EECS 230 Deep Learning
Lecture 3: Neural Network

Some slides from Simon Prince, Dan Jurafsky, Roni Sengupta, and Olga Veksler

Shallow Neural Network

From last lecture: 1D Linear regression

- Model:

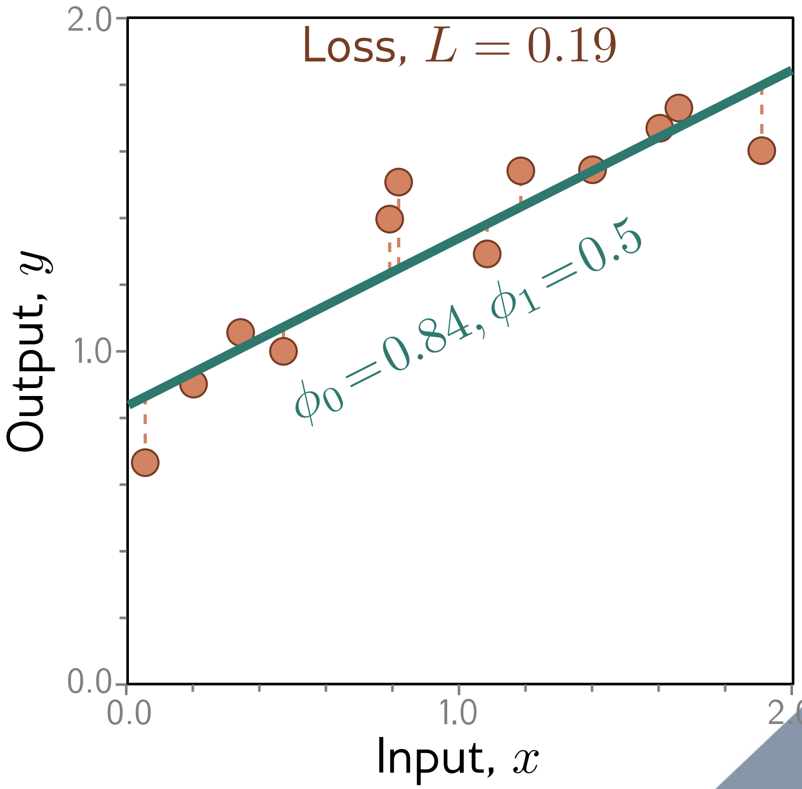
$$\begin{aligned} y &= \mathrm{f}[x, \boldsymbol{\phi}] \\ &= \phi_0 + \phi_1 x \end{aligned}$$

- Parameters

$$\boldsymbol{\phi} = \begin{bmatrix} \phi_0 \\ \phi_1 \end{bmatrix}$$

y-offset

slope



From last lecture: Linear Classification

- For example: fish classification - *salmon* or *sea bass*?
- extract two features, *fish length* and *fish brightness*

feature vector

$$\mathbf{x}^1 = \begin{bmatrix} 3.3 \\ 5.7 \end{bmatrix} \quad \mathbf{x}^2 = \begin{bmatrix} 6.3 \\ 8.7 \end{bmatrix} \quad \mathbf{x}^3 = \begin{bmatrix} 2.3 \\ 1.7 \end{bmatrix} \quad \mathbf{x}^4 = \begin{bmatrix} 6.4 \\ 7.0 \end{bmatrix}$$

individual features $\mathbf{x}_k^i$
e.g. fish length and brightness


salmon
$\mathbf{y}^1=0$


sea bass
$\mathbf{y}^2=1$

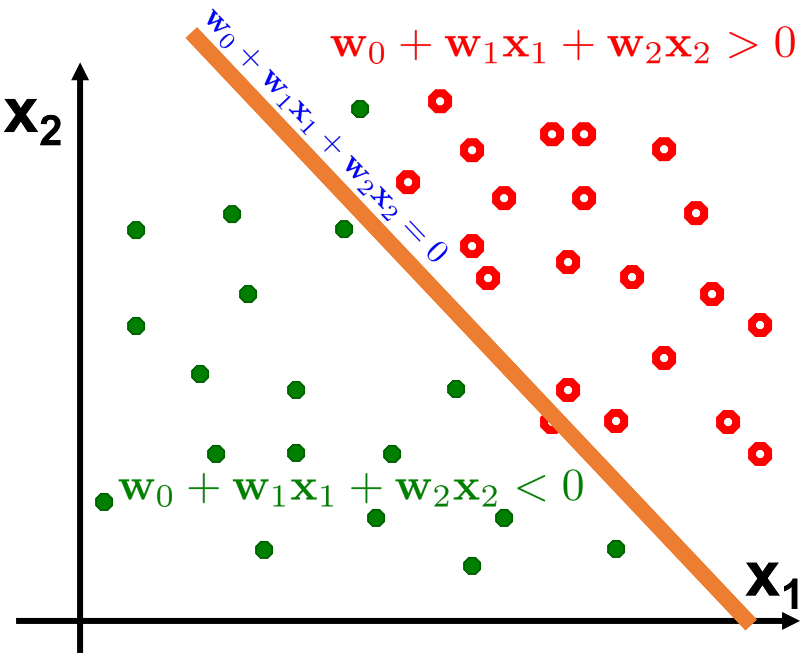

salmon
$\mathbf{y}^3 = 0$


sea bass
$\mathbf{y}^4=1$

- $\mathbf{y}^i$ is the output (label or target) for example $\mathbf{x}^i$

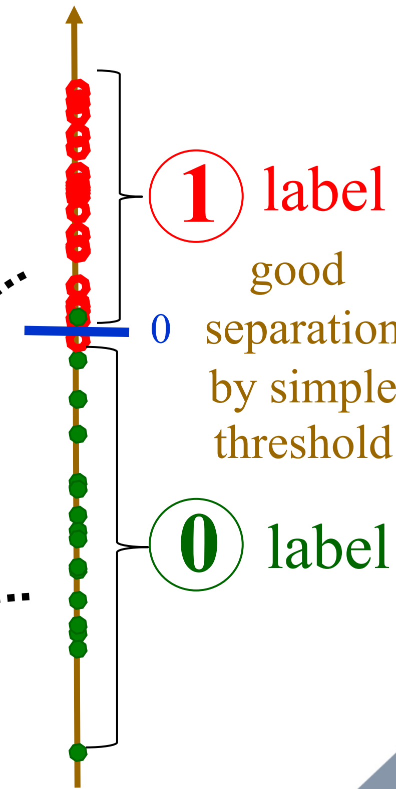
Linear Classification (perceptron)

- For two class problem and 2-dimensional data (feature vectors)



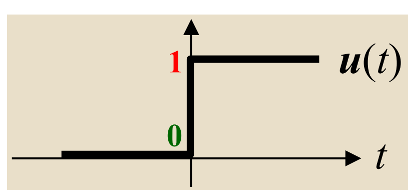
"good"
linear transformation
from 2D space to 1D

$$\mathbf{w}_0^* + \mathbf{w}_1^*\mathbf{x}_1 + \mathbf{w}_2^*\mathbf{x}_2$$



thresholding can be formally represented by this prediction function

$$\mathbf{f}(\mathbf{w},\mathbf{x}) = u\,(\mathbf{w}_0 + \mathbf{w}_1\mathbf{x}_1 + \mathbf{w}_2\mathbf{x}_2) \qquad \mathbf{f}(\mathbf{w},\mathbf{x}) \in \{0, 1\}$$



unit step function
(a.k.a. *Heaviside* function)

$$u(t) := \begin{cases} 1 & \text{if } t > 0 \\ 0 & O.W. \end{cases}$$

Neural Unit

- Take weighted sum of inputs, plus a bias

$$z = b + \sum_i w_i x_i$$

$$z = w \cdot x + b$$

- Instead of just using z, we'll apply a nonlinear activation function f:

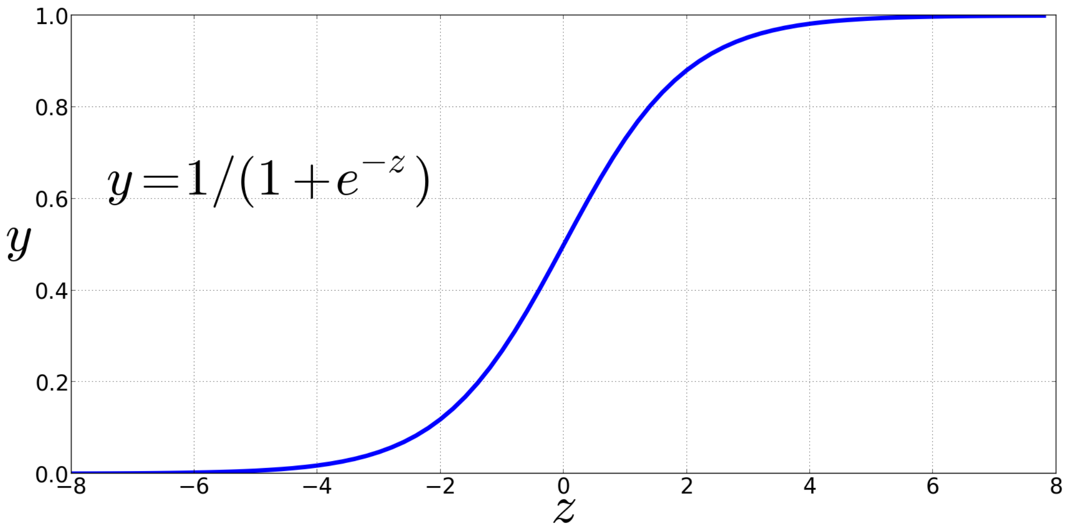
$$y = a = f(z)$$

UCMERCED

Non-linear Activation Function

Sigmoid

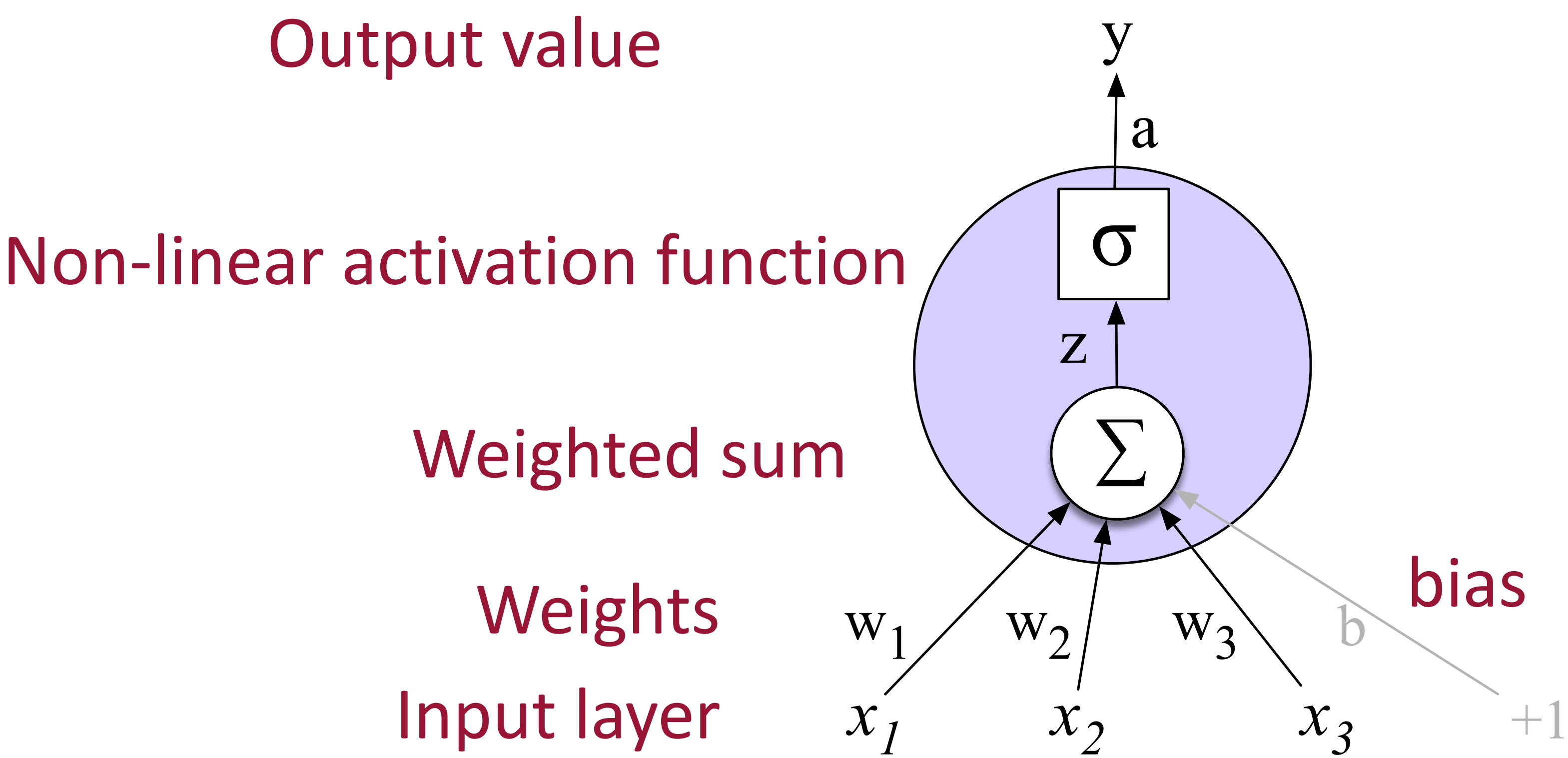
$$y = \sigma(z) = \frac{1}{1 + e^{-z}}$$



Final function the unit is computing

$$y = \sigma(w \cdot x + b) = \frac{1}{1 + \exp(-(w \cdot x + b))}$$

Neural Unit



Output value
Non-linear activation function
Weighted sum
Weights
Input layer
y
a
σ
z
Σ
w1
w2
w3
b
bias
x1
x2
x3
+1

An example

Suppose a unit has:

$w = [0.2, 0.3, 0.9]$

$b = 0.5$

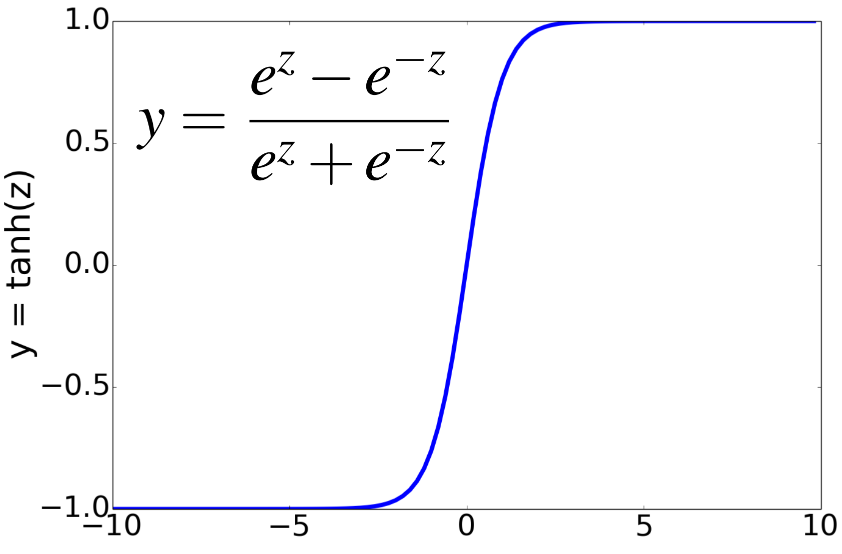
What happens with input x:

$x = [0.5, 0.6, 0.1]$

$$y = \sigma(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}} =$$

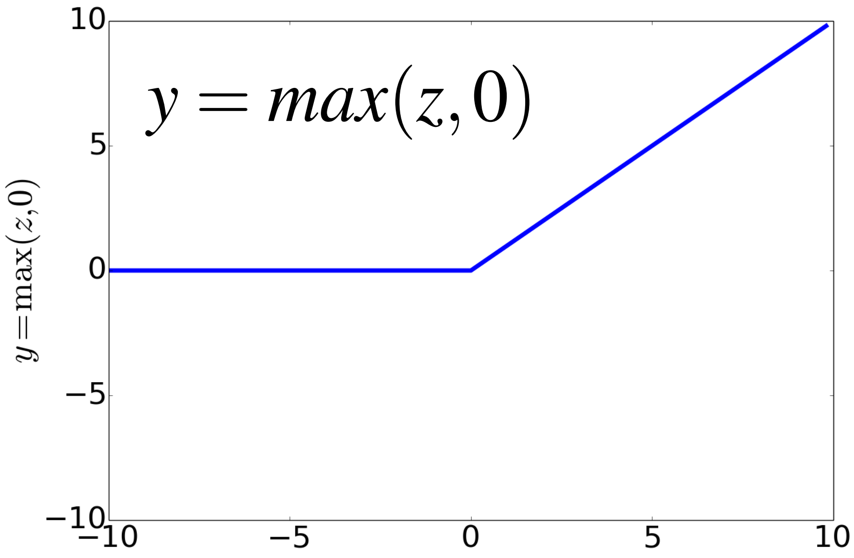
$$\frac{1}{1 + e^{-(.5*.2+.6*.3+.1*.9+.5)}} = \frac{1}{1 + e^{-0.87}} = .70$$

Other non-linear activation function



tanh

Most Common:



