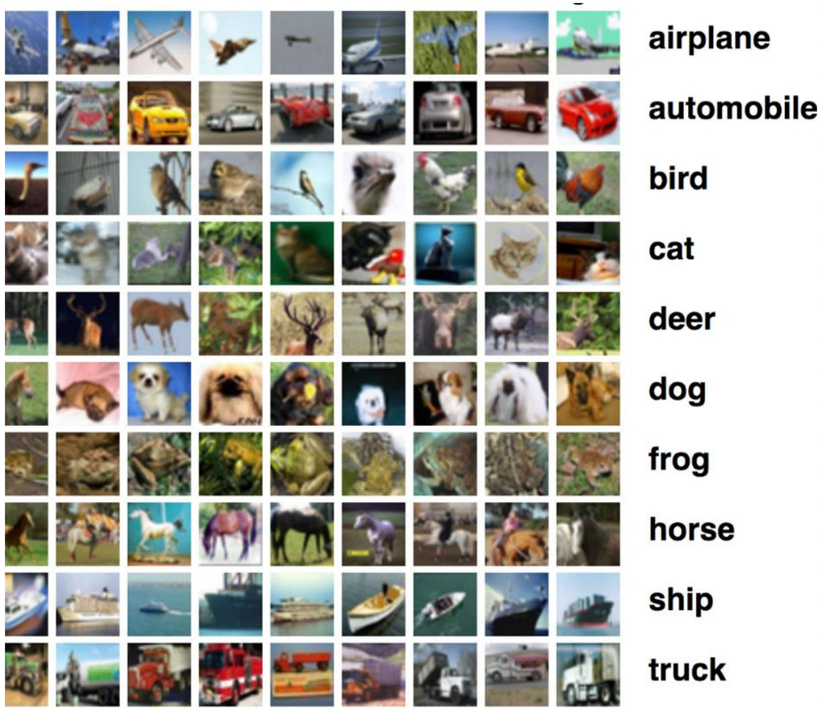
ECE 4252/8803: Fundamentals of Machine Learning (FunML)
Fall 2024

Lecture 5: Classifiers (Introduction to Neural Networks)



Reminders

- Reminders
  - HW 1 is due this Friday, 6-Sept
  - HW 2 was posted and due next Friday, 13-Sept
- Lecture 2 Scribe notes were posted
  - Will be posted by weekends going forward
  - Please see the posted notes to get an idea of expectation
  - Q/A at the end added by GTA, Shiva
  - Focus must be on:
    - Visuals
    - Numerical examples
    - Concepts

Classifier Comparison

Methods	Assumptions on Feat. Dist.	Feat. Normalization	Cost Function	Regularization	Linear Classifier	Prob. View of Prediction	Generative/Discriminative	Parametric/ Non-parametric	Overfitting
Logistic Regression	No	Required	BCE (convex)	Additional term	Linear	Yes	Discriminative	Parametric	Not often
K Nearest Neighbors	No	Required	N/A	N/A	Non-linear	N/A	Discriminative	Non-parametric	when k is too small
Decision Trees	No	Not Required	N/A	N/A	Non-linear	N/A	Discriminative	Non-parametric	with large depth
Support Vector Machines	No	Required	Hinge (convex)	C (control robustness)	Linear/ Non-linear(kernel)	N/A	Discriminative	Parametric	Not often
Naïve Bayes	Conditional independent	Not Required	N/A	N/A	Non-linear /Linear (Gaussian)	Yes	Generative	Parametric	Not often
Artificial Neural Networks	No	Required	Non-convex	Additional term	Non-linear	Yes	Discriminative/ Generative	Parametric	with many layers

Overview

In this Lecture..

- Nearest Neighbor
- Naïve Bayes
- Logistic Regression
- Decision Trees
- Support Vector Machines
- Artificial Neural Networks
  - Overview
  - Activation Function
  - Perceptron Network
  - Multi-layer ANN
  - Feedforward and Backward Error Propagation
  - Learning Algorithm
  - Image Classification using ANNs

Artificial Neural Networks

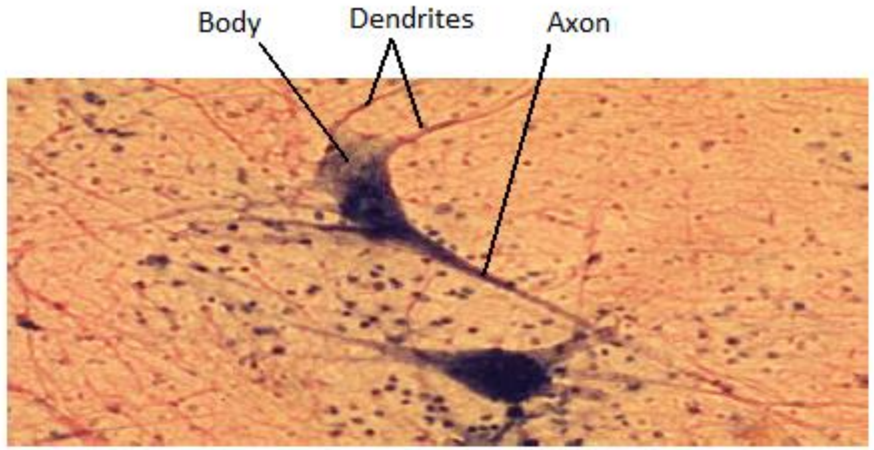
Overview

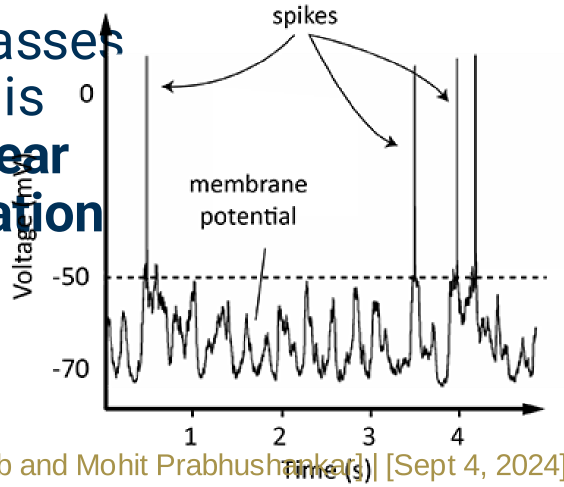
- Overview
- Activation Function
- Perceptron Network
- Multi-layer ANN
- Feedforward and Backward Error Propagation
- Learning Algorithm
- Image Classification using ANNs

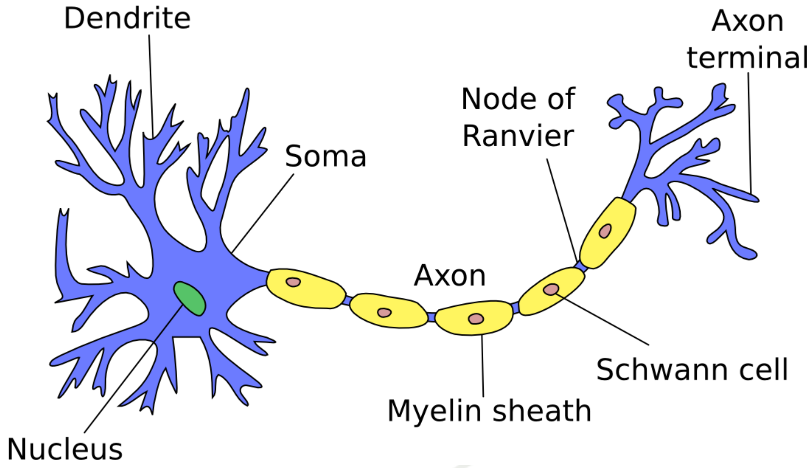
Artificial Neural Networks

Biological Neurons

- The brain has approximately 100 billion **neurons**
- Neurons communicate through electro-chemical signals. Neurons are connected through junctions called synapses.
- Each neuron receives thousands of connections with other neurons, constantly receiving incoming signals.
- If the resulting sum of the signals surpasses a certain voltage threshold, a response is sent through the axon. This is a **non linear** relation and motivates the use of **activation**



