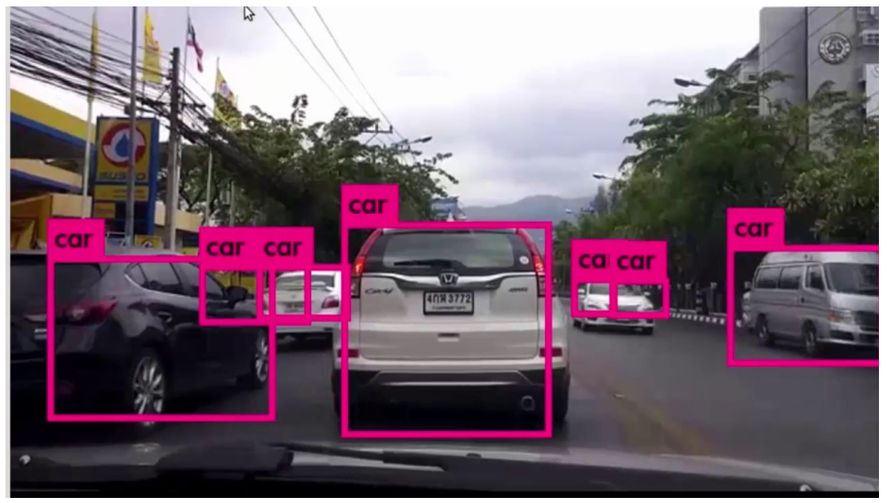
Knowledge Representation

- knowledge base approach
  - Hard-code the knowledge
  - Shows problems in real-world tasks
- AI systems need the ability to acquire their own knowledge!
  - Extracts patterns from raw data
  - Heavily depends on the representation of the data



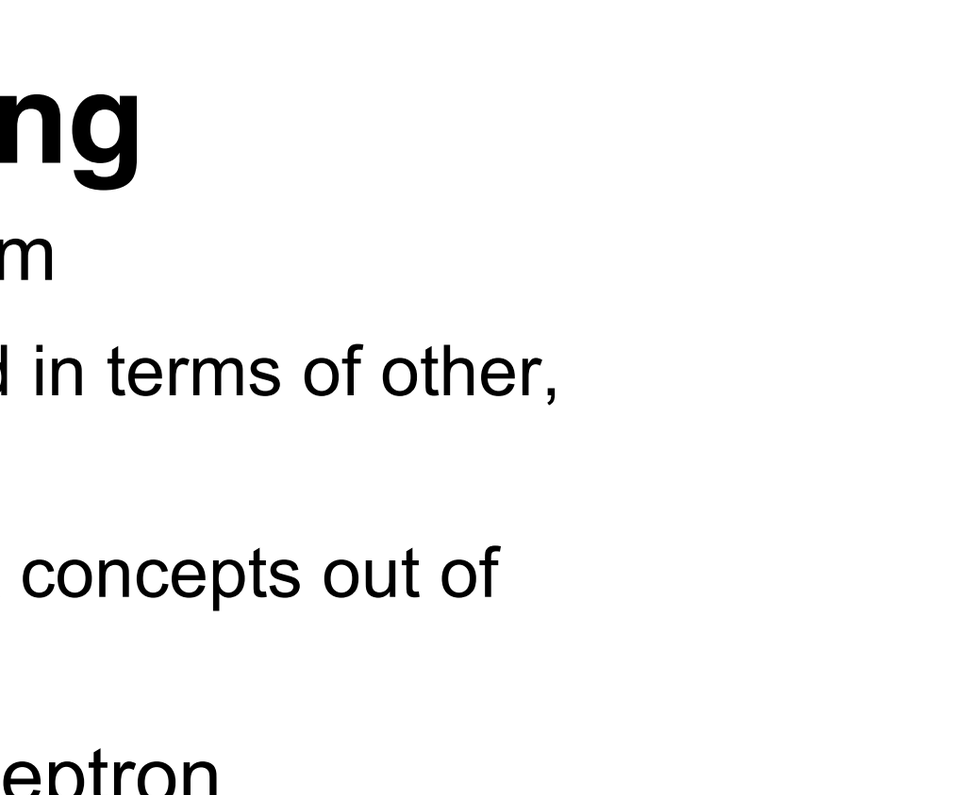
Representation learning

- Many tasks can be solved by designing the right set of features
- But in general, it is difficult to know which features should be extracted
- Features have to highlight the factors of variation
  - E.g: the position of the car, colour, angle, brightness of the sun
  - Factors of variation influence every single piece of data
  - Can be very difficult to extract such high-level, abstract features from raw data!
- use machine learning to discover:
  - The mapping from representation to output
  - The representation itself!

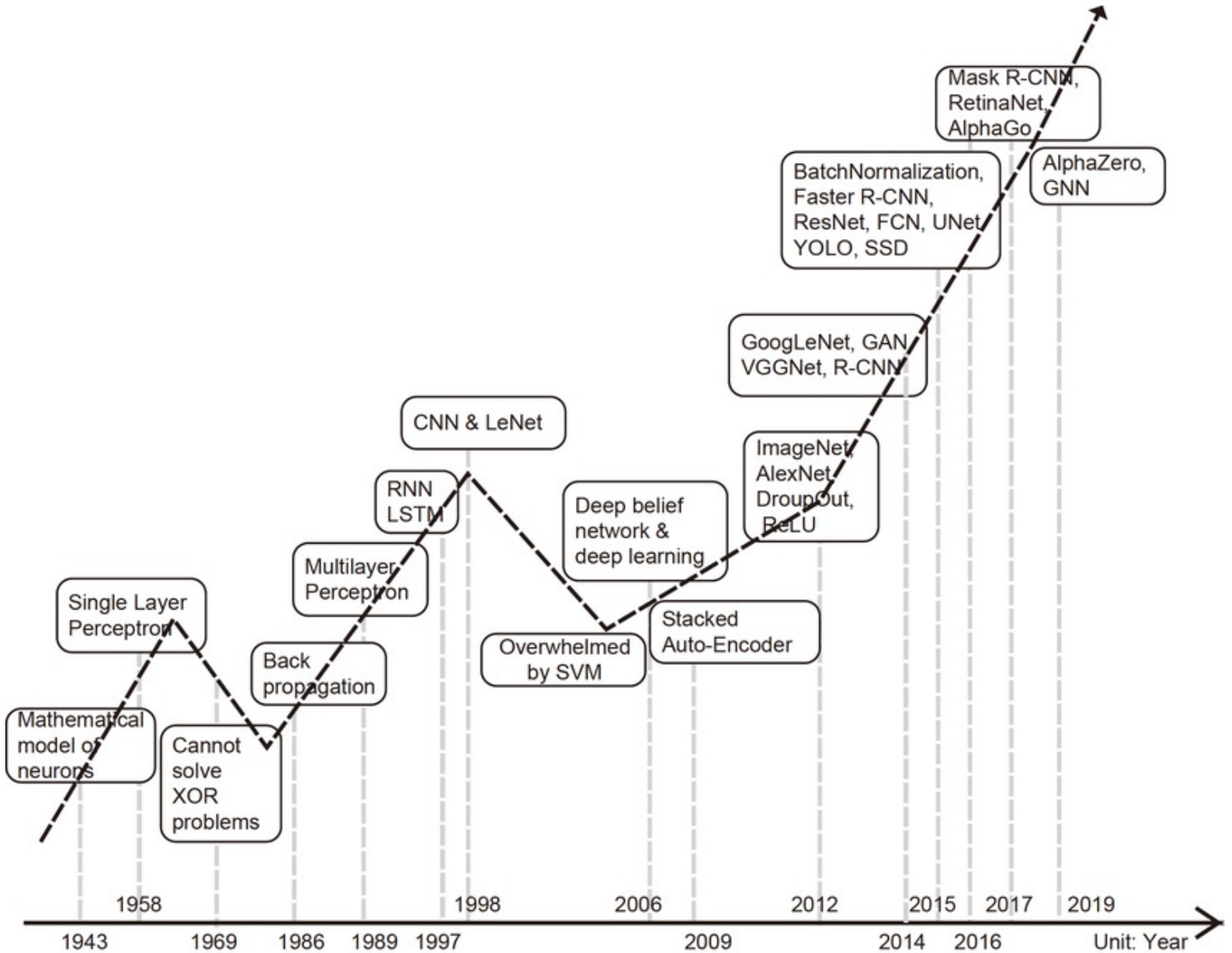


Deep Learning

- Deep learning solves this central problem
  - Extracts representations are expressed in terms of other, simpler representations
  - Enables the computer to build complex concepts out of simpler concepts
- Deep Neural Network or Multilayer Perceptron
  - Mathematical function mapping some set of input values
  - Formed by composing many simpler functions
  - Different function providing a new representation of the input



