# Artificial Neural Networks

Artificial Intelligence for Digital Health (AID)
M.Sc. in Digital Health – University of Pisa
Davide Bacciu (davide.bacciu@unipi.it)

# Lecture Outline

- Modeling the artificial neuron
- Artificial neural networks and the multilayer perceptron
  - Layered structure
  - Activation functions
  - Outputs and losses
- Training Artificial neural networks
  - Backpropagation algorithm
  - Loss optimization
  - Some basic tricks

# Why Using Neural Networks?

- They are universal function approximators
- Non-parametric
- Scalable
- Heavily supported at SW and HW
- Fast solutions by Lego bricking

# Neural Networks in Healthcare

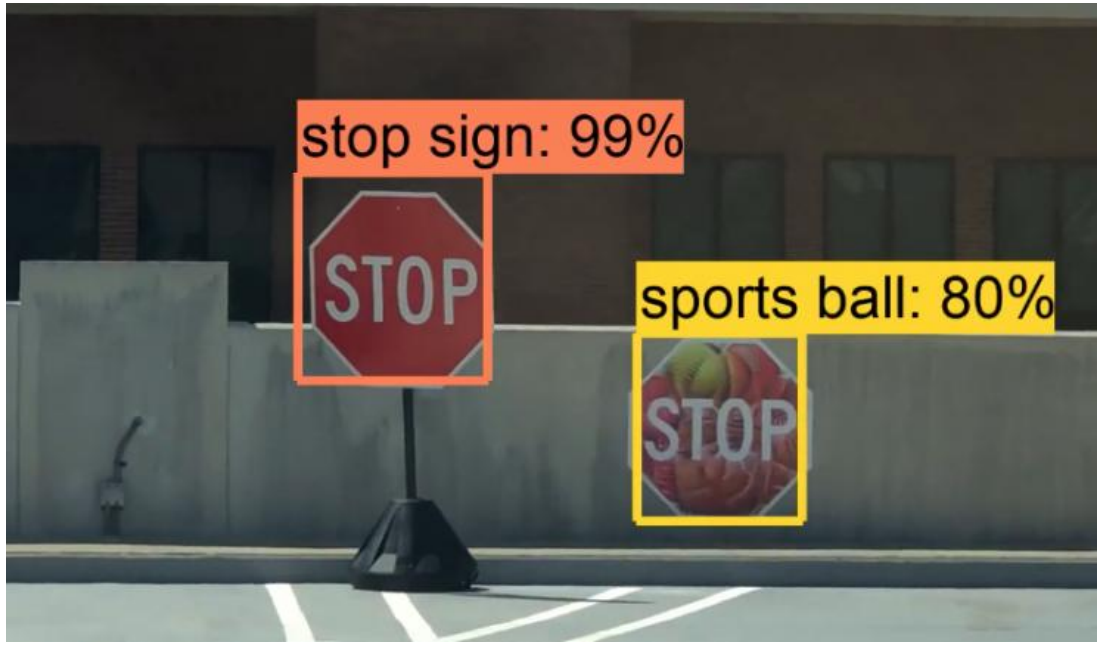
Learn complex patterns from noisy and highly diverse data

Handle vast amounts of data in a scalable way, possibly in real time

Provide highly accurate predictions

# When to be careful in using them?

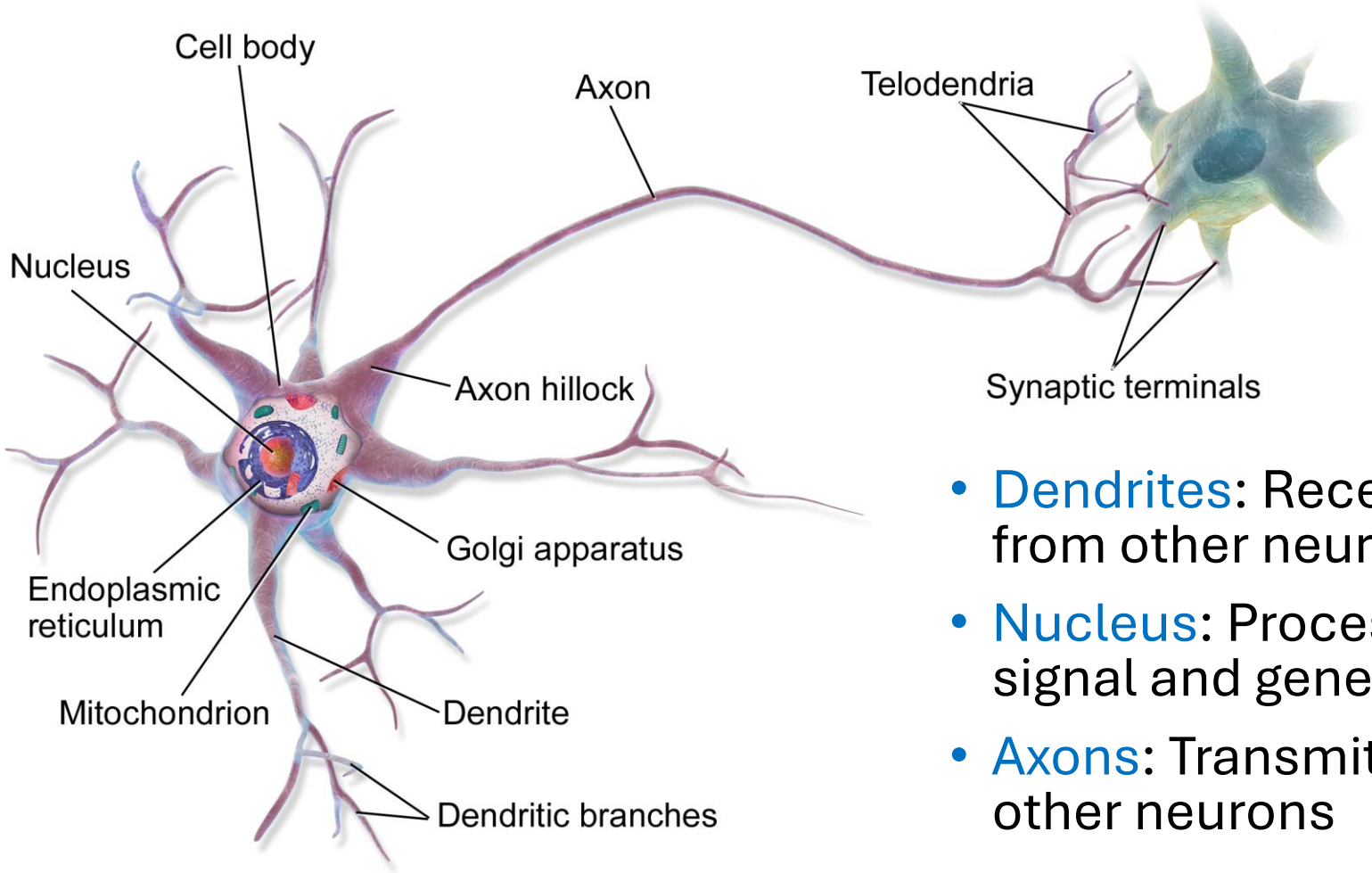
- Stringent interpretability and safety/security requirements
- Need to incorporate background/prior knowledge available
- Little-data (as opposed to Big-Data)




Essentially in our biomedical setting!

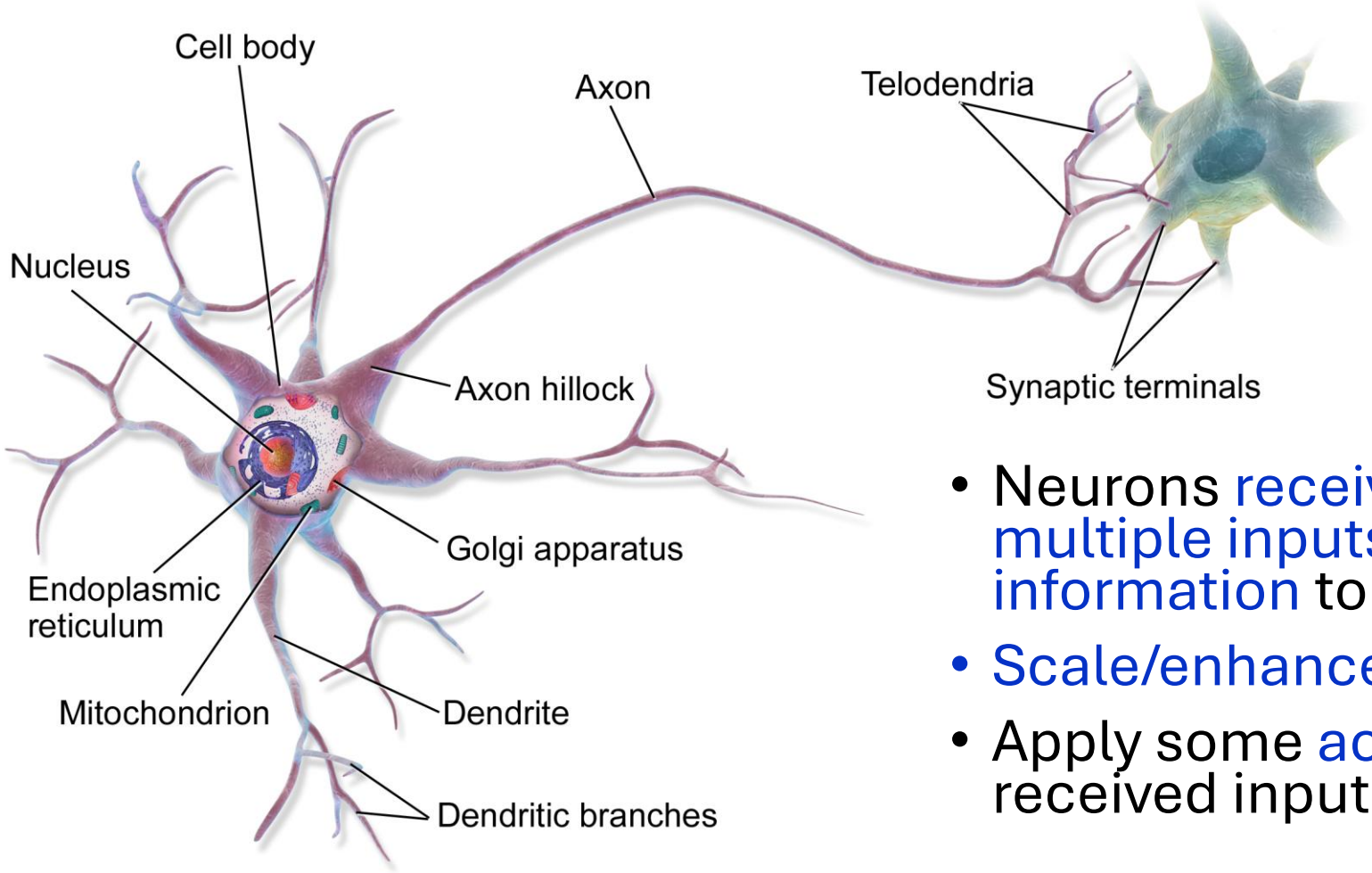
# Modeling the artificial neuron

# The Biological Neuron




- Dendrites: Receive electrical signals from other neurons
- Nucleus: Processes the electrical signal and generates an output signal
- Axons: Transmit the output signal to other neurons

