# Data Analysis and Machine Learning using Python

Lecture 6: Neural Networks and Deep Learning;
*April 26 2024*

# Today:

- Homework 7 will be posted tomorrow
- Quick review of home work 5
- Some useful functions in scikit-learn
- Perceptrons, neural Networks and deep learning

# Artificial Neural Networks

# Artificial Neural Networks

Human brain

- Neuron switching time ~.001 second
- Number of neurons ~$10^{10}$
- Connections per neuron ~$10^{4-5}$
- Scene recognition time ~.1 second
- Requires a lot of training
- Lots of parallel computation!

Artificial neural networks (ANNs):

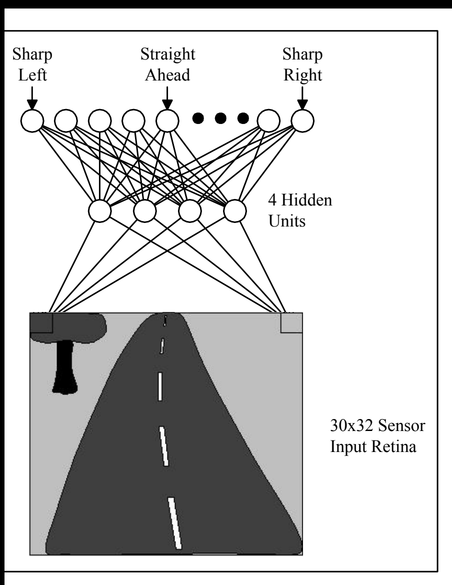
- Many neuron-like threshold switching units

# When to Consider Neural Networks

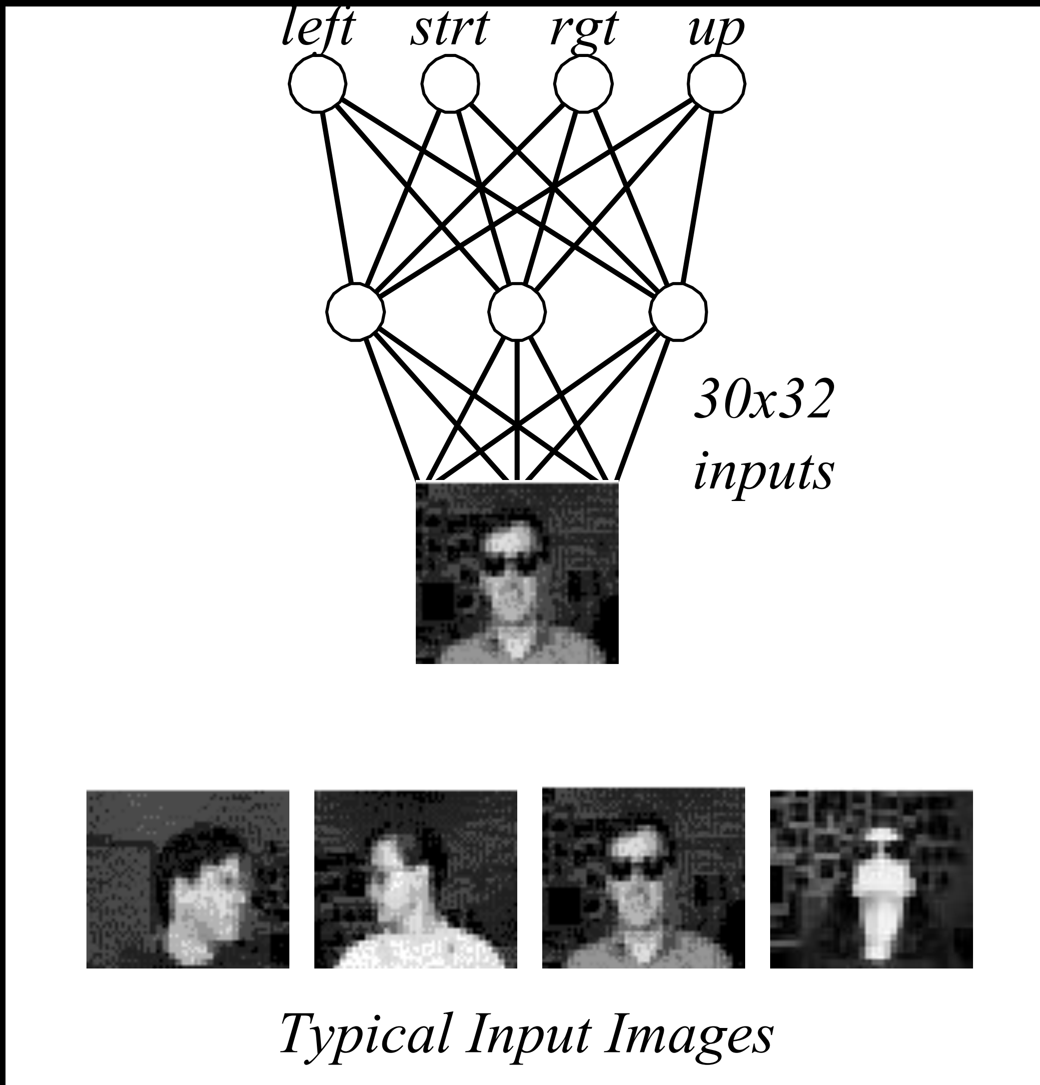
- Input is high-dimensional discrete or real-valued (e.g., combining information from different measurements or detectors)
- Output is discrete or real valued
- Output is a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is *unimportant*