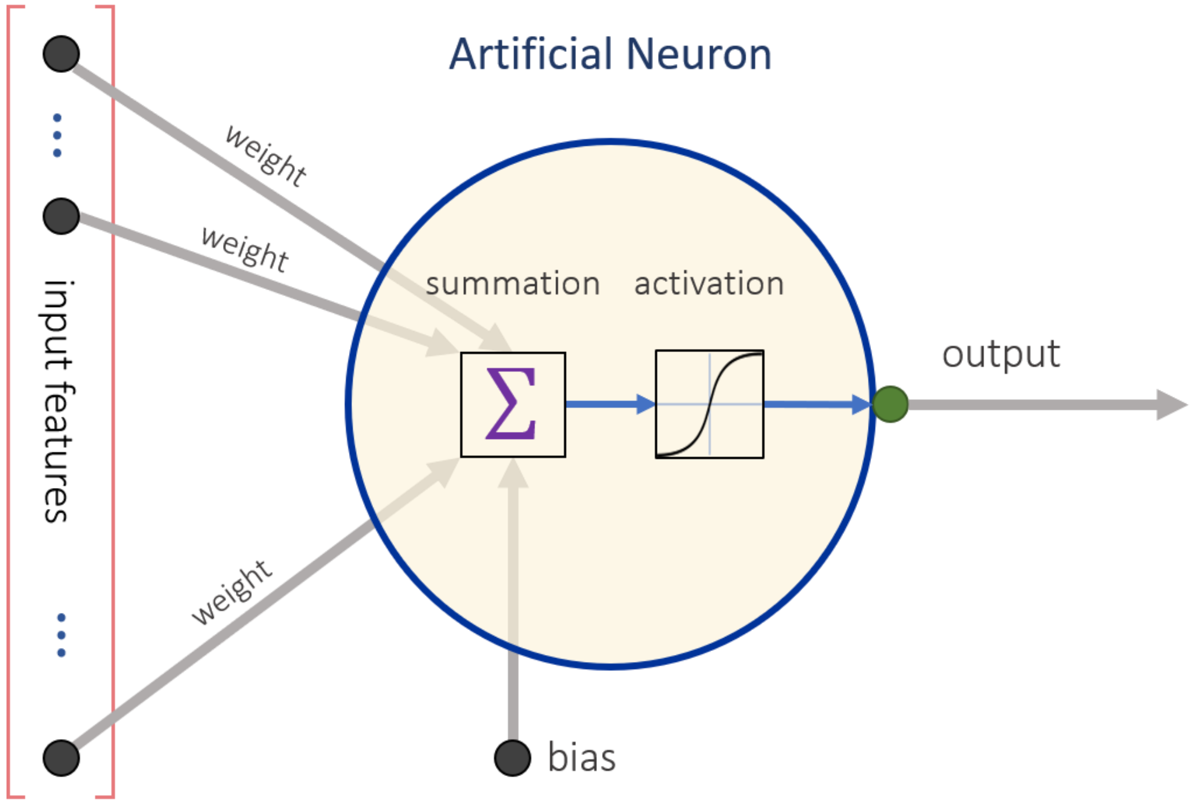




Artificial Neural Networks

Artificial Neuron

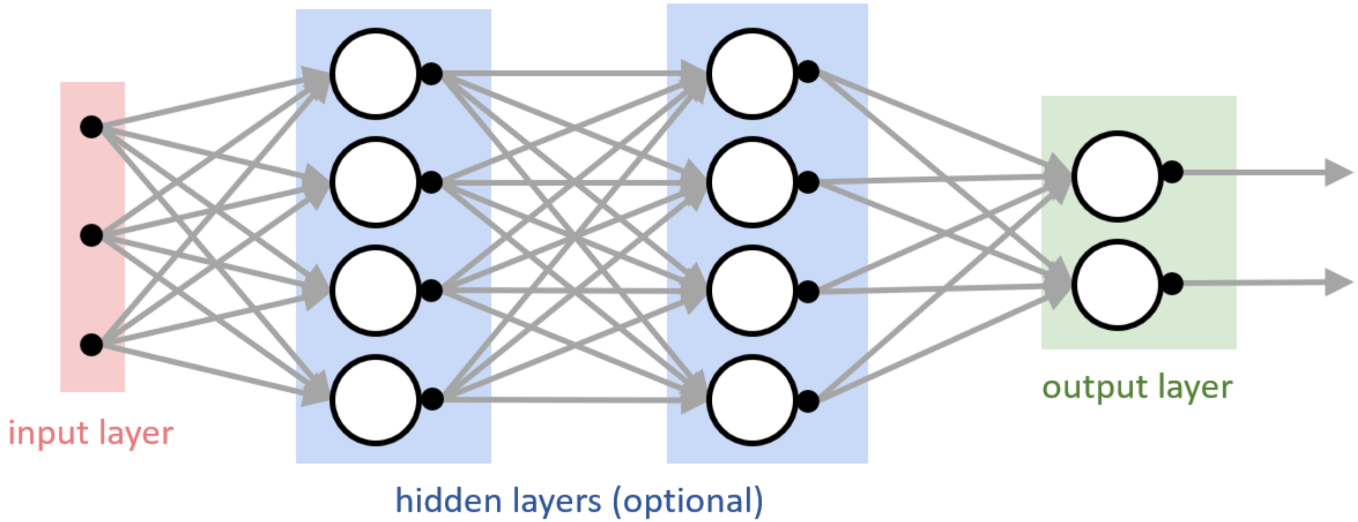
- A computational unit consisting of:
  - A single **output**
  - Multiple **inputs**
  - Input **weights**
  - A **bias** input
  - An **activation function**



Artificial Neural Networks

Artificial Neurons within a Network

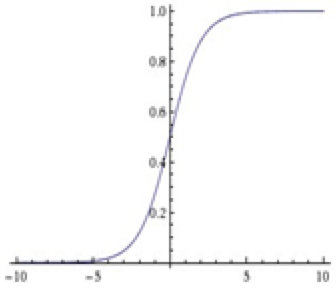
- Typically, artificial neurons in ANNs are connected in **layers**:
  - An **input** layer (Layer 0)
  - An **output** layer (Layer K)
  - Zero or more **hidden** (middle) layers (Layers $1 \dots K-1$)



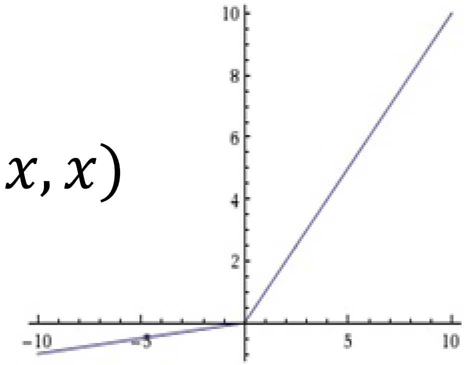
Artificial Neural Networks

Common Activation Functions

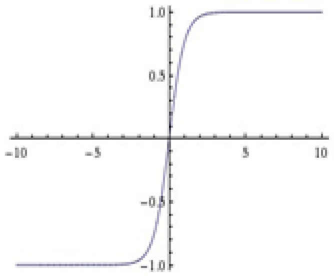
Sigmoid $\sigma(x) = \frac{1}{(1+e^{-x})}$



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

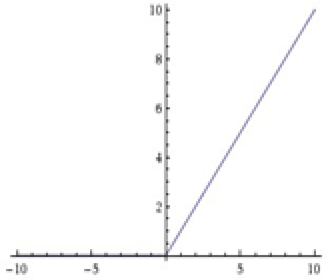


Tanh $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$

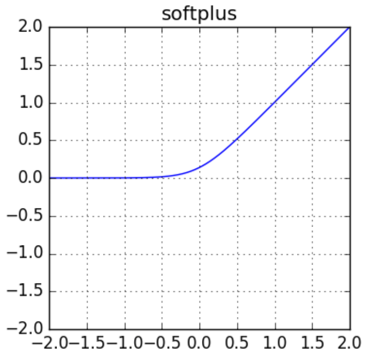


Maxout $max(w_1^T x + b_1, w_2^T x + b_2)$

ReLU $\max(0, x)$



SoftPlus $f(x) = ln(1 + e^{x})$



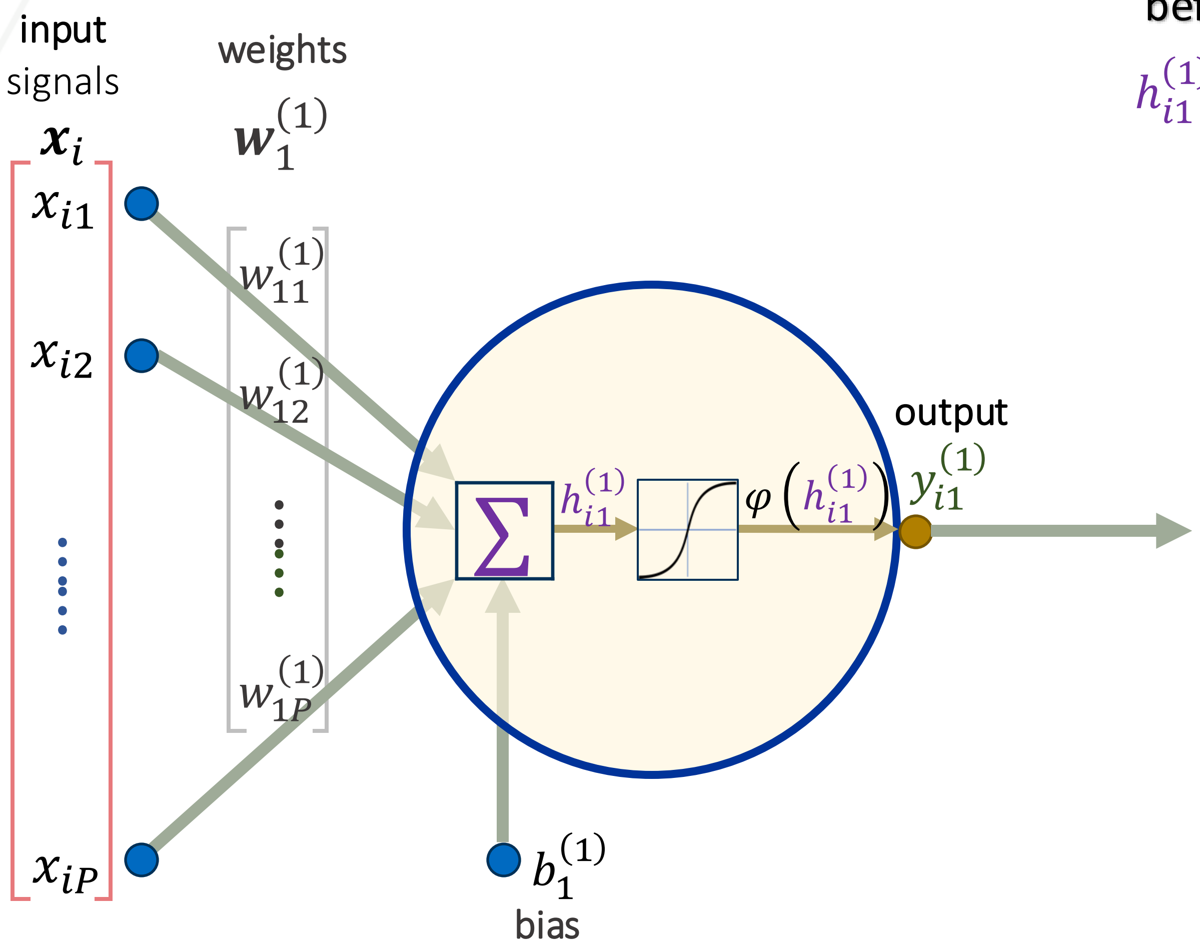
OLIVES @GeorgiaTech

Georgia Tech

Artificial Neural Networks

The Perceptron

- Single-layer Perceptron (SLP)



weighted input to the neuron before activation φ

$$h_{i1}^{(1)} = \left(\boldsymbol{w}_j^{(1)}\right)^T \boldsymbol{x}_i + b_1^{(1)}$$

output of the neuron after activation φ

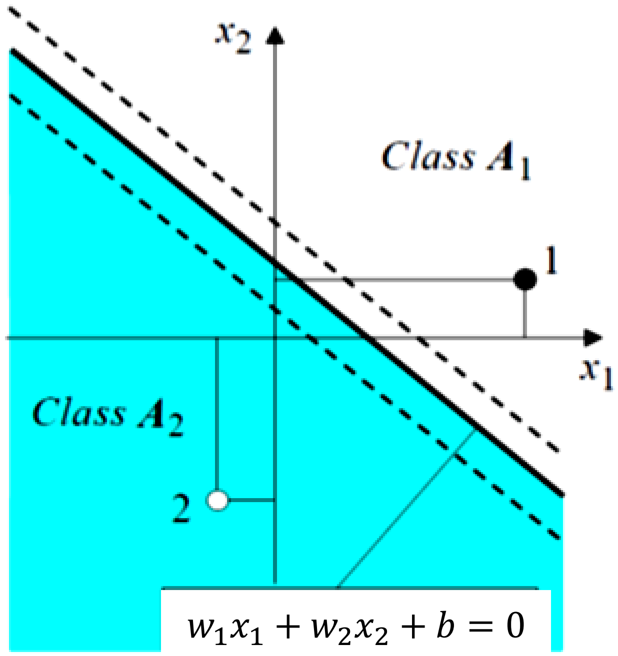
$$\varphi\left(h_{i1}^{(1)}\right)$$

The simplest form of the perceptron uses linear activation $\varphi\left(h_{i1}^{(1)}\right) = h_{i1}^{(1)}$, the outputs are binary (i.e. $y_{i1}^{(1)} \in \{1, -1\}$):