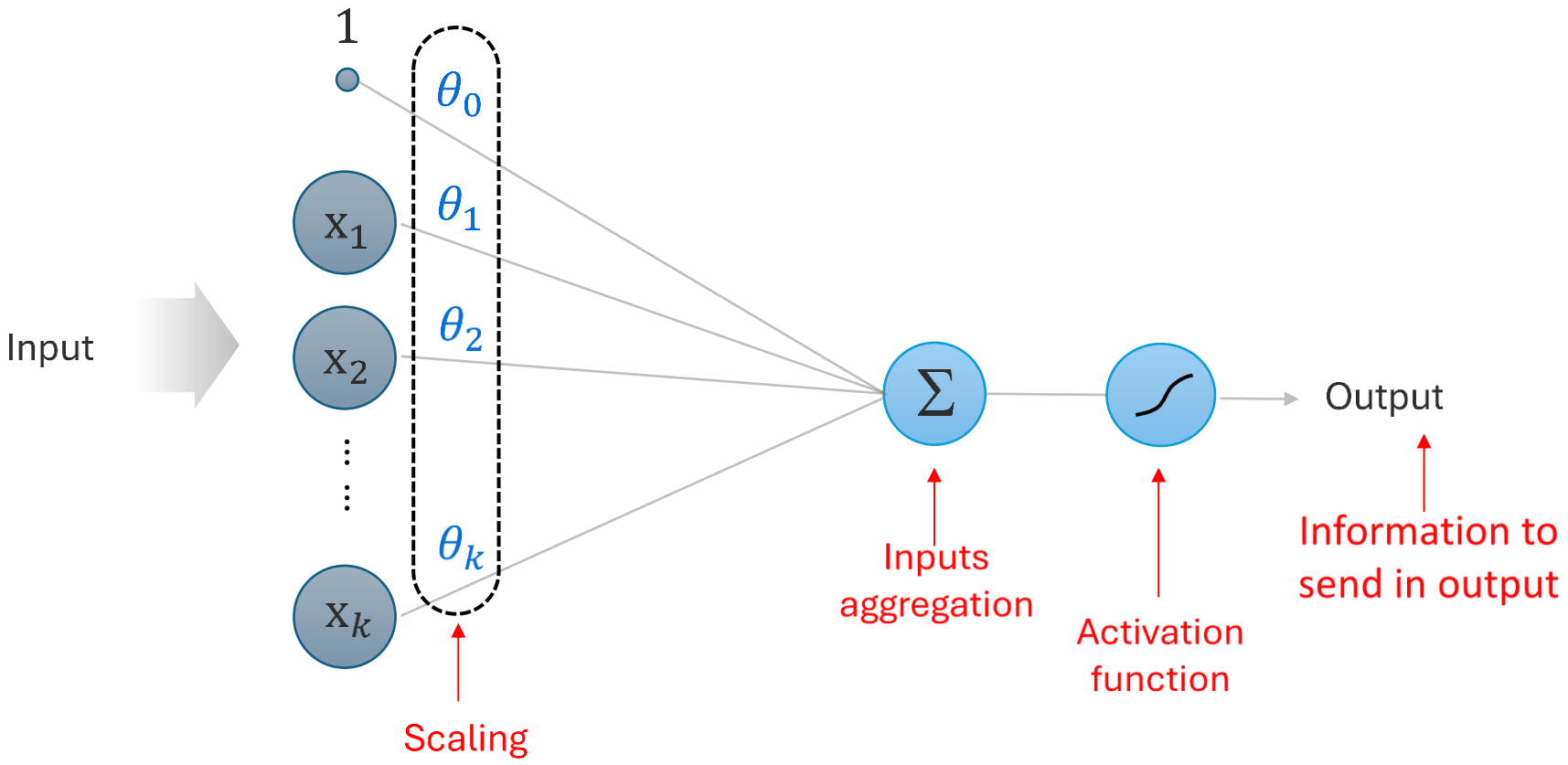
# The Neuron Metaphor



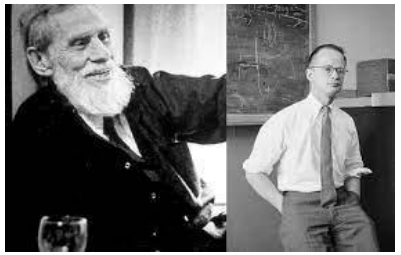

Looks like something we have already seen!

- Neurons receive information from multiple inputs and transmit information to other neurons
- Scale/enhance inputs
- Apply some activation function to received input information