From Neural Networks To Deep Learning

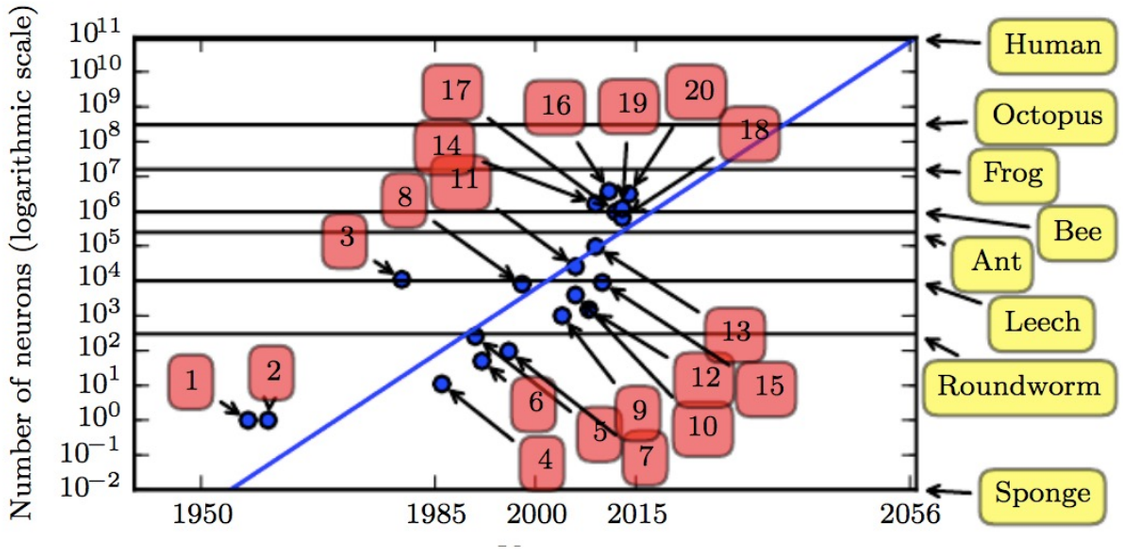


What Happened in 2006?

- Neural Computation and Adaptive Perception (NCAP) program
  - Deep networks were generally believed to be very difficult to train
  - Algorithms were too computationally costly
- In 2006:
  - Deep Belief Networks (Hinton et al., 2006)
    - Could be efficiently trained
      - Greedy layer-wise pretraining (unsupervised)
      - Can be used to train many other kinds of deep networks
  - Availability of faster CPUs, the advent of general-purpose GPUs

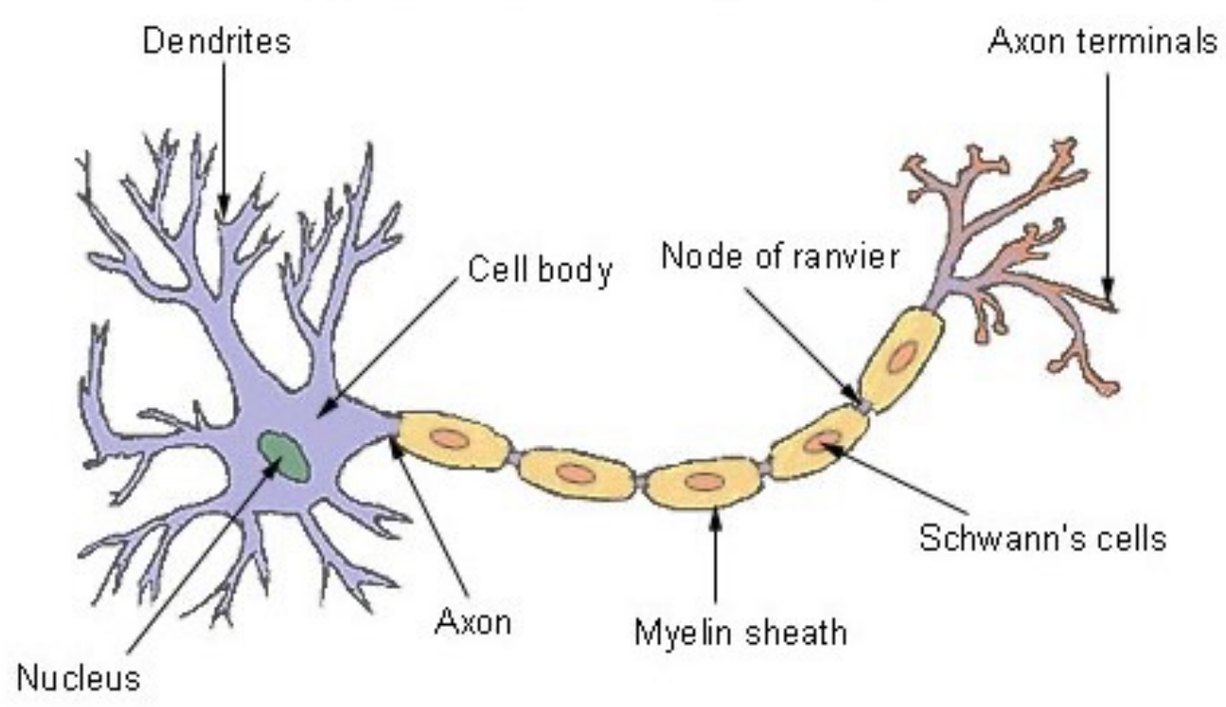
Evolution of Deep Learning

- DNN outperformed competing ML methods
- Increasing Dataset Sizes
  - Achieve acceptable performance: around 5,000 labeled examples per category
  - Match or exceed human performance: at least 10 million labeled examples
  - Big Data
- Increasing the model size
  - Hidden units doubled in size roughly every 2.4 years
  - Same number of neurons as the human brain in the 2050s



Neural Networks

Biological inspiration: **neuron**



The major components are:

- **Dendrites:** takes the input from other neurons in form of an electrical impulse
- **Cell Body**: elaborates those inputs and decide what action to take
- **Axon terminals**: transmit outputs in form of electrical impulse

Each neuron:

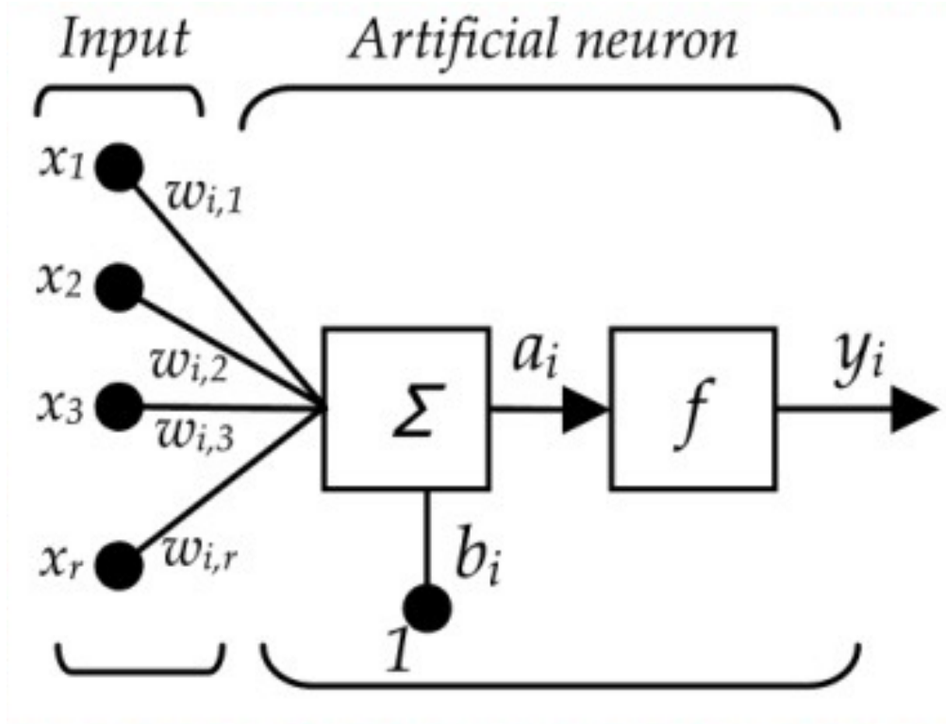
1. takes input from numerous other neurons through the **dendrites**
2. performs the required **processing** on the input
3. sends another electrical pulse through the **axon** into the terminal nodes from where it is transmitted to numerous other neurons.