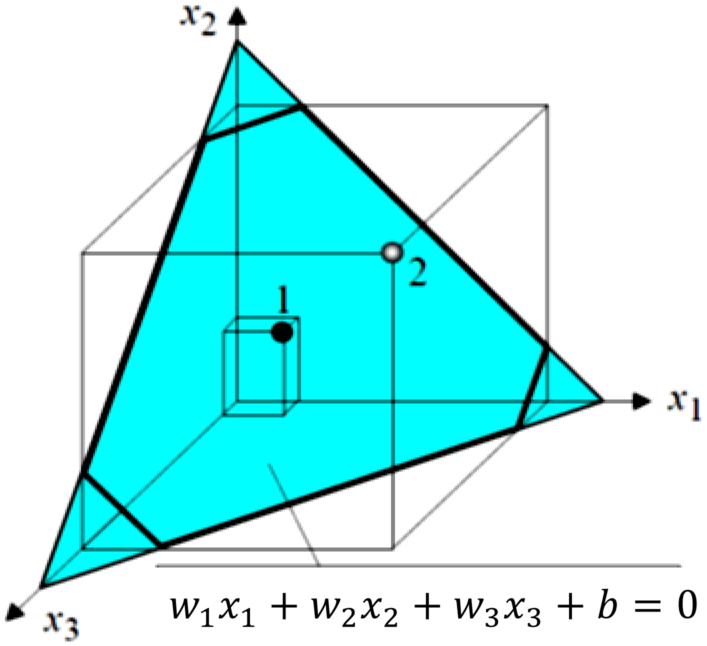
$$y_{i1}^{(1)} = \begin{cases} +1, if \left(\boldsymbol{w}_j^{(1)}\right)^T \boldsymbol{x}_i + b_1^{(1)} \geq 0 \\ -1, if \left(\boldsymbol{w}_j^{(1)}\right)^T \boldsymbol{x}_i + b_1^{(1)} < 0 \end{cases}$$

The above model applies to linearly separable data

- Linear separability in the perceptron



(a) Two-input perceptron.



(b) Three-input perceptron.

The SLP Perceptron Learning Algorithm

1. **Weight Initialization**

   Initial weights $\boldsymbol{w} = [w_1, w_2, \dots, w_P]$ are set to random values

2. **Neuron Activation**

   Calculate perceptron outputs with activation function such as sigmoid:

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

3. **Weight Update**

   Weights are updated based on the learning rule:

   $$w_i^{t+1} = w_i^t + \alpha x_i^t e^t$$

   Where

   - e: difference between the calculated output and desired output
   - α: learning rate which is a positive constant less than 1

4. **Iteration**

   Input next training sample, and the algorithm keeps iterating between steps 2 and 3 until convergence

Artificial Neural Networks

The XOR Problem

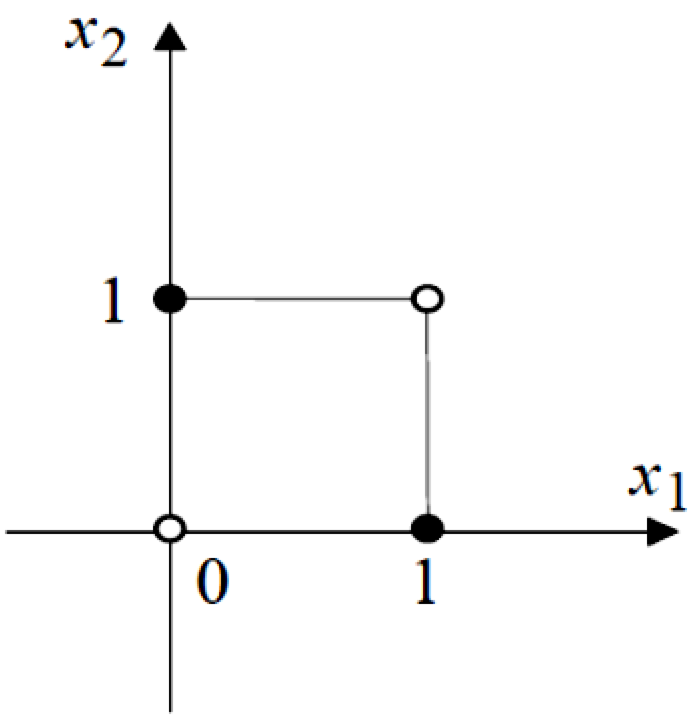
The XOR, or “exclusive or”, problem is a classic problem in ANN research. It is the problem of using a neural network to predict the outputs of XOR logic gates given two binary inputs.

o is the output=0

● is the output=1

These are not linearly separable using one neuron (a line hyperplane).

We need a decision plane! ➔ hidden layer (MLP)



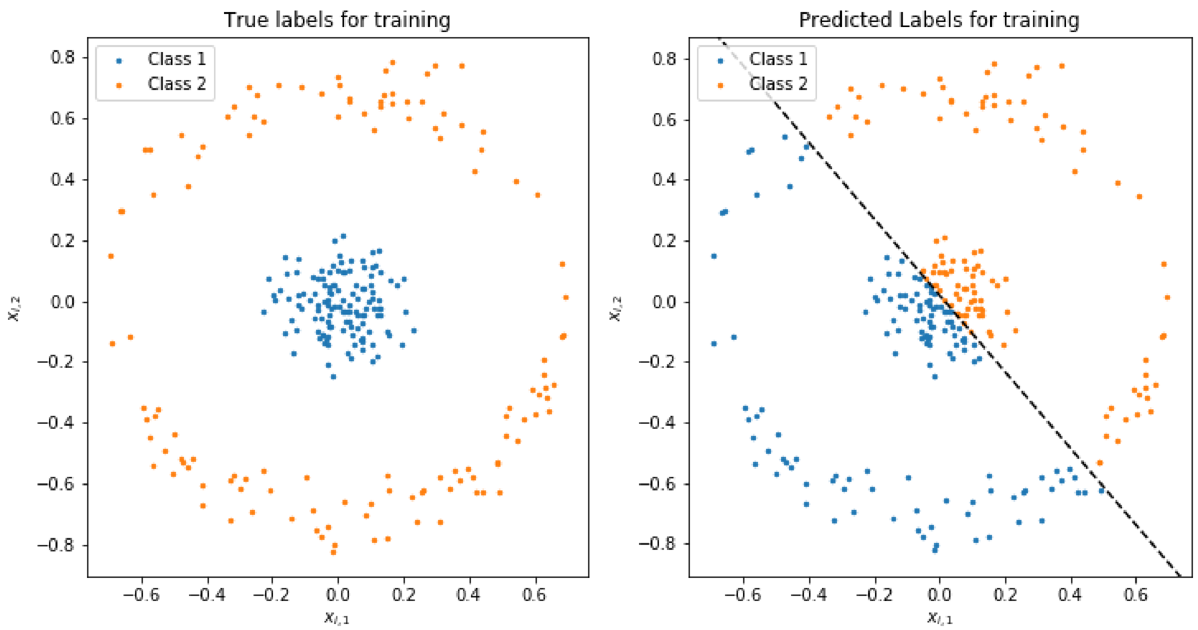
(c) *Exclusive-OR* ($x_1 \oplus x_2$)

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

Artificial Neural Networks
Hidden Layers

- Single neurons with linear activation fail to classify non-linearly separable dataset

- Non-linearly separable data requires
  - Non-linear Activation
  - Multi-layer networks of neurons (Multi-layer perceptron)



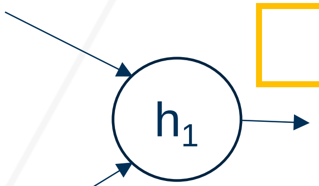
SLP with linear activation classifying non-linearly separable data

Artificial Neural Networks

Solving XOR with Hidden Layers

Input Neurons

Output Neuron

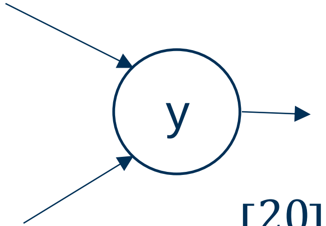


$$w_{h1} = \begin{bmatrix} 20 \\ 20 \end{bmatrix}$$

$$b_{h1} = -10$$

$$w_{h2} = \begin{bmatrix} -20 \\ -20 \end{bmatrix}$$

$$b_{h2} = 30$$



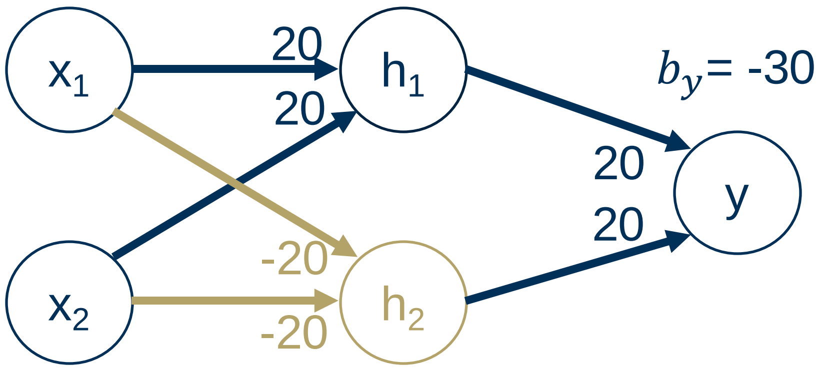
$$w_y = \begin{bmatrix} 20 \\ 20 \end{bmatrix}$$

$$b_y = -30$$

$$out_{h1} = \sigma_{h1}(w_{h1}[x_1 \quad x_2] + b_{h1}) \qquad out_{h2} = \sigma_{h2}(w_{h2}[x_1 \quad x_2] + b_{h2}) \qquad out_y = \sigma_y(w_y[out_{h1} \quad out_{h2}] + b_y)$$

b_{h1}= -10 $\quad out_{h1} = \sigma_{h1}$ (20 x_1 + 20 x_2 – 10)