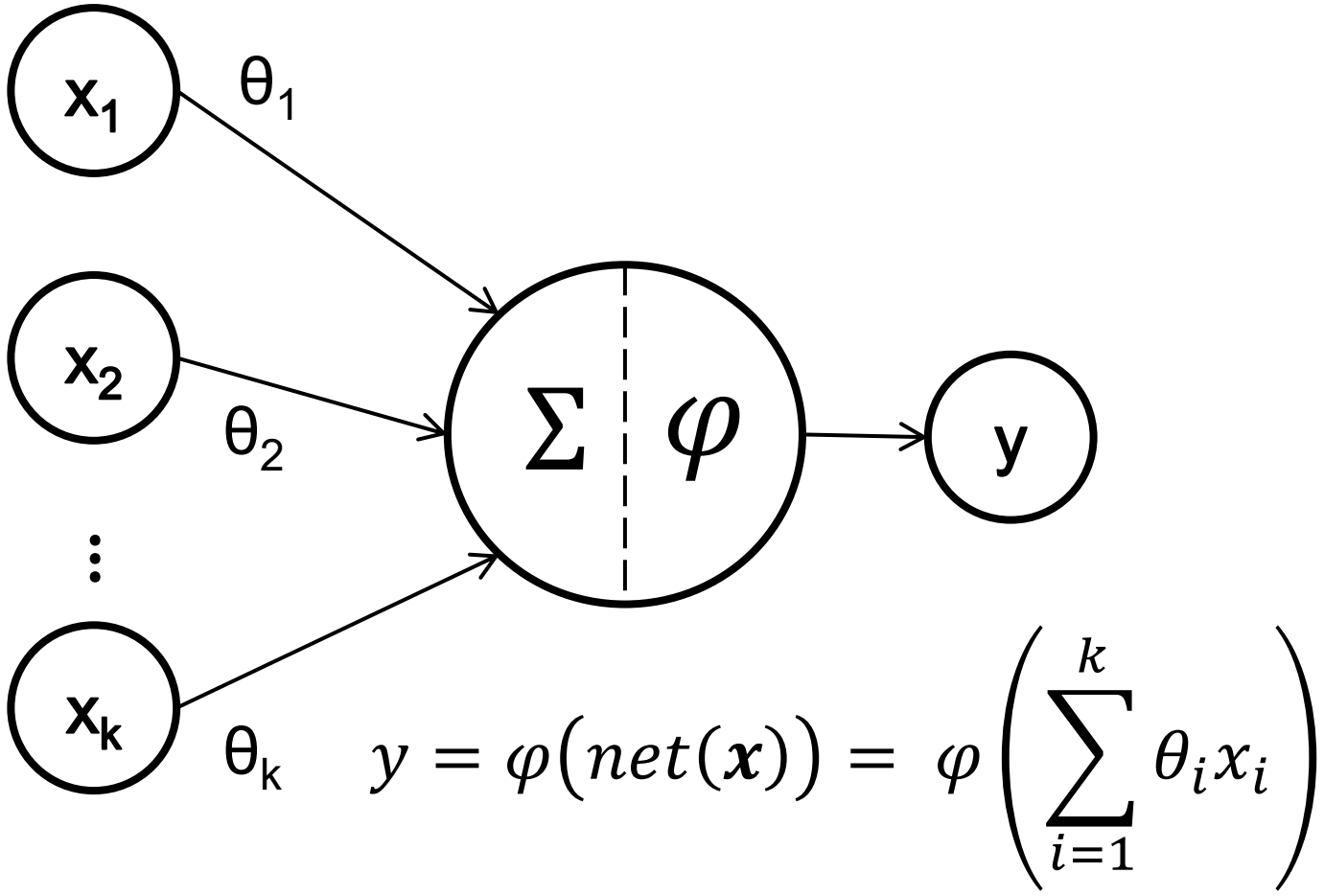
# The Return of the Logistic Regression



# The Artificial Neuron



McCulloch & Pitts, 1943



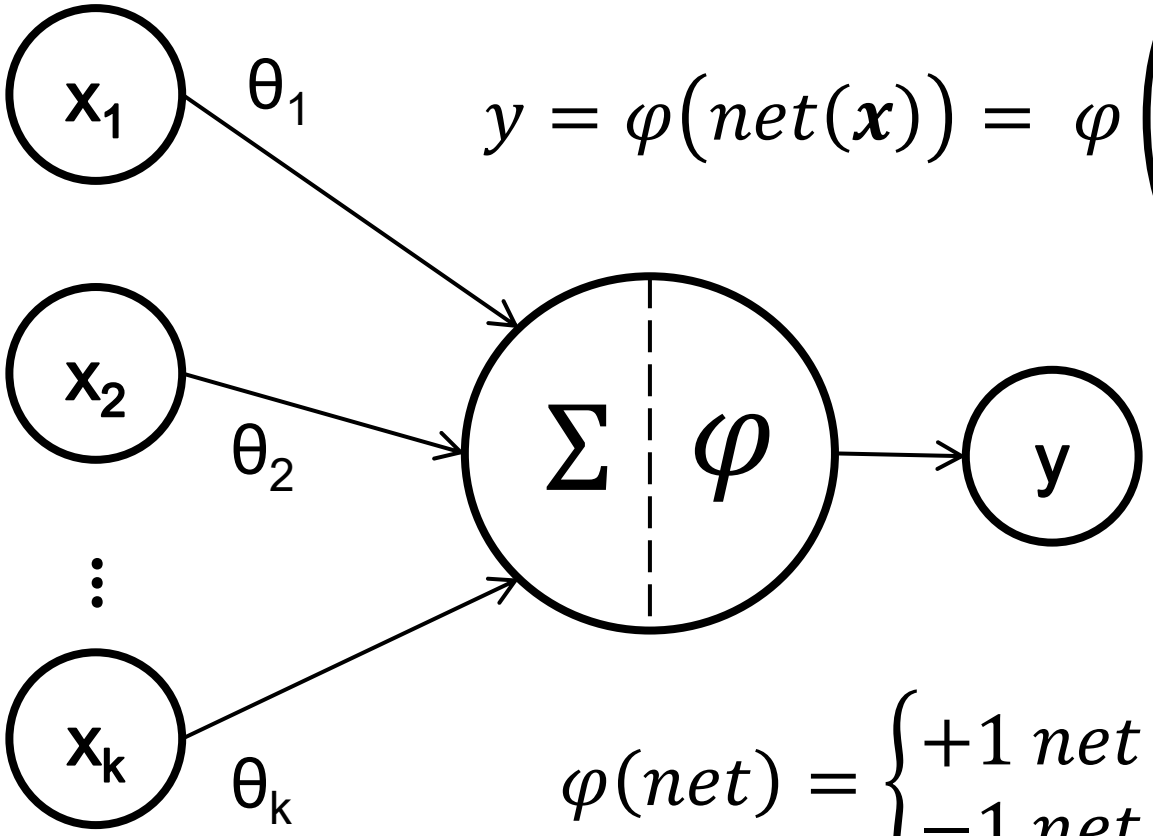

$$y = \varphi\big(net(\boldsymbol{x})\big) = \varphi\left(\sum_{i=1}^{k} \theta_i x_i\right)$$

- Input $\boldsymbol{x}$
- Synaptic weights $\boldsymbol{\theta}$
- Local potential

$$\text{net}(\mathbf{x}) = \sum_{i=1}^{k} \theta_i x_i$$

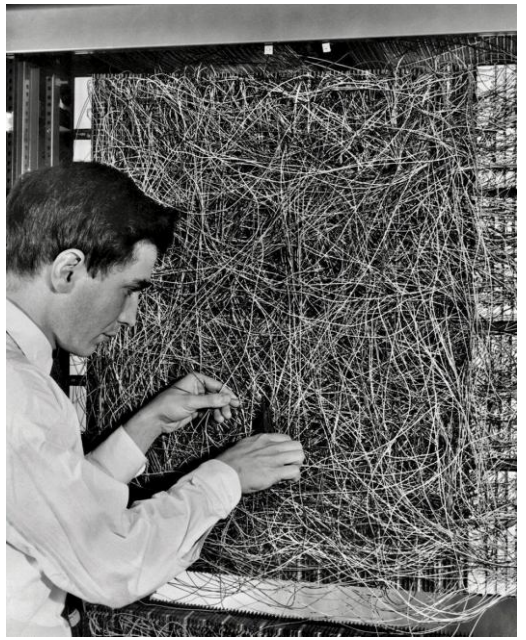
- Activation function $\varphi$
- Output $\boldsymbol{y}$

# The Perceptron




$$y = \varphi\big(net(\boldsymbol{x})\big) = \varphi\left(\sum_{i=1}^{k} \theta_i x_i\right)$$

$$\varphi(net) = \begin{cases} +1 \; net \geq 0 \\ -1 \; net < 0 \end{cases}$$



Rosenblatt, 1953

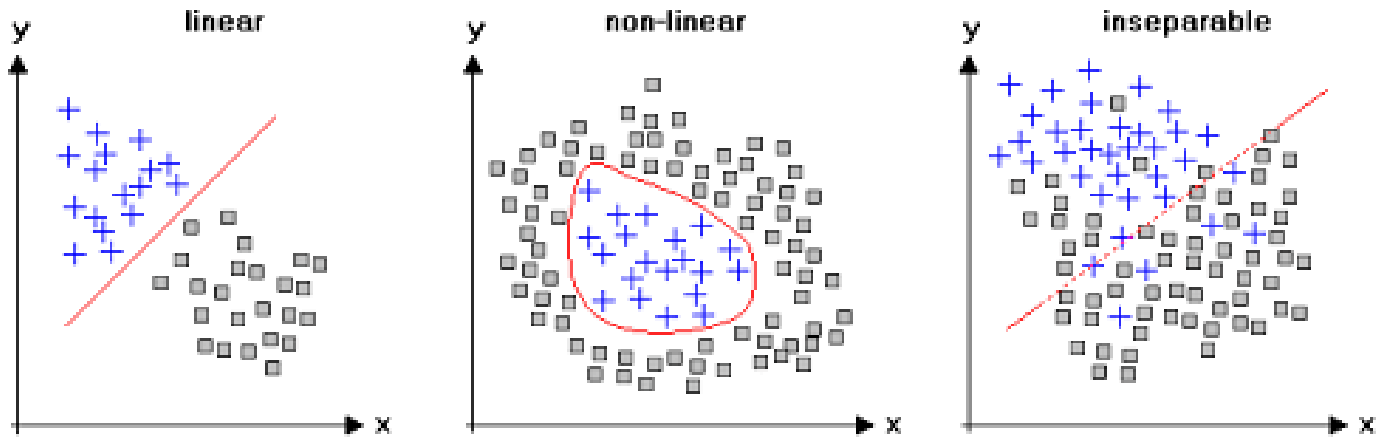
The first learnable neuron (the delta-rule)

# The Perceptron –> The Issue

Still too similar to a logistic regression

$$y = \varphi\big(net(\boldsymbol{x})\big) = \sigma\left(\sum_{i=1}^{k} \theta_i x_i\right)$$

With its limitations

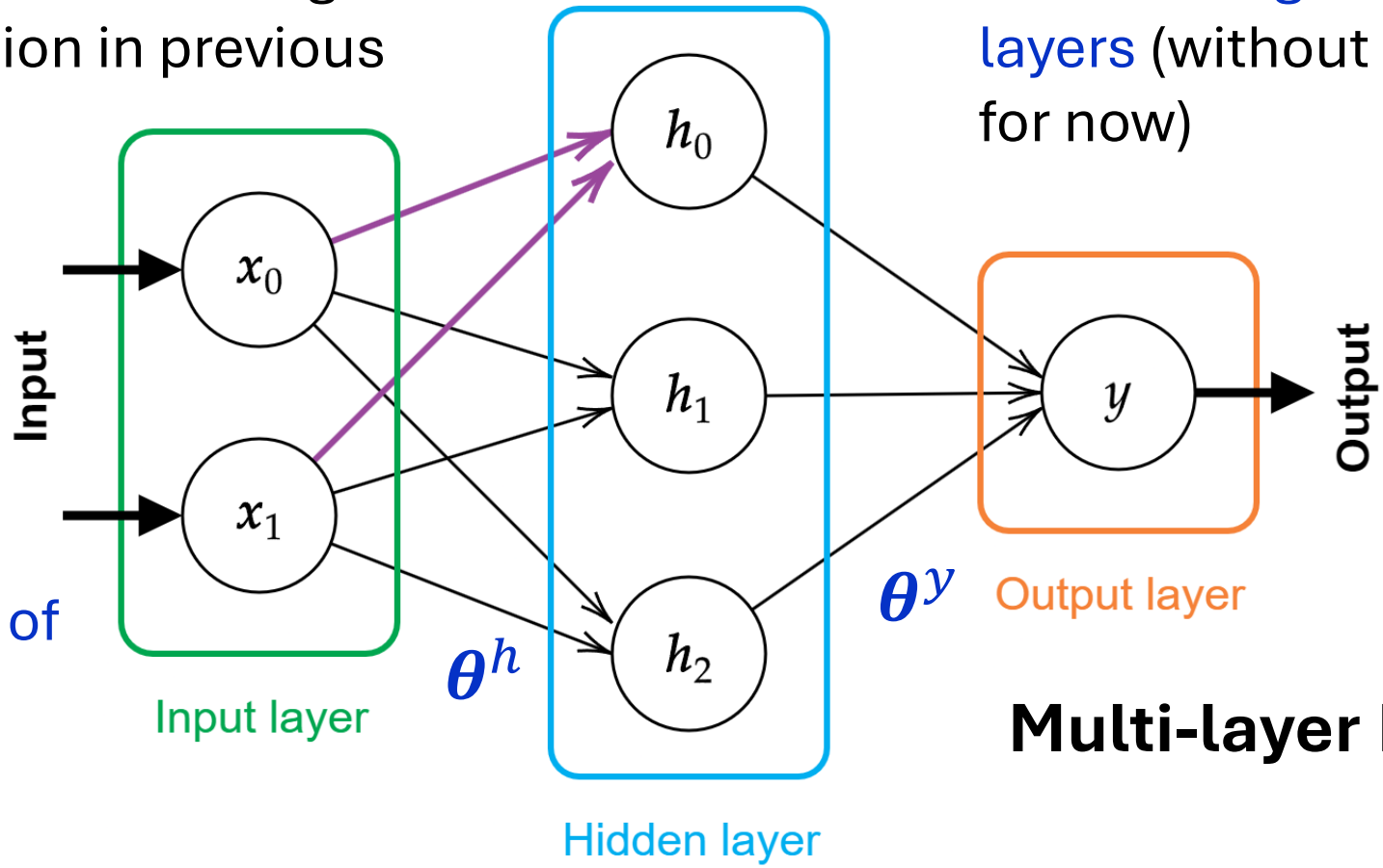


# Artificial Neural Networks

# Artificial neural networks – Interconnected neurons

Each neuron following the equation in previous slides

Neurons organized in layers (without cycles, for now)