Neural Networks

- Inspired by the human brain
  - Human brain constitute by about 10^10 strongly interconnected neurons
  - Each Neuron possesses a number of connections that goes from 10^4 to about 10^5
  - The response time of neuron is about 0.001 seconds
  - Considering that for recognizing the content of a scene a human takes about 0.1 seconds, it follows that the human brain heavily exploits parallel computing
    - In fact it can not make more than 100 serial calculation [0,1/0.001=100]

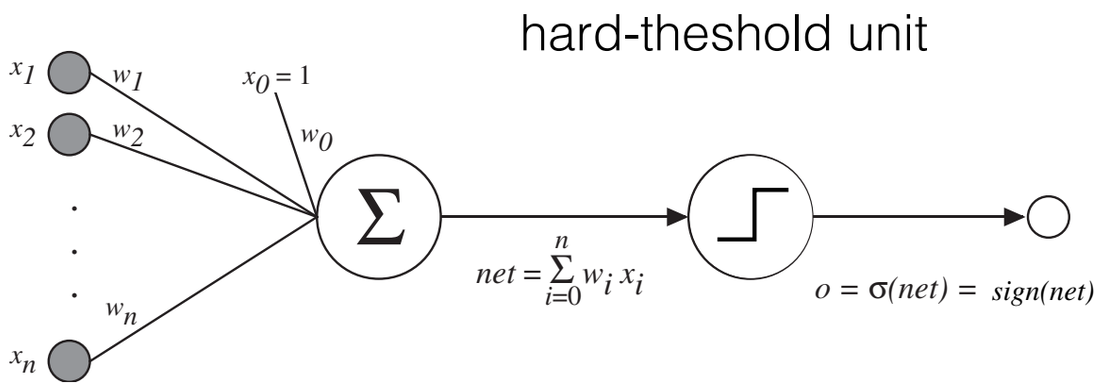
Artificial neuron

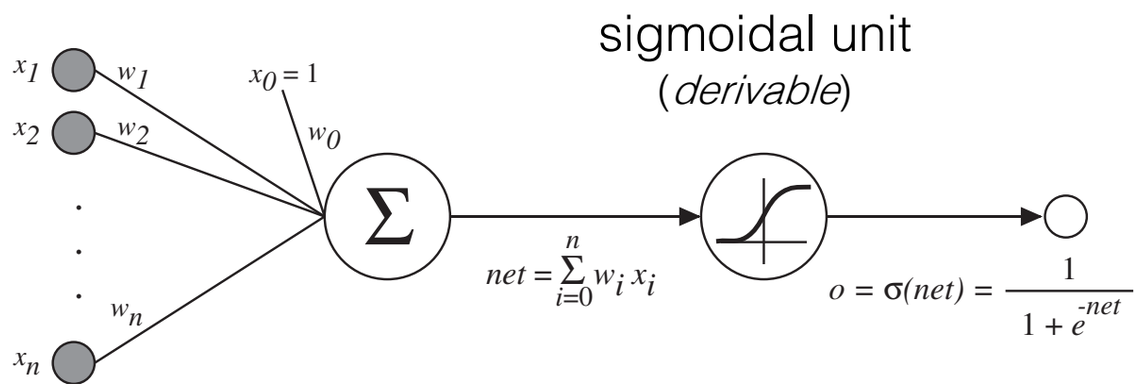
- A neuron computes a non-linear function over the inputs
  - Perceptron
- Its output depends on the input and a set of **weights**
- The weights have to be **learned**
- Some relationships **cannot be modelled using a single neuron**.



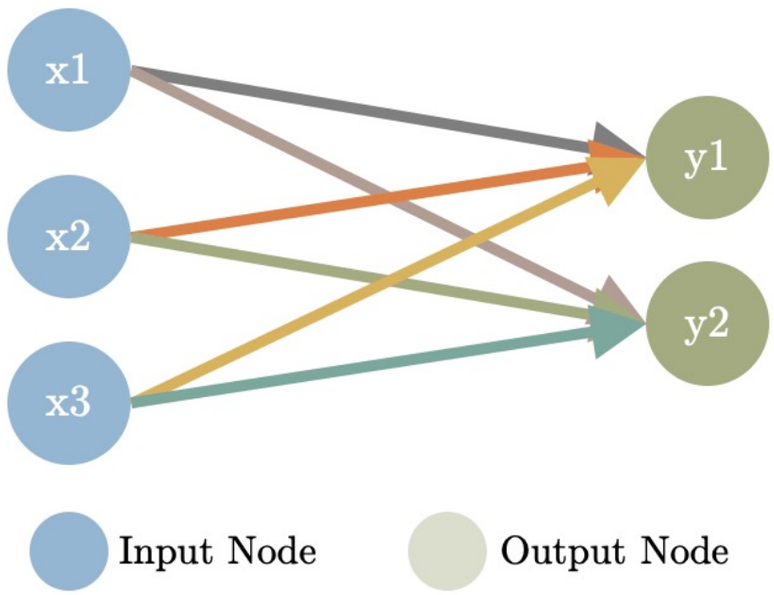
Artificial Neural Networks

- A System consisting of interconnected units that compute nonlinear (numerical) functions





- Adjustable weights are associated with connections among units

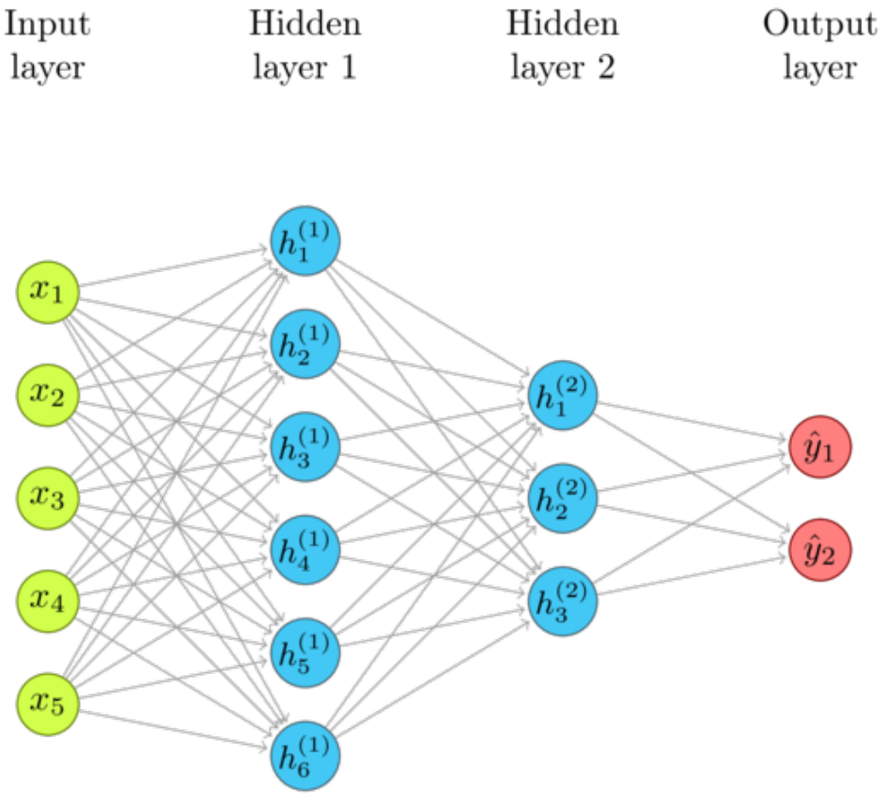


- *Input units:* represent input variables
- *output units:* represent output variables

Deep Feed Forward Neural Networks

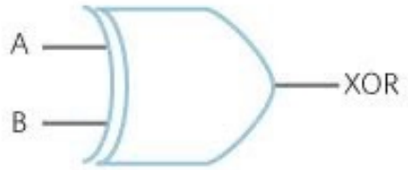
- Also called multilayer perceptrons (**MLPs**)
- Approximate some function f^*: $y = f^*(x)$
  - Maps an input x to a category y
  - The MLP defines a mapping $\hat{y} = f(x; \Theta)$
    - Θ: values of the parameters
- Typically represented as a composition of many different functions
  - $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$
  - $f^{(i)}$: i-th layer
- Intermediate layers: hidden layer
  - Behaviors not directly specified by the training data
- Final layer: output layer
  - The training examples specify directly what the output layer must do

The idea behind using multiple layers is that complex relations can be broken into simpler functions and combined.