ReLU

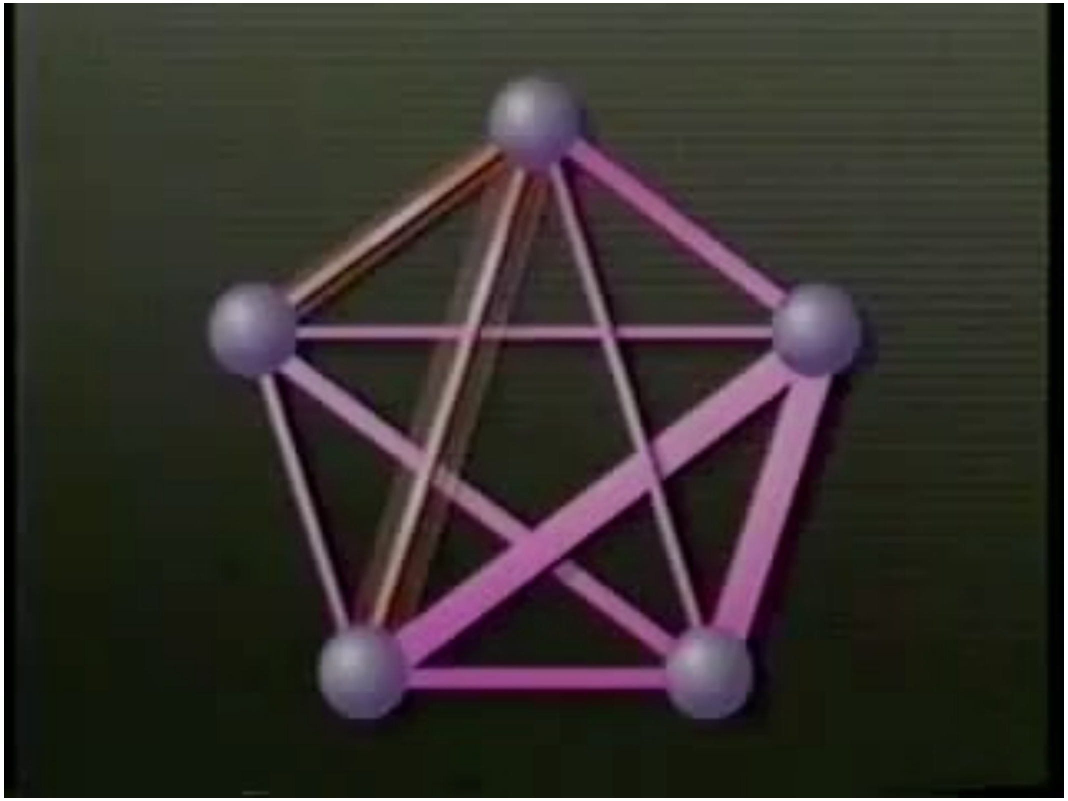
Rectified Linear Unit

Perceptron

- A very simple neural unit
- Binary output (0 or 1)
- No non-linear activation function

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$

Perceptron from the 50's and 60's



https://www.youtube.com/watch?v=cNxadbrN_aI&t=71s

The XOR problem

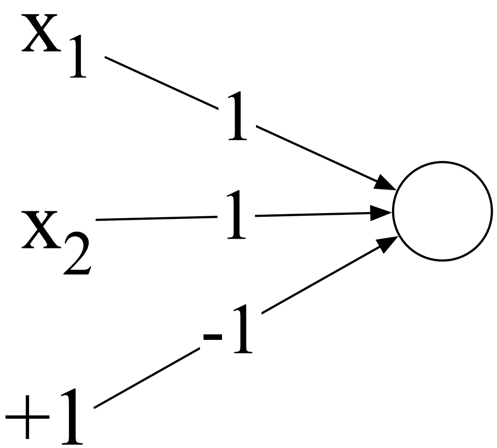
Minsky and Papert (1969)

- Can perceptron compute simple functions of input?

AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

Easy to build AND or OR with perceptron

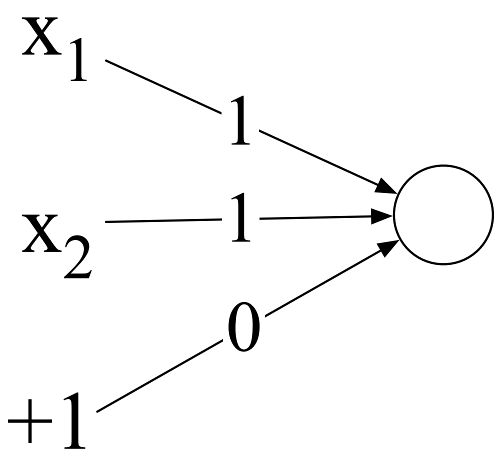
$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$



AND

AND

x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1



OR

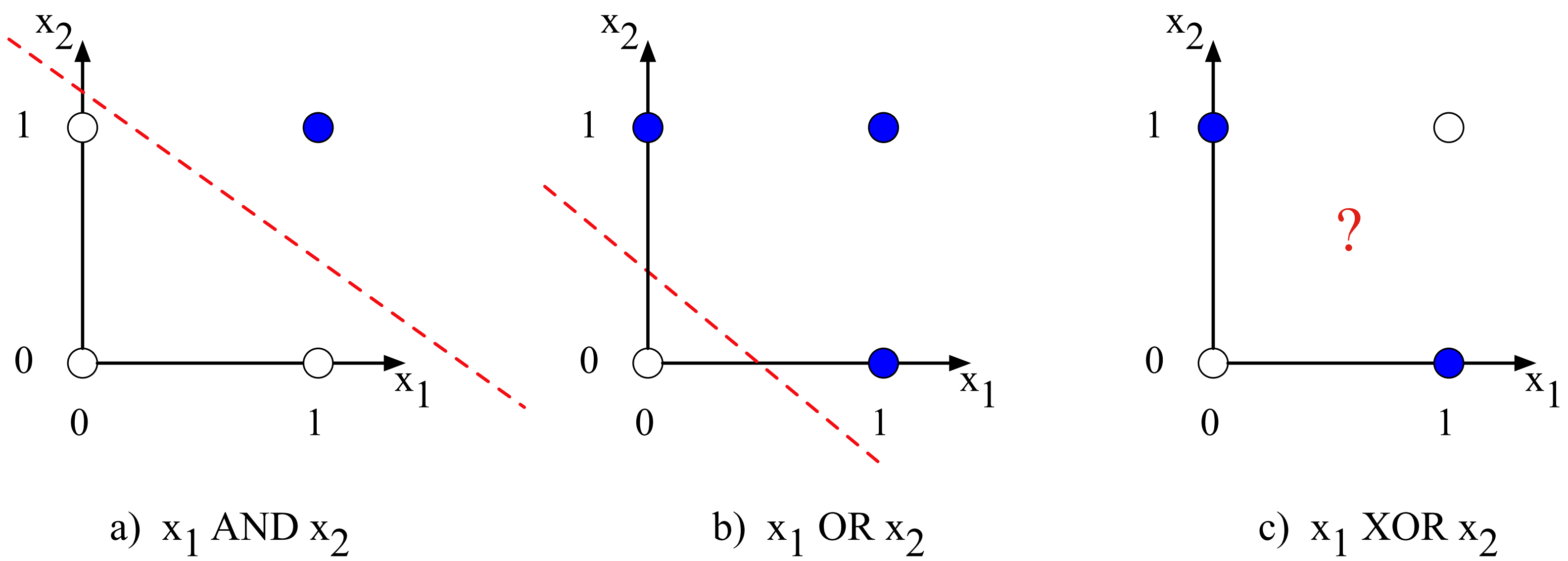
OR

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

Is it possible to capture XOR with perceptrons?

- Pause the lecture and try for yourself!
- No!
- Why? Perceptrons are linear classifiers

Decision boundaries



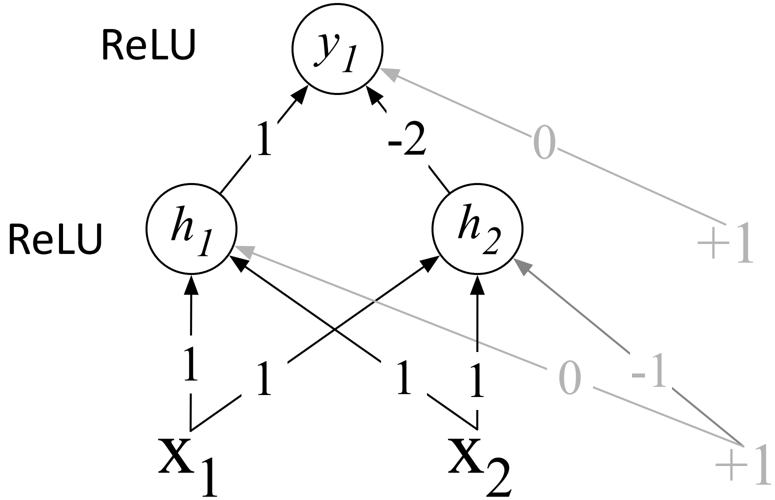
XOR is not a **linearly separable** function!

Solution to the XOR problem

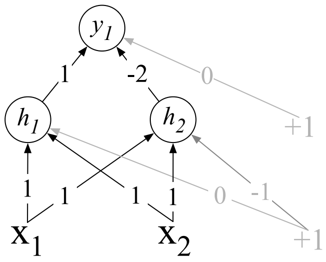
- XOR **can't** be calculated by a single perceptron
- XOR **can** be calculated by a layered network of units.

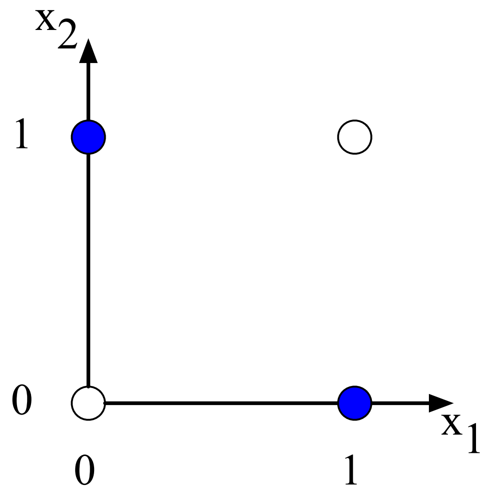
XOR

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

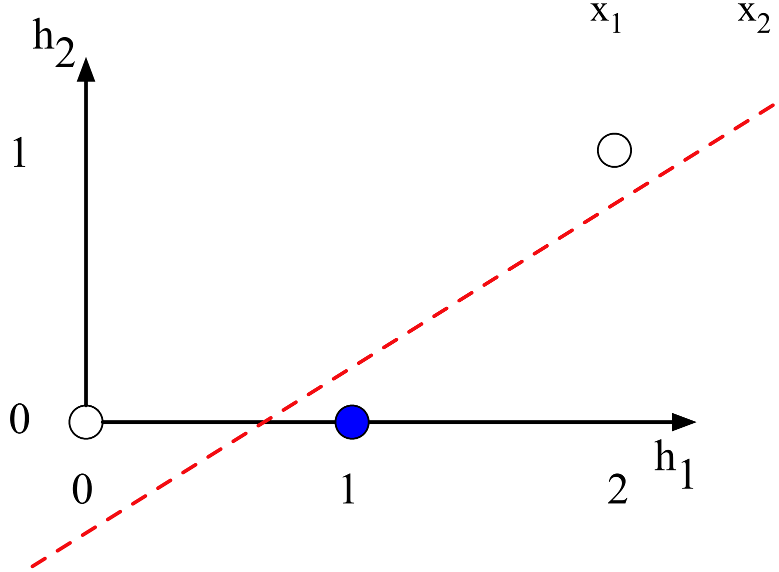


The hidden representation h





a) The original *x* space



b) The new (linearly separable) *h* space

(With learning: hidden layers will learn to form useful representations)

Shallow Neural Network with Hidden Units

$$y = \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11}x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21}x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31}x].$$

Break down into two parts:

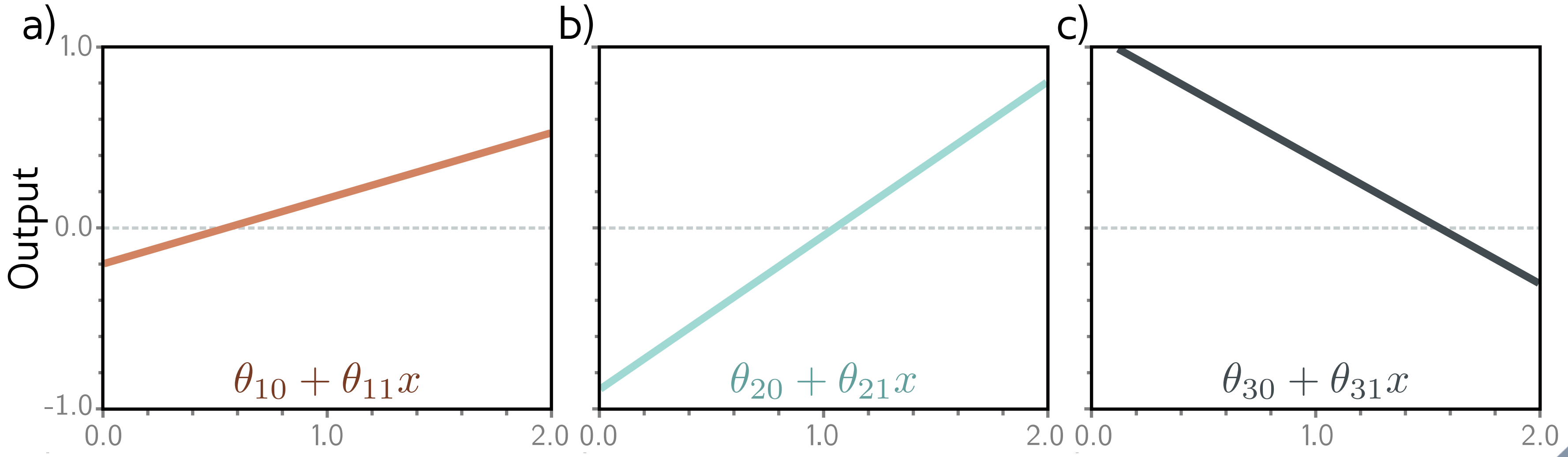
$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

where:

$$\text{Hidden units} \left\{ \begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x] \end{aligned} \right.$$