Note the multiple set of weights

**Multi-layer Perceptron**

# Multi-Layer Perceptron (MLP)

Three main components (layers):

- Input layer: feeds the input $\boldsymbol{x}$
- Hidden layer: transforms the input $\boldsymbol{x}$ into a new (vectorial) representation $\mathrm{h}(\boldsymbol{x})$ (size of the number of hidden neurons) depending on the hidden layer parameters $\boldsymbol{\theta}^h$
- Output layer: generates the prediction $\boldsymbol{y}$ (output) by combining the intermediate representation of the input provided by the hidden layer and the output layer parameters $\boldsymbol{\theta}^y$

Two key aspects

- Can learn any decision boundary (even non-linear ones)
- Adaptable to any type of task by changing the output layer

# MLP – Input Layer

- The input layer is simply a replica of the input data
- However, preprocessing is crucial for good performance
- Feature scaling (normalization or standardization)
  - If features have different scales, normalization is needed to ensure stable training.
- Handling categorical features
  - Convert categorical variables into numerical form using one-hot encoding
  - For large categories, embedding layers can be used instead

# Categorical Features

- A categorical variable is a variable that can belong to one of k discrete categories
- Categorical variables are usually encoded using 1-out-of-k coding (one hot)
  - E.g. for three colors: red = (1 0 0), green =(0 1 0), Blue =(0 0 1)
  - I.e. the MLP above will have 3 input
- If we used red = 1, green = 2, blue = 3, then this type of encoding imposes a representational bias which is not semantically supported
- More recently dense embeddings have taken over (especially for natural language)

# Numerical Features

- A numerical variable (continuous, ordinal) can be directly fed to a neural network
- However, it is good practice to normalize data so that the dynamic range of inputs is limited
  - Min-Max Scaling - Scales values to [0,1] or [−1,1]

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}} \rightarrow x' \in [0,1] \text{ or } x' = 2\frac{x - x_{min}}{x_{max} - x_{min}} - 1 \rightarrow x' \in [-1,1]$$

  - Z-score Standardization- Centers data around mean 0 and std 1

$$x' = \frac{x - \mu}{\sigma}$$

- Need also care in choosing w.r.t. whom you are normalizing population or individual (especially in biomedical applications)
- Normalization is applied also in timeseries and graphs

# MLP – Hidden Layer

The hidden layer transforms the input $\boldsymbol{x}$ into another vector $\boldsymbol{h}$ of arbitrary size

- Size of $\boldsymbol{h}$
  - $\boldsymbol{h}$ can be smaller or larger than $\boldsymbol{x}$, depending on the complexity of the task
  - More neurons → More capacity to learn complex patterns (but risk of overfitting)
  - Fewer neurons → Simpler models, better generalization
- Helps solve the task by learning representations that make classification, regression, or other tasks easier
- The transformation from $\boldsymbol{x}$ to $\boldsymbol{h}$ is governed by trainable parameters $\boldsymbol{\theta}^h$ fitted during training

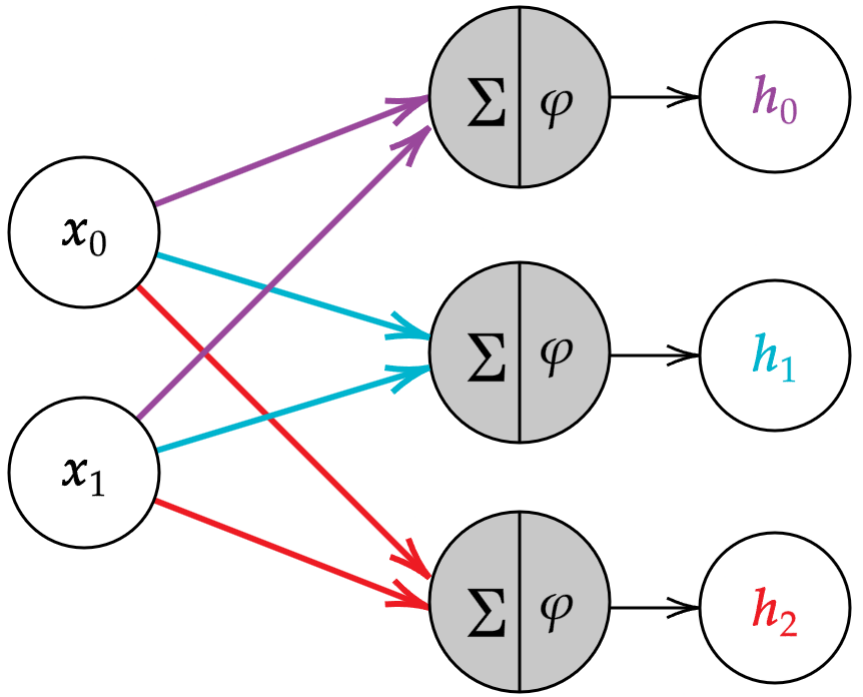
$$\boldsymbol{h} = g_{\theta^h}(\boldsymbol{x}) \text{ with } g_{\theta^h}(\cdot) \text{ nonlinear}$$

# Hidden layer step-by-step (I)

- Input: A vector $\boldsymbol{x}$ with $k = 2$ features $\rightarrow$ $\boldsymbol{x} \in \mathbb{R}^2$
- Output: A vector $\boldsymbol{h}$ with $s = 3$ features $\rightarrow \boldsymbol{h} \in \mathbb{R}^3$

$$h_0 = \varphi(\boldsymbol{\theta}_0 \boldsymbol{x}) = \varphi(\theta_{00} x_0 + \theta_{10} x_1)$$
$$h_1 = \varphi(\boldsymbol{\theta}_1 \boldsymbol{x}) = \varphi(\theta_{01} x_0 + \theta_{11} x_1)$$
$$h_2 = \varphi(\boldsymbol{\theta}_2 \boldsymbol{x}) = \varphi(\theta_{02} x_0 + \theta_{12} x_1)$$

We transformed a 2D input vector into a 3D hidden representation by a linear combination of $\boldsymbol{x}$ features + an activation function $\varphi$



# Hidden layer step-by-step (II)

- $\boldsymbol{\theta}$ is a parameter matrix with k=2 rows and s=3 columns

$$\boldsymbol{\theta} = \begin{bmatrix} 2 & 3 & -1 \\ 2 & 6 & -8 \end{bmatrix}^T$$

- $\boldsymbol{x} = [-1 \quad 1]^T$ is a sample input vector

$$h_0 = \varphi\left([2 \quad 2]\begin{bmatrix} -1 \\ 1 \end{bmatrix}\right) = \varphi(0)$$

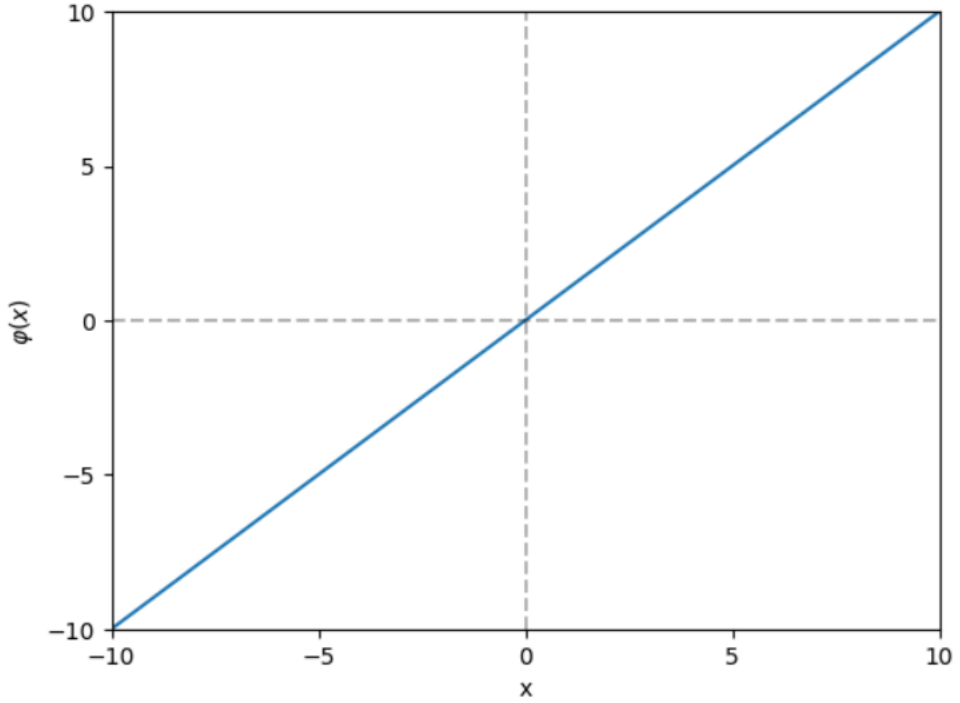
$$h_1 = \varphi\left([6 \quad 3]\begin{bmatrix} -1 \\ 1 \end{bmatrix}\right) = \varphi(-3)$$