Examples:

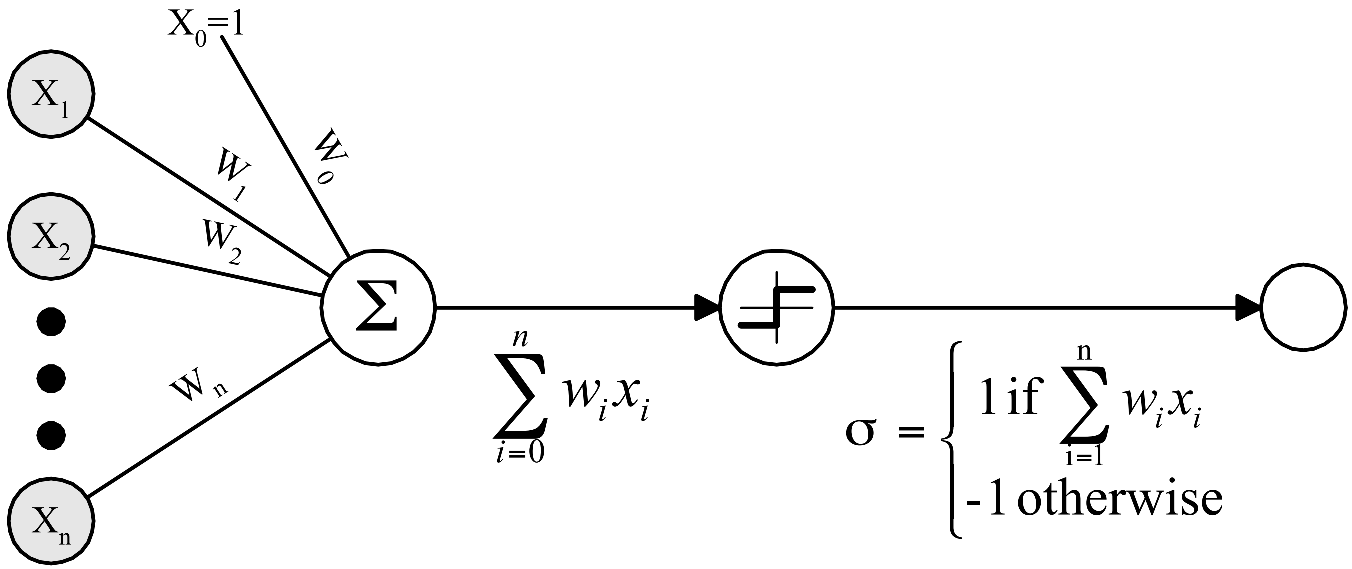
# ANN drives through Paris




# ANN recognizes faces



# Perceptron

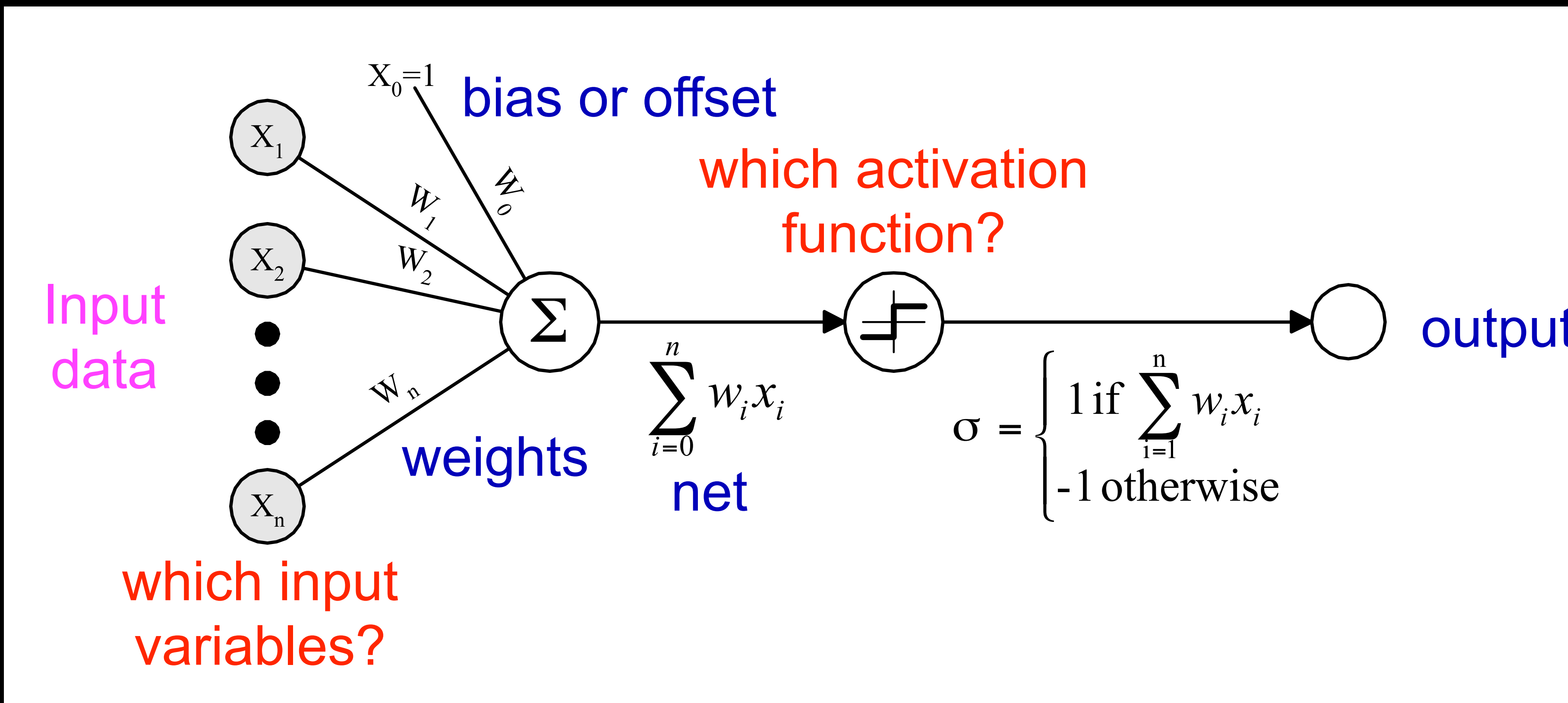