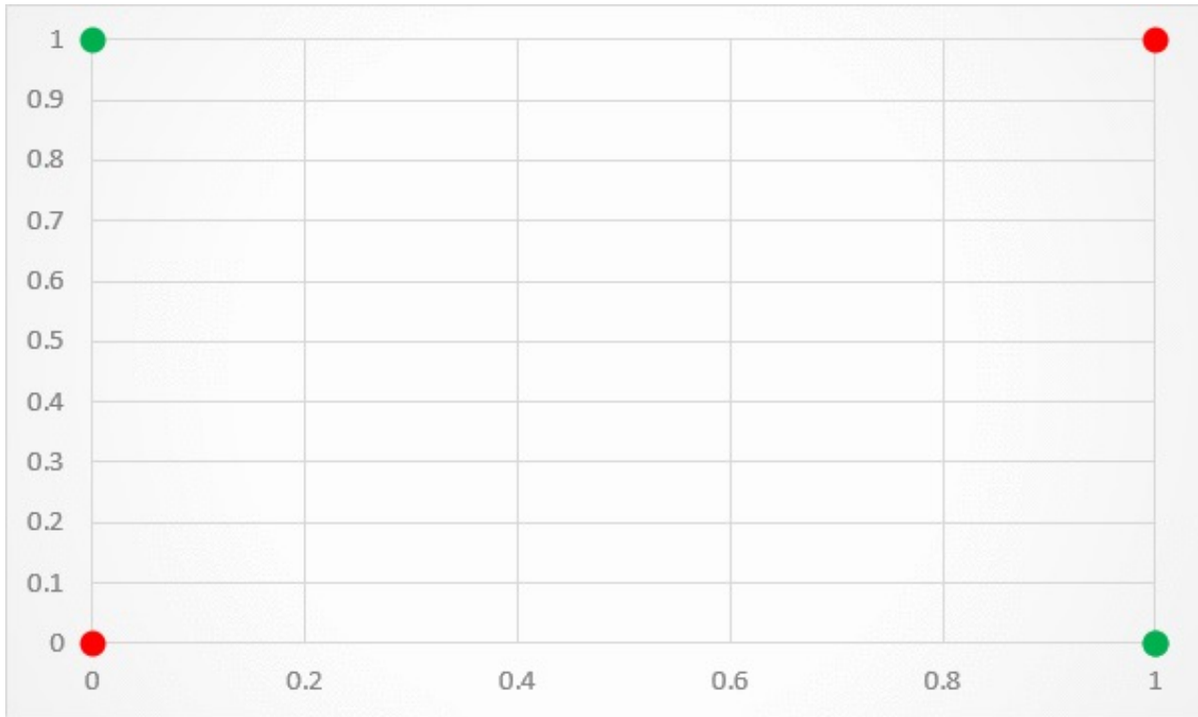
Why deep? An example

- XOR function $X = A \oplus B$



A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

- $Tr = \begin{Bmatrix} ([0,0],0),([0,1],1), \\ ([1,0],1),([1,1],0) \end{Bmatrix}$

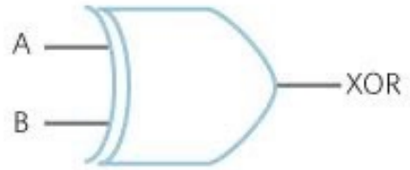


- Let us define our model as
  - $f(\boldsymbol{x}; \boldsymbol{w}; b) = x\boldsymbol{w}^T + b$

can you find a model (a hyperplane) that correctly classifies the four points?

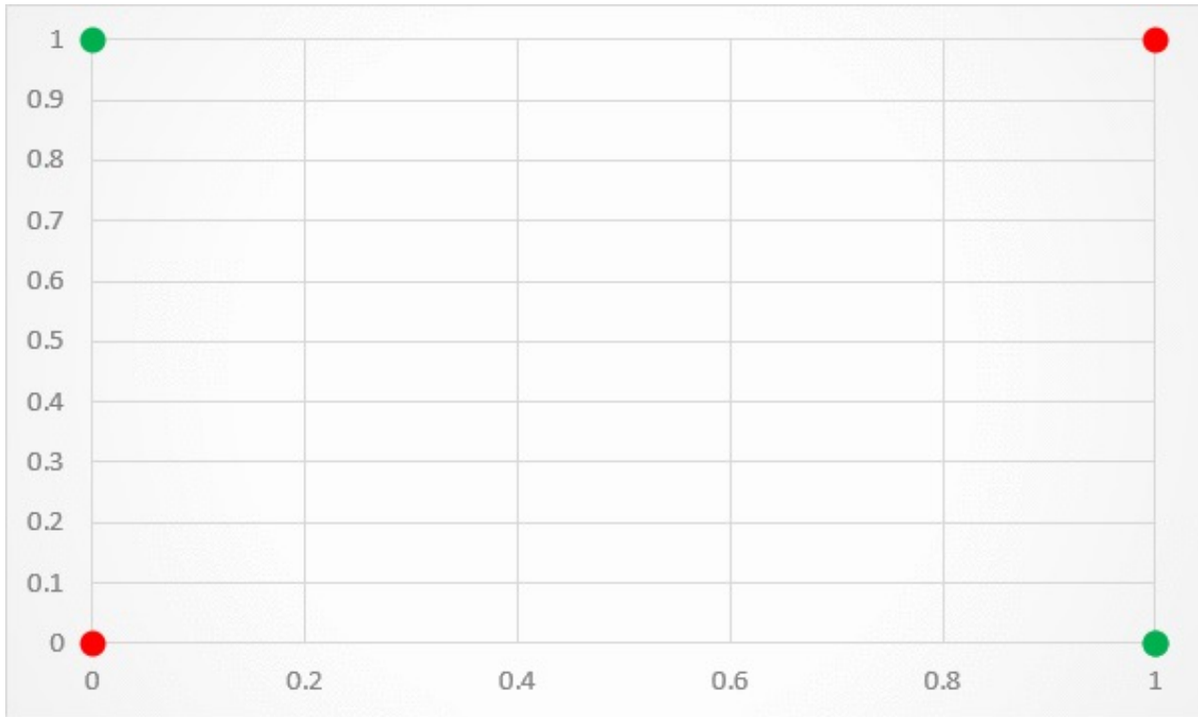
Why deep? An example

- XOR function $X = A \oplus B$



A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

- $Tr = \begin{Bmatrix} ([0,0],0),([0,1],1), \\ ([1,0],1),([1,1],0) \end{Bmatrix}$



- Let us define our model as
  - $f(\boldsymbol{x}; \boldsymbol{w}; b) = x\boldsymbol{w}^T + b$

- The XOR function cannot be represented by any linear classifier
- Including a single-layer neural network

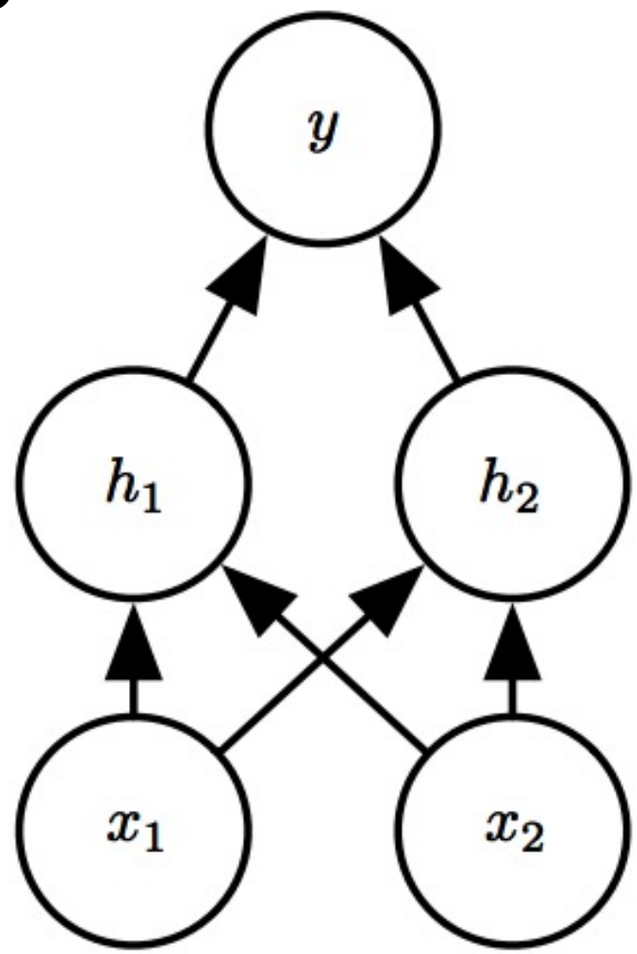
2-layers network

- We want to model $f^2(f^1(\boldsymbol{x}))$
  - Hidden representation

$$\boldsymbol{h} = f^1\left(\boldsymbol{x}; W^{(1)}; \boldsymbol{b}^1\right) = xW^{(1)T} + \boldsymbol{b}^{(1)}$$

  - Now $\boldsymbol{W}^{(1)}$ is a matrix, and $\boldsymbol{h}$ is a vector!
- Since the output is a number,
  - $f^2\left(\boldsymbol{h}; \boldsymbol{w}^{(2)}; b^2\right) = \boldsymbol{h}\boldsymbol{w}^{(2)T} + b^{(2)}$
- Note: both f^1 and f^2 are linear

Affine transformation from vector to vector



Is this 2-layer network ANY BETTER?