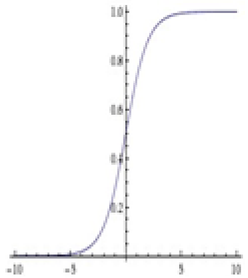
X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0



b_y= -30

$out_y = \sigma_y$ (20 h_1 + 20 h_2 – 30)

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



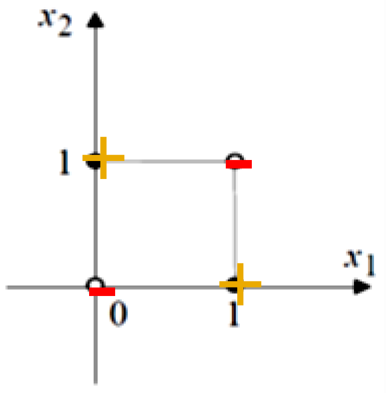
b_{h2}= 30 $\quad out_{h2} = \sigma_{h2}$ (-20 x_1 - 20 x_2 + 30)

OLIVES @GeorgiaTech

Georgia Tech

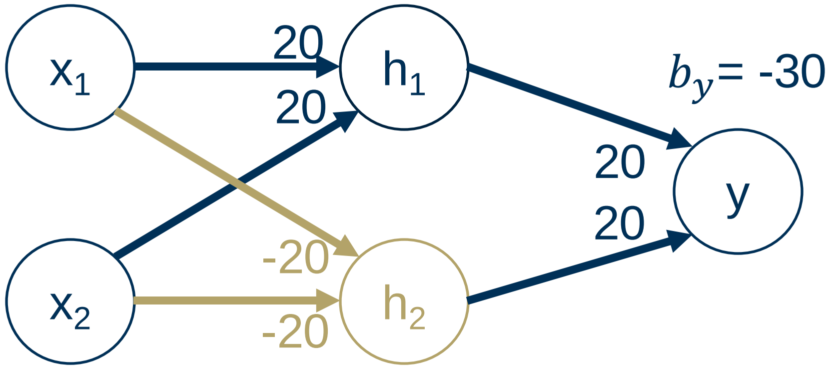
Artificial Neural Networks

Solving XOR with Hidden Layers



(c) *Exclusive-OR*
$(x_1 \oplus x_2)$

b_{h1} = -10 $\quad out_{h1} = \sigma_{h1}$ (20 x_1 + 20 x_2 – 10)

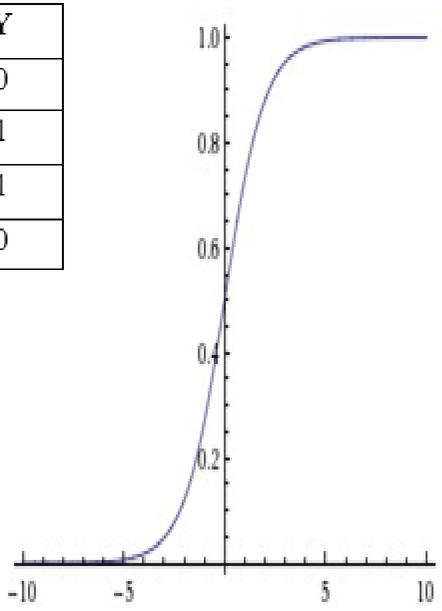


b_y= -30

$out_y = \sigma_y$ (20 h_1 + 20 h_2 – 30)

b_{h2}= 30 $\quad out_{h2} = \sigma_{h2}$ (-20 x_1 - 20 x_2 + 30)

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0



Input (0,0) ➔ σ (20 X 0 + 20 X 0– 10) ≈ 0 σ (-20 X 0 - 20 X 0 +30) ≈ 1 σ (20 X 0 + 20 X 1 – 30) ≈ 0

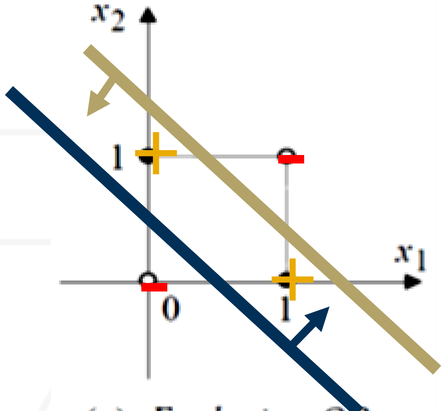
Input (1,1) ➔ σ (20 X 1 + 20 X 1– 10) ≈ 1 σ (-20 X 1 - 20 X 1 +30) ≈ 0 σ (20 X 1 + 20 X 0 – 30) ≈ 0

Input (0,1) ➔ σ (20 X 0 + 20 X 1– 10) ≈ 1 σ (-20 X 0 - 20 X 1 +30) ≈ 1 σ (20 X 1 + 20 X 1 – 30) ≈ 1

Input (1,0) ➔ σ (20 X 1 + 20 X 0– 10) ≈ 1 σ (-20 X 1 - 20 X 0 +30) ≈ 1 σ (20 X 1 + 20 X 1 – 30) ≈ 1

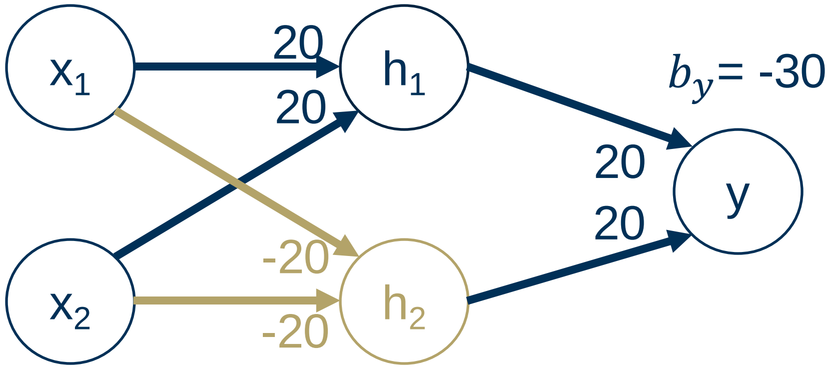
Artificial Neural Networks

Solving XOR with Hidden Layers



(c) *Exclusive-OR*
$(x_1 \oplus x_2)$

b_{h1} = -10 $\quad out_{h1} = \sigma_{h1}$ (20 x_1 + 20 x_2 – 10)

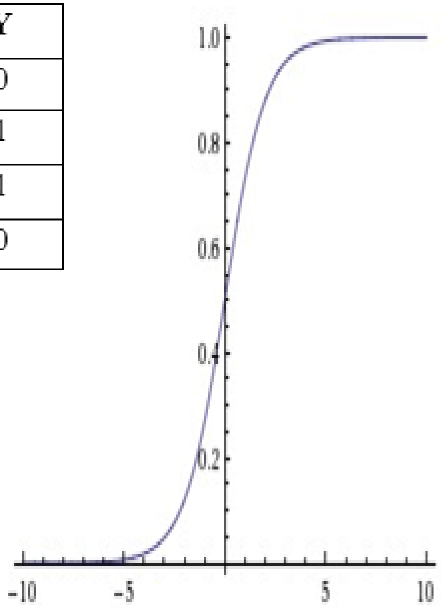


b_y= -30

$out_y = \sigma_y$ (20 h_1 + 20 h_2 – 30)

b_{h2}= 30 $\quad out_{h2} = \sigma_{h2}$ (-20 x_1 - 20 x_2 + 30)

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0



Input (0,0) ➔ σ (20 X 0 + 20 X 0− 10) ≈ 0 σ (-20 X 0 - 20 X 0 +30) ≈ 1 σ (20 X 0 + 20 X 1 − 30) ≈ 0

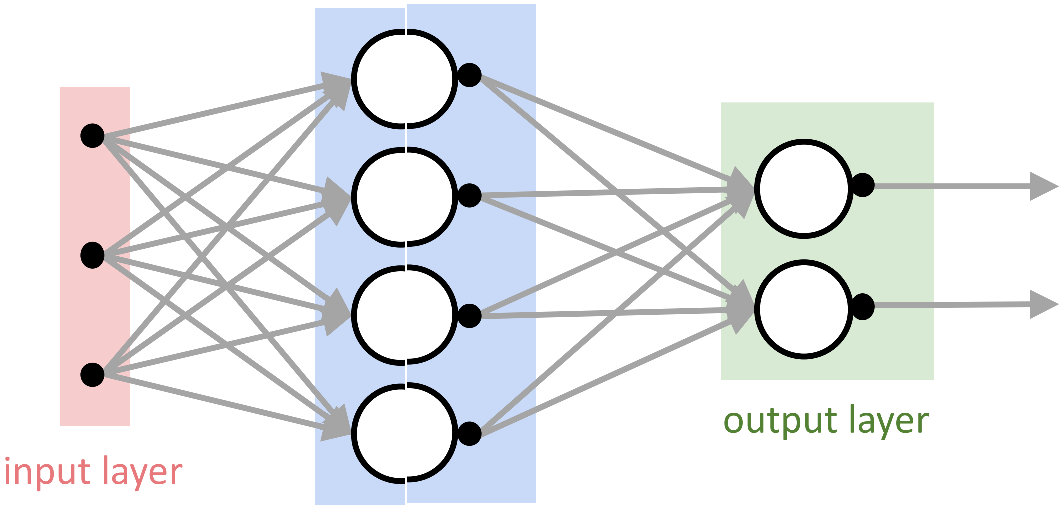
Input (1,1) ➔ σ (20 X 1 + 20 X 1− 10) ≈ 1 σ (-20 X 1 - 20 X 1 +30) ≈ 0 . σ (20 X 1 + 20 X 0 − 30) ≈ 0