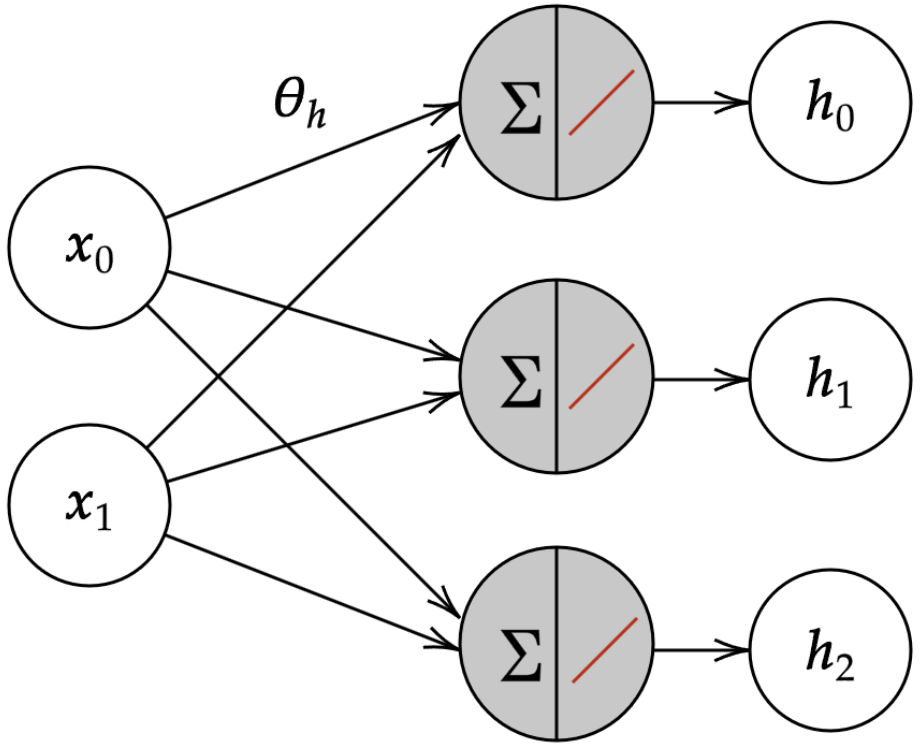
$$h_2 = \varphi\left([-8 \quad -1]\begin{bmatrix} -1 \\ 1 \end{bmatrix}\right) = \varphi(7)$$

$$\boldsymbol{h} = \varphi([0 \ -3 \ 7])$$

# Linear activation function

$$\varphi(\boldsymbol{\theta x}) = \boldsymbol{\theta x}$$

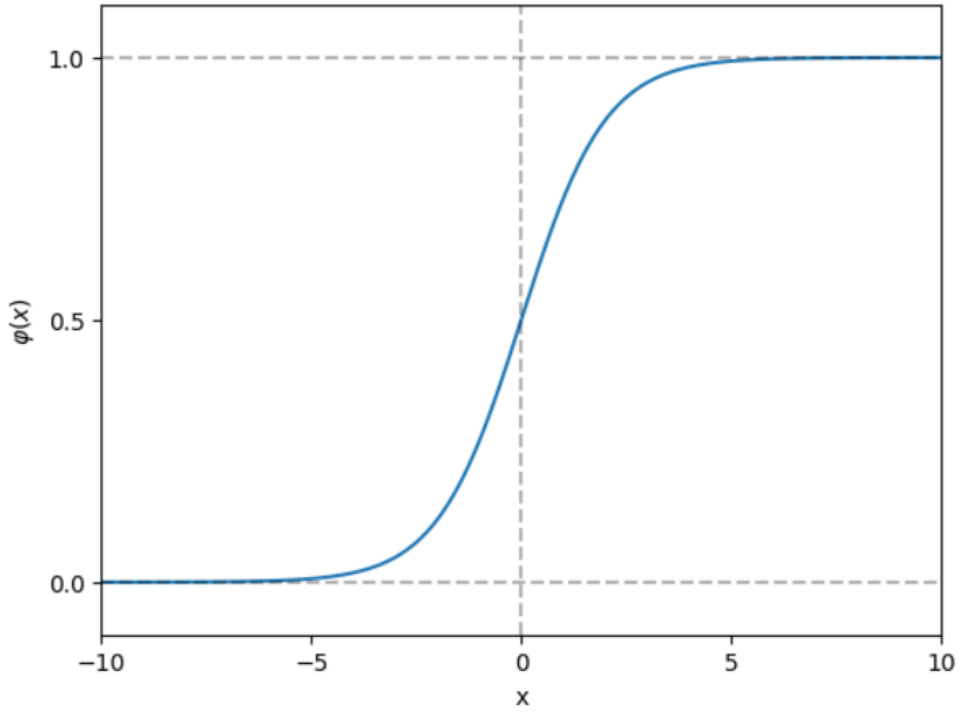


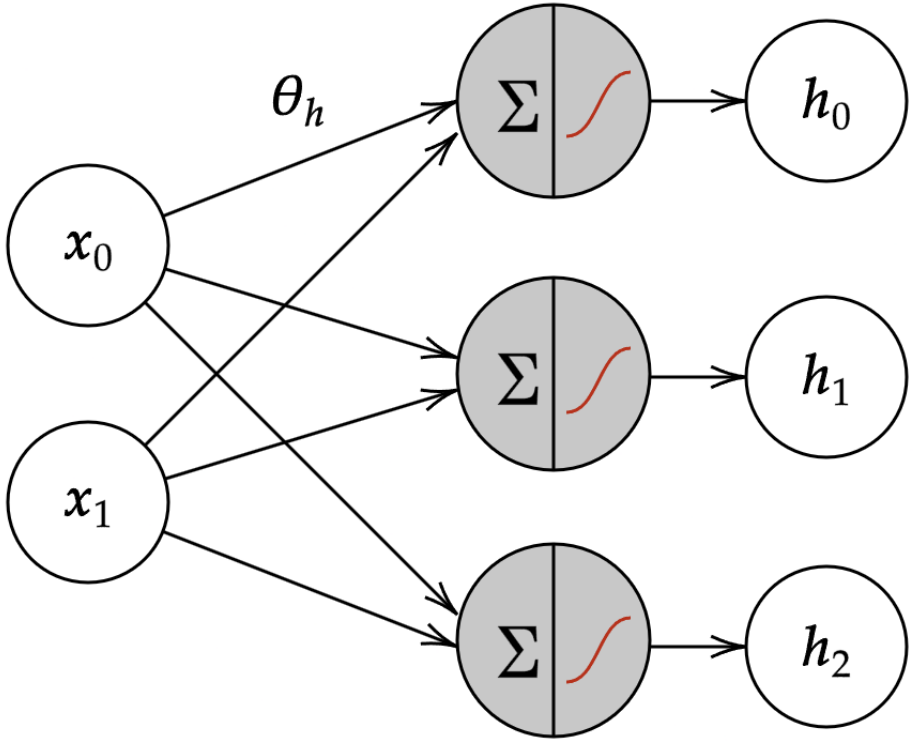




It returns the input without changing it $\boldsymbol{h} = \varphi([0\ -3\ 7]) = [0\ -3\ 7]$

# Sigmoid activation function

$$\varphi(\boldsymbol{\theta x}) = \frac{1}{1 + e^{-\boldsymbol{\theta x}}}$$

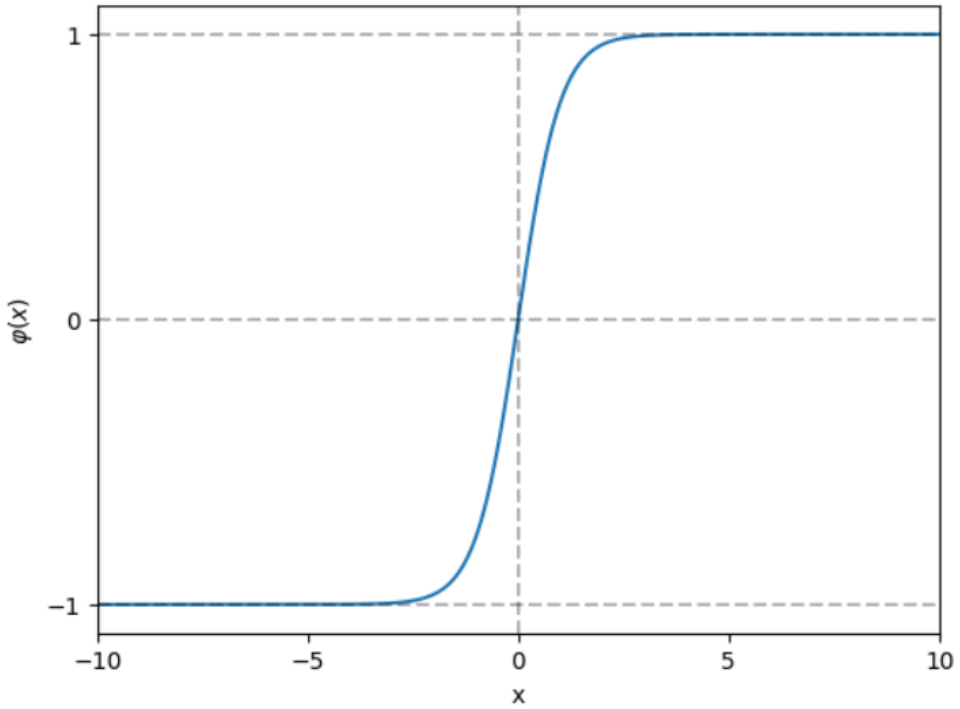


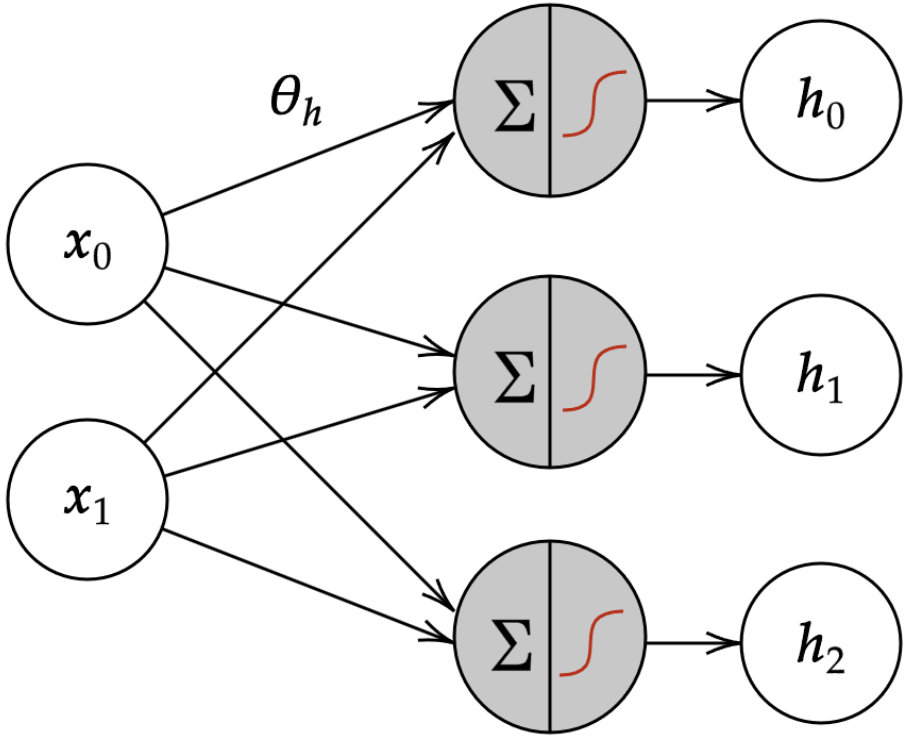




Squashes the input in [0,1] → $\varphi([0\ \ -3\ 7]) = [0.5\ 0.047\ 0.999]$

# Hyperbolic tangent activation function

$$\varphi(\boldsymbol{\theta x}) = \frac{e^{\boldsymbol{\theta x}} - e^{-\boldsymbol{\theta x}}}{e^{\boldsymbol{\theta x}} + e^{-\boldsymbol{\theta x}}}$$