The need for Non-linearity

- Consider a network with n layers
- If the functions $f^{(i)}$ are linear, we can represent them as a matrix multiplication

$$f^{(1)}(\boldsymbol{x}) = \boldsymbol{x}W^{(1)T},\ f^{(2)}(\boldsymbol{x}) = \boldsymbol{x}W^{(2)T}..$$

- Then we can rewrite the output layer as

$$f^{(n)}(\boldsymbol{x}) = \boldsymbol{x}W^{(1)T}W^{(2)T} \ldots \boldsymbol{w}^{(n)T} = \boldsymbol{x}\boldsymbol{w}^{(L)T}$$

with $\boldsymbol{w}^{(L)} = W^{(1)}\ W^{(2)} \ldots \boldsymbol{w}^{(n)}$

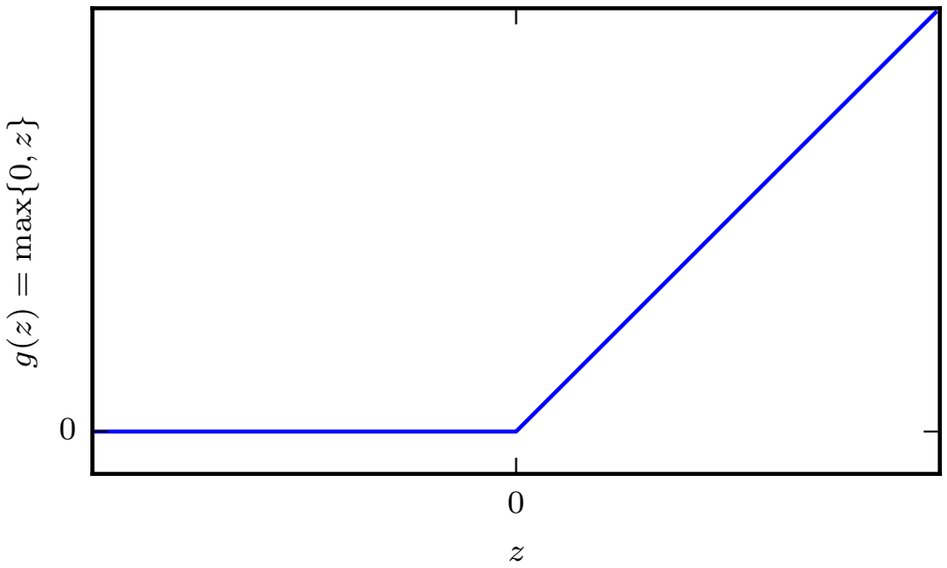
- The deep network would be linear, i.e. it is equivalent to a single-layer network

Non-linearity and XOR

- Thus, $f^{(1)}$ must be non-linear:
- **Affine** transformation controlled by learnable parameters followed by
- fixed non-linear function: **activation** function

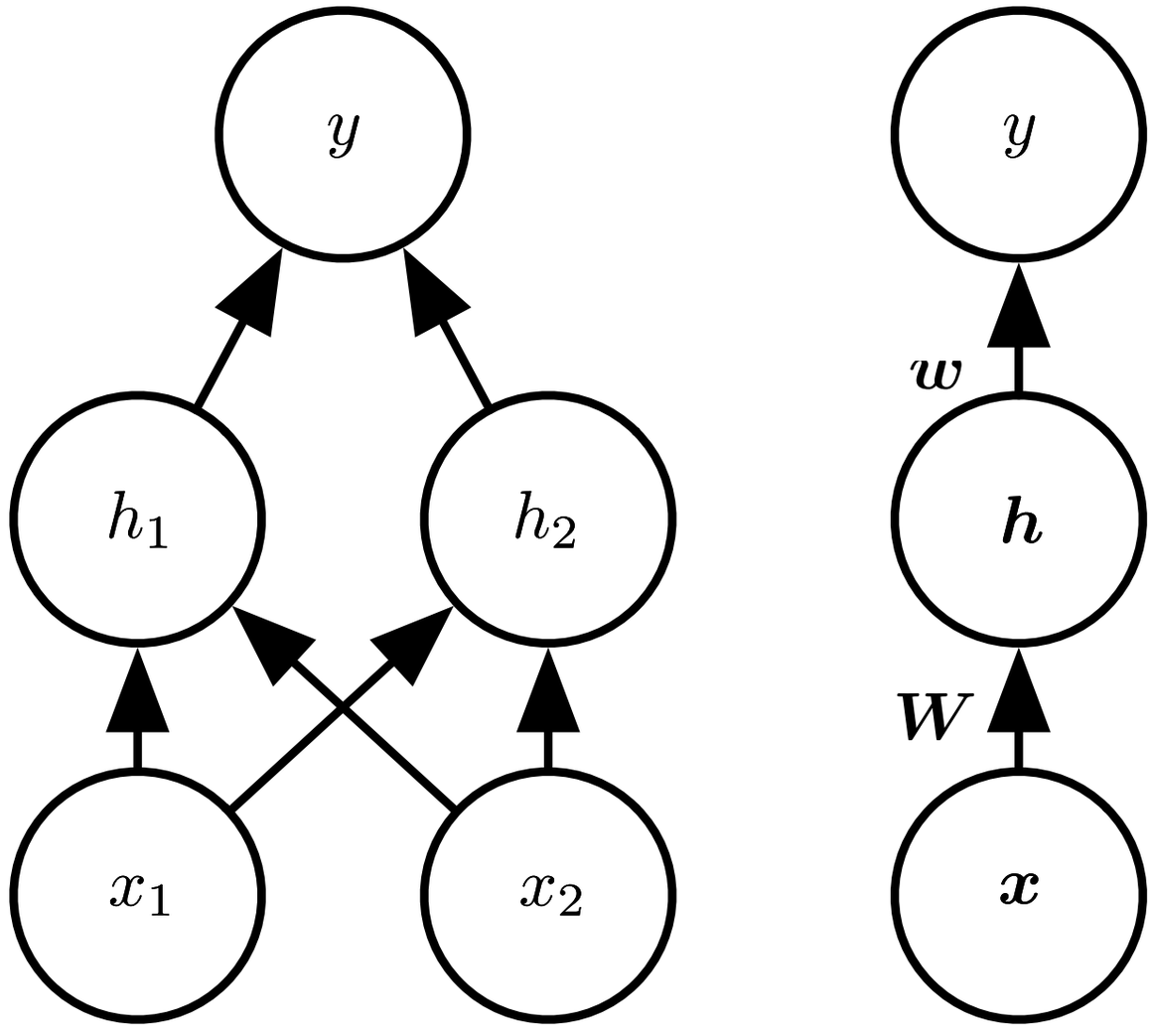
$$\boldsymbol{h} = \sigma(\boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{x} + \boldsymbol{c})$$

Element-wise non-linear transformation, e.g. $ReLU(z) = \max\{0, z\}$



- The network becomes
  - $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \mathbf{w}^{\mathrm{T}}\max\{0, \mathbf{W}^{\mathrm{T}}\mathbf{x} + \mathbf{c}\} + \mathrm{b}$

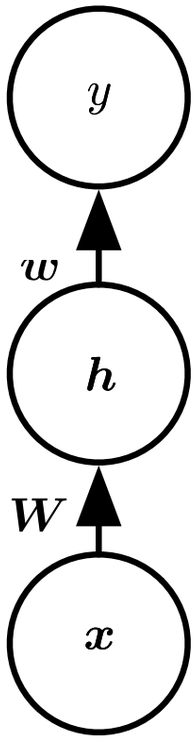
Network diagrams



- Left: every unit has a node (neuron)
- Right: each node represents a whole layer

Solution for XOR problem

- $Tr = \{([0,0], 0), ([0,1], 1), ([1,0], 1), ([1,1], 0)\}$
- $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \mathbf{w}^{\mathrm{T}} \max\{0, \mathbf{W}^{\mathrm{T}}\mathbf{x} + \mathbf{c}\} + \mathrm{b}$

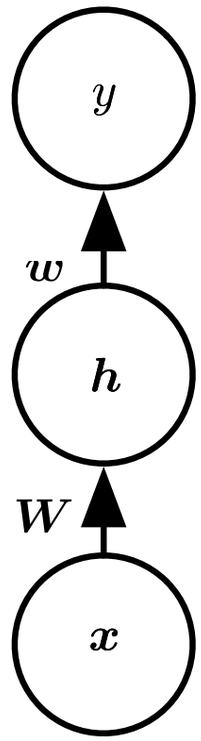


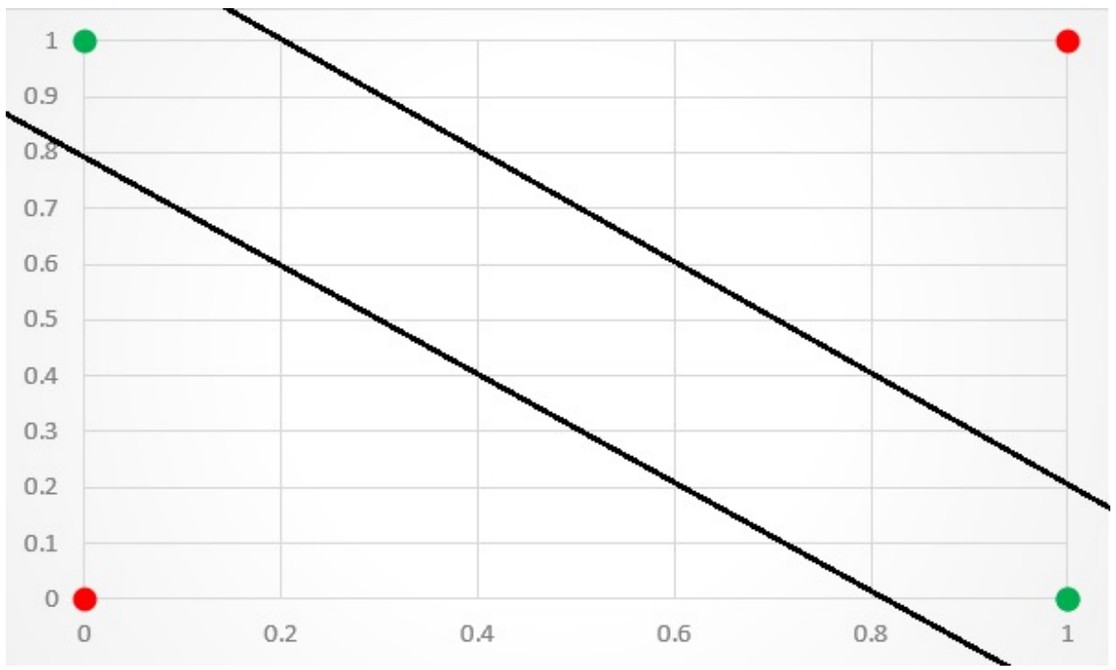
Check if such a network classifies correctly all the 4 inputs