$$o(x_1,\ldots,x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + \ldots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

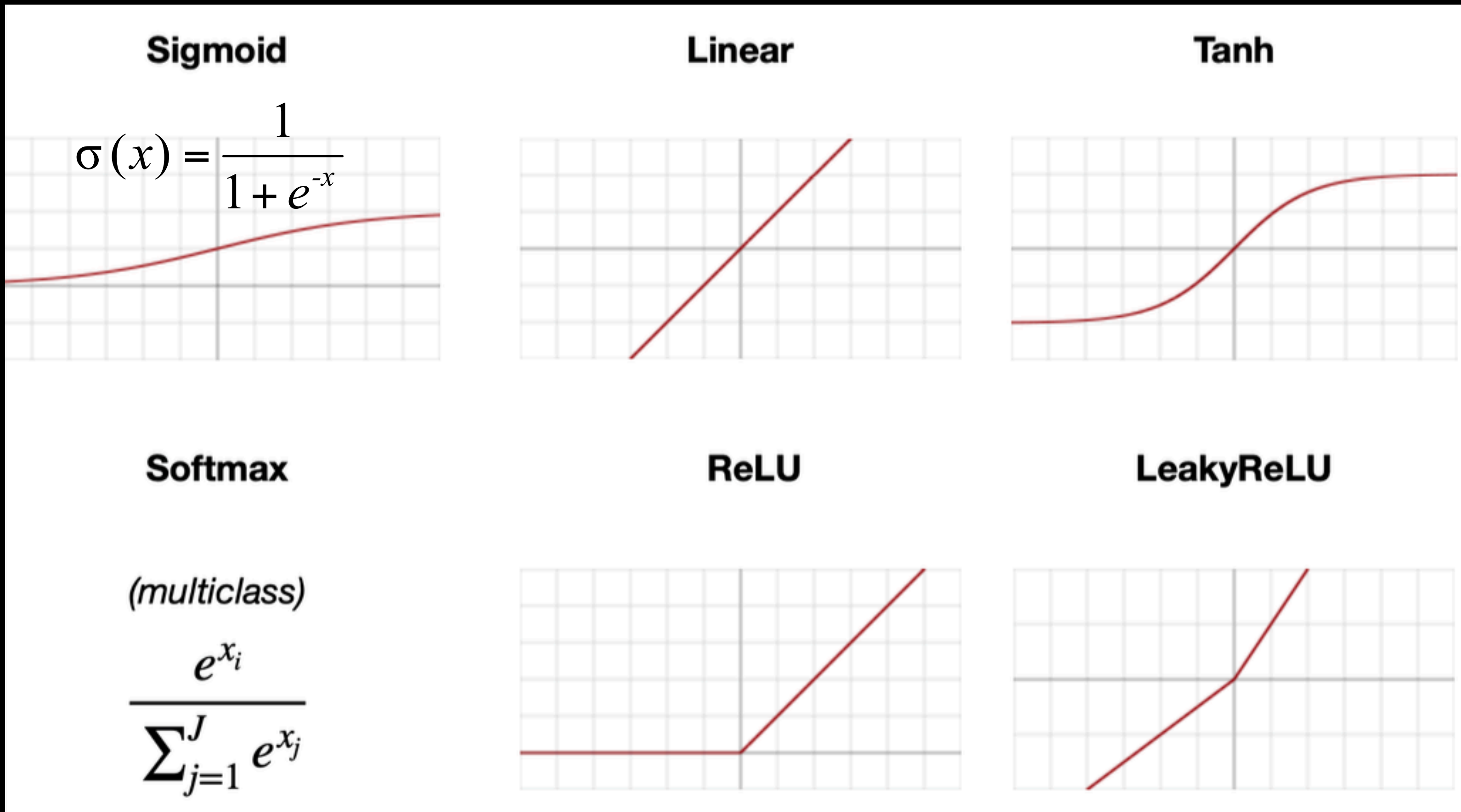
Sometimes we will use simpler vector notation :