Input (0,1) ➔ σ (20 X 0 + 20 X 1− 10) ≈ 1 σ (-20 X 0 - 20 X 1 +30) ≈ 1 σ (20 X 1 + 20 X 1 − 30) ≈ 1

Input (1,0) ➔ σ (20 X 1 + 20 X 0− 10) ≈ 1 σ (-20 X 1 - 20 X 0 +30) ≈ 1 σ (20 X 1 + 20 X 1 − 30) ≈ 1

Artificial Neural Networks
Hidden Layers

- A NN with one hidden layer can represent:
  - any bounded continuous function (to some arbitrary ε) [Universal Approximation Theorem, Cybenko 1989]
  - any Boolean Function, but it requires 2^k hidden units for 1K inputs



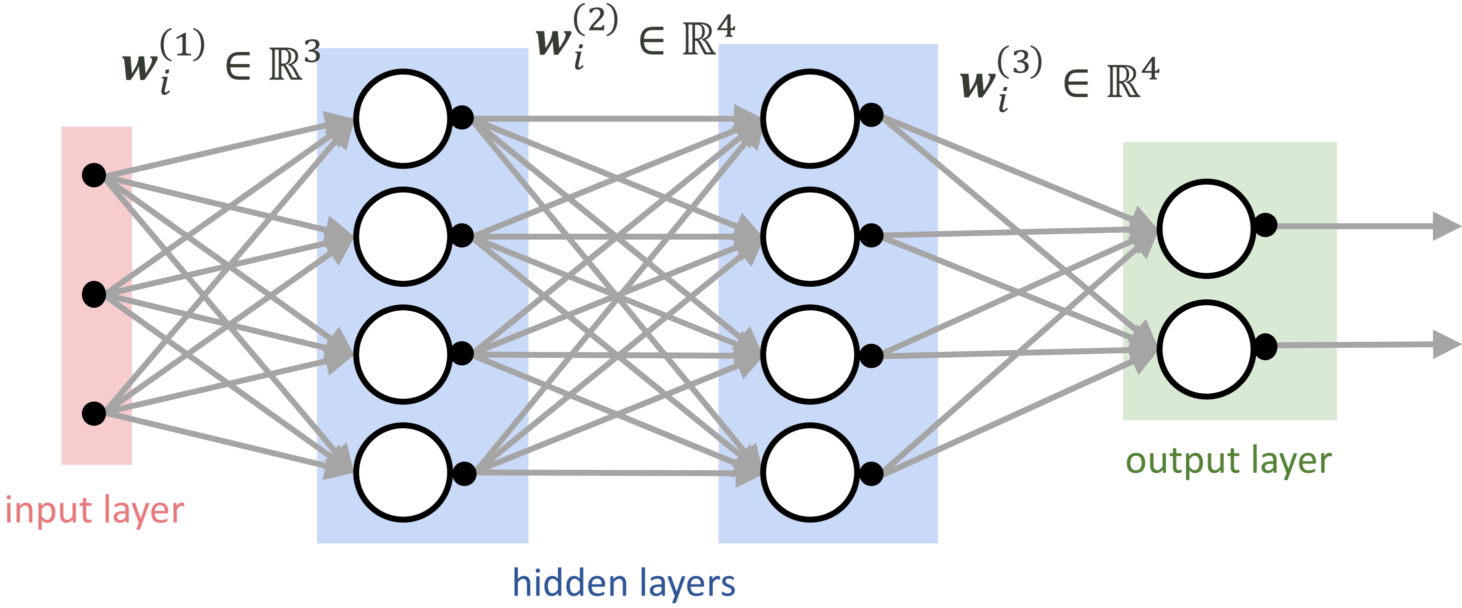
Artificial Neural Networks

Multiple Hidden Layers

Layer-wise organization: MLP consists of *fully-connected* layers in which neurons between two adjacent layers are fully pairwise connected, while neurons within a single layer share no connections.

Sizing MLP: the number of neurons or the number of parameters (more commonly)

- The network has 4 + 4 + 2=10 neurons, [3 x 4] + [4 x 4] + [4 x 2] = 12 + 16 + 8 = 36 weights and 4 + 4 + 2 = 10 biases, for a total of 46 learnable parameters.



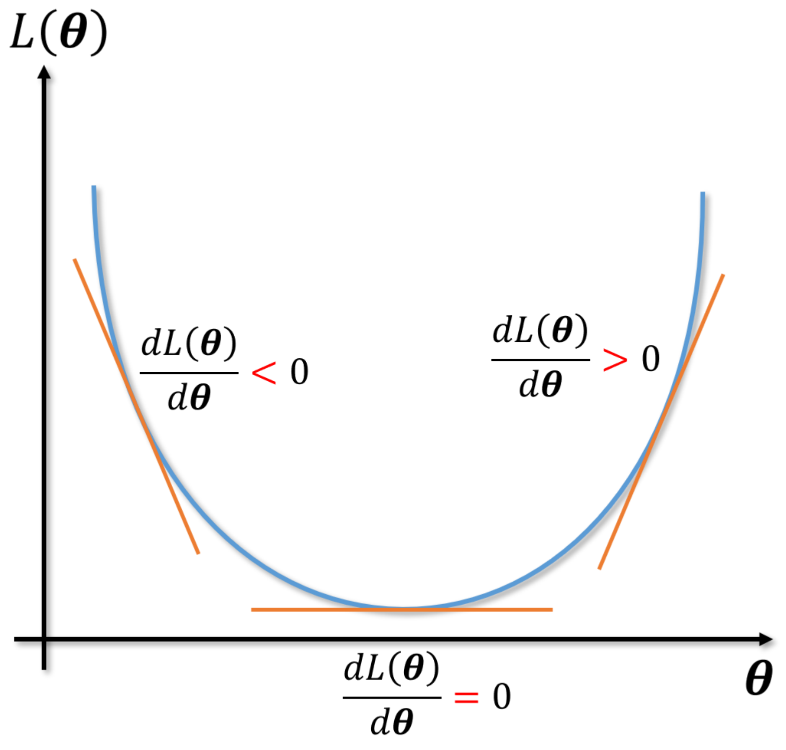
Artificial Neural Networks

Learning the Optimal Parameters

- ANN is essentially a function $f_{\boldsymbol{\theta}}: \mathcal{X} \rightarrow \mathcal{Y}$ mapping feature vectors $\boldsymbol{x}_i \in \mathcal{X}$ to predicted output vectors $\hat{\boldsymbol{y}}_i \in \mathcal{Y}$ based on a set of parameters $\boldsymbol{\theta} = \{\boldsymbol{W}, \boldsymbol{b}\}$
- Let $L(\boldsymbol{\theta})$ be the loss function that measures the difference between predictions $\hat{\boldsymbol{y}}$ and desired outputs $\boldsymbol{y}$
- We want to find the **optimum** $\boldsymbol{\theta}^*$ such that:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, L(\boldsymbol{\theta})$$

- Considering the change in the loss function with respect to $\boldsymbol{\theta}$, the loss function is minimum when $\frac{dL(\boldsymbol{\theta})}{d\boldsymbol{\theta}} = 0$.
- Finding $\boldsymbol{\theta}^*$ can be achieved by backpropagation and gradient descent

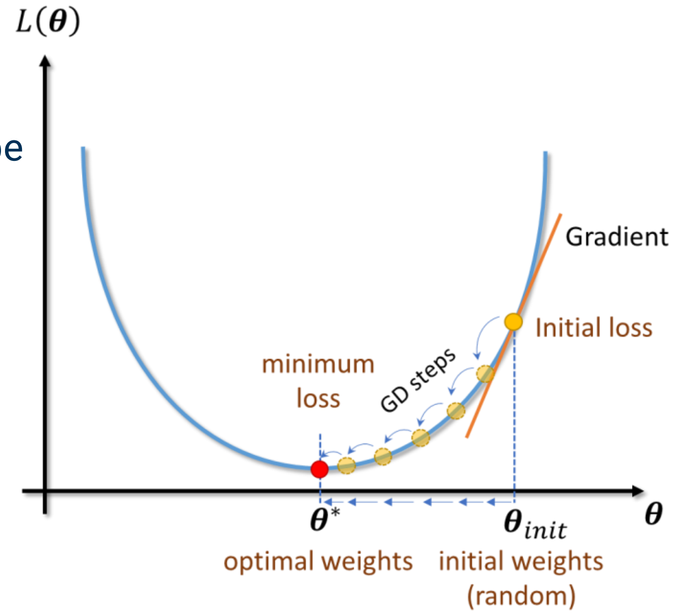


Artificial Neural Networks

Quick Review of Derivatives

Derivative of a function, $f(\boldsymbol{\theta})$:

- how fast *f* changes around $\boldsymbol{\theta}$
- will *f* increase or decrease if we increase $\boldsymbol{\theta}$
- is $\boldsymbol{\theta}$ higher or lower than it should be ($\boldsymbol{\theta}^*$); shall we make it bigger or smaller to be where it should be



$\frac{\partial}{\partial\theta} f(g(\theta)) = \frac{\partial}{\partial g} f(g(\theta)) \cdot \frac{\partial}{\partial\theta} g(\theta)$ chain rule

$\frac{\partial}{\partial\theta} \sum_i f_i(\theta) = \sum_i \frac{\partial}{\partial\theta} f_i(\theta)$ derivative of the sum is the sum of the derivatives