Solution for XOR problem

- $Tr = \{([0,0], 0), ([0,1], 1), ([1,0], 1), ([1,1], 0)\}$
- $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \mathbf{w}^{\mathrm{T}} \max\{0, \mathbf{W}^{\mathrm{T}}\mathbf{x} + \mathbf{c}\} + \mathrm{b}$

With $\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0$

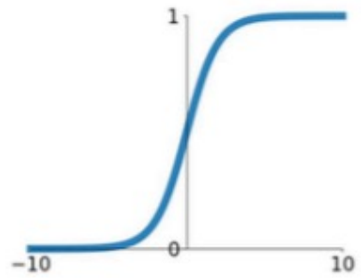




Nonlinear activation functions

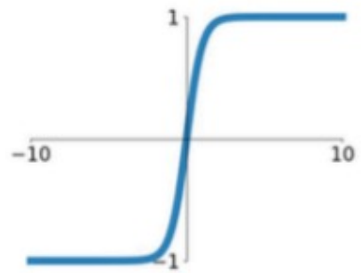
Sigmoid

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



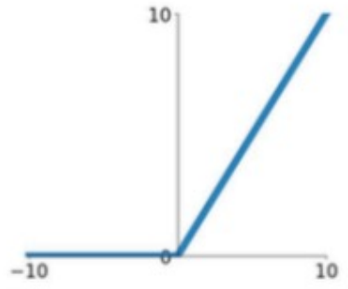
tanh

$$\tanh(x)$$



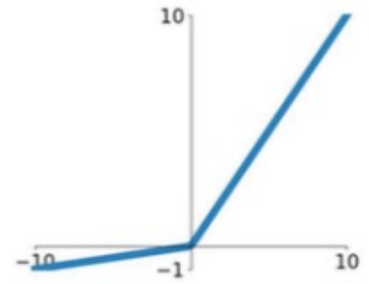
ReLU

$$\max(0, x)$$



Leaky ReLU

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

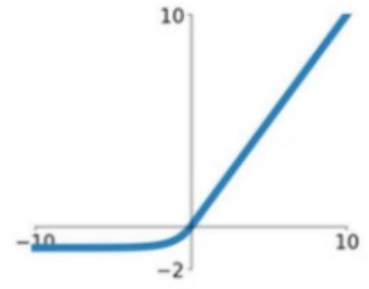


Maxout

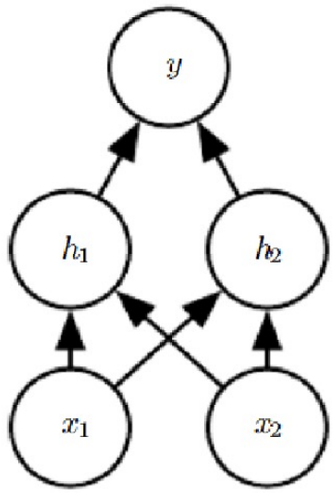
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

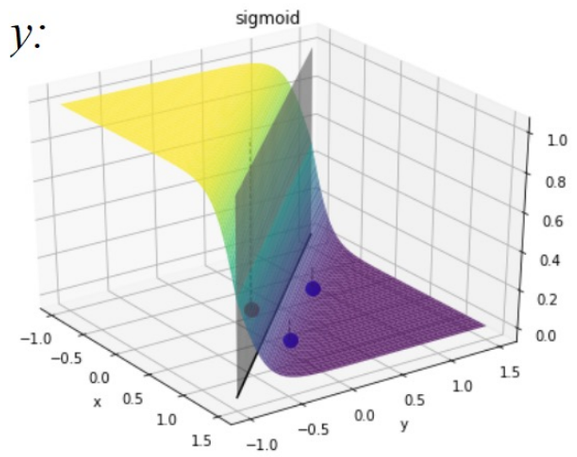
ELU

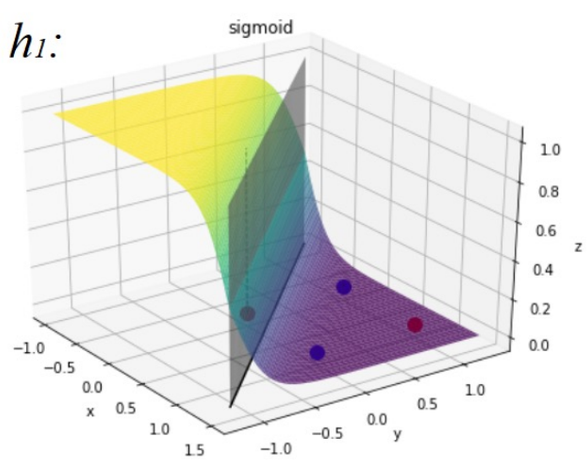
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

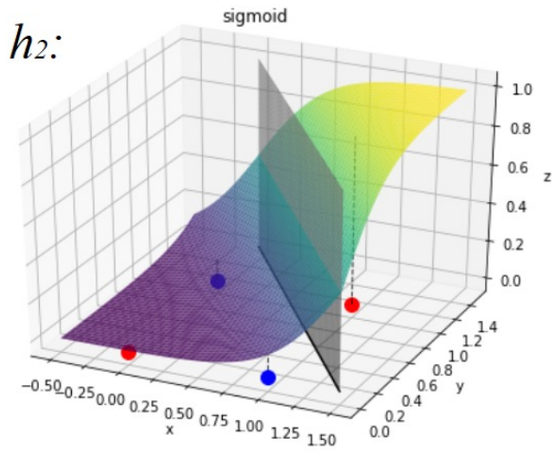


- Goals: have a more complex decision surface
  - Use non-linear activation functions
  - Stack several hidden layers









Learning in a Network of Perceptrons

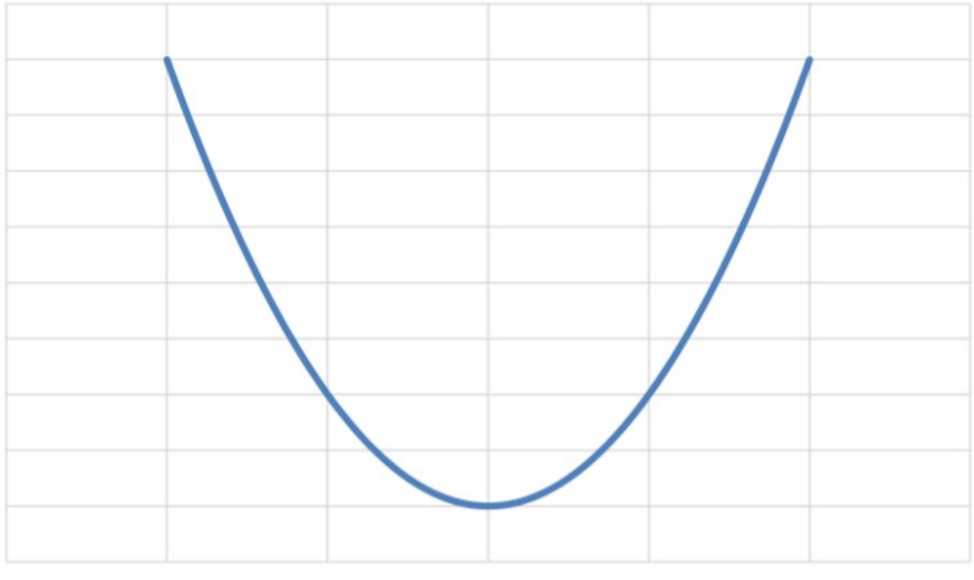
A single Perceptron in not able to learn all possibile boolean functions (e.g., XOR)
However a network of Perceptrons can implement any boolean function (via AND, OR, NOT).
Problem: how to train a network of Perceptrons ?
It is not clear how to assign "credit" or "blame" to the hidden units:

CREDIT ASSIGNMENT PROBLEM

A possibile solution is to make a single neuron derivable and exploit the Gradient Descent technique to learn the "right" weights.