Visualize shallow neural network

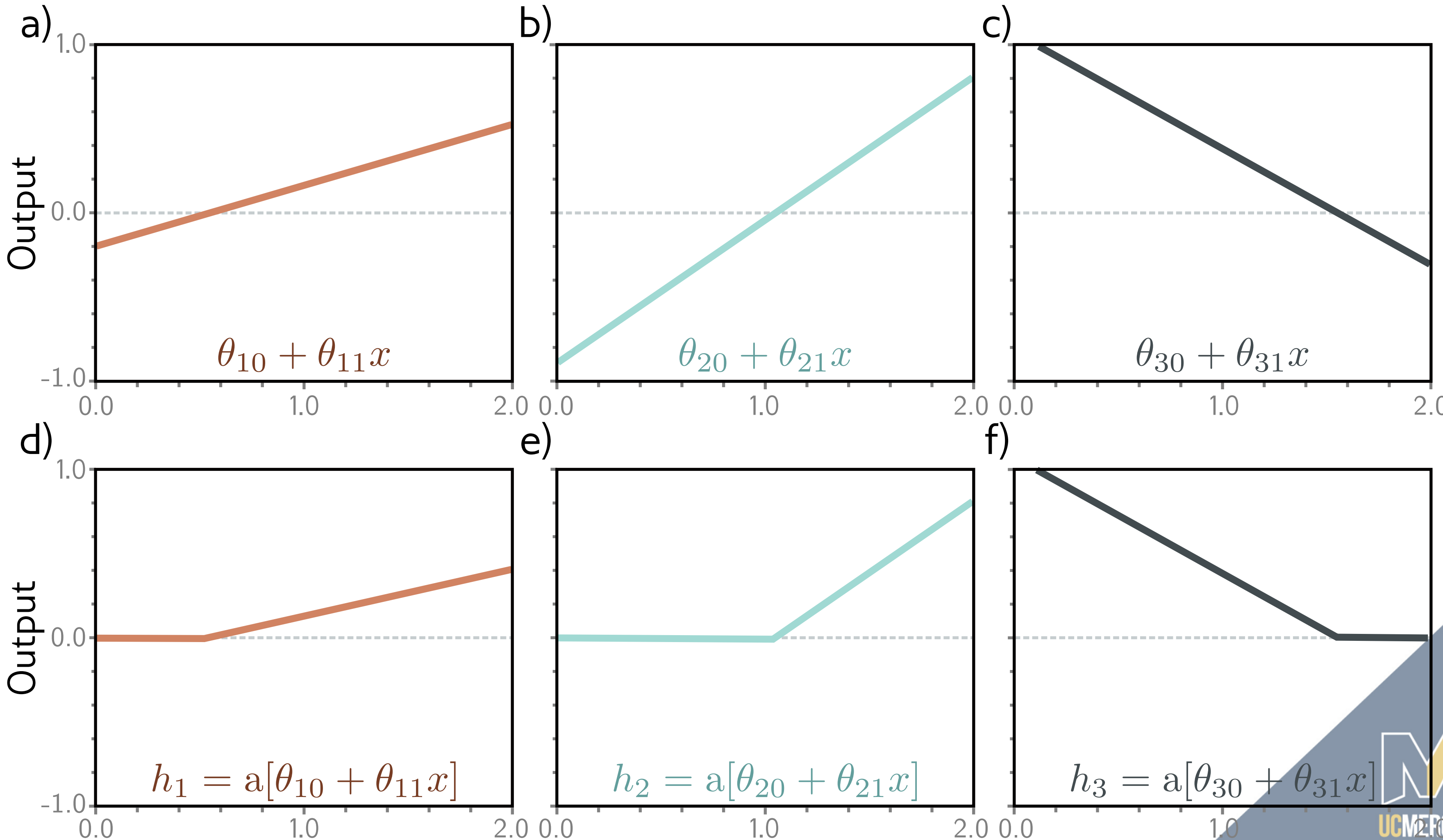
1. Compute the linear function



Visualize shallow neural network

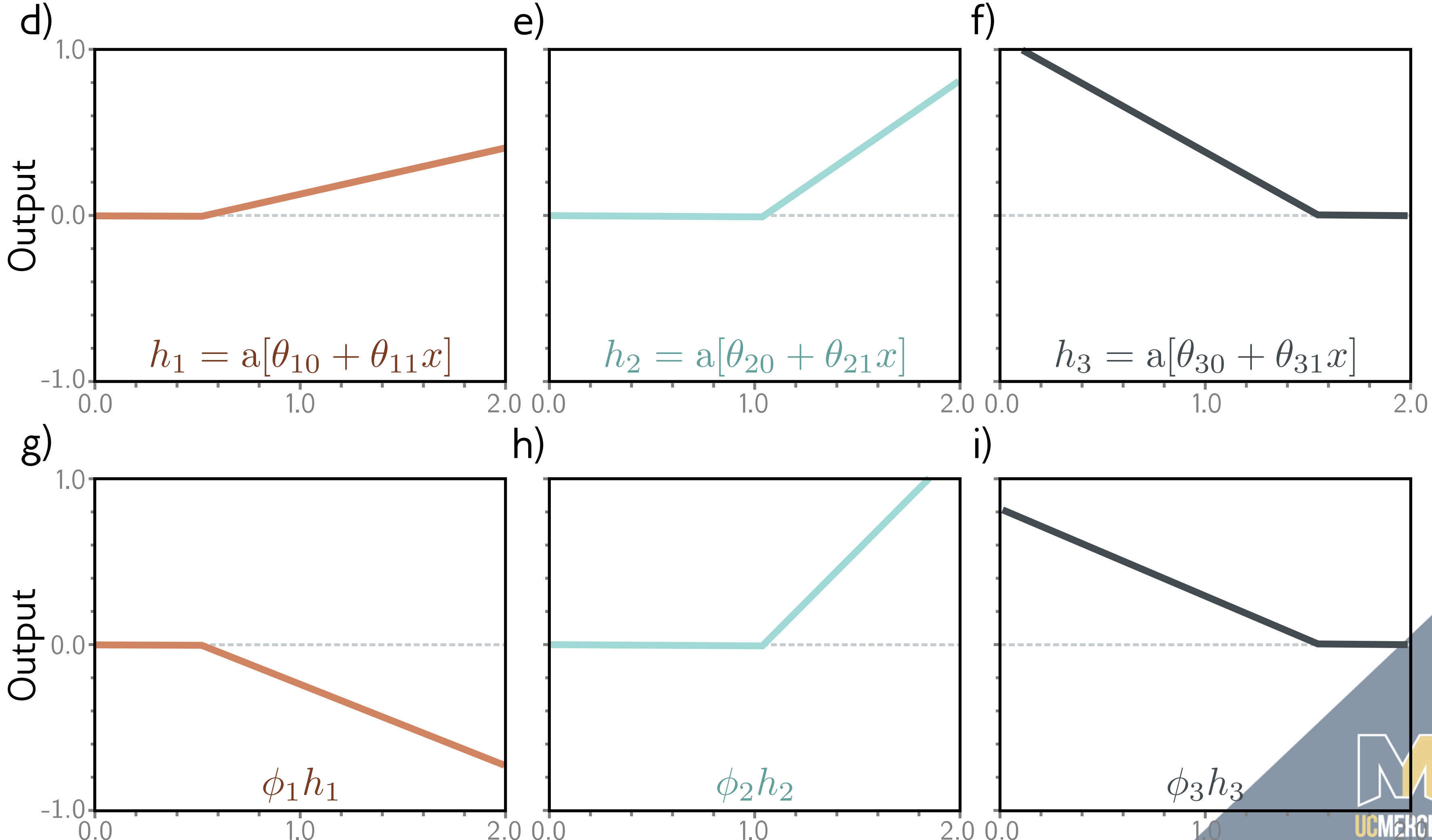
$$h_1 = \mathrm{a}[\theta_{10} + \theta_{11}x]$$
$$h_2 = \mathrm{a}[\theta_{20} + \theta_{21}x]$$
$$h_3 = \mathrm{a}[\theta_{30} + \theta_{31}x],$$

2. Pass through Relu (create hidden units)



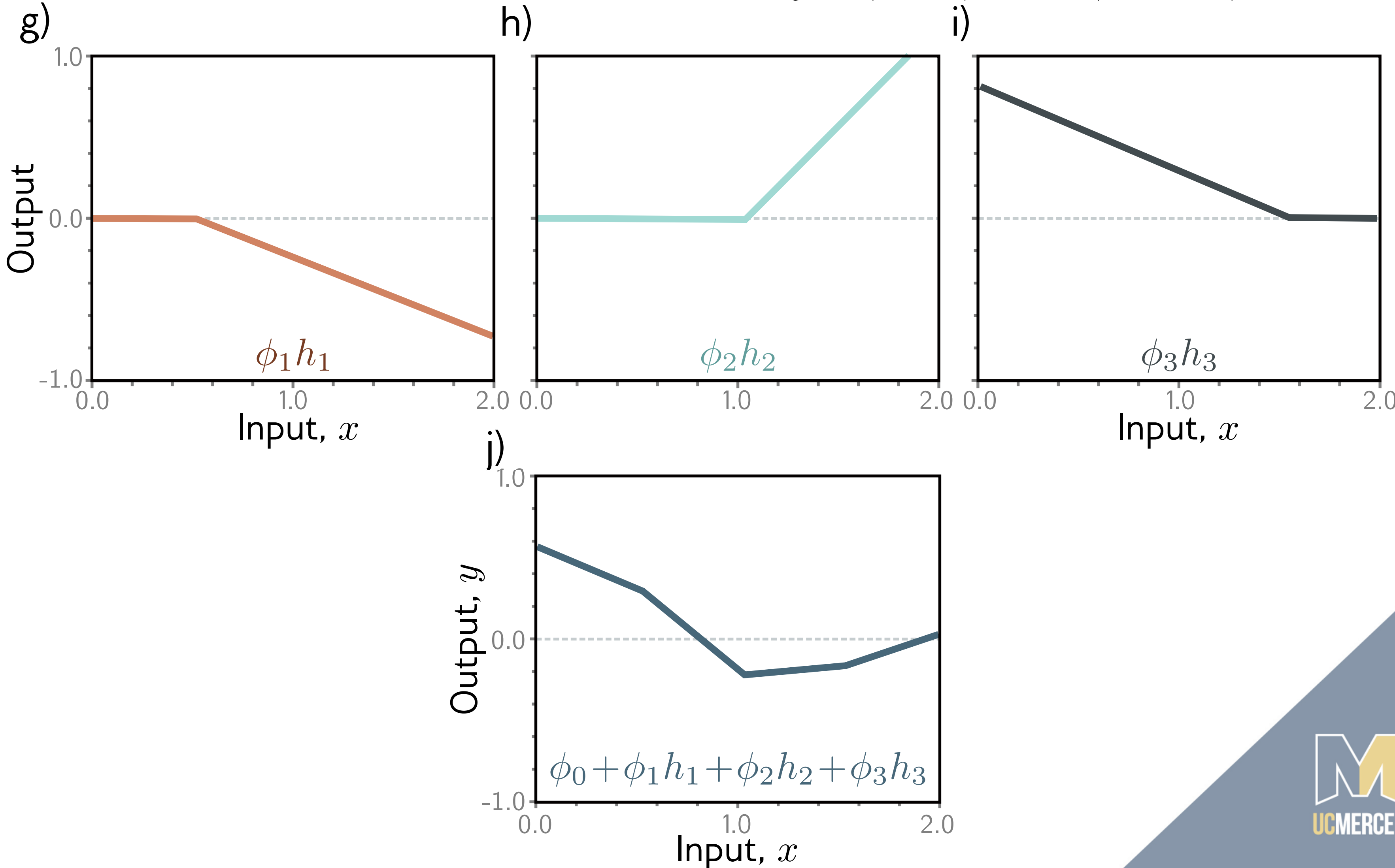
Visualize shallow neural network

3. Weight the hidden units



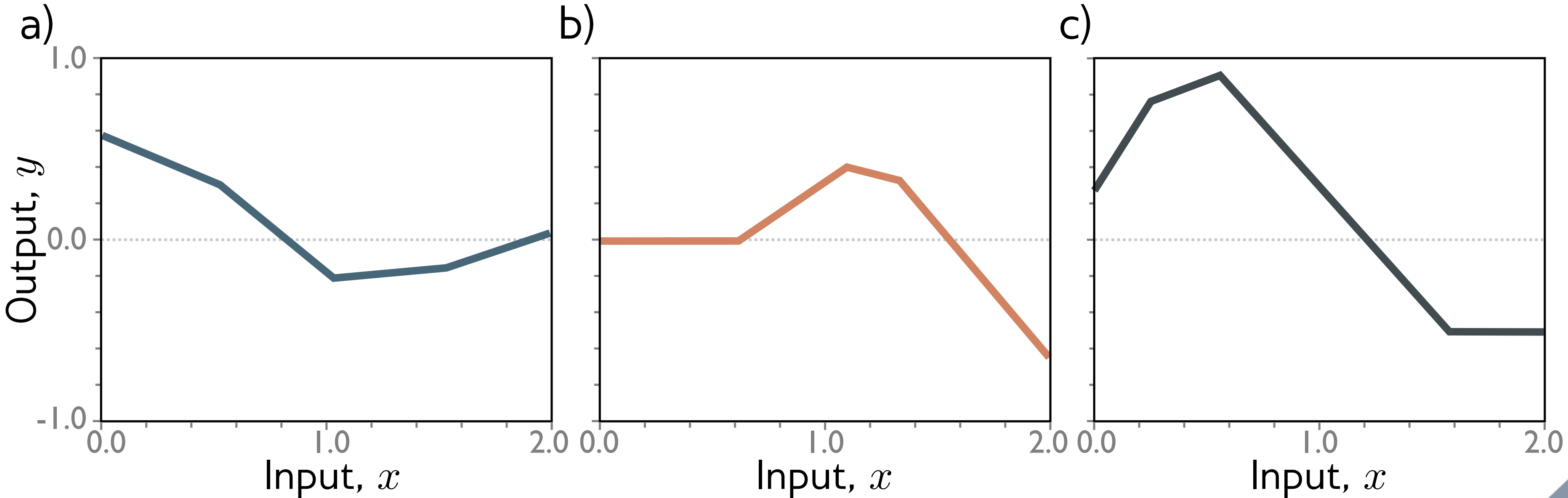
Visualize shallow neural network

4. Sum the weighted hidden units $y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$



Visualize shallow neural network

$$y = \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11}x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21}x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31}x].$$



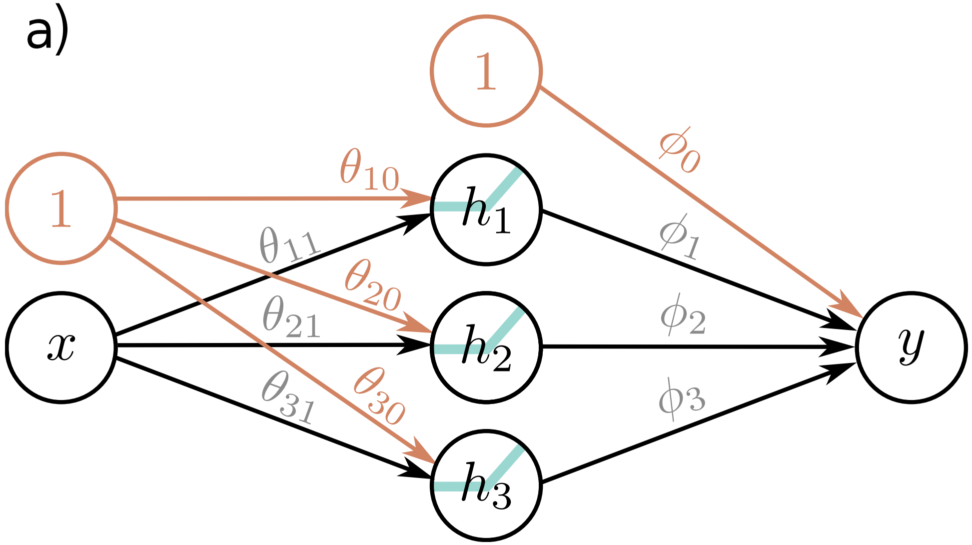
Example shallow network = piecewise linear functions
1 "joint" per ReLU function

Depicting shallow neural networks

$$h_1 = \mathrm{a}[\theta_{10} + \theta_{11}x]$$
$$h_2 = \mathrm{a}[\theta_{20} + \theta_{21}x]$$
$$h_3 = \mathrm{a}[\theta_{30} + \theta_{31}x]$$

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

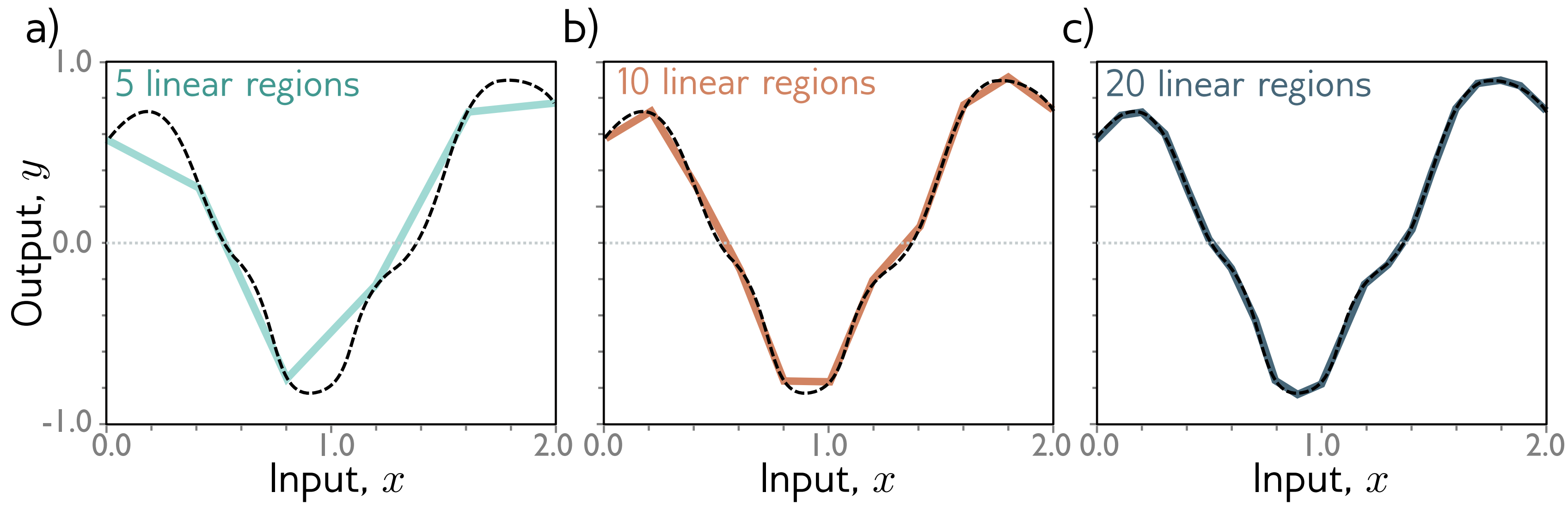
a)



Each parameter multiplies its source and adds to its target

With enough hidden units

- ❑… we can describe any 1D function to arbitrary accuracy



Universal approximation theorem

"a formal proof that, with enough hidden units, a shallow neural network can describe any continuous function on a compact subset of $\mathbb{R}^D$ to arbitrary precision"

Universal approximation theorem

Approximation by Superpositions of a Sigmoidal Function*

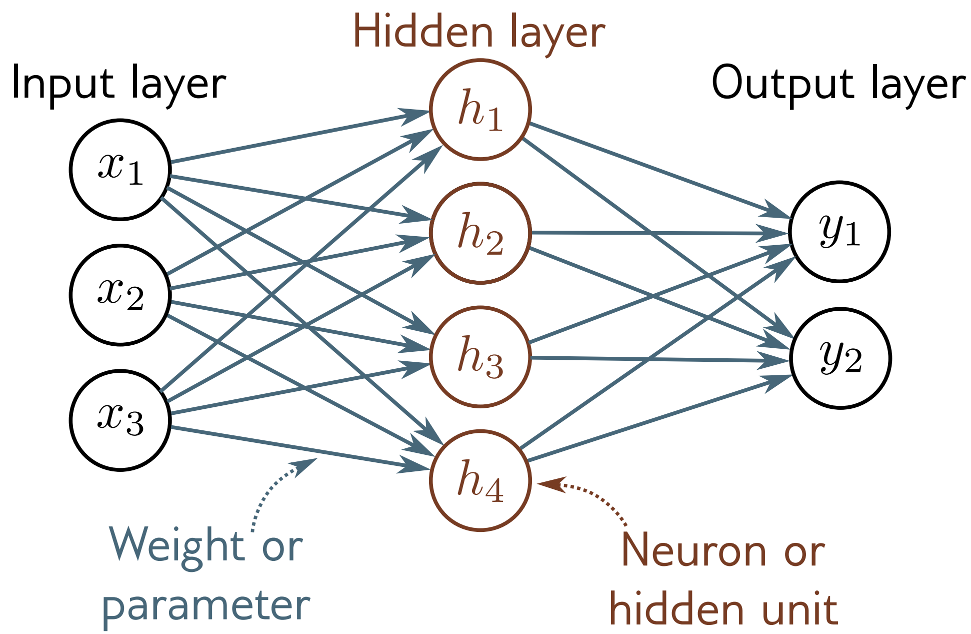
G. Cybenko†

Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Key words. Neural networks, Approximation, Completeness.