$$o(\vec{x}) = \begin{cases} 1 \text{ if } \vec{w} \cdot \vec{x} > 0 \\ -1 \text{ otherwise} \end{cases}$$

# Elements of perceptrons

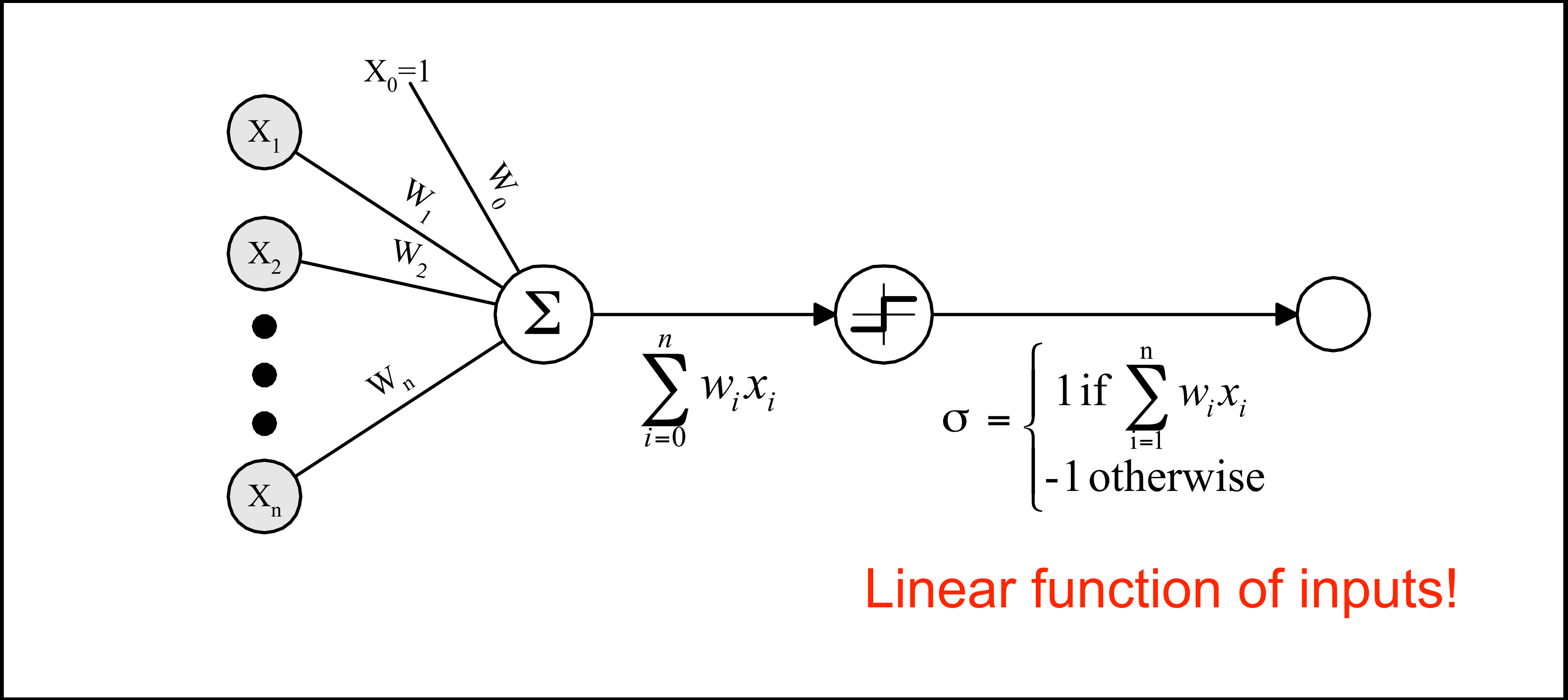


# Common activation functions

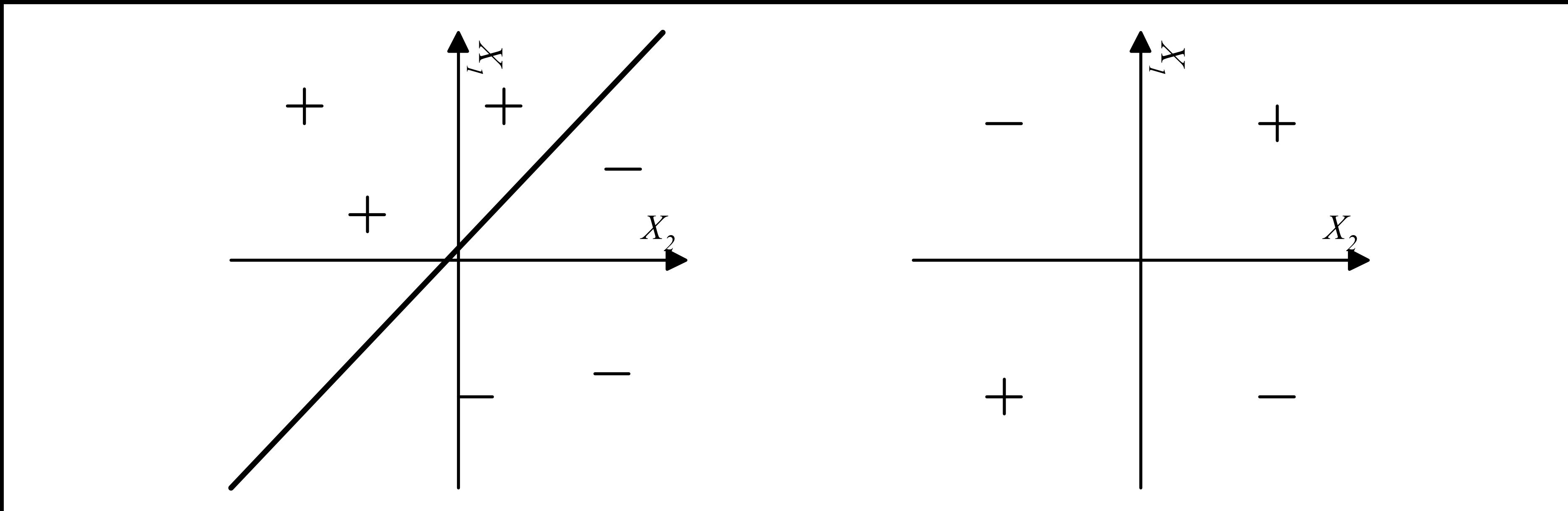



- Sigmoid function: “rounded” step function