Gradient-based learning in NNs

- Linear models are formulated as convex models
- The non-linearity in neural networks makes the problem non-convex
  - No guarantee of achieving the global optimum
  - Sensitive to the starting point (usually small random values are used)
- We will discuss in detail in a following lecture



Cost Function

- An important aspect of the design of a deep neural network is the choice of the cost function
- In most cases the model defines a distribution $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$
- We can use the principle of maximum likelihood, so…
  - The cost function is the cross-entropy between training labels and network predictions (negative log-likelihood)

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

- The specific form of the cost function changes from model to model
  - The cost function choice is tightly coupled with the choice of output units
  - Output representation $\boldsymbol{h} = f(\boldsymbol{x}; \boldsymbol{\theta})$ determines the form of the cross-entropy function

Linear units for Gaussian output

- Very simple kind of output unit is based on an affine transformation with no nonlinearity **(linear units)**

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

- Linear output layers are often used to produce the mean of a gaussian distribution

$$p_{model}(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I})$$

- Maximizing the likelihood corresponds to minimize the mean squared error

$$J(\boldsymbol{\theta}) = \frac{1}{2}\mathbb{E}_{p\sim\hat{p}_{data}}\left(y - f(\boldsymbol{x};\boldsymbol{\theta})\right)^2 + constant$$

Sigmoid units for Bernoulli out distr

- Binary output y (e.g. binary classification problems)

Bernoulli distribution: single parameter $\phi \in [0,1]$

$$P(\mathrm{x} = 1) = \phi, P(\mathrm{x} = 0) = 1 - \phi$$

In general:

$$P(\mathrm{x} = x) = \phi^{x}(1 - \phi)^{1-x}$$

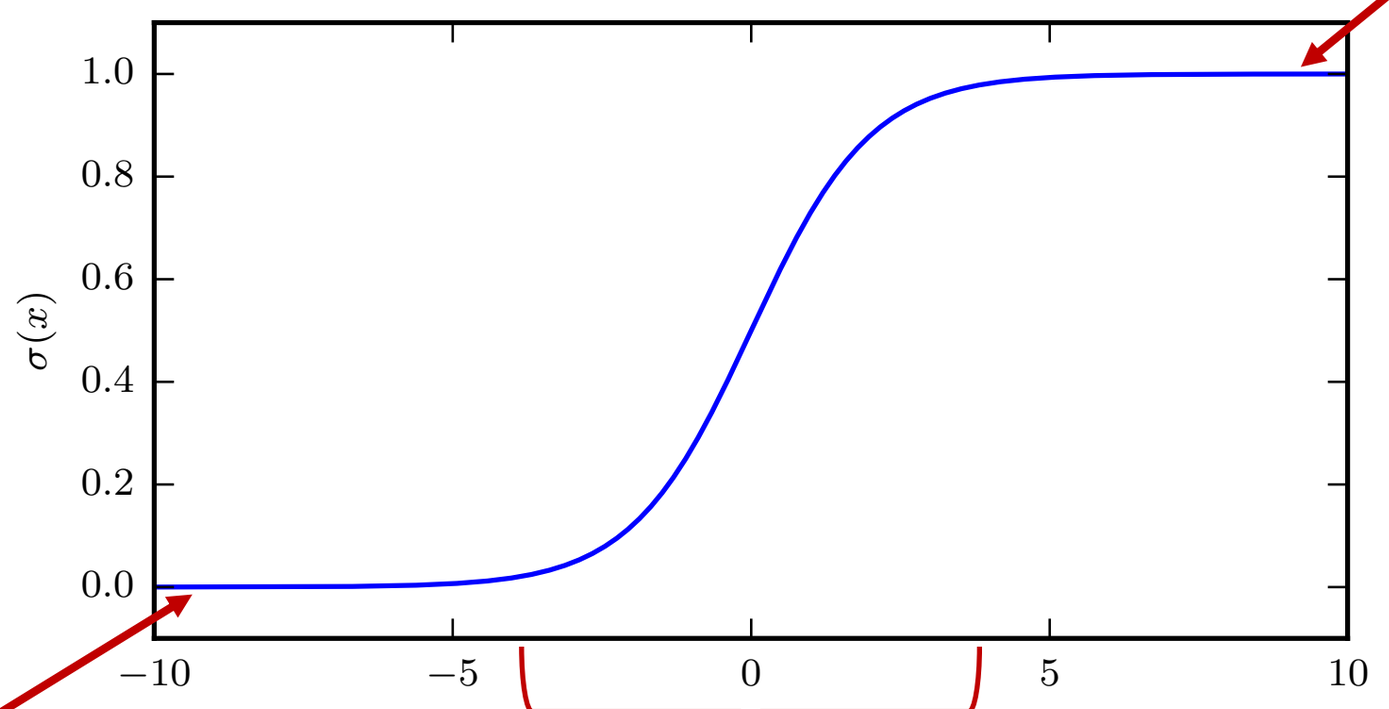
- The NN just needs to predict $P(\mathrm{y} = 1|\boldsymbol{x})$
  - We want to force this number to lie in $[0,1]$
- E.g. $P(\mathrm{y} = 1|\boldsymbol{x}) = max\{0, min\{1, \boldsymbol{w}^T \boldsymbol{h} + b\}\}$
  - not a good choice for gradient descent
    - output outside $[0,1]$ → gradient $\mathbf{0}$
- We want to ensure there is always some gradient when the model is wrong: linear layer + **sigmoid activation**

$$\hat{y} = \sigma(\boldsymbol{w}^T \boldsymbol{h} + b)$$

Sigmoid function

- Sigmoid or Logistic function

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



Sigmoid units for Bernoulli Output

- An output unit is composed of two components
  - $z = \boldsymbol{w}^T \boldsymbol{h} + b$
  - $\sigma(\cdot)$: the activation function to convert z into a probability
- How can we define a probability distribution over y using the value z ?
  - Let's construct an unnormalized probability distribution $\tilde{P}(y)$
    (We omit the dependence on x for the moment)
    - Assumption: unnormalized log probabilities are linear in y and z

$$log\tilde{P}(y) = yz \quad [\text{ i.e. } log\tilde{P}(\mathrm{y}=1) = z,\ log\tilde{P}(\mathrm{y}=0) = 0\ ]$$

  - we can exponentiate to obtain the unnormalized probabilities

$$\tilde{P}(y) = e^{yz}$$

  - then we normalize to obtain a proper probability (sum = 1)

$$P(y) = \frac{e^{yz}}{\sum_{y'=0}^{1} e^{y'z}} = \sigma((2y-1)z)$$

$$[\text{ i.e. } P(\mathrm{y}=1) = \frac{e^z}{1+e^z} = \frac{1}{\frac{1+e^z}{e^z}} = \frac{1}{1+\frac{1}{e^z}} = \frac{1}{1+e^{-z}}, P(\mathrm{y}=0) = \frac{1}{1+e^z}\ ]$$

Maximum likelihood learning of a Bernoulli

- This approach to predicting the probabilities in log space is natural to use with maximum likelihood learning
  - Because the cost function used with maximum likelihood is

$$-logP(y|\boldsymbol{x})$$

    - the log in the cost function undoes the exp of the sigmoid
      - Without this effect, the saturation of the sigmoid could prevent gradient based learning from making good progress

softplus
$\zeta(x) = \log(1 - \exp(x))$

$$J(\boldsymbol{\theta}) = -\log P(y|\boldsymbol{x}) = -\log \sigma\big((2y-1)z\big) = \zeta((1-2y)z)$$

- Saturation occurs when $y = 1$ and z is very positive or $y = 0$ and z is very negative
  - So, when the model has the right answer!
- Note: other loss functions such as MSE would saturate when $\sigma(z)$ saturates

Softmax for Multinoulli out distr

- Output: probability ditribution over a discrete variable with n possible values
  - **Softmax**
    - generalization of sigmoid
    - used as the output of a classifier, to represent the probability distribution over n different classes
- We have to generate a vector $\hat{\boldsymbol{y}} = [\hat{y}_0, \hat{y}_1, \ldots, \hat{y}_{n-1}]$, where $\hat{y}_i = P(y = i \mid \boldsymbol{x})$
- We require $\forall i, 0 \leq \hat{y}_i \leq 1$ and $\sum_i \hat{y}_i = 1$
- Same approach for the Bernoulli distribution generalizes to the Multinoulli distribution
  - Linear layer predicts unnormalized log probabilities :

$$\boldsymbol{z} = \boldsymbol{W}^T \boldsymbol{h} + \boldsymbol{b} \quad \text{Where} \quad z_i = \log \tilde{P}(y = i \mid x)$$

Softmax for Multinoulli out distr

$$softmax(\mathbf{z})_i = \frac{e^{z_i}}{\sum_j^n e^{z_j}}$$

- use of e works well when training the softmax to output a target value y using maximum log-likelihood
  - Defining the $softmax$ in terms of e is natural
  - the log in the log-likelihood undo the e of the $softmax$

$$\log\ softmax(\mathbf{z})_i = z_i - \log \sum_j e^{z_j}$$

Pushes the correct label up

Pushes the uncorrect labels down

Note: softmax is a continuous and differentiable

Output functions in general

- Linear, sigmoid, and softmax output units are the most common
  - NN can generalize to almost any kind of output layer