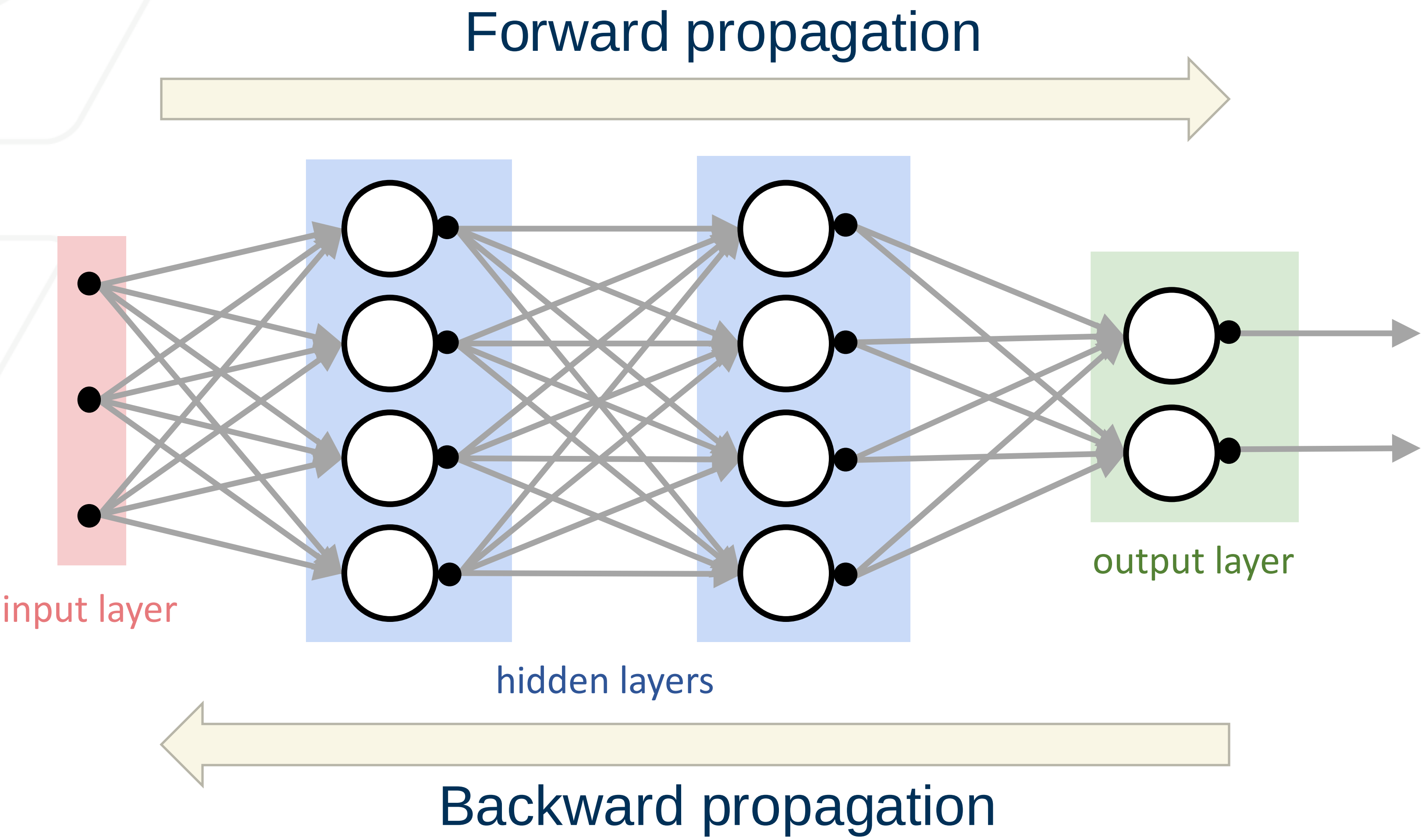
$\frac{\partial}{\partial\theta_k} \sum_i a_i f(\theta_i) = a_k \frac{\partial}{\partial\theta_k} f(\theta_k)$ derivative wrt one element of the sum collapses the sum

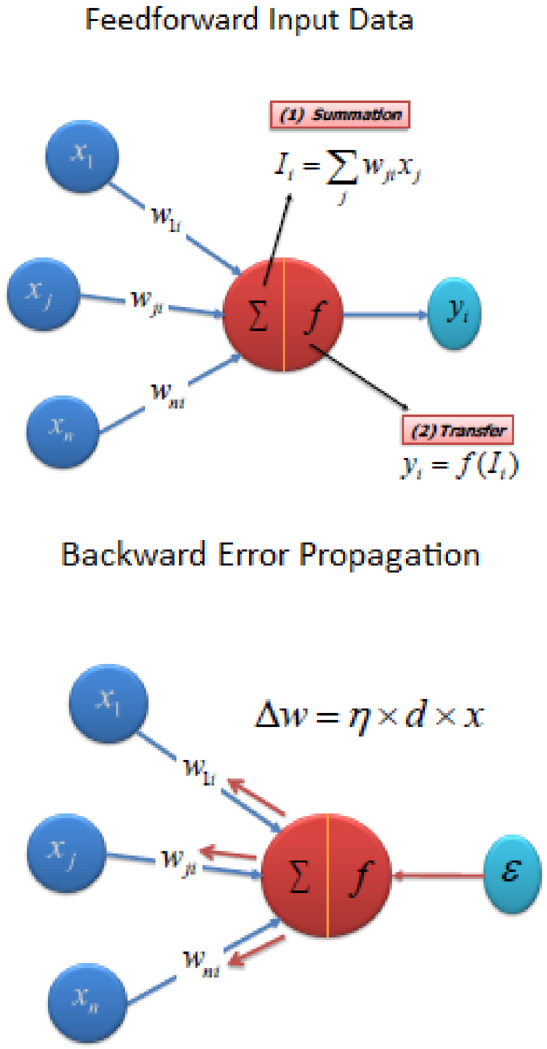
The sigmoid has a special property:

$$\sigma'(x) = \frac{\partial}{\partial x}(1+e^{-x})^{-1} = -(1+e^{-x})^{-2}(-e^{-x}) = \frac{1}{1+e^{-x}} \frac{e^{-x}}{1+e^{-x}} = \sigma(x)\big(1-\sigma(x)\big)$$

$$\frac{\partial}{\partial x}\sigma(f_x) = \sigma(f_x)\big(1-\sigma(f_x)\big)\frac{\partial}{\partial x} f_x$$

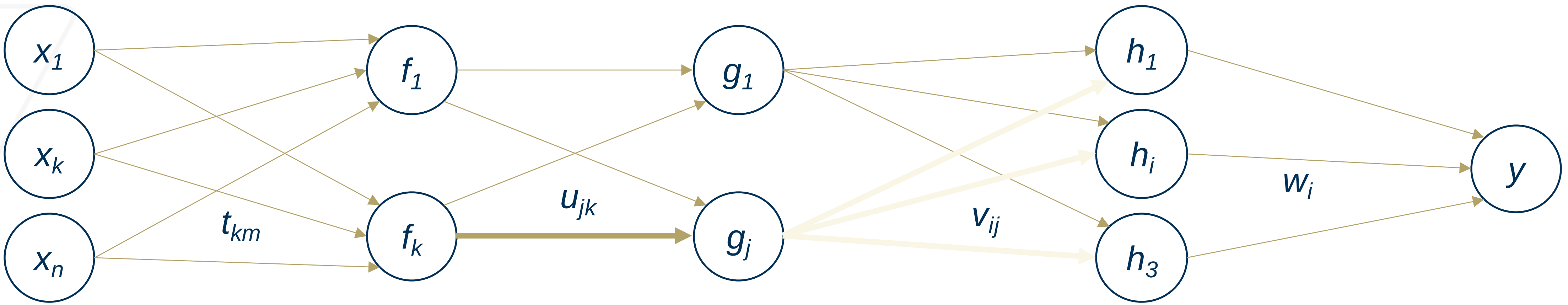
Backpropagation
Terminologies





Artificial Neural Networks

Backpropagation

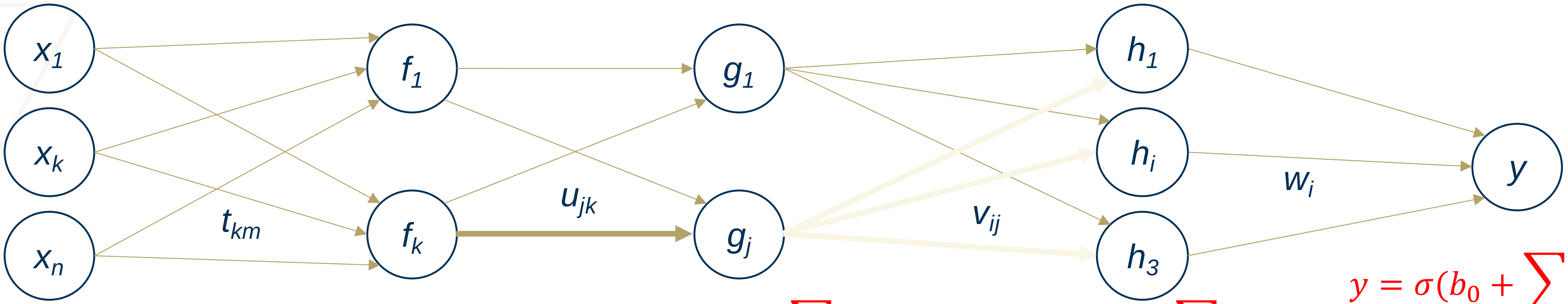


1. For a new sample $\boldsymbol{x} = [x_1, \dots x_n]$
2. Feed forward: compute g_j based on units f_k *from the previous layer using* $g_j = \sigma(b_{j0} + \sum_k u_{jk} f_k)$
3. Predict *y as y**
4. Backpropagate error: $e = y - y^*$
   1. Determine whether we need g_j to be lower or higher (assuming we already determined the better values for *h* from a previous step) by $\frac{\partial e}{\partial g_j} = \sum_i \sigma'(h_i) v_{ij} \frac{\partial e}{\partial h_i}$, or in other words by determining how h_i will change as g_j changes AND by inspecting if h_i was too high or too low
   2. Update weights: update the weight u_{jk} that feeds g_j by $\frac{\partial e}{\partial u_{jk}} = \frac{\partial e}{\partial g_j} \sigma'(g_j) f_k$, or in other words by determining if we want g_j to be higher or lower AND by determining how g_j will change if u_{jk} becomes higher or lower; now update the weight with a learning rate multiplier of $\frac{\partial e}{\partial u_{jk}}$