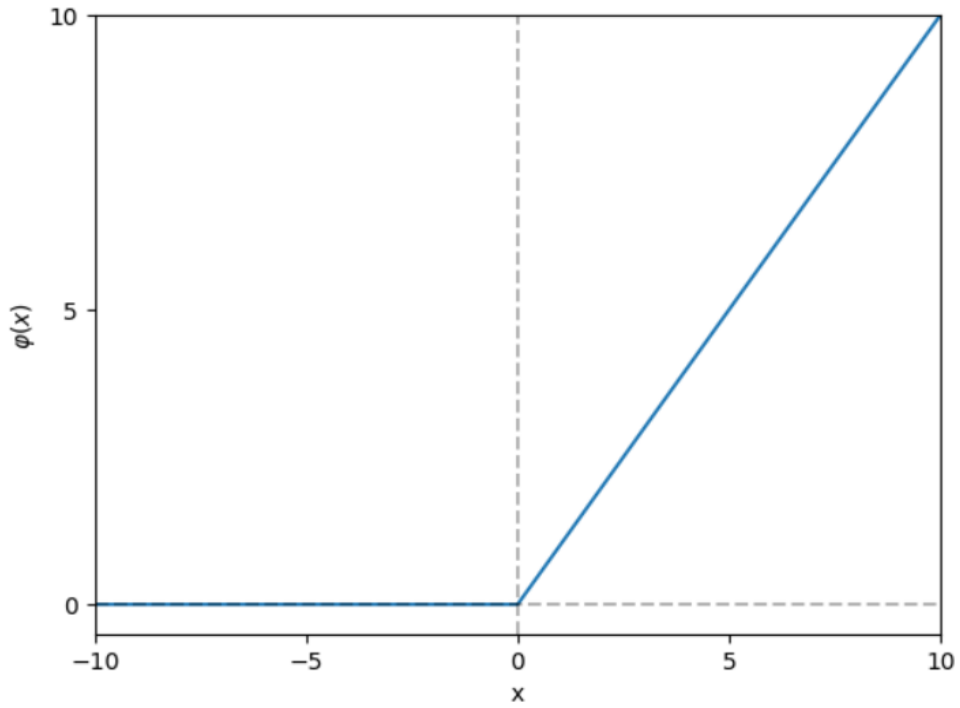


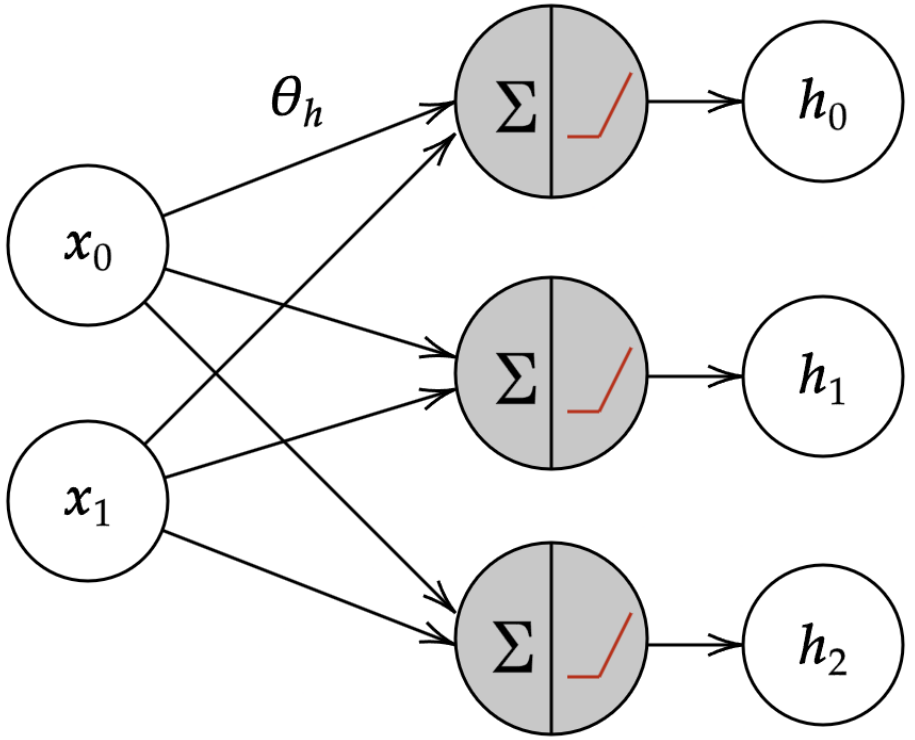




Squashes the input in [-1,1] → $\varphi([0 \ -3 \ 7]) = [0 \ -0.995 \ 0.999]$

# Rectified Linear Unit (ReLU)

$$\varphi(\boldsymbol{\theta x}) = \max(0, \boldsymbol{\theta x})$$




Zeroes the negative components of the input vector, leaving the rest unchanged

$\rightarrow \varphi([0\ \ -3\ 7]) = [0\ 0\ 7]$

# MLP – Output Layer

The output layer transforms the hidden layer output $\boldsymbol{h}$ into the prediction $y$

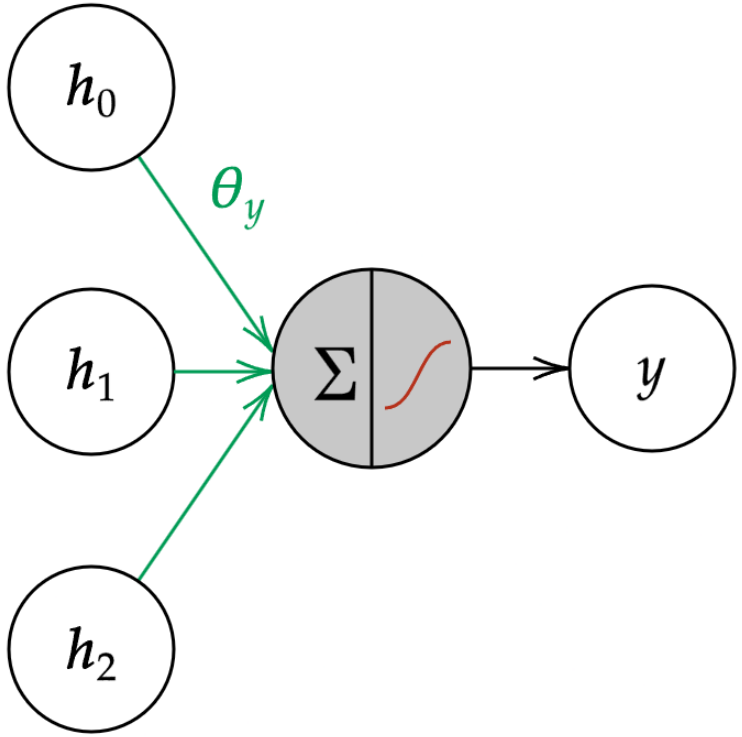
- Provides the final prediction of the MLP
- The transformation from $\boldsymbol{h}$ to $y$ is parameterized by $\boldsymbol{\theta}^{y}$ which are learned during training
- It implements a parameterized function $y = g_{\theta^{y}}(\boldsymbol{h})$
- Computes similarly to the hidden layer, but it is associated to activation functions which are task-specific and are linked to a loss function

# Output layer for binary classification

Goal: classify a sample $\boldsymbol{x}$ into either class $y \in \{0,1\}$

- Linear combination of the inputs
- Followed by sigmoid activation

$$\hat{y} = \sigma(\boldsymbol{\theta}^y \boldsymbol{h})$$




Binary cross entropy loss

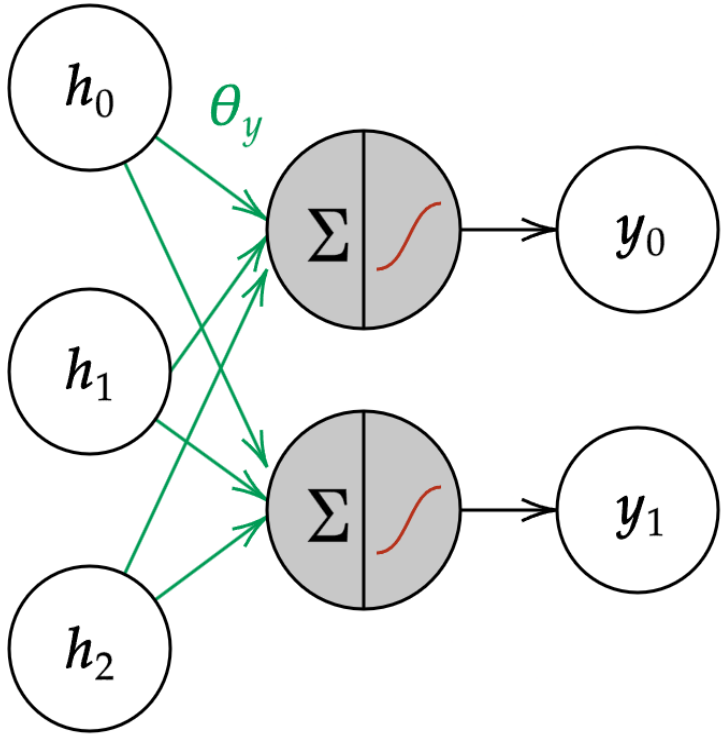
$$BCE(\hat{y}, y) = -y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