- Unlike step function, can take derivative → helpful for learning

# Elements of perceptrons



# Perceptron decision boundaries




Can epresent some useful functions

But some functions not representable

- e.g., not linearly separable
- therefore, we will want networks of perceptrons

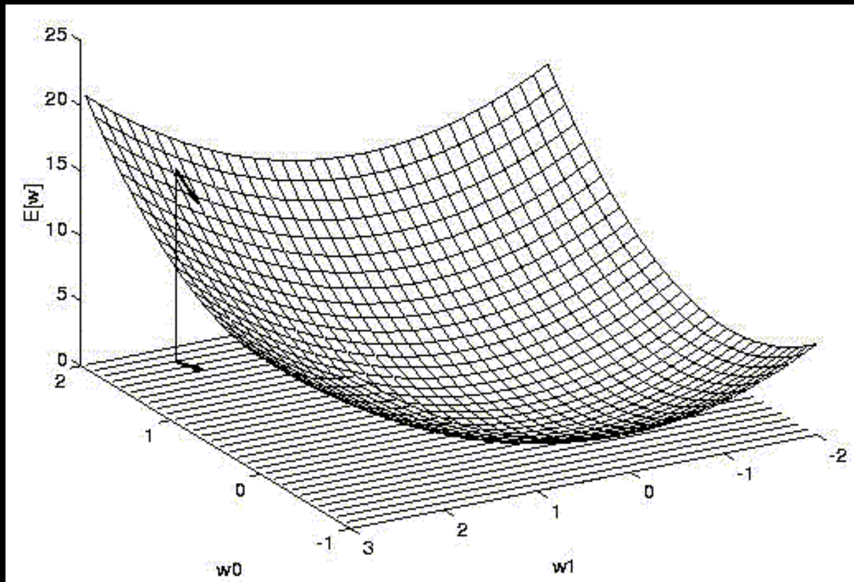
# How to train the perceptron?