Cybenko, George. "Approximation by superpositions of a sigmoidal function." *Mathematics of control, signals and systems* 2.4 (1989): 303-314.

Terminology



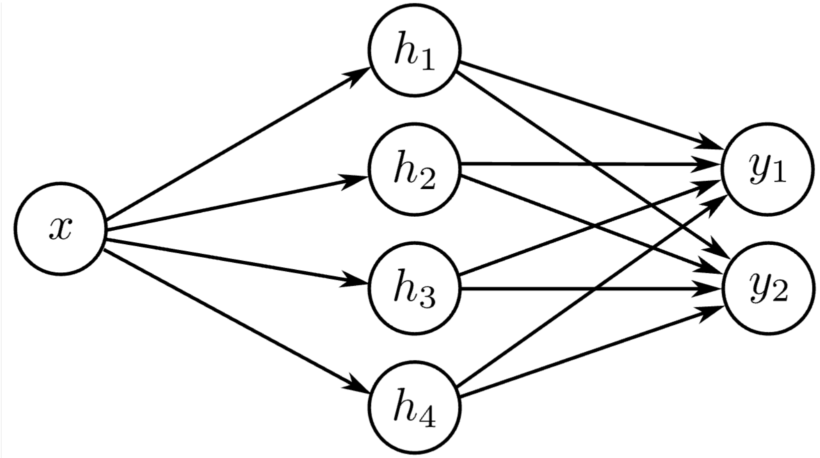
- Y-offsets = biases
- Slopes = weights
- Everything in one layer connected to everything in the next = fully connected network
- No loops = feedforward network
- Values after ReLU (activation functions) = activations
- Values before ReLU = pre-activations
- One hidden layer = shallow neural network
- More than one hidden layer = deep neural network
- Number of hidden units $\approx$ capacity

Deep Neural Network

Shallow network

- 1 input, 4 hidden units, 2 outputs

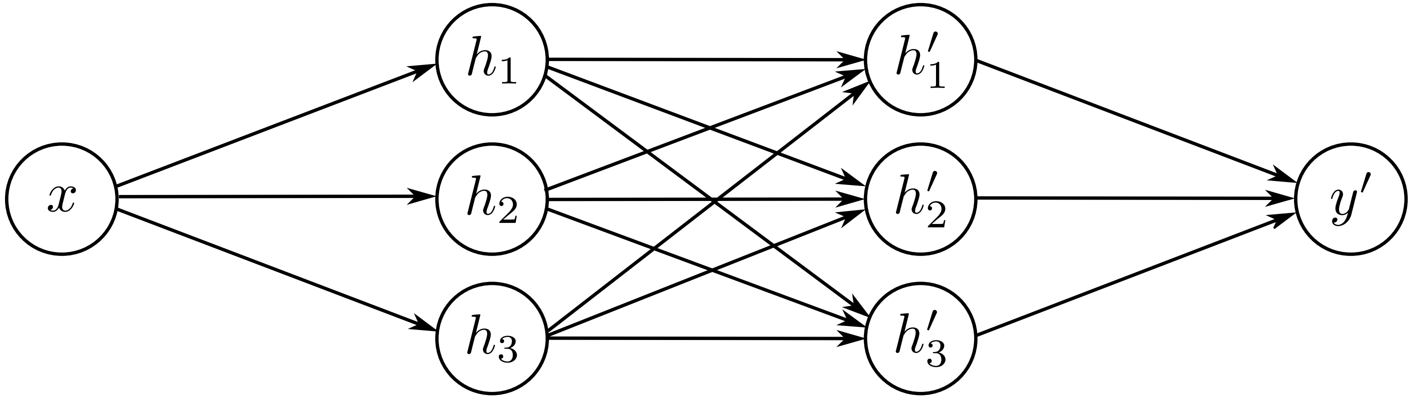
$$
\begin{aligned}
h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\
h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\
h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x] \\
h_4 &= \mathrm{a}[\theta_{40} + \theta_{41}x]
\end{aligned}
$$



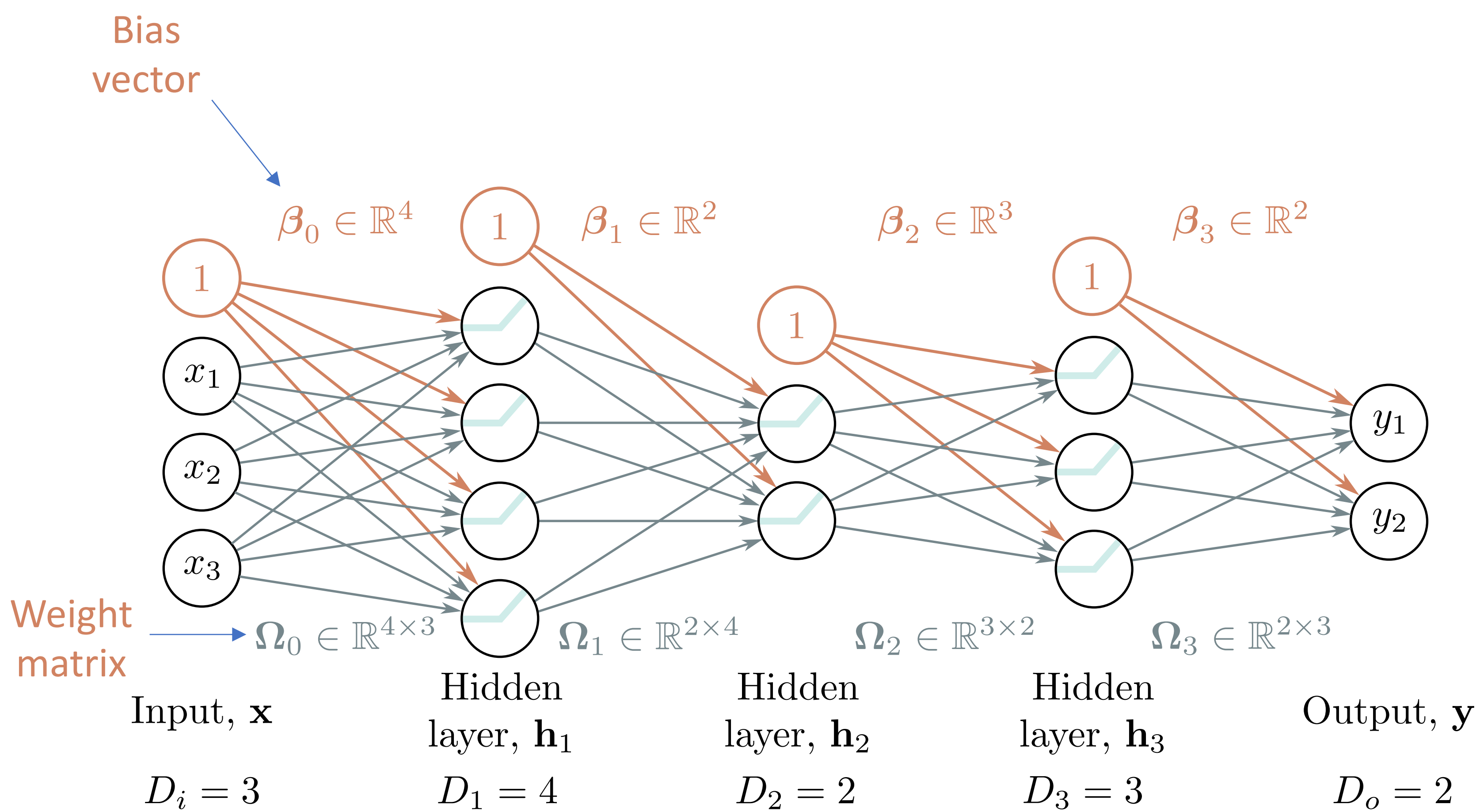
Network as composing function

$$h_1 = \mathrm{a}[\theta_{10} + \theta_{11}x]$$
$$h_2 = \mathrm{a}[\theta_{20} + \theta_{21}x]$$
$$h_3 = \mathrm{a}[\theta_{30} + \theta_{31}x]$$
$$h_4 = \mathrm{a}[\theta_{40} + \theta_{41}x]$$

$$h'_1 = \mathrm{a}[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3]$$
$$h'_2 = \mathrm{a}[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3]$$
$$h'_3 = \mathrm{a}[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3]$$



Example of Multi Layer Perceptron (MLP)



Shallow vs deep networks

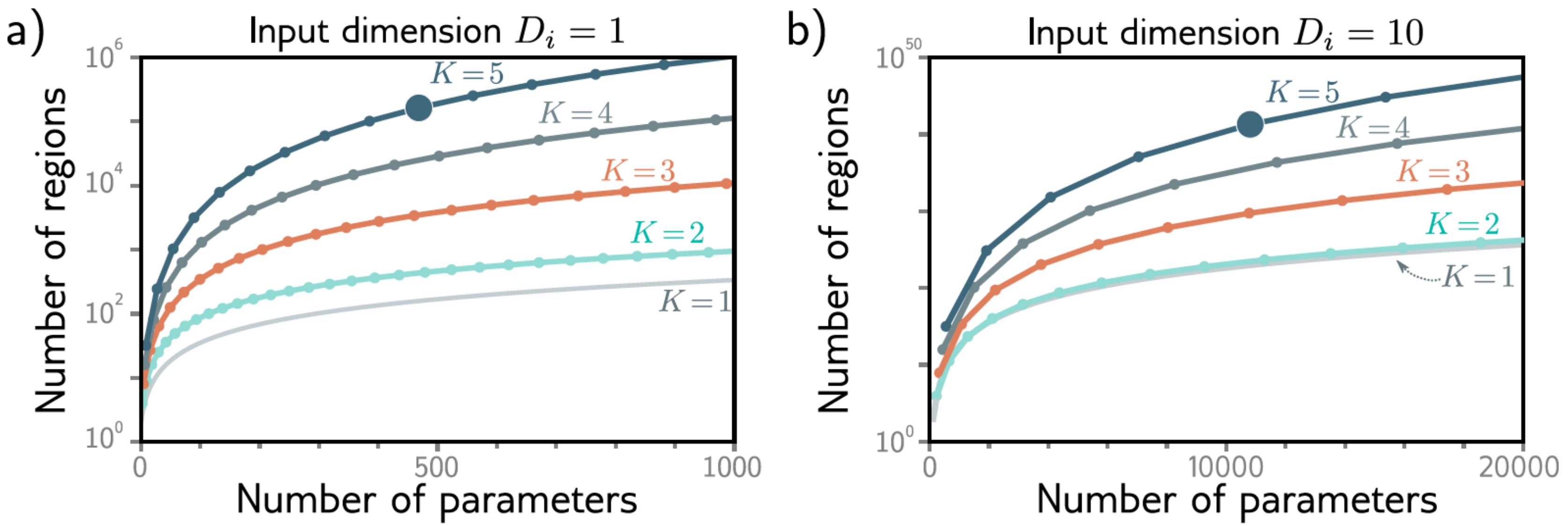
- The best results are created by deep networks with many layers.
  - 50-1000 layers for most applications
  - Best results in
    - Computer vision
    - Natural language processing
    - Graph neural networks
    - Generative models
    - Reinforcement learning

All use deep networks. But why?

- Ability to approximate different functions?
- Both obey the universal approximation theorem.
- Argument: One layer is enough, and for deep networks could arrange for the other layers to compute the identity function.

Shallow vs deep networks

- Number of linear regions per parameter
  - Deep networks create many more regions per parameters



5 layers
10 hidden units per layer
471 parameters
161,501 linear regions

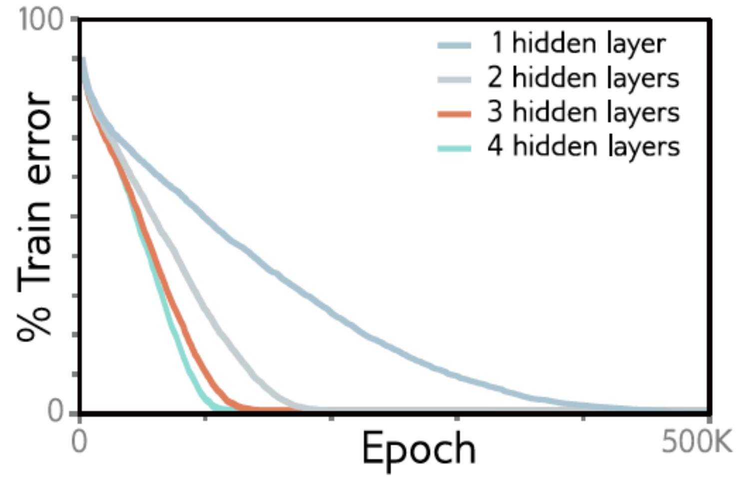
5 layers
50 hidden units per layer
10,801 parameters
$>10^{40}$ linear regions

UCMERCED

Shallow vs deep networks

- Fitting and generalization

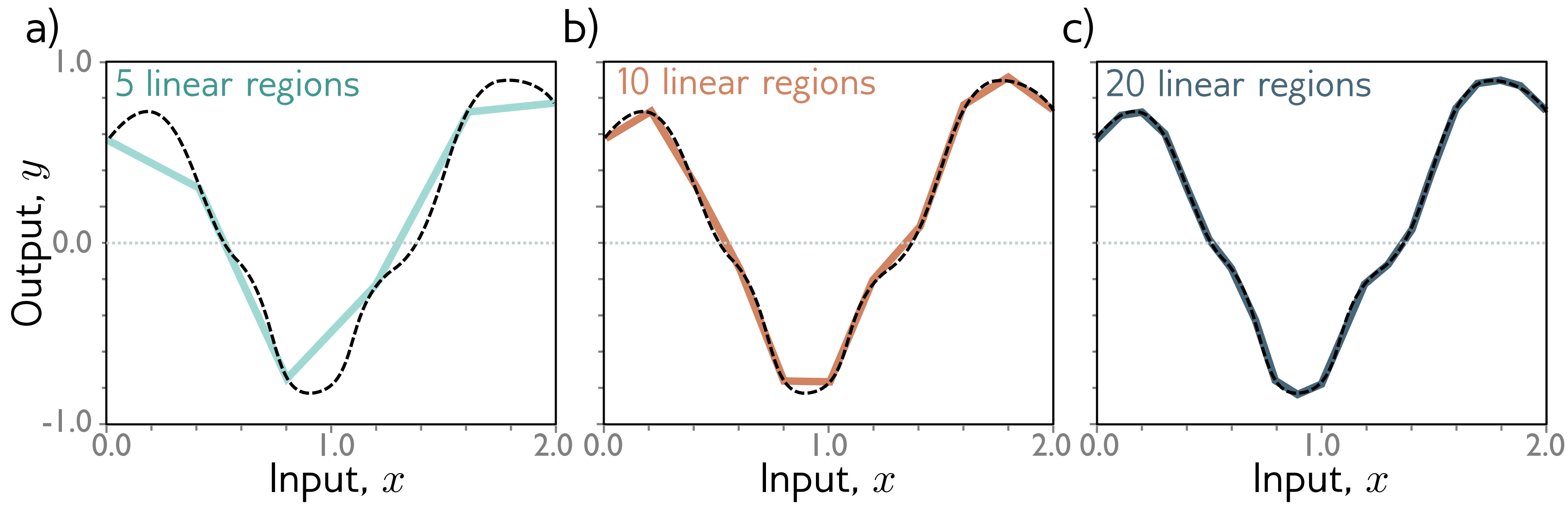
Figure 20.2 MNIST-1D training. Four fully connected networks were fit to 4000 MNIST-1D examples with random labels using full batch gradient descent, He initialization, no momentum or regularization, and learning rate 0.0025. Models with 1,2,3,4 layers had 298, 100, 75, and 63 hidden units per layer and 15208, 15210, 15235, and 15139 parameters, respectively. All models train successfully, but deeper models require fewer epochs.



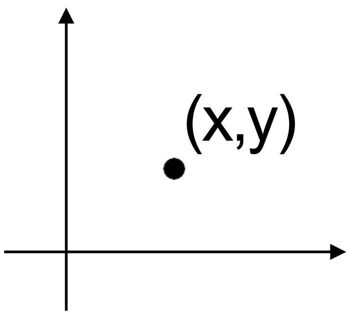
Implicit Neural Field
(An example of multi layer perceptron)

Neural network as function estimation

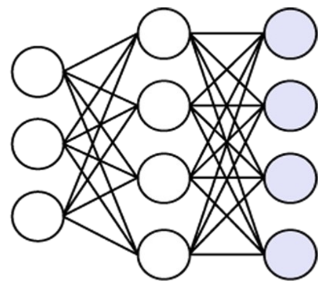
- ... we can describe any 1D function to arbitrary accuracy



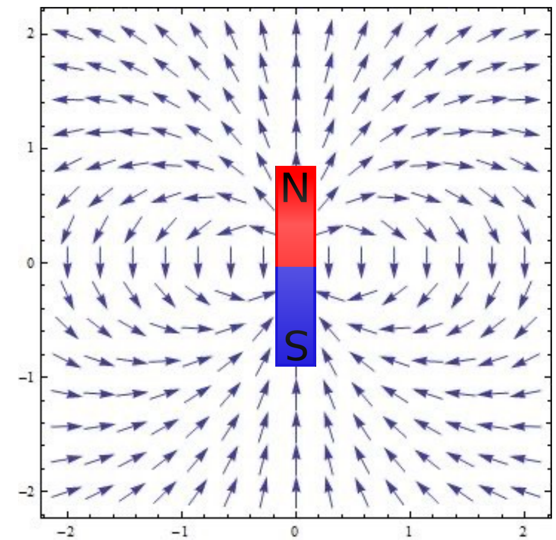
What are neural fields?