# Output layer for multiple binary classification

Goal: predict a d-dimensional vector $\boldsymbol{y}$ of 0 and 1 for a sample $\boldsymbol{x}$

- Output vector $\boldsymbol{y} \in \{0,1\}^d$
- Repeated application of the sigmoid for the d-dimensions

$$\hat{\boldsymbol{y}} = \sigma(\boldsymbol{\theta}^y \boldsymbol{h})$$




Average binary cross entropy loss

$$BCE_{multi}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{d}\sum_{d} BCE(\widehat{y_d}, y_d)$$

# Output layer for regression

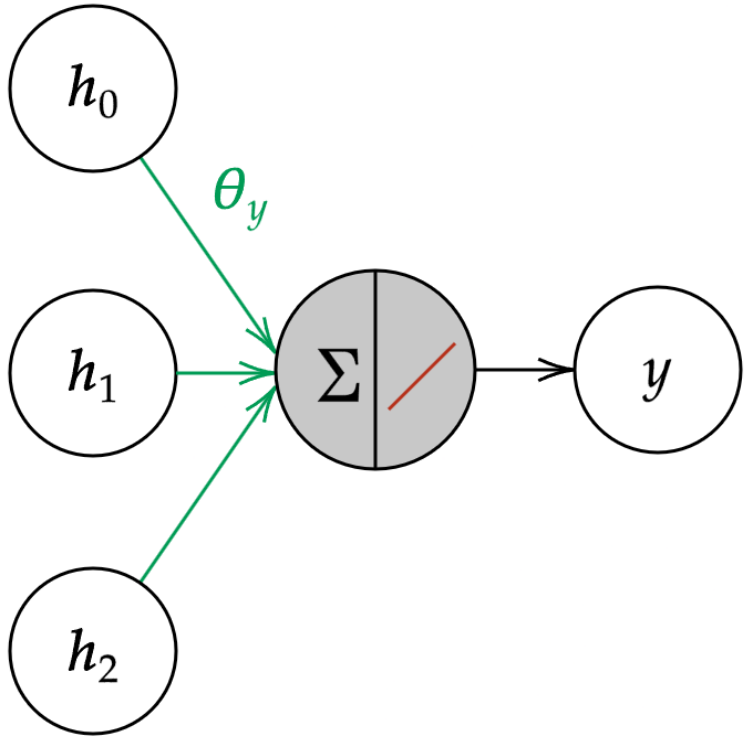
Goal: predict a real value $y$ in response to sample $\boldsymbol{x}$

- Just a linear combination of the inputs
- Followed by linear (identity) activation

$$\hat{y} = (\boldsymbol{\theta}^{y}\boldsymbol{h})$$

Mean squared error loss

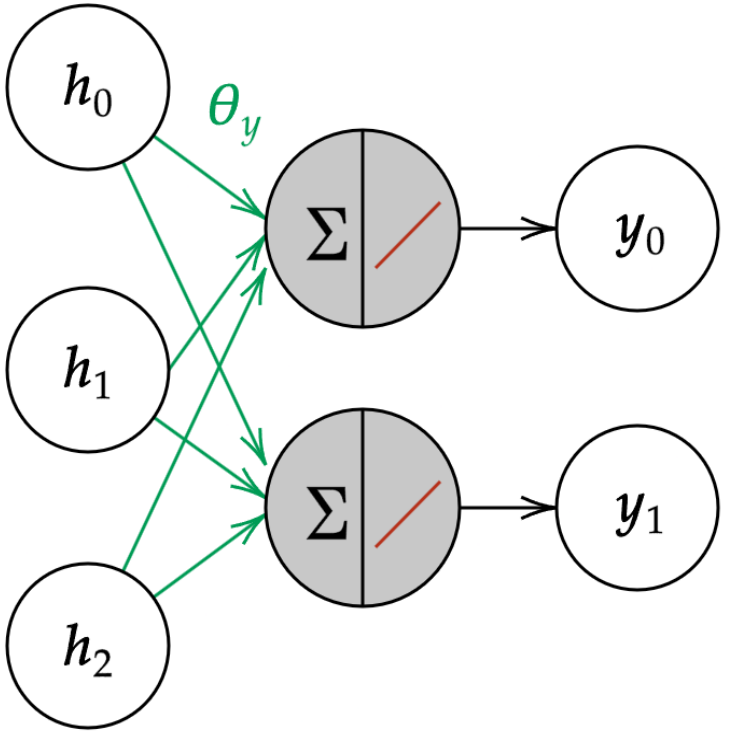
$$MSE(\hat{y}, y) = (y - \hat{y})^2$$



# Output layer for multiple regression

Goal: predict a real valued vector $\boldsymbol{y} \in \mathbb{R}^d$ in response to sample $\boldsymbol{x}$

- Multiple linear combination of $\boldsymbol{h}$, one for each component of $\boldsymbol{y}$
- Followed by multiple application of identity




Mean squared error loss

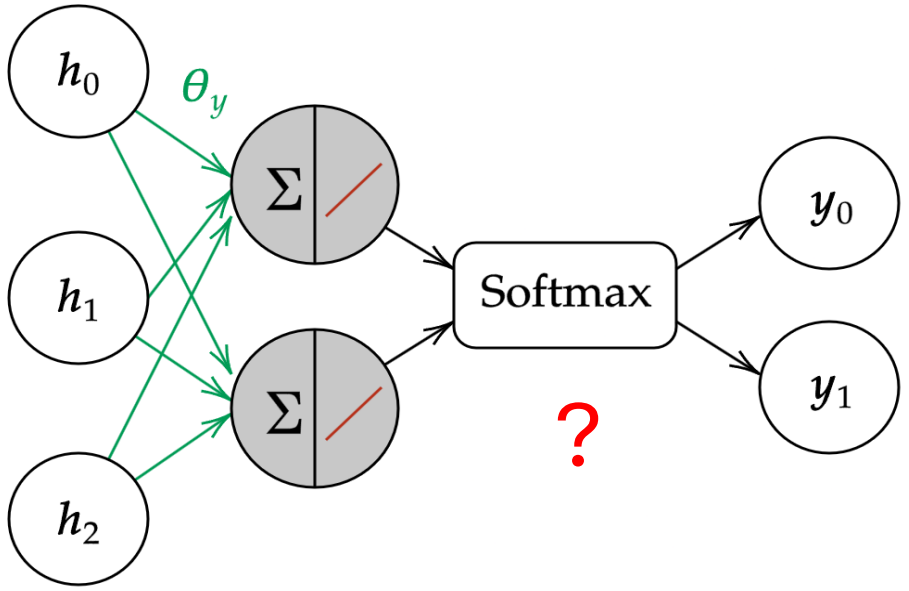
$$MSE_{multi}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{d}\sum_d MSE(\widehat{y_d}, y_d)$$

# Output layer for multiclass classification

Goal: assign a sample $\boldsymbol{x}$ to one of $d$ classes $\{c_1, \dots, c_d\}$

- Output $\boldsymbol{y} \in \{0,1\}^d$ as a one-hot encoding of the class
- Linear combination of $\boldsymbol{h}$ followed by identify
- Output generated by a softmax function

$$\hat{y} = softmax(\boldsymbol{\theta}^y \boldsymbol{h})$$




## Cross-Entropy loss

$$CE(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_d y_d \log(\widehat{y_d})$$

# Softmax function

- Transforms a (dense) vector in a categorical (discrete) probability distribution
- Given a vector $\boldsymbol{x}$ with $n$ components

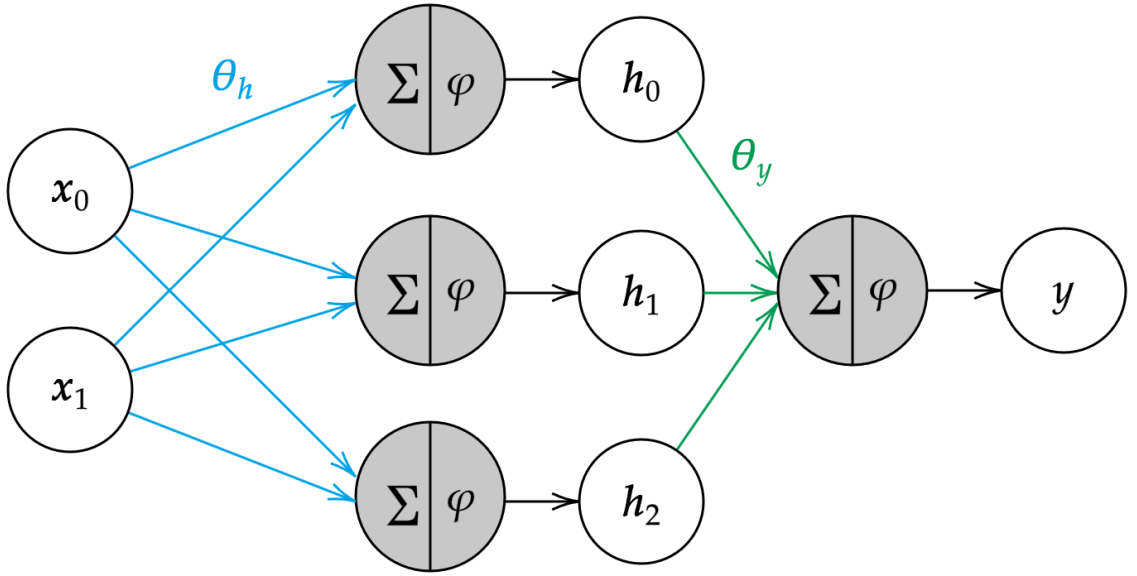
$$softmax_i(\boldsymbol{z}) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