- What does training the perceptron mean?
  - We want to minimize a loss function
  - Change the weights to find the minimal value of the loss function in the space of weights

# How to train the perceptron?

- What does training the perceptron mean?
  - We want to minimize a loss function
  - Change the weights to find the minimal value of the loss function in the space of weights
- Example loss function: Mean squared error
  - $E(\overrightarrow{w}) = \frac{1}{n} \sum (t_i - o(\vec{x}))^2$
  - $E$ is loss function depending on weights $\overrightarrow{w}$
  - $t_i$ are the truth (target) values we want to predict
  - $o(\vec{x})$ is the output of the perceptron (or NN), as a function of the input values $\vec{x}$

# How to train the perceptron?

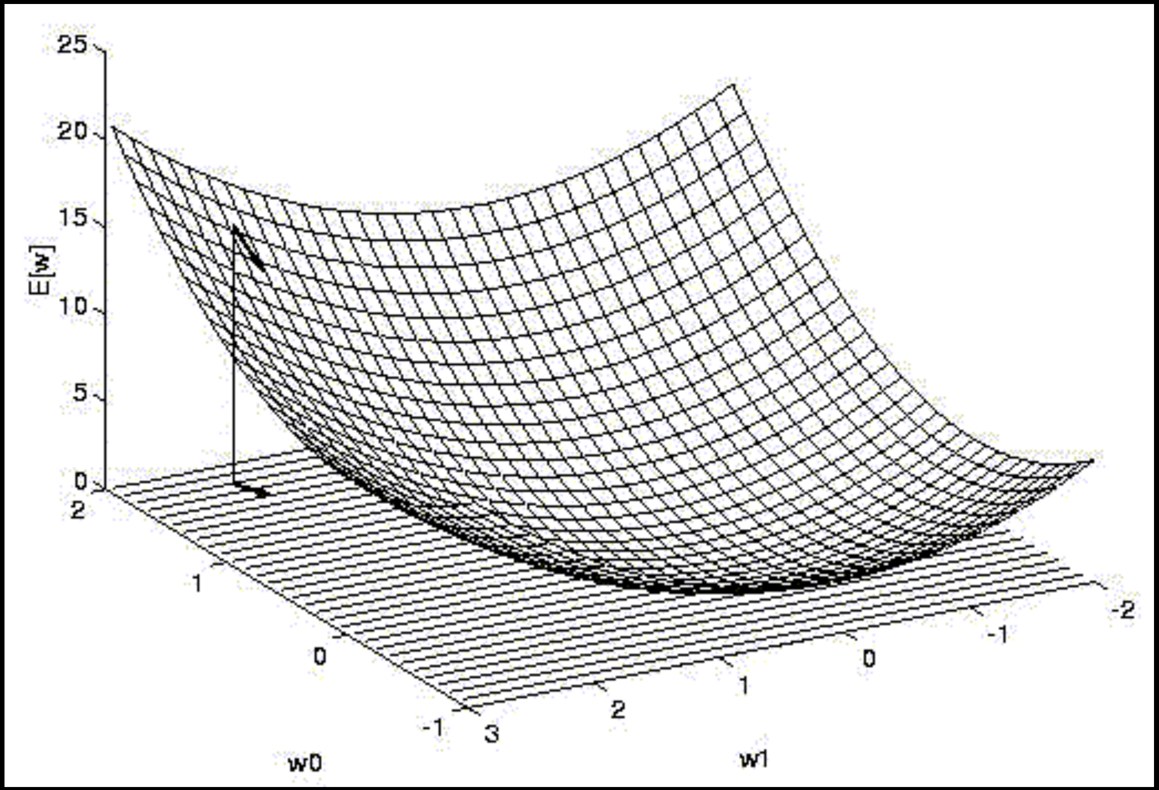



Example: Loss function for perceptron with two inputs (two weights), or one input + bias

How to find weights w that minimize loss function?

- One global fit will usually not work
  - noisy data → many local minima
- Solution: Gradient descent optimization

# Gradient descent




- Start somewhere (initial weights)
- Make steps in the direction of the steepest slope of E(**w**)

# Gradient descent

The direction of the steepest slope is given by the gradient $\nabla E(\overrightarrow{w})$ if the loss function w.r.t. the weights $\overrightarrow{w}$

Gradient $\quad \nabla E[\vec{w}] \equiv \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n} \right]$ Slope of $E(\overrightarrow{w})$ surface

Training rule: $\Delta w_i = -\eta \, \nabla E[\vec{w}]$ small step η down slope ∇E

i.e., $$\Delta w_i = -\eta \, \frac{\partial E}{\partial w_i}$$

# Gradient descent

Differentiate using chain rule:

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2$$

$$= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2$$

$$= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d)$$

$$= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x}_d)$$

$$\frac{\partial E}{\partial w_i} = \sum_d (t_d - o_d)(-x_{i,d})$$

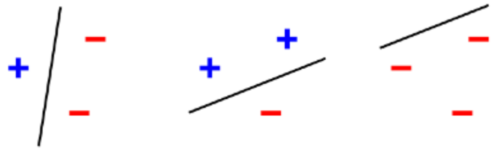
# How to train your Perceptron

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta\,(t - o) x_i$$

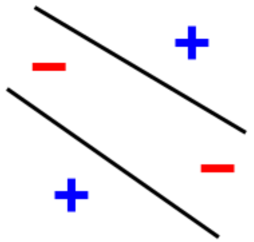
- $t = c(\vec{x})$ is target value
- $o$ is perceptron output
- $\eta$ is small constant (e.g., .1) called learning rate




linearly separable

Can prove it will converge

- If training data is linearly separable
- and $\eta$ is sufficiently small



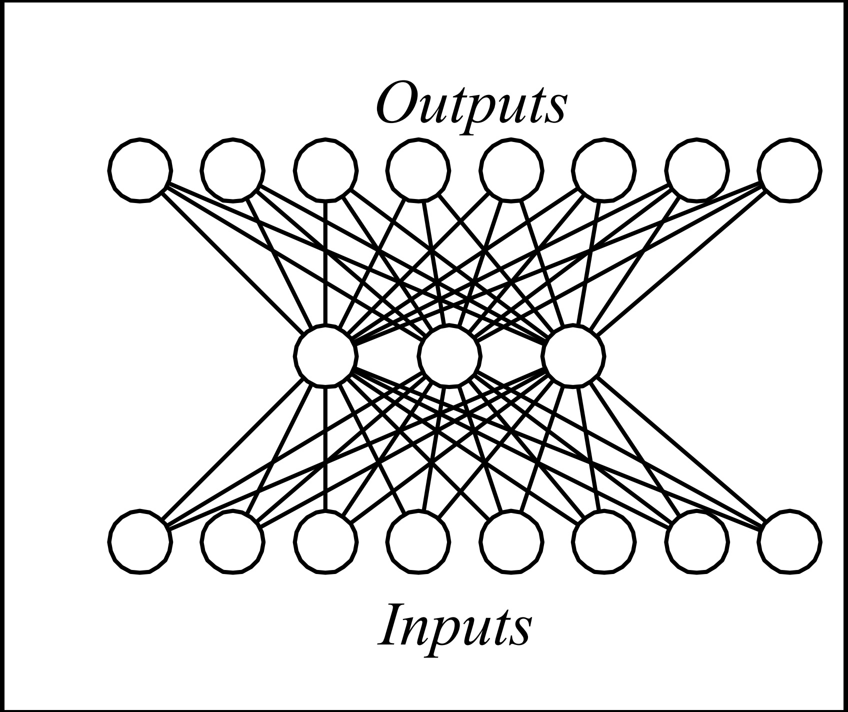

not l.s.

# Implementing the training

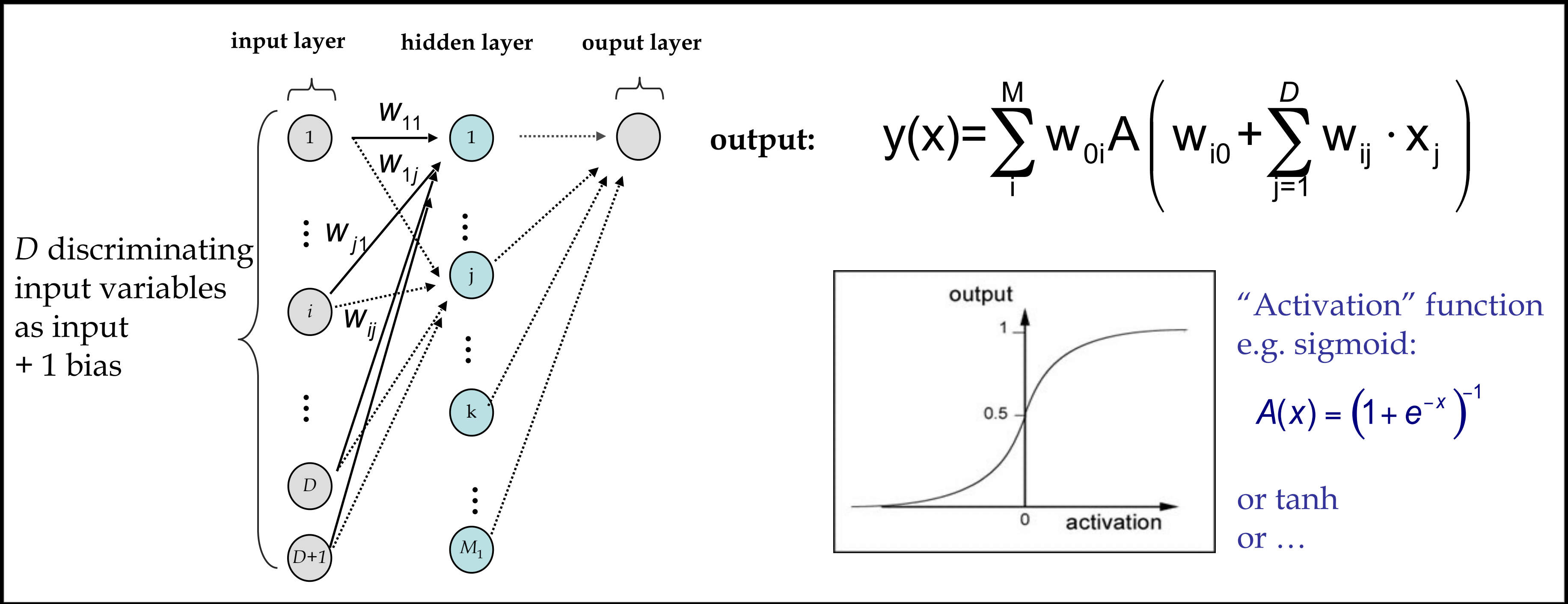
- Pick a small learning rate $\eta$, e.g. 0.05 - 0.1
- Initialize all $w_i$ to some small value
- Until training converges:
  - Initialize all $\Delta w = 0$
  - For each $\vec{x}$ in training sample
    - Calculate $o(\vec{x})$ for all $\vec{x}$in training sample
    - Update $\Delta w_i \rightarrow \Delta w_i + (t - o)x_i$
  - Then update $w_i \rightarrow w_i + \Delta w_i$