$\Phi : \mathbb{R}^2 \rightarrow \mathbb{R}^2$

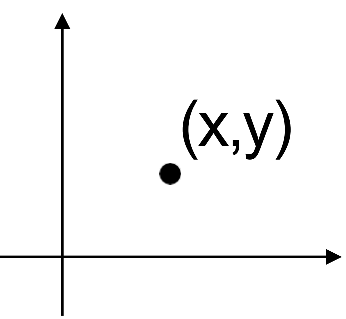


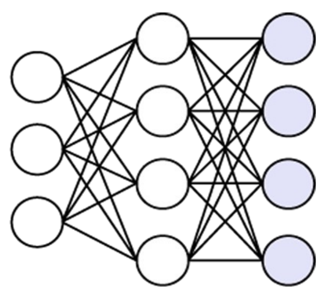
Neural Network (Φ)



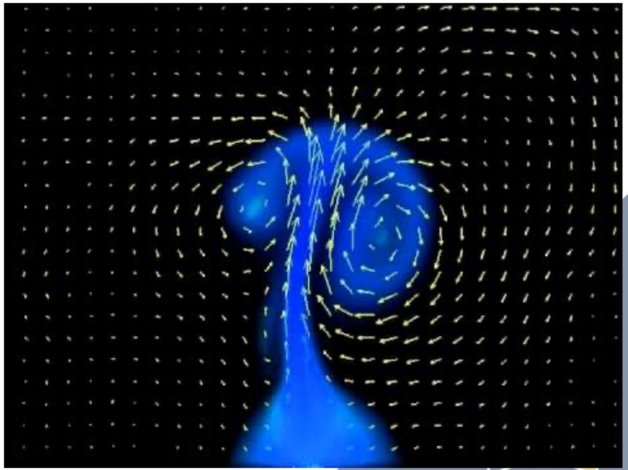
Magnetic Field

$\Phi : \mathbb{R}^2 \rightarrow \mathbb{R}^2$





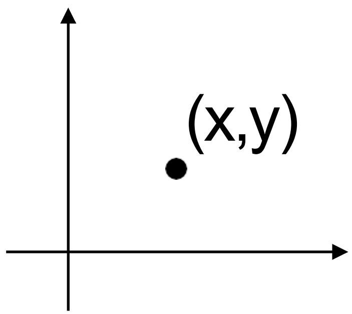
Neural Network (Φ)



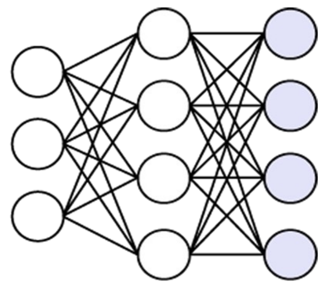
Eulerian Flow Field of a Fluid

[Koldora CC]

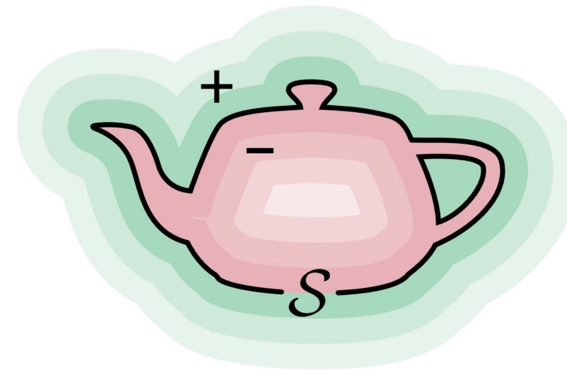
What are neural fields?



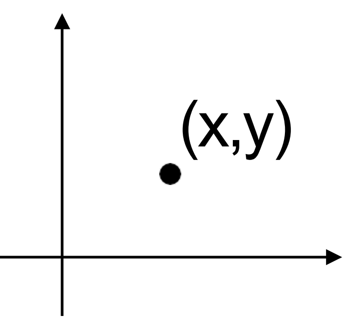
$\Phi : \mathbb{R}^2 \rightarrow \mathbb{R}^2$



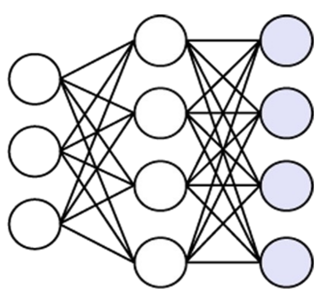
Neural Network (ϕ)



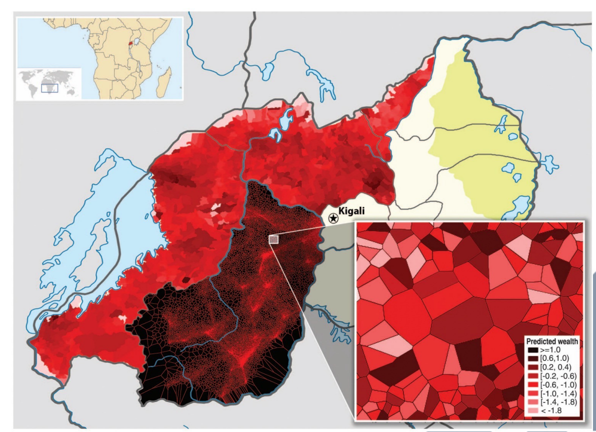
Signed Distance Function (SDF)



$\Phi : \mathbb{R}^2 \rightarrow \mathbb{R}^2$



Neural Network (ϕ)

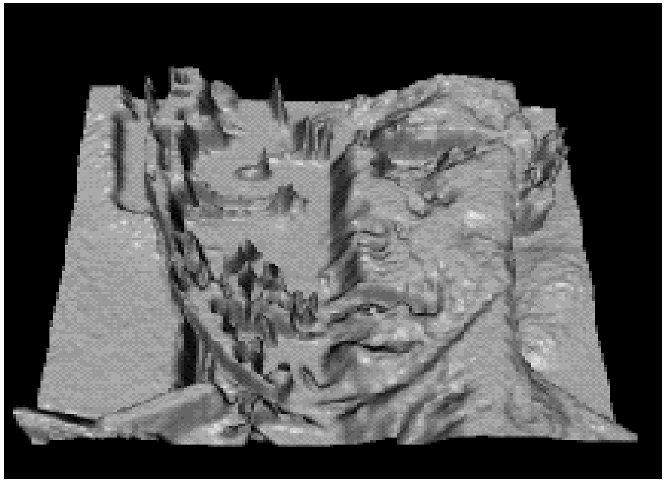


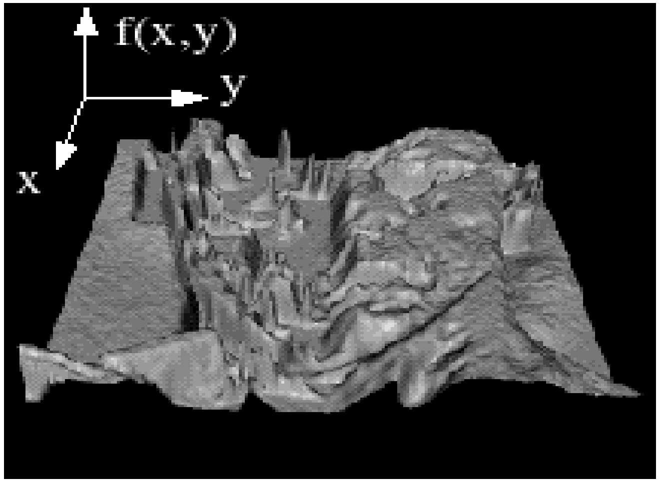
Geospatial Data

[Blumenstock et al. 2015]

Image as function

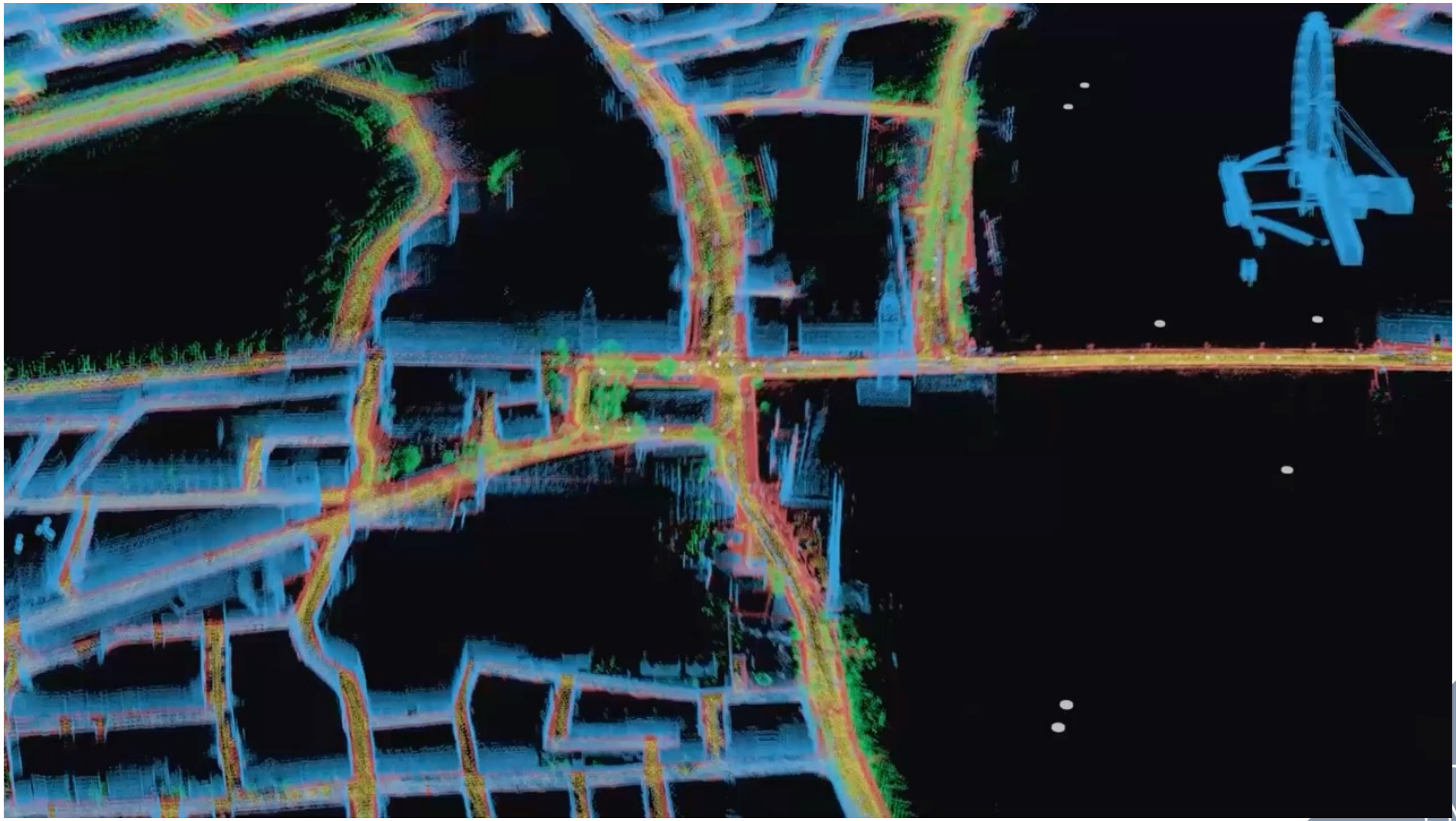
$$f(x, y) : \mathcal{R}^2 \rightarrow \mathcal{R}$$





Neural fields

- NeRF (Neural Radiance Field) has revolutionalized Computer Vision & Graphics in past 2 years!

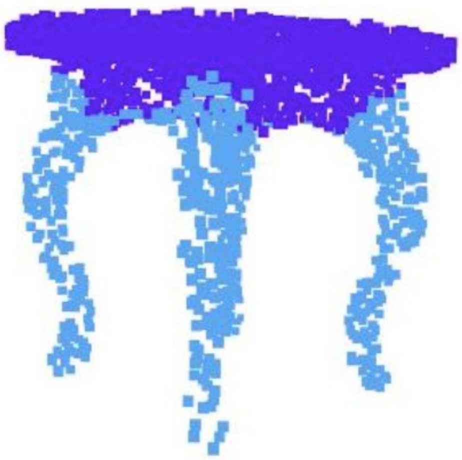


Google maps immersive view

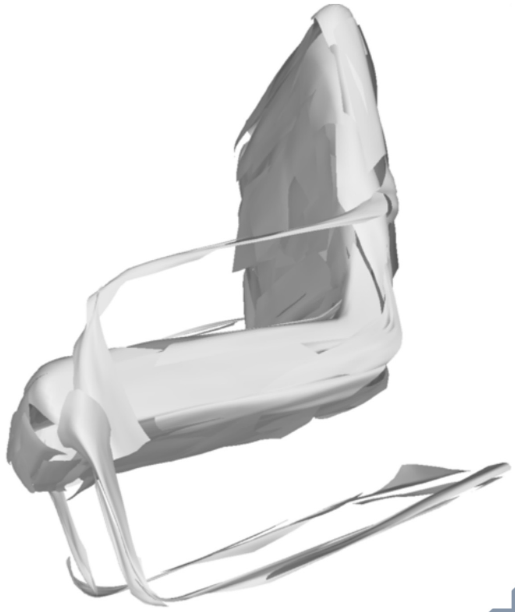
Representation for 3D deep learning

Voxel

Points

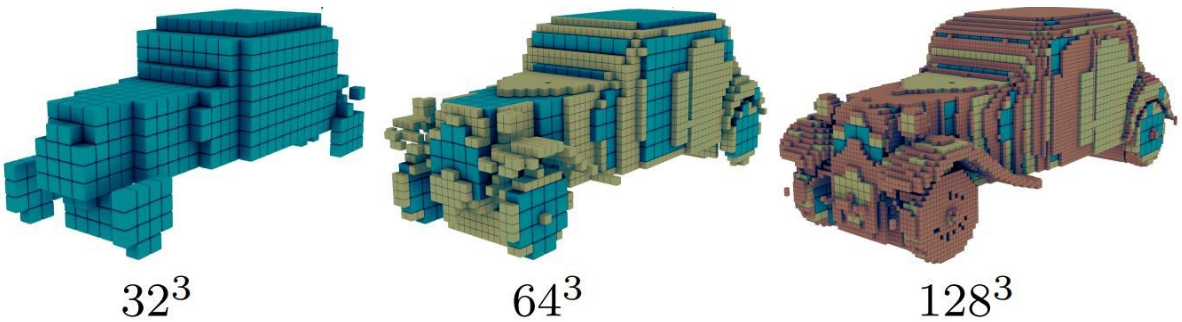


Meshes



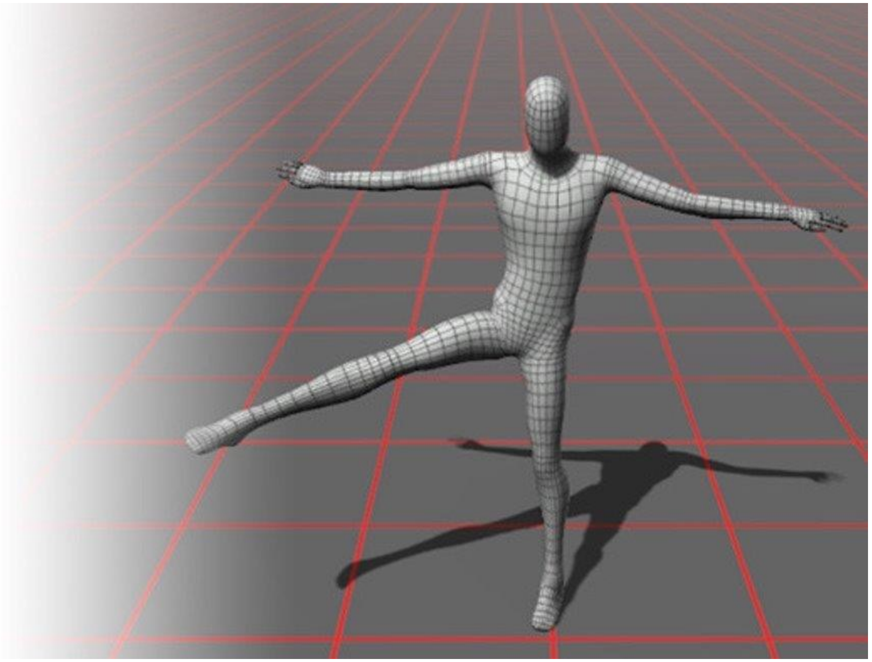
Voxel representation

- Memory expensive, computationally expensive (N^3)



Mesh representation

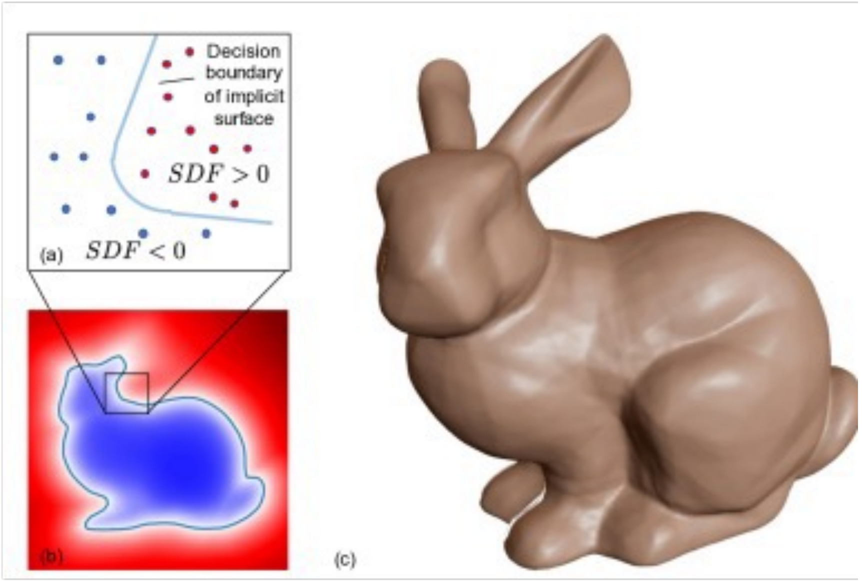
- ❑Fixed topology
- ❑Discrete vertices and connections



Point cloud representation

- ❑Does not define a surface
- ❑Not suitable for visualization, texturing, etc.