In other words

- Take the exponentiation of each component of $\boldsymbol{z}$
- Compute the sum of exponentiated components
- Divide each exponentiated component by the overall sum
- The resulting softmax vector will sum-to-1 (like a probability distribution)

$$softmax([-1\ 0\ 1]) = [0.09\ 0.244\ 0.666]$$

# MLP: putting things back together




- Transform $\boldsymbol{x}$ into $\boldsymbol{h}$ through the hidden layer $\boldsymbol{h} = g_{\theta^h}(\boldsymbol{x}) = \varphi(\boldsymbol{\theta}^h \boldsymbol{x})$
- Transform $\boldsymbol{h}$ into $y$ through the output layer $y = g_{\theta^y}(\boldsymbol{h}) = \varphi(\boldsymbol{\theta}^y \boldsymbol{h})$
- The model is $MLP_\theta(\boldsymbol{x}) = g_{\theta^y}(g_{\theta^h}(\boldsymbol{x}))$ with $\theta = \{\theta^h, \theta^y\}$ being learned parameters

# What do we need to build an MLP?

- Input preprocessing
- Size of the hidden layer $s$
- Hidden layer activation functions $\varphi$

} Model selection choices

- Configuration of the output layer (guided by the task)

# Training an Artificial Neural Network

# Let's recall logistic regression training

For a certain number of iterations (epochs):

For each training pair $\left(\boldsymbol{x}^i, y^i\right) \in D_{train}$:

1. Compute the prediction $h_\theta\left(\boldsymbol{x}^i\right) = \sigma(\boldsymbol{x}^i\boldsymbol{\theta})$
2. Compute the loss L of the prediction $h_\theta\left(\boldsymbol{x}^i\right)$ compared to the true label $y^i$
3. Compute the gradient of the loss $\boldsymbol{\nabla_\theta L} = \boldsymbol{x}(y^i - h_\theta\left(\boldsymbol{x}^i\right))$
4. Update the parameters $\boldsymbol{\theta}_{new} = \boldsymbol{\theta} - \eta\boldsymbol{\nabla_\theta L}$
5. Use the updated parameters in the next iteration $\boldsymbol{\theta} = \boldsymbol{\theta}_{new}$

# MLP training

For a certain number of iterations (epochs):

For each training pair $(\boldsymbol{x}^i, y^i) \in D_{train}$:

1. Compute the prediction $MLP_\theta(\boldsymbol{x}^i)$
2. Compute the loss $L$ of $MLP_\theta(\boldsymbol{x}^i)$ compared to the true label $y^i$
3. Compute the gradient of the loss $\boldsymbol{\nabla_\theta L}$
4. Update the parameters $\boldsymbol{\theta}_{new} = \boldsymbol{\theta} - \eta \boldsymbol{\nabla_\theta L}$
5. Use the updated parameters in the next iteration $\boldsymbol{\theta} = \boldsymbol{\theta}_{new}$

Actually, it is a bit more articulated than that

# MLP training, more realistically

For a certain number of iterations (epochs):

Shuffle $D_{train}$

For each subsample $B$ of pairs $(\boldsymbol{x}^B, y^B)$ extracted from $D_{train}$ :

1. Compute the prediction $MLP_\theta(\boldsymbol{x}^B)$
2. Compute the loss $L$ of $MLP_\theta(\boldsymbol{x}^B)$ compared to the true labels $y^B$
3. Compute the gradient of the loss $\boldsymbol{\nabla_\theta L}$
4. Update the parameters $\boldsymbol{\theta}_{new} = \boldsymbol{\theta} - \eta \boldsymbol{\nabla_\theta L}$
5. Use the updated parameters in the next iteration $\boldsymbol{\theta} = \boldsymbol{\theta}_{new}$

The gradient of the loss needs to be computed w.r.t. all parameters, including those from the hidden layer!

# Minibatching

Mini-batching is a technique used to split the dataset into smaller subsets, each containing B examples. Instead of updating the parameters after processing all examples or after each single example, updates are done after processing each mini-batch

1. Divide the dataset into multiple mini-batches of size B (batch size, hyperparameter)
2. Shuffle the dataset at the beginning of each epoch to avoid bias in training.
3. Update model parameters once per mini-batch.

- Why is Mini-Batching Useful? It is a compromise between:
  - Stochastic Gradient Descent (SGD) (B = 1): Updates after each example → fast but noisy updates.
  - Batch Gradient Descent (B = n): Updates after the entire dataset → stable but slow updates.
- Mini-batching provides a balance
  - Faster convergence than full batch updates
  - More stable updates than SGD
  - Efficient computation by leveraging GPU parallelism

# Gradient of the loss

We have different loss functions based on the output layer, and we can compute their gradients as follows

| Loss Function | Formula | Gradient |
|---|---|---|
| Binary Cross-Entropy (BCE) | $-[y \log(\hat{y}) + (1-y) \log(1-\hat{y})]$ | $x(\hat{y} - y)$ |
| Mean Squared Error (MSE) | $(\hat{y} - y)^2$ | $x(\hat{y} - y)$ |
| Categorical Cross-Entropy (CE) | $-\sum_{k=1}^{K} y_k \log(\hat{y}_k)$ | $\hat{y} - y$ |

We can update the output layer parameters ($\theta^y$) using these gradients

However, we don't yet know how to compute the gradient for the hidden layer parameters ($\theta^h$)

# Backpropagation at the rescue

Key Idea: We apply the Chain Rule to propagate the gradient from the output layer to the hidden layer

**1. Compute the gradient for the output layer**

- Example: For BCE, the gradient is

$$\nabla_{\theta^y} L = x(\hat{y} - y)$$

**2. Propagate the gradient back to the hidden layer**

- Using the Chain Rule, the gradient for the hidden layer parameters $\theta^h$ is:

$$\nabla_{\theta^h} L = \nabla_h L \cdot \nabla_{\theta^h} h$$

- $\nabla_h L$ is the gradient of the loss with respect to the hidden layer output $\boldsymbol{h}$
- $\nabla_{\theta^h} h$ is the gradient of $\boldsymbol{h}$ with respect to $\theta^h$

**3. Compute the gradient of $\boldsymbol{h}$**

- If the hidden layer transformation is:

$$\boldsymbol{h} = \varphi(\boldsymbol{\theta}^h \boldsymbol{x})$$

- Then, using the chain rule:

$$\nabla_{\theta^h} L = \nabla_h L \cdot \left(\varphi'\left(\boldsymbol{\theta}^h \boldsymbol{x}\right)\right) \cdot \boldsymbol{x}$$

- Where $\varphi$' is the derivative of the activation function

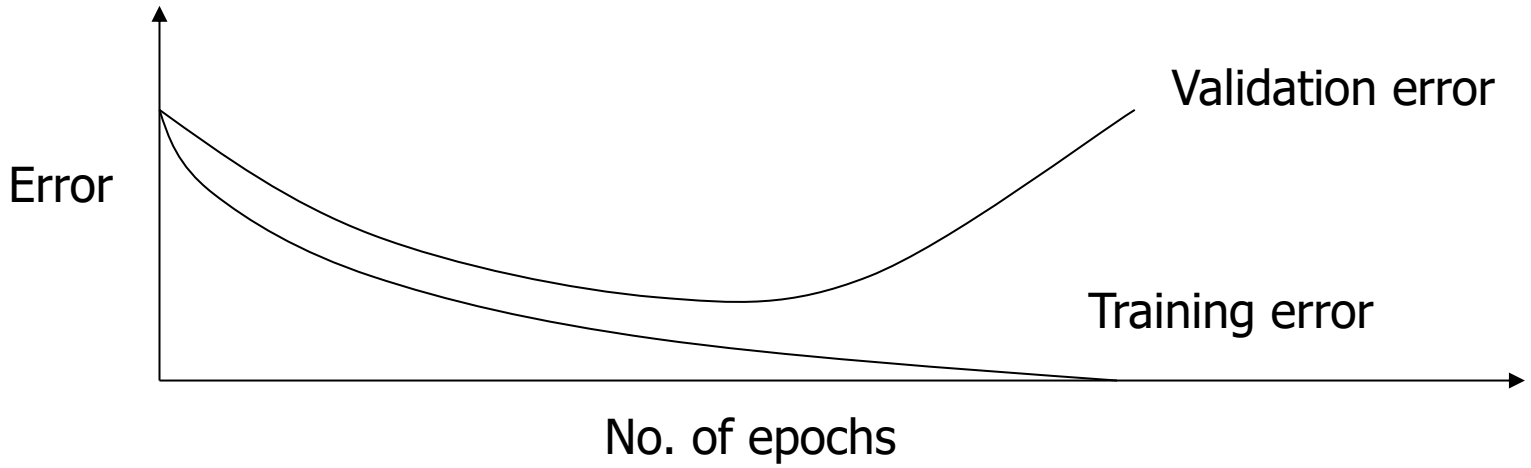
Luckily for you backpropagation is automatically handled by the NN libraries

# Convergence Criteria

- Learning is obtained by iteratively supplying shuffled training data and adjusting by backpropagation
  - Typically, 1 training set presentation = 1 epoch
- We need a stopping criteria to define convergence
  - Validation for generalization performance: stop when generalization performance reaches a peak

# Early Stopping

- Keep a hold-out validation set and assess accuracy after (every/some) epoch.
- Maintain weights for best performing network on the validation set and stop training when error increases beyond this
- Always let the network run for some epochs before deciding to stop (patience parameter), then backtrack to best result



# Wrap-up

# Take home lessons

- Artificial Neural Networks are universal function approximators that can learn complex patterns from noisy and diverse data
  - In healthcare they allow handling vast amounts of heterogenous data in a scalable way, providing highly accurate predictions
- The artificial neuron is loosely inspired by biological neurons
  - Artificial neurons receive and aggregate inputs, apply activation functions, and transmit outputs
- The MLP has a layered structure consisting of input, hidden, and output layers
  - Capable of learning any decision boundary and adaptable to various tasks
  - Several architectural/model selection choices need to be taken even for simple networks
- Training ANNs involves a specialized gradient descent algorithm (backpropagation)
  - Optimization aspects needs to be carefully curated
  - Stopping conditions are relevant as it is an iterative process
- Optimize neural networks from the start
  - Applying the right weight initialization strategies significantly impacts effectiveness
  - Needs to be tailored to the activation function

# Next Lecture

- Next week lecture
  - Risk scoring and stratification for population health management
  - Machine learning for risk stratification
  - Identifying risk factors
  - Assessment and validation of risk predictors
  - Censoring and its impact in risk scoring