Think of $\sigma'(h_i)v_{ij}$ as a scalar; if h is around 0, then changing g will impact h a lot; if h is close to 1 or 0, then this expression has no impact

Artificial Neural Networks

Backpropagation



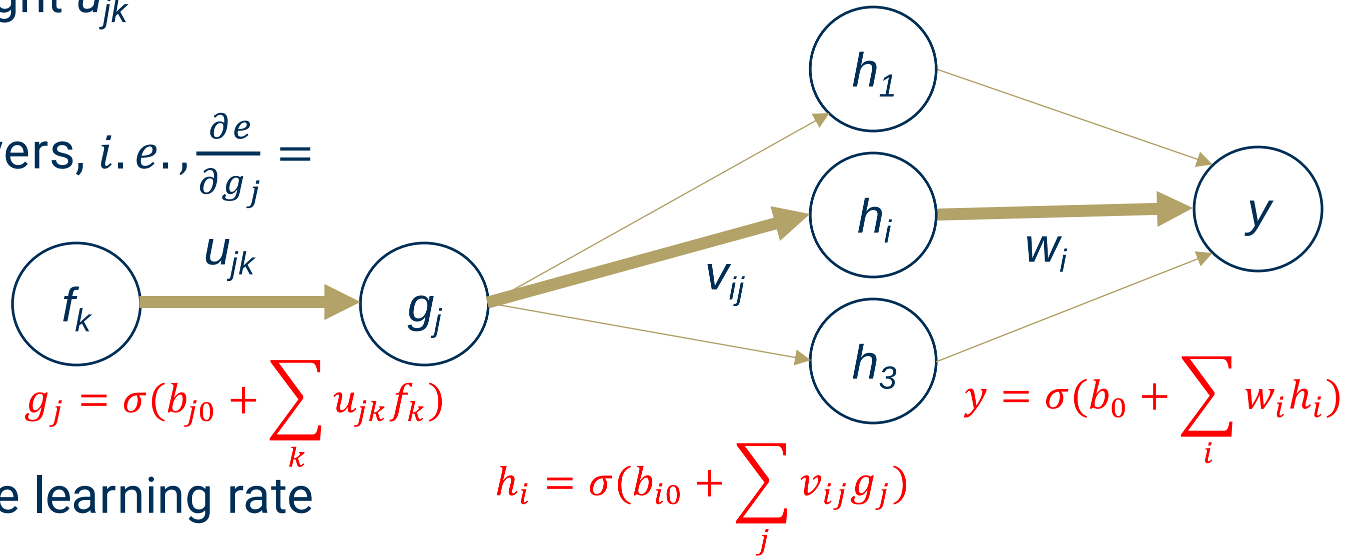
$y = \sigma(b_0 + \sum_i w_i h_i)$

$f_k = \sigma(b_{k0} + \sum_m t_{km} x_m)$ $\quad g_j = \sigma(b_{j0} + \sum_k u_{jk} f_k)$ $\quad h_i = \sigma(b_{i0} + \sum_j v_{ij} g_j)$

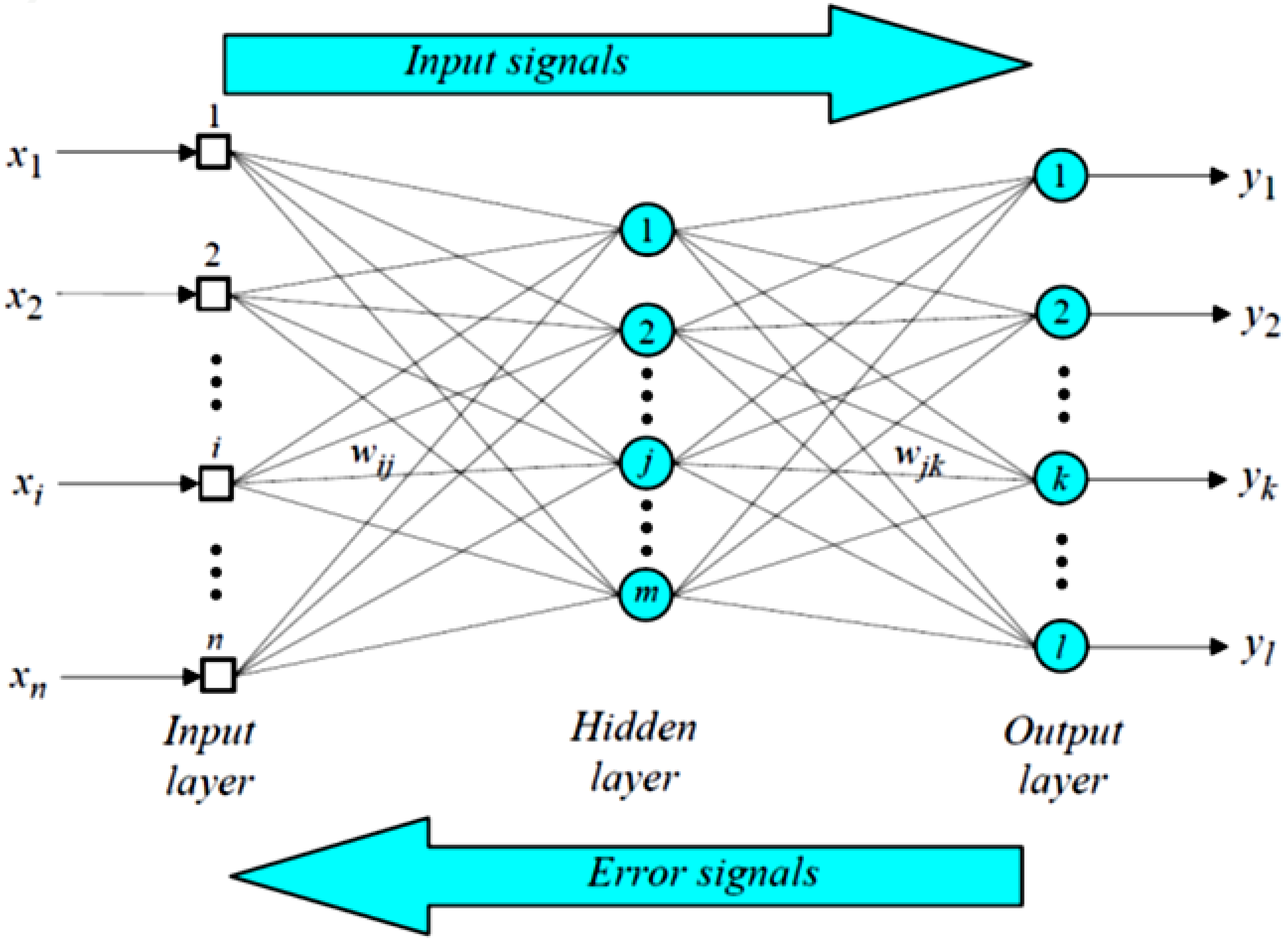
- $e = \frac{1}{2}(y - y^*)^2$
- $\frac{\partial e}{\partial h_i} = (y - y^*)\frac{\partial y}{\partial h_i} = (y - y^*)\, y(1 - y) w_i$; the sum goes away because we are differentiating wrt to one of the elements
- $\frac{\partial e}{\partial g_j} = (y - y^*)\frac{\partial y}{\partial g_j} = (y - y^*)\, y(1 - y) \sum_i w_i \frac{\partial h_i}{\partial g_j} = (y - y^*)\, y(1 - y) \sum_i w_i h_i (1 - h_i) v_{ij}$
- By observing the nesting pattern, we can write $\frac{\partial e}{\partial g_j} = \sum_i h_i(1 - h_i) v_{ij} \frac{\partial e}{\partial h_i}$ and so on
- $\frac{\partial e}{\partial u} = (y - y^*)\frac{\partial y}{\partial u} = (y - y^*)\, y(1 - y) \sum_i w_i h_i (1 - h_i) \sum_j \frac{\partial g_j}{\partial u} = (y - y^*)\, y(1 - y) \sum_i w_i h_i (1 - h_i) v_{ij}\, g_j(1 - g_j) f_k = g_j(1 - g_j) f_k \frac{\partial e}{\partial g_j}$

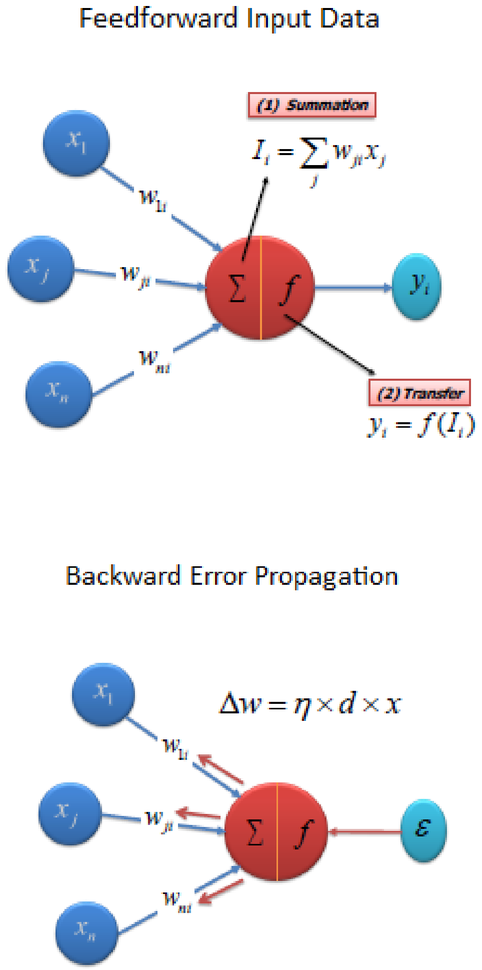
Backpropagation Summary

- **Feed forward propagate** the sample $\boldsymbol{x}$, compute activation outputs of neurons from previous layers and compute the predicted output y
- Compute the **backpropagation error**: $e = L(y, y^*)$, where L is the loss function
- Assume using sigmoid activation σ, for a weight u_{jk} compute $\frac{\partial e}{\partial u_{jk}}$ using **chain rule:**
  - Compute $\frac{\partial e}{\partial g_j}$ by carrying $\frac{\partial e}{\partial h_i}$ from later layers, $i.e.$, $\frac{\partial e}{\partial g_j} = \sum_i h_i(1 - h_i)v_{ij}\frac{\partial e}{\partial h_i}$
  - Compute $\frac{\partial e}{\partial u_{jk}} = g_j(1 - g_j)f_k\frac{\partial e}{\partial g_j}$
- Update weights using **gradient descent**
  - $u_{jk}(t+1) = u_{jk}(t) - \alpha\frac{\partial e}{\partial u_{jk}}$, where α is the learning rate
- Iterate the above steps until the error e converges