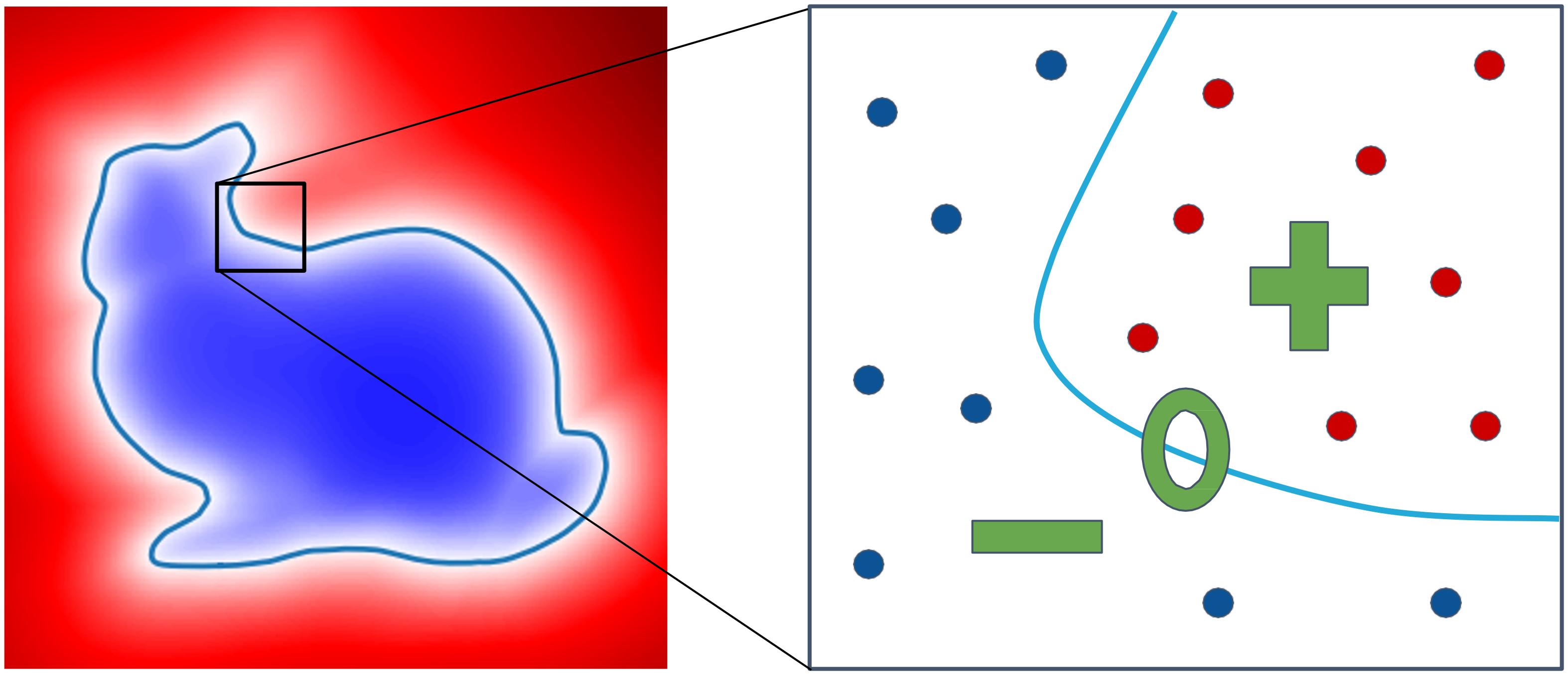
Signed Distance Function (SDF)



- ❑SDF(X) = 0, when X is on the surface.
- ❑SDF(X) > 0, when X is outside the surface
- ❑SDF(X) < 0, when X is inside the surface
- ❑Deep SDF: Use a neural network (co-ordinate based MLP) to represent the SDF function.

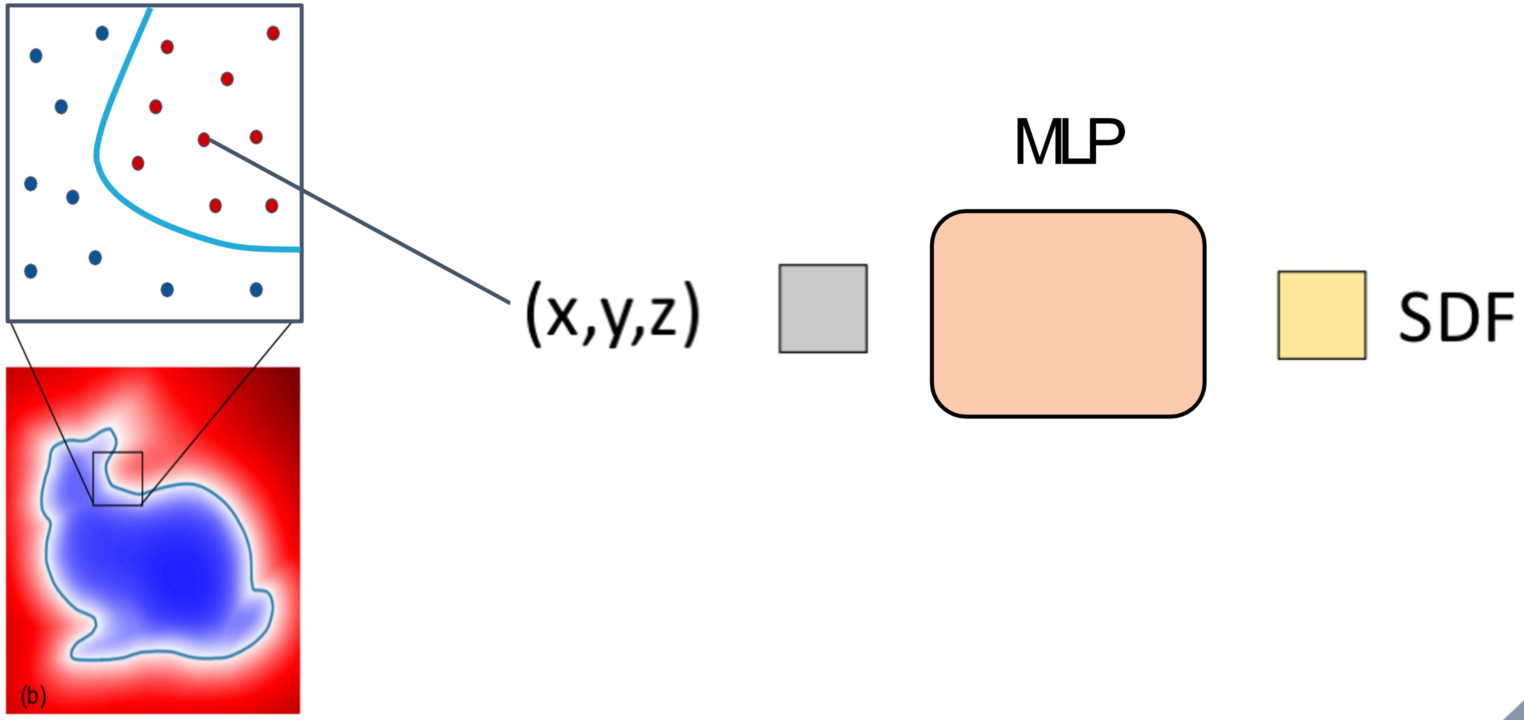
Park, Jeong Joon, et al. "Deepsdf: Learning continuous signed distance functions for shape representation." CVPR. 2019.

Surface as decision boundary



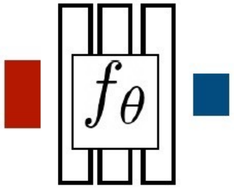
Regression of continuous SDF

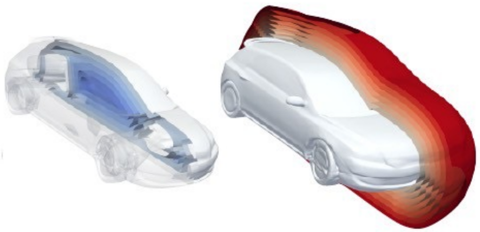
- ❑ Multi layer preception maps a point (X,Y,Z) to SDF value



SDF

Instance-specific SDFs

$\mathbf{x} \in \mathbb{R}^3$  $s \in \mathbb{R}$



$$f_\theta : \mathbb{R}^3 \to \mathbb{R}$$

Signed Distance Field (for a single instance):

(position) → (distance)

if 6 layer network with 1000-dim feature space, about 6M parameters per instance!

Loss

Training Perceptron - First Attempt

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \underbrace{\sum_{i \in \text{train}} L(\mathbf{y}^i, \overbrace{\boxed{\mathbf{f}(\mathbf{w}, \mathbf{x}^i)}}^{\text{single example loss}})}_{\substack{L(\mathbf{w}) \\ \textbf{total loss}}}$$

prediction on example $\mathbf{x}^{\mathbf{i}}$

Consider **perceptron**: $\mathbf{f}(\mathbf{w}, \mathbf{x}) = u(W^T X)$

vector representation of $\mathbf{w}$

$$W^T = [\mathbf{w}_0, \mathbf{w}_1, ..., \mathbf{w}_m]$$

$$X^T = [1, \mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m]$$

homogeneous representation of $\mathbf{x}$

Classification error loss: $L(\mathbf{y}, \mathbf{f}) = \overset{\text{Iverson brackets}}{[\mathbf{y} \neq \mathbf{f}]}$

perceptron's prediction on example $\mathbf{x}^{\mathbf{i}}$

$$\Rightarrow \quad L(W) = \underbrace{\sum_{i \in \text{train}} [\mathbf{y}^i \neq \boxed{u(W^T X^i)}]}_{\textbf{classification error counts}}$$

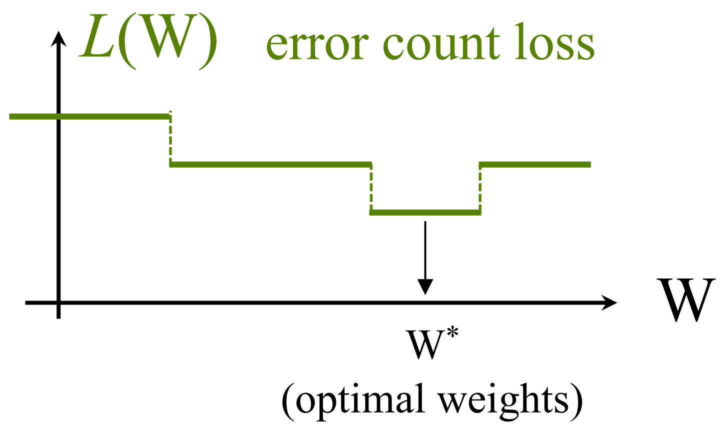
classification error <u>counts</u>

since both $\mathbf{y}^{\mathbf{i}}, u \in \{0,1\}$

extreme case of (so-called) *vanishing gradients*

Zero Gradients Problem

Classification error loss function L(W) is **piecewise constant**:



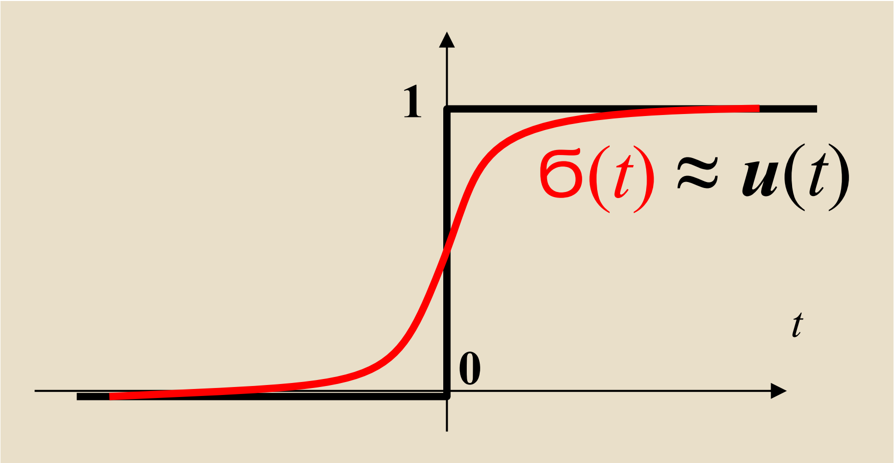
NOTE: in this case gradient ∇L is always either zero or does not exist

"error count" loss function cannot be optimized via *gradient descent*

Work-around for Zero Gradients

Perceptron: $\quad \mathbf{f}(\mathbf{w}, \mathbf{x}^i) \; = \; u(W^T X^i) \approx \sigma(W^T X^i)$

approximate decision function *u* using its **softer** version (relaxation)



$u(t)$ - ***unit step*** function
(a.k.a. *Heaviside* function)

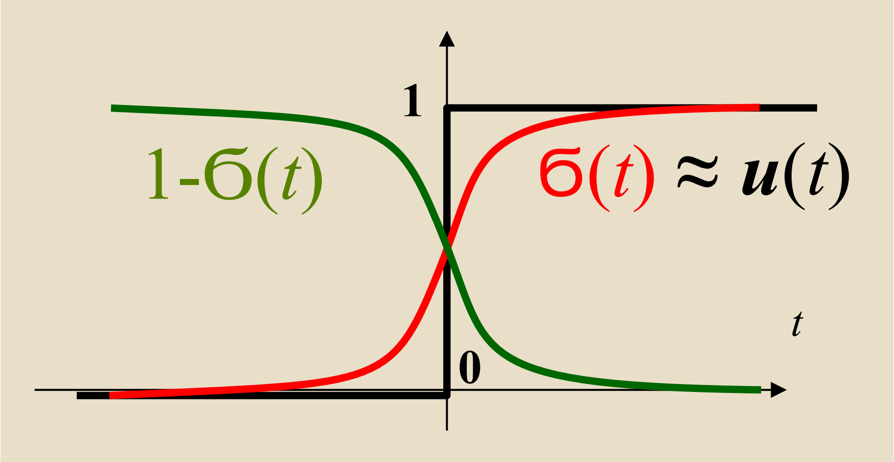
б(t) - ***sigmoid*** function

$$\boxed{\sigma(t) := \frac{1}{1+\exp(-t)}}$$

Work-around for Zero Gradients

Perceptron: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) \; = \; u(W^T X^i) \approx \; \sigma(W^T X^i)$

approximate decision function *u* using its **softer** version (relaxation)



$u(t)$ - ***unit step*** function
(a.k.a. *Heaviside* function)

$\sigma(t)$ - ***sigmoid*** function

$$\boxed{\sigma(t) := \frac{1}{1+\exp(-t)}}$$

Relaxed predictions are often interpreted as prediction **"probabilities"**

$$\Pr(\mathbf{x}^i \in \mathrm{Class1} \,|\, W) \;\; = \;\; \sigma(W^T X^i)$$

$$\Pr(\mathbf{x}^i \in \mathrm{Class0} \,|\, W) \;\; = \; 1 - \sigma(W^T X^i) \; \equiv \; \sigma(-W^T X^i)$$

Training Perceptron - Second Attempt

Perceptron approximation: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) = \boxed{u(W^T X^i) \approx \sigma(W^T X^i)}$

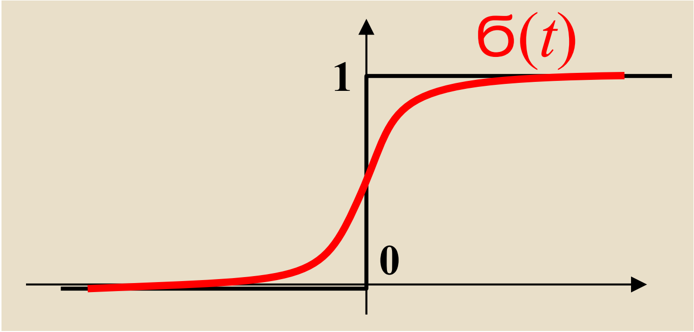
$\mathbf{y} \in \{0, 1\}$

Classification error loss: $\cancel{L(\mathbf{y}, \sigma) = [\mathbf{y} \neq \sigma]}$