- Maximum likelihood provides a guide to design almost any output layer
  - We define a conditional distribution $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$
  - As cost function maximum likelihood suggests to use

$$-\log p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$$

- We can think of the NN as $f(\boldsymbol{x};\boldsymbol{\theta}) = \boldsymbol{\omega}$
  - $\boldsymbol{\omega}$: parameters of a distribution over $\boldsymbol{y}$
  - Cost function $-\log p(\boldsymbol{y};\boldsymbol{\omega}(\boldsymbol{x}))$

Hidden units

- Hidden units can be described as accepting an input x

$$z = W^T x + b$$

  - applying an element-wise nonlinear function $g(z)$

- Design of hidden units not yet have many definitive guiding theoretical principle
  - Many other types of hidden units are available
  - Impossible to define the best beforehand
    - Evaluating hidden unit performance on a validation set

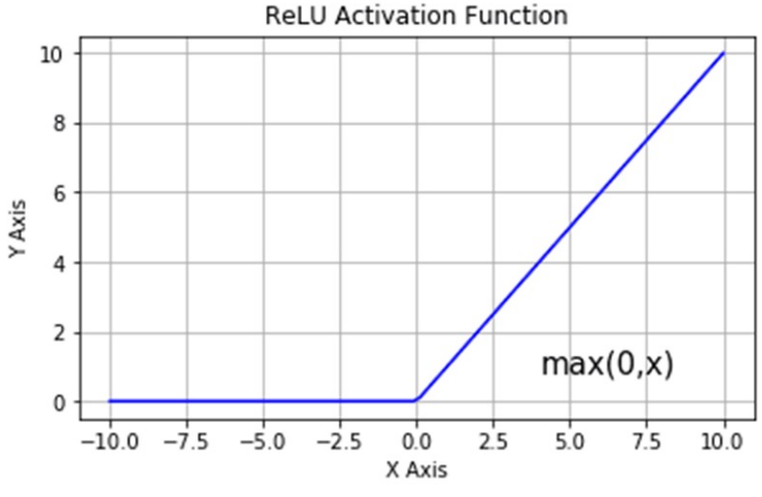
To select the most suitable activation function we can rely on some basic intuitions motivating each type of hidden unit

Hidden units: ReLU

$$g(z) = \max(0, z), \qquad \mathrm{z} = g(\boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b})$$

- Similar to linear units (easier to optimize)
- Relu (as some other units) is not actually differentiable at all input points ($\mathrm{z} = 0$)
  - return right or left derivative
  - Hidden units that are not differentiable are usually nondifferentiable at only a small number of points
- ReLU does not learn on examples that gives 0 gradient
  - Slope α_i when $z_i \leq 0$:
    - generalizations of Relu: e.g leakyReLU

$$g \quad (z, \alpha)_i = \max(0, z_i) + \alpha_i \min(0, z_i)$$

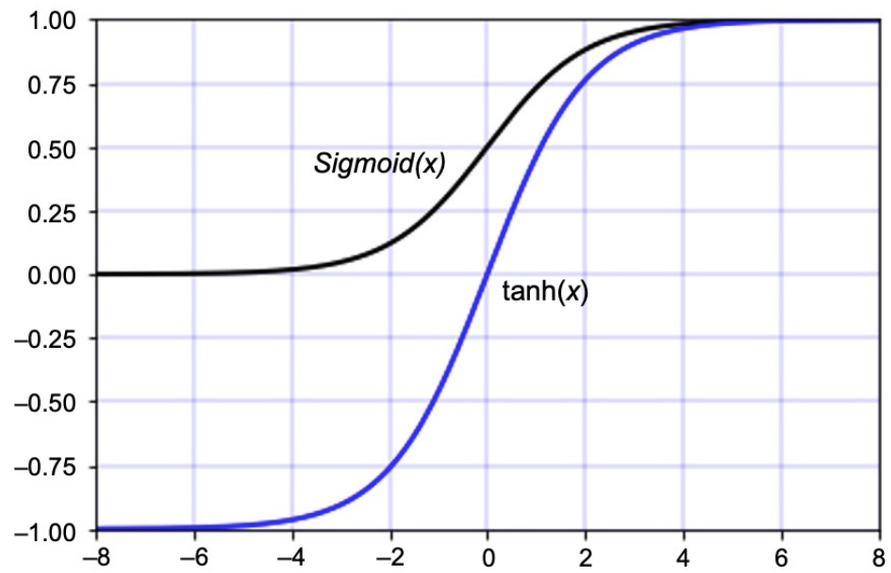


Hidden units: Tanh and sigmoid

Logistic Sigmoid: $g(z) = \sigma(z)$

Hyperbolic Tangent: $g(z) = tanh(z) = 2\sigma(2z) - 1$

- Saturate across most of their domain
  - To a high value when z is very positive, to a low when z is very negative
  - Strongly sensitive when z is near 0
- Tanh typically performs better than the logistic sigmoid
  - similar to the identity function near 0
- In general, many differentiable functions are reasonable (e.g. cos(x)), but they show no significant advantage over common ones



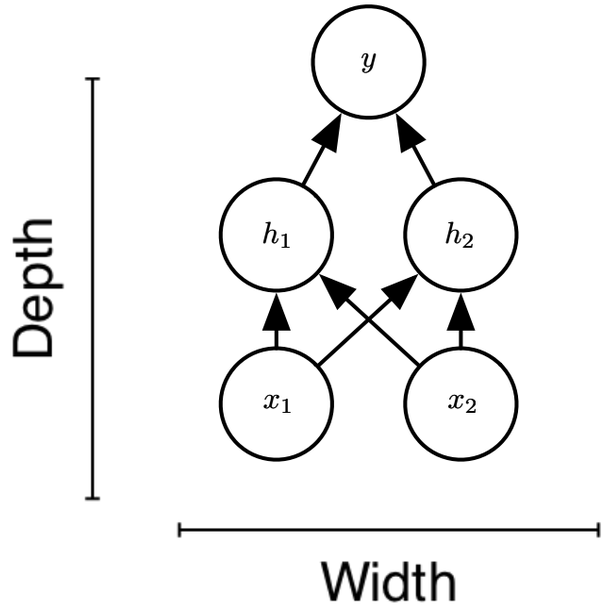
Architecture

- Architecture : the overall structure of the neural network
- NNs are generally organised in layers
  - Each layer is a function of the preceding one

$$\boldsymbol{h}^{(1)} = g^{(1)}(\boldsymbol{W}^{(1)T}\boldsymbol{x} + \boldsymbol{b}^{(1)})$$

$$\boldsymbol{h}^{(i)} = g^{(i)}(\boldsymbol{W}^{(i)T}\boldsymbol{h}^{(i-1)} + \boldsymbol{b}^{(i)})$$

- Main architectural considerations
  - Depth of the network
  - Width of each layer



- Deeper networks
  - Often able to use less parameters
  - Tend to generalize better
  - harder to train

The ideal network architecture for a task must be found

Universal approximation Theorem

A feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation can approximate any [continuous function] with any desirable nonzero amount of error, given enough hidden units

- So, regardless of what function we are trying to learn, we know that a large MLP will be able to represent this function
  - The theorem does not say how large this network will be
- We are not guaranteed that the training algorithm will be able to **learn** it
  - The optimization algorithm may not be able to find the value of the parameters that corresponds to the desired function
  - The training algorithm might choose the wrong function (overfitting)

Universal Approximator Theorem

- Feedforward network with a single layer is sufficient to represent any function
  - The layer may be infeasibly large
  - May fail to learn and generalize correctly
- Using deeper models can reduce the number of units required to represent the desired function!
  - Can reduce the amount of generalization error
  - Note: having deep architecture sometimes is not useful!
    - Validation phase is crucial!
- Universal approximation theorems have also been proved for a wider class of activation functions, which includes ReLU

Advantage of Depth

- Choosing a deep model encodes a very general belief that the function we want to learn should involve composition of several simpler functions
  - From representation learning point of view: learning problem consists of discovering a set of features that can in turn be described in terms of other simpler features
  - The function we want to learn is a computer program consisting of multiple steps, where each step makes use of the previous step's output
- Empirically, greater depth does seem to result in better generalization for a wide variety of tasks

Other Architectural Considerations

- Depth of the network and the width of each layer are not the only parameters that influence a model architecture
  - In practice, neural networks show considerably more diversity
- Many neural network architectures have been developed for specific tasks
  - Convolutional Neural Networks (CNN)
  - Recurrent Neural Networks (RNN)
- In general, the layers need not be connected in a chain
  - Skip connections
    - Make it easier for the gradient to flow from output layers to layers nearer the input
  - sparse connections
    - each unit in a layer is connected to only a small subset of units in the next layer