Artificial Neural Networks
Backpropagation Summary



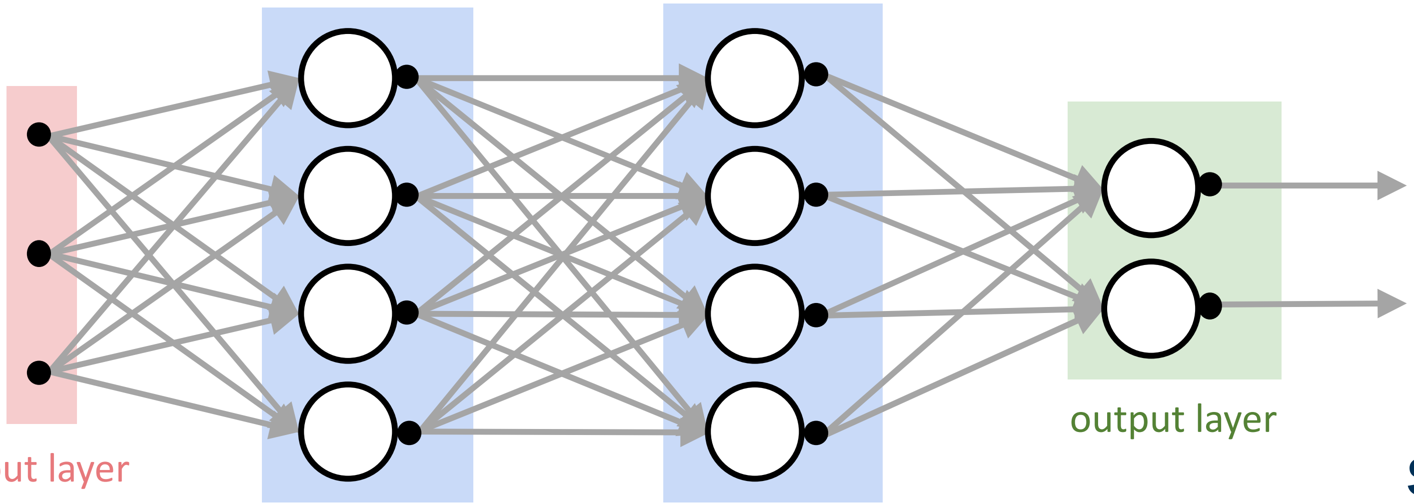


Artificial Neural Networks

Backpropagation Summary

During the inference, predict the class with the highest probability with SoftMax activation: $\mathbf{argmax}_j\ f(y_{ij})$

Sample x_i



Assume the raw outputs $y_\mathrm{i} = [y_{i1}, y_{i2}, y_{i3}]$
$= [3.2, 5.1, -1.7]$

SoftMax activation:

$$f(y_{ij}) = \frac{e^{y_{ij}}}{\sum_k e^{y_{ik}}}$$

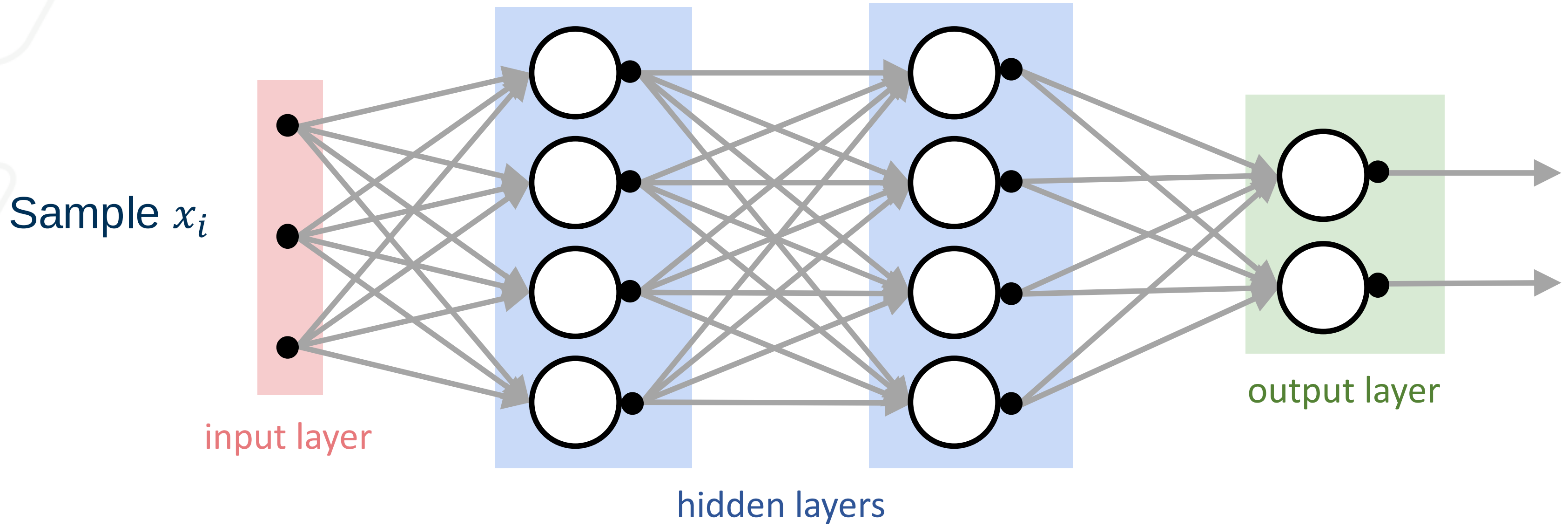
Raw outputs		unnormalized probabilities		probabilities
y_{i1}= 3.2	exp $\Rightarrow$	24.5	normalize $\Rightarrow$	$f(y_{i1}) = 0.13$
y_{i2}= 5.1		164.0		$\boxed{f(y_{i2}) = 0.87}$
y_{i3}= -1.7		0.18		$f(y_{i3}) = 0.00$

predict class 2 with highest probability

Artificial Neural Networks

Backpropagation Summary

During training, for every sample, we set the ground-truth label as a one-hot vector $[1, 0, \ldots, 0]^T$ with 1 for the correct class and 0 for every other class. Backpropagate the error and repeat for every sample.



Assume the raw outputs $y_i = [y_{i1}, y_{i2}, y_{i3}] = [3.2, 5.1, -1.7]$

Assume **ground-truth class is class 1**

Raw outputs		unnormalized probabilities		probabilities		One-hot ground-truth label
y_{i1} = 3.2	exp ⇨	24.5	normalize ⇨	$f(y_{i1}) = 0.13$	Loss function ⇨ to obtain error ⇦	**1.00**
y_{i2} = 5.1		164.0		$\boxed{f(y_{i2}) = 0.87}$		0.00
y_{i3} = -1.7		0.18		$f(y_{i3}) = 0.00$		0.00

Image Classification: mapping the image pixels to probabilities for each category

For simplicity, we take a linear function:

$$\widehat{\boldsymbol{Y}} = \boldsymbol{\phi}(\boldsymbol{X}\boldsymbol{W}^T + \boldsymbol{b}^T)$$

where

- $\boldsymbol{X} \in \mathbb{R}^{N \times P}$: dataset containing N vectorized images
- $P \in \mathbb{R}^{H \times W \times C}$: the number of pixels (*features*) of each image
- $\widehat{\boldsymbol{Y}} \in \mathbb{R}^{N \times P^{(k)}}$: the associated probabilities of each category $1,2,\dots,P^{(k)}$
- $\boldsymbol{\phi}(\boldsymbol{X}, \boldsymbol{W}, \boldsymbol{b})$: the activation function
- $\boldsymbol{W} \in \mathbb{R}^{P \times P^{(k)}}$: the weight matrix
- $\boldsymbol{b} \in \mathbb{R}^{P^{(k)} \times 1}$: the bias vector

Single image binary classification (predicting dog/cat)

stretch pixels into single column vector

Input 32x32 RGB image

$$\phi\left(\begin{bmatrix} 0.2 & 2.1 \\ -0.5 & 0.0 \\ 0.1 & 0.25 \\ 2.0 & 0.2 \\ 1.5 & -0.3 \\ \vdots & \vdots \\ 1.3 & 1.2 \end{bmatrix}^{T} \begin{bmatrix} 56 \\ 231 \\ 24 \\ 188 \\ 75 \\ \vdots \\ 32 \end{bmatrix} + \begin{bmatrix} 0.2 \\ 2.4 \end{bmatrix}\right) = \begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix} \begin{matrix} \text{Cat score} \\ \text{Dog score} \end{matrix}$$

$w^T \in \mathbb{R}^{2\times(32)(32)(3)}$ $x_i \in \mathbb{R}^{(32)(32)(3)\times 1}$ $b \in \mathbb{R}^{2\times 1}$ $y_i \in \mathbb{R}^{2\times 1}$

Example Datasets:

- MNIST (hand-written digits)
    - # total: 70,000 grayscale images
    - # classes: 10
    - # size: 28x28
    - # training samples: 60,000 images
    - # test samples: 10,000 images

https://www.researchgate.net/figure/Sample-images-of-MNIST-data_fig3_222834590

- CIFAR-10 (subsets of the 80 million tiny images)
    - # total: 60,000 color images
    - # classes: 10
    - # size: 32x32
    - # training samples: 50,000 images
    - # test samples: 10,000 images

https://www.kaggle.com/c/cifar-10

Appendix A: Notations

- x_i: a single feature
- $\boldsymbol{x}_i$: feature vector (data sample)
- $\boldsymbol{X}$: matrix of feature vectors (dataset)
- N: number of data samples
- m: degree of polynomial
- P: number of features in a feature vector
- θ_i: a single model coefficient (parameter)
- $\boldsymbol{\theta}$: coefficient vector
- ε: error margin
- α: learning rate
- γ: bias factor
- Bold letter/symbol: vector
- Bold capital letters/symbol: matrix