<u>now makes no sense at all</u>

relaxed decision function (sigmoid) never returns exactly 0 or 1

NOTE:
To be able to use **gradient descent** we need to "**soften**" both the **decision function** and the **loss function**



$$0 < \quad \sigma(t) \equiv \frac{1}{1 + \exp(-t)} \quad < 1$$

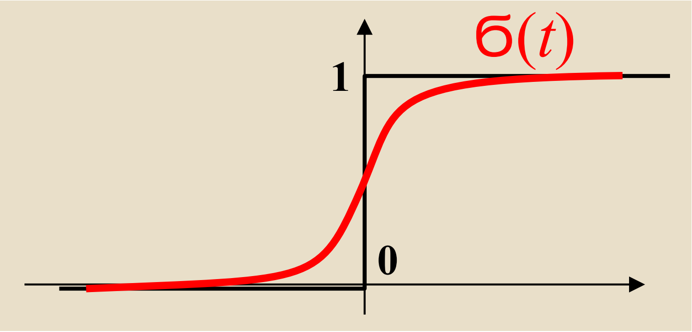
Quadratic Loss

Perceptron approximation: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) = u(W^T X^i) \approx \sigma(W^T X^i)$

$\mathbf{y} \in \{0, 1\}$

Consider **quadratic loss**: $L(\mathbf{y}, \sigma) = (\mathbf{y} - \sigma)^2$

NOTE:
Loss $L(\mathbf{y}, \sigma(W^T X))$ is now differentiable with respect to W because $L(\mathbf{y}, \sigma)$ is differentiable w.r.t. σ

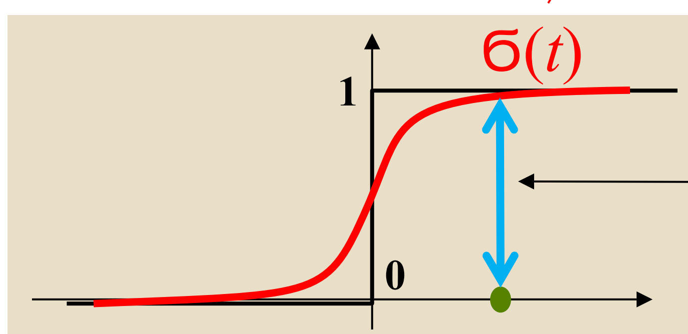


Quadratic Loss

Perceptron approximation: $$\mathbf{f}(\mathbf{w}, \mathbf{x}^i) = \boxed{u(W^T X^i) \approx \sigma(W^T X^i)}$$

$$\mathbf{y} \in \{0, 1\}$$

Consider **quadratic loss**: $$L(\mathbf{y}, \sigma) = (\mathbf{y} - \sigma)^2$$



$$(\mathbf{y}^i - \sigma(W^T X^i))$$

$$W^T X^i > 0$$

$$\mathbf{y}^i = 0$$

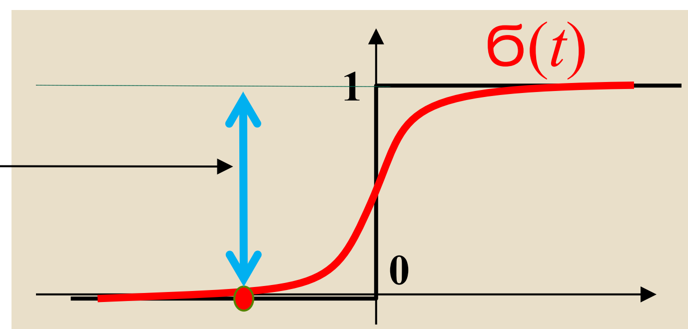
misclassified example

Quadratic Loss

Perceptron approximation: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) = \boxed{u(W^T X^i) \approx \sigma(W^T X^i)}$

Consider **quadratic loss**: $L(\mathbf{y}, \sigma) = (\mathbf{y} - \sigma)^2$, $\quad \mathbf{y} \in \{0, 1\}$

$(\mathbf{y}^j - \sigma(W^T X^j))$



$W^T X^j < 0$

$\mathbf{y}^j = 1$

another misclassified example

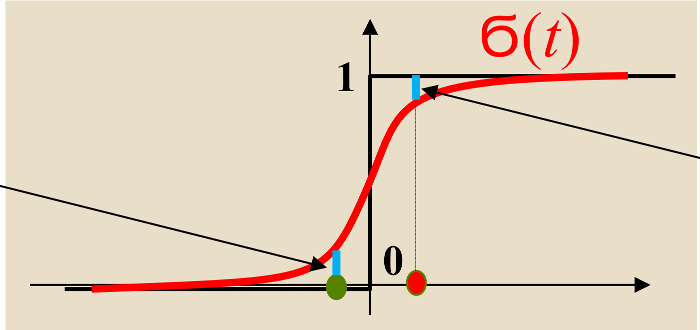
Quadratic Loss

Perceptron approximation: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) \; = \; u(W^T X^i) \;\approx\; \sigma(W^T X^i)$

$\mathbf{y} \in \{0, 1\}$

Consider **quadratic loss**: $L(\mathbf{y}, \sigma) \; = \; (\mathbf{y} - \sigma)^2$

$(\mathbf{y}^j - \sigma(W^T X^j))$



$(\mathbf{y}^i - \sigma(W^T X^i))$

$W^T X^j$ $\quad$ $W^T X^i$

$\mathbf{y}^j = 0$ $\quad$ $\mathbf{y}^i = 1$

correctly classified examples

NOTE:

loss function encourages W s.t.
correctly classified points are moved further from the decision boundary,
i.e. $W^T X^i \gg 0$ and $W^T X^j \ll 0$.

Quadratic Loss

Perceptron approximation: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) = u(W^T X^i) \approx \sigma(W^T X^i)$

Consider **quadratic loss**: $L(\mathbf{y}, \sigma) = (\mathbf{y} - \sigma)^2$

approximation for perceptron's prediction on example $\mathbf{x^i}$

Total loss $\Rightarrow$

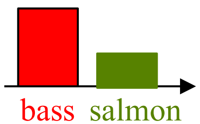
$$L(W) = \underbrace{\sum_{i \in \text{train}} (\mathbf{y}^i - \boxed{\sigma(W^T X^i)})^2}_{\textbf{Sum of Squared Differences (SSD)}}$$

(binary case)

Cross-Entropy Loss (related to *logistic regression* loss)

Perceptron approximation: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) = \boxed{u(W^T X^i) \approx \sigma(W^T X^i)}$

Consider two probability distributions
over two classes (e.g. bass or salmon) : $(\mathbf{y}, 1-\mathbf{y})$ and $(\sigma, 1-\sigma)$



$$\Pr(\mathbf{x}^i \in \text{Class1} \mid W) = \sigma(W^T X^i)$$

$$\Pr(\mathbf{x}^i \in \text{Class0} \mid W) = 1 - \sigma(W^T X^i)$$

Distance between two distributions can be evaluated via **cross-entropy**
(equivalent to ***KL divergence*** for fixed target)

From the last (optional) part of topic 9B:

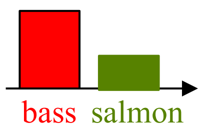
$$\boxed{H(\boldsymbol{p}, \boldsymbol{q}) := -\sum_k p_k \ln q_k}$$

(binary case)

Cross-Entropy Loss (related to *logistic regression* loss)

Perceptron approximation: $\mathbf{f}(\mathbf{w},\mathbf{x}^i) = u(W^T X^i) \approx \sigma(W^T X^i)$

Consider two probability distributions
over two classes (e.g. bass or salmon) : $(\mathbf{y}, 1-\mathbf{y})$ and $(\sigma, 1-\sigma)$



(binary)
Cross-entropy loss: $L(\mathbf{y},\sigma) = -\mathbf{y}\ln\sigma - (1-\mathbf{y})\ln(1-\sigma)$

Distance between two distributions can be evaluated via **cross-entropy**
(equivalent to ***KL divergence*** for fixed target)

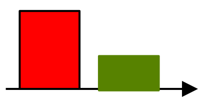
$$H(\boldsymbol{p},\boldsymbol{q}) := -\sum_k p_k \ln q_k$$

(binary case)

Cross-Entropy Loss (related to *logistic regression* loss)

Perceptron approximation: $\mathbf{f}(\mathbf{w},\mathbf{x}^i) \;=\; u(W^T X^i) \;\approx\; \sigma(W^T X^i)$

Consider two probability distributions
over two classes (e.g. bass or salmon) : $(\mathbf{y}, 1-\mathbf{y})$ and $(\sigma, 1-\sigma)$



(binary)
Cross-entropy loss:

$$L(\mathbf{y},\sigma) \;=\; -\mathbf{y}\ln\sigma \;-\; (1-\mathbf{y})\ln(1-\sigma)$$

Each data label $\mathbf{y}$ provides "deterministic" distribution $(\mathbf{y}, 1-\mathbf{y})$ that is either (1,0) or (0,1). This implies an equivalent alternative expression:

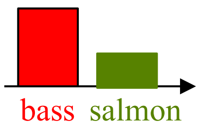
$$L(\mathbf{y},\sigma) \;=\; \begin{cases} -\ln\sigma & \text{if } \mathbf{y}=1 \\ -\ln(1-\sigma) & \text{if } \mathbf{y}=0 \end{cases}$$

(binary case)

Cross-Entropy Loss (related to *logistic regression* loss)

Perceptron approximation: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) \;=\; u(W^T X^i) \;\approx\; \sigma(W^T X^i)$

Consider two probability distributions
over two classes (e.g. bass or salmon) : $(\mathbf{y}, 1-\mathbf{y})$ and $(\sigma, 1-\sigma)$

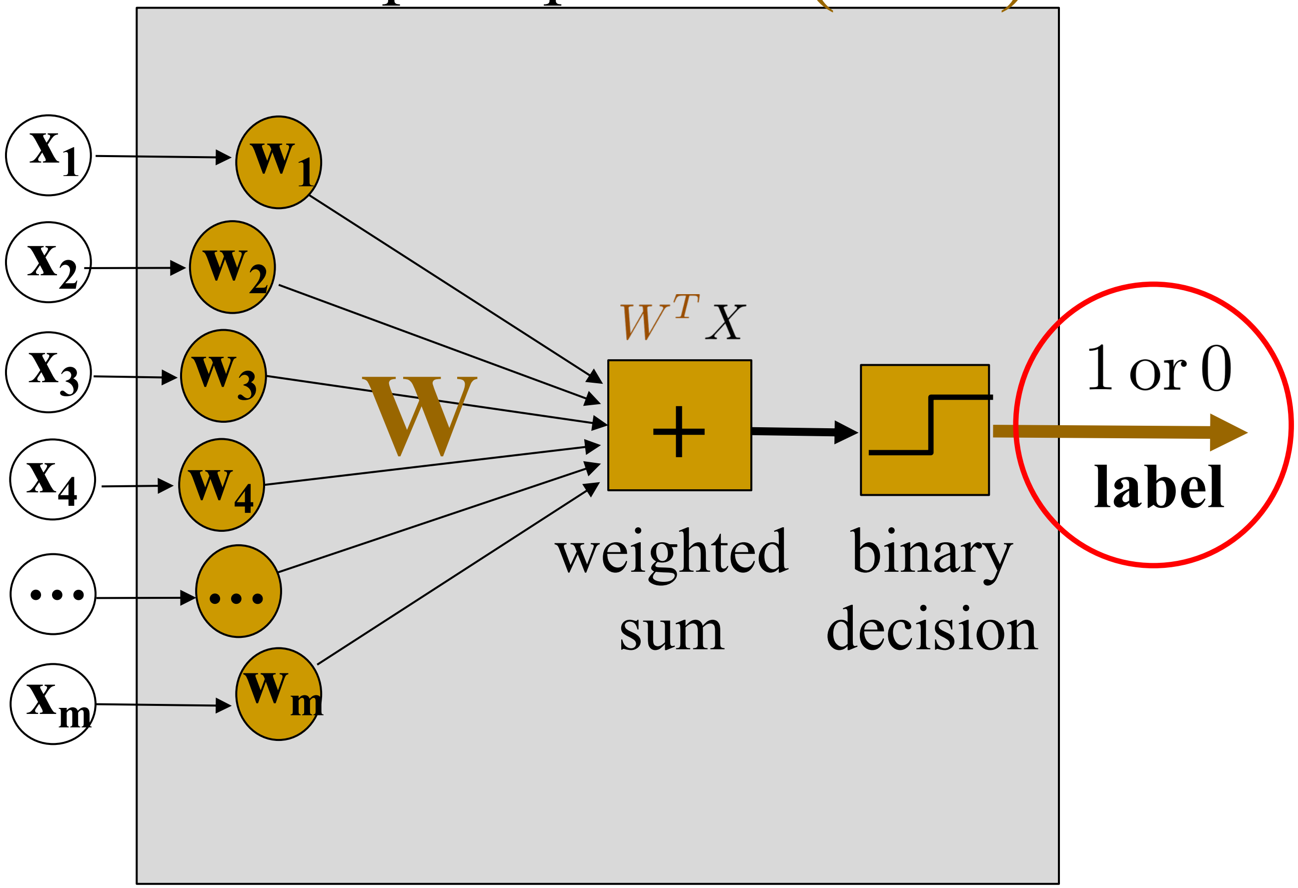


Total loss: $\sum\limits_{i \in \text{train}} \left(-\mathbf{y}^i \ln \sigma(W^T X^i) \;-\; (1-\mathbf{y}^i) \ln(1-\sigma(W^T X^i)) \right)$

$$\Rightarrow \quad \boxed{L(W) \;=\; - \underbrace{\sum_{\substack{i \in \text{train} \\ \mathbf{y}^i = 1}} \ln \sigma(W^T X^i) \;-\; \sum_{\substack{i \in \text{train} \\ \mathbf{y}^i = 0}} \ln(1-\sigma(W^T X^i))}_{\text{sum of } \textbf{Negative Log-Likelihoods (NLL)}}}$$

Towards Multi-label Classification

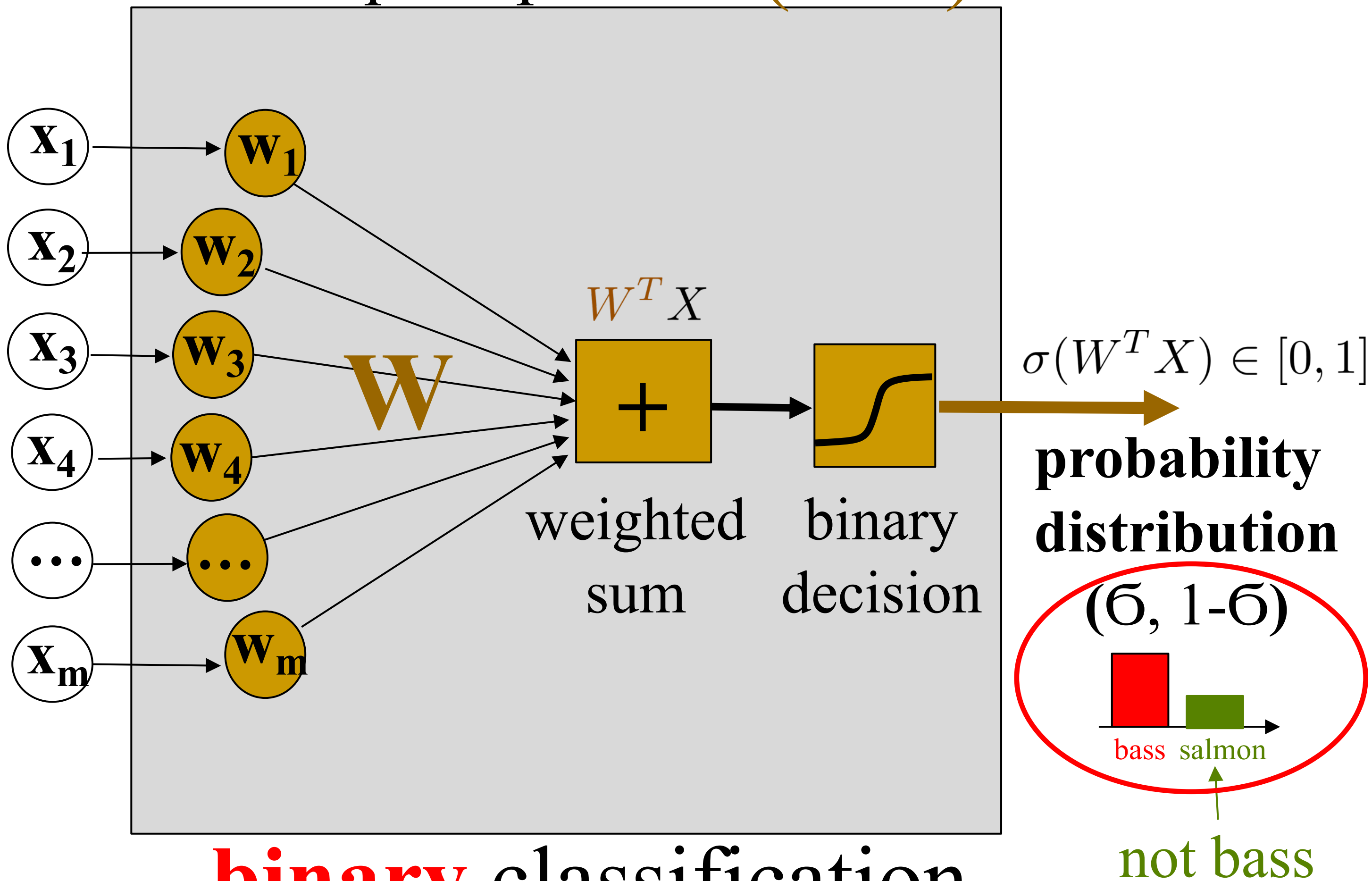
Remember: basic perceptron $\mathbf{u}(\mathbf{w}^T\mathbf{x})$



binary classification

Towards Multi-label Classification

Remember: "relaxed" perceptron $\sigma(\mathbf{w}^T\mathbf{x})$

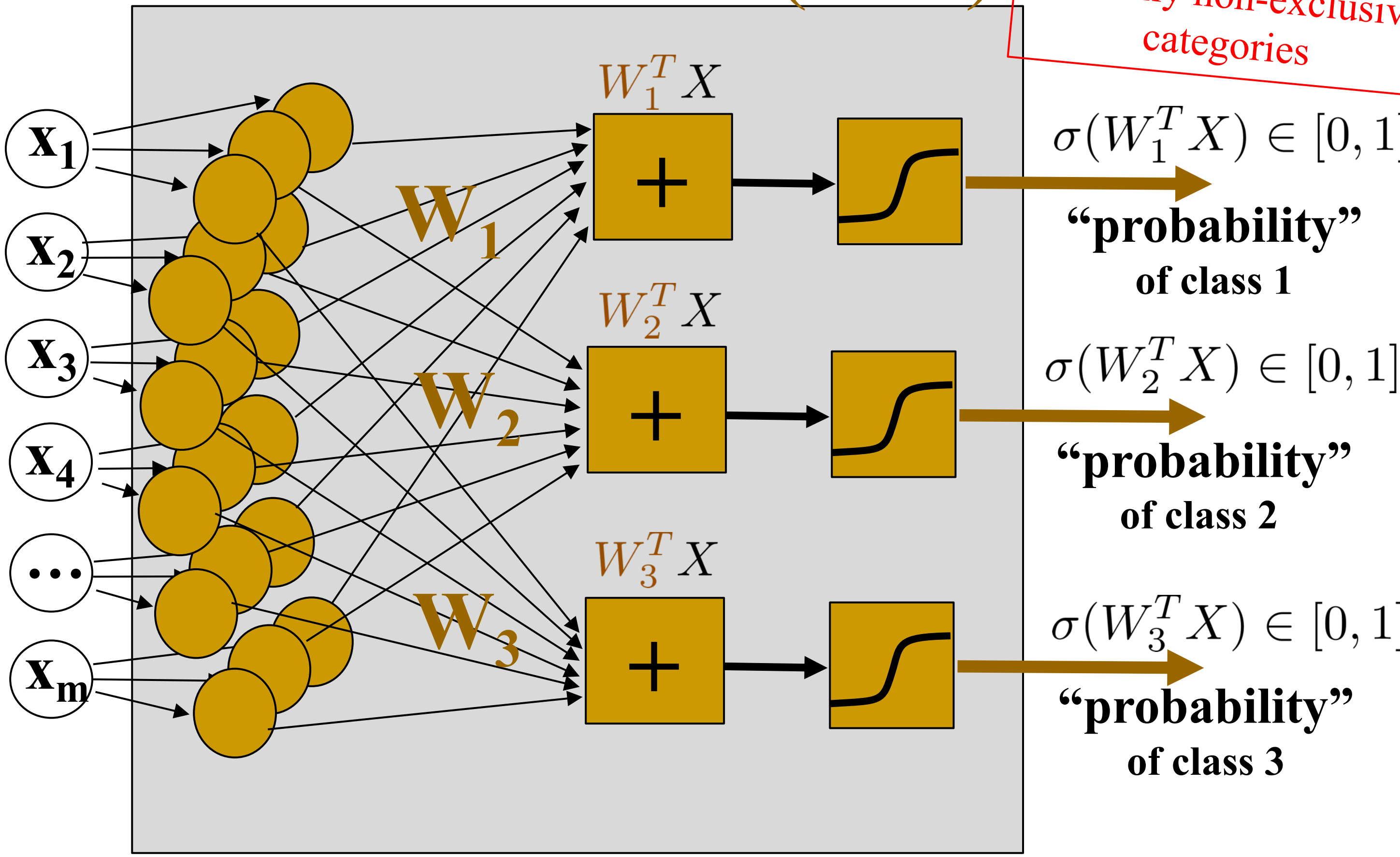


binary classification

Towards Multi-label Classification

use K linear transforms W_k and sigmoids $\sigma(\mathbf{w}^T\mathbf{x})$

This is used for mutually non-exclusive categories

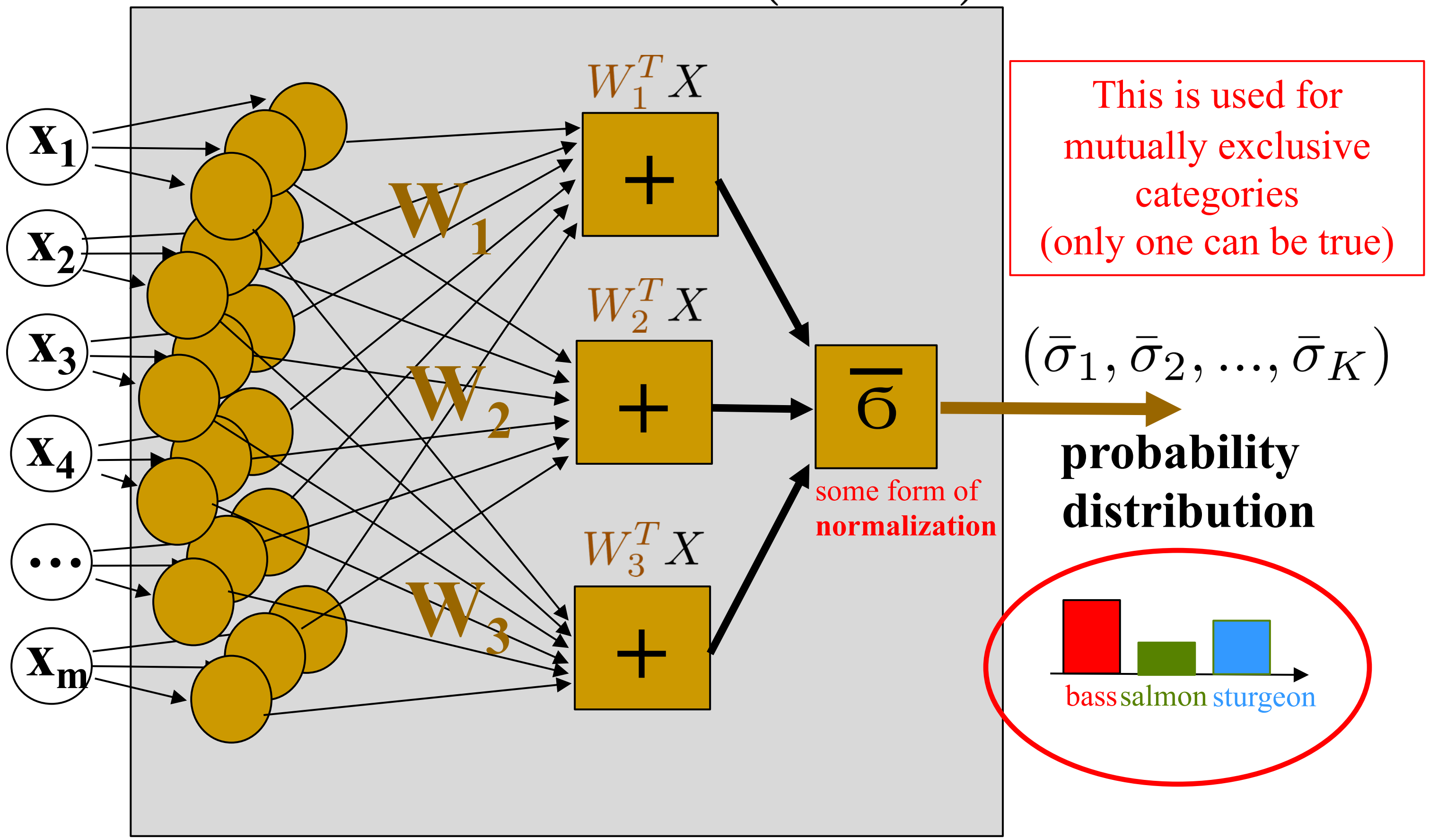


multi-label classification

Such "probability scores" $\sigma_1, \sigma_2, \ldots, \sigma_K$ over K classes do not add up to 1

Common Approach: Soft-Max

use K linear transforms W_k and **soft-max** $\bar{\sigma}(\mathbf{W}X)$



multi-label classification

Notation: K rows of matrix **W** are vectors W_k so that vector **W**X has elements $W_k^T X$

Soft-Max Function $\bar{\sigma} : \mathbb{R}^K \rightarrow \Delta_K$

$$\begin{pmatrix} a^1 \\ a^2 \\ \cdots \\ a^K \end{pmatrix} \xrightarrow{\text{softmax}} \begin{pmatrix} \frac{\exp a^1}{\sum_k \exp a^k} \\ \frac{\exp a^2}{\sum_k \exp a^k} \\ \cdots \\ \frac{\exp a^K}{\sum_k \exp a^k} \end{pmatrix}$$

$\mathbf{a} \in \mathbb{R}^K$ $\qquad \bar{\sigma}(\mathbf{a}) \in \Delta_K$

Example:

$$\begin{bmatrix} -3 \\ 2 \\ 1 \end{bmatrix} \xrightarrow{\text{softmax}} \begin{bmatrix} \frac{\exp(-3)}{\mathbf{exp}(-3)+\mathbf{exp}(2)+\mathbf{exp}(1)} \\ \frac{\exp(2)}{\mathbf{exp}(-3)+\mathbf{exp}(2)+\mathbf{exp}(1)} \\ \frac{\exp(1)}{\mathbf{exp}(-3)+\mathbf{exp}(2)+\mathbf{exp}(1)} \end{bmatrix}$$

$$= \begin{bmatrix} 0.005 \\ 0.7275 \\ 0.2676 \end{bmatrix}$$

Soft-max normalizes logits vector $\mathbf{a}$ converting it to distribution over classes

Soft-Max Function $\bar{\sigma} : \mathbb{R}^K \to \Delta_K$

$$\begin{pmatrix} a^1 \\ a^2 \\ \dots \\ a^K \end{pmatrix} \xrightarrow{\text{softmax}} \begin{pmatrix} \frac{\exp a^1}{\sum_k \exp a^k} \\ \frac{\exp a^2}{\sum_k \exp a^k} \\ \dots \\ \frac{\exp a^K}{\sum_k \exp a^k} \end{pmatrix}$$

$\mathbf{a} \in \mathbb{R}^K$

$\bar{\sigma}(\mathbf{a}) \in \Delta_K$

NN Example:

$$\begin{pmatrix} W_1^T X \\ W_2^T X \\ \dots \\ W_K^T X \end{pmatrix} \xrightarrow{\text{softmax}} \begin{pmatrix} \frac{\exp W_1^T X}{\sum_k \exp W_k^T X} \\ \frac{\exp W_2^T X}{\sum_k \exp W_k^T X} \\ \dots \\ \frac{\exp W_K^T X}{\sum_k \exp W_k^T X} \end{pmatrix}$$

$\mathbf{W} X$

$\bar{\sigma}(\mathbf{W} X)$

$(\bar{\sigma}_1, \bar{\sigma}_2, \dots, \bar{\sigma}_K)$

Soft-max normalizes logits vector **a** converting it to distribution over classes

Soft-Max Function $\bar{\sigma} : \mathbb{R}^K \to \Delta_K$

NN Example:

$$\begin{pmatrix} W_1^T X \\ W_2^T X \\ \cdots \\ W_K^T X \end{pmatrix} \xrightarrow{\text{softmax}} \begin{pmatrix} \frac{\exp W_1^T X}{\sum_k \exp W_k^T X} \\ \frac{\exp W_2^T X}{\sum_k \exp W_k^T X} \\ \cdots \\ \frac{\exp W_K^T X}{\sum_k \exp W_k^T X} \end{pmatrix}$$

$\mathbf{W}X$

$\bar{\sigma}(\mathbf{W}X)$

$(\bar{\sigma}_1, \bar{\sigma}_2, \ldots, \bar{\sigma}_K)$

NOTE:

soft-max generalizes sigmoid

to multi-class predictions. Indeed, consider binary perceptron with scalar linear discriminator $W^{\mathrm{T}}X$ (e.g. for class 1)

$$\underset{\text{sigmoid}}{\sigma(W^T X)} = \frac{1}{1+e^{-W^T X}}$$

$$\equiv \frac{e^{\frac{1}{2}W^T X}}{e^{\frac{1}{2}W^T X} + e^{-\frac{1}{2}W^T X}} = \bar{\sigma}_1 \begin{pmatrix} \frac{1}{2}W^T X \\ -\frac{1}{2}W^T X \end{pmatrix}$$

class 1 output of **soft-max** for a combination of two linear predictors: $\frac{1}{2}W^{\mathrm{T}}X$ for class 1 and - $\frac{1}{2}W^{\mathrm{T}}X$ for class $\neg 1$ (class 0)

Soft-max normalizes logits vector **a** converting it to distribution over classes

(general multi-class case)

Cross-Entropy Loss

K-label perceptron's output: $\bar{\sigma}(\mathbf{W}X^i)$ for example X^i

k-th index

Multi-valued label $\mathbf{y}^i = k$ gives **one-hot** distribution $\bar{\mathbf{y}}^i = (0, 0, 1, 0, \ldots, 0)$

Consider two probability distributions
over K classes (e.g. bass, salmon, sturgeon) : $\bar{\mathbf{y}}^i$ and $(\bar{\sigma}_1, \bar{\sigma}_2, \bar{\sigma}_3, ..., \bar{\sigma}_K)$

$$\Pr(\mathbf{x}^i \in \text{Class}\, k \mid W) \;=\; \bar{\sigma}_k(WX^i)$$

Total loss: $$L(W) = \sum_{i \in \text{train}} \overbrace{\sum_k -\bar{\mathbf{y}}^i_k \ln \bar{\sigma}_k(WX^i)}^{\textbf{cross entropy}}$$

$$\Rightarrow \quad \boxed{L(W) = -\sum_{i \in \text{train}} \ln \bar{\sigma}_{\mathbf{y}^i}(WX^i)}$$

sum of **Negative Log-Likelihoods (NLL)**

soft-max vs *arg-max*

Multi-label (linear) Classification

Define K linear transforms, from features X to K "*logits*"

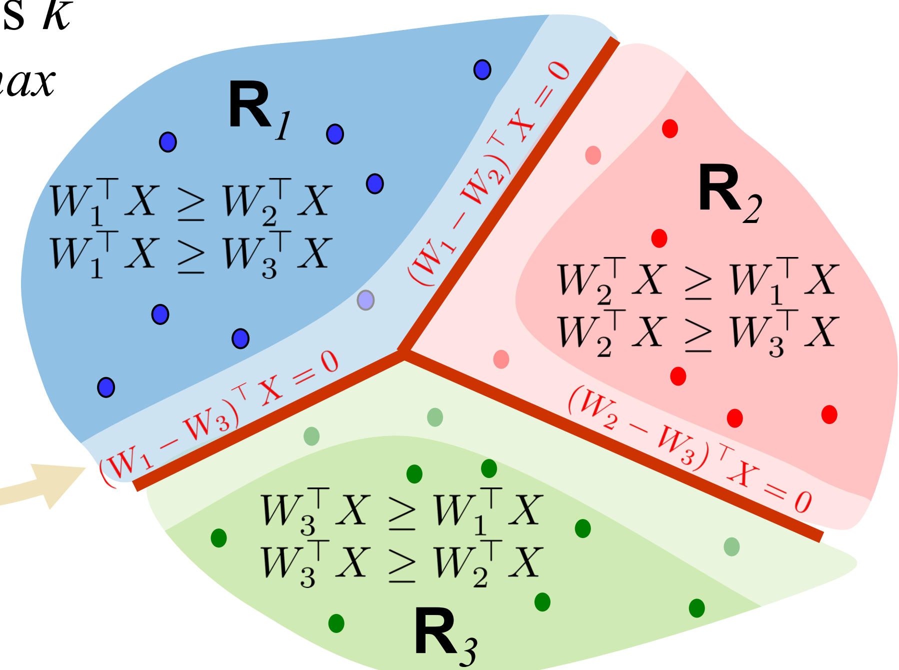
$$logit_k(X) = W_k^T X \qquad \text{for } k = 1, 2, \ldots K$$

- **arg-max** assigns X to class k corresponding to the largest logit

$$\arg\max_k \{W_k^T X\}$$

- Let $\mathbf{R}_k$ be decision region for class k
  all points X assigned to class k by *arg-max*

soft-max $\bar{\sigma}\{W_k^\top X\}$ softens hard **arg-max** predictions similarly to how sigmoid softens unit-step function



Summary

- ❑Shallow neural network
- ❑Universal function approximation theorem
- ❑Deep neural network
  - ❑Multi layer perceptron
  - ❑An example: Implicit neural filed for shape representation
- ❑Loss
  - ❑Sigmoid, Softmax
  - ❑Cross entropy loss, quadratic loss

Next

- ❑How to train neural networks?