# Networks of perceptrons

- Linear units correspond to *hyperplanes* as decision boundary
- How to approximate arbitrary hyper surfaces?
- → Multi-layer perceptron (MLP)



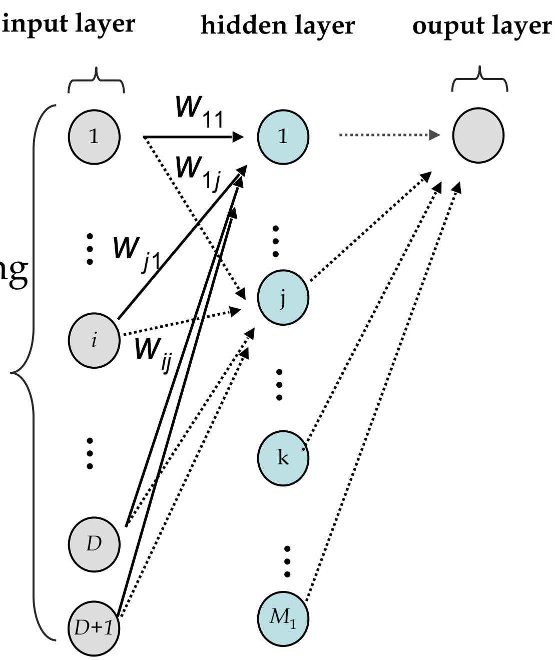
# Multilayer Perceptron - MLP



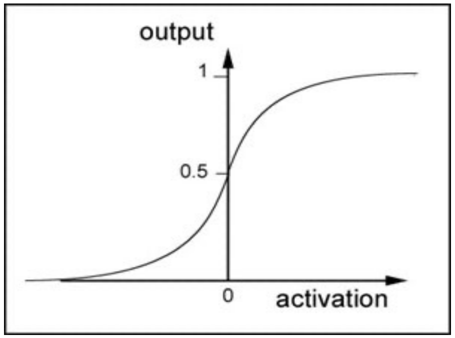

- Nodes in hidden layer have "activation functions" whose arguments are linear combinations of input variables → non-linear response to the input
- The output is a linear combination of the output of the activation functions at the internal nodes
- Input to the layers from preceding nodes only → feed forward network (no backward loops)
- It is straightforward to extend this to additional layers

# Multilayer Perceptron - MLP

*D* discriminating input variables as input + 1 bias




**output:** $$y(x) = \sum_{i}^{M} w_{0i} A\left( w_{i0} + \sum_{j=1}^{D} w_{ij} \cdot x_j \right)$$




"Activation" function e.g. sigmoid:

$$A(x) = \left(1 + e^{-x}\right)^{-1}$$

or tanh
or …

- Many connections: many independent weights
- Learning: Use analytic derivatives and gradient descent to optimize weights
- Can manually tune architecture, solver, activation function,….

# Backpropagation

- How to change weights for hidden layers?
- Can't take derivative of E wrt hidden layer weights directly
- Use same idea (gradient descent), but recursively
- Start with output layer, calculate update to weights from hidden layer
- Then update hidden layer weights
  - continue if multiple hidden layers
  - repeat until satisfied with network performance

# Backpropagation

Initialize all weights to small random numbers. Until satisfied, do

- For each training example, do

  1. Input the training example and compute the outputs

  2. For each output unit $k$

     $$\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)$$

  3. For each hidden unit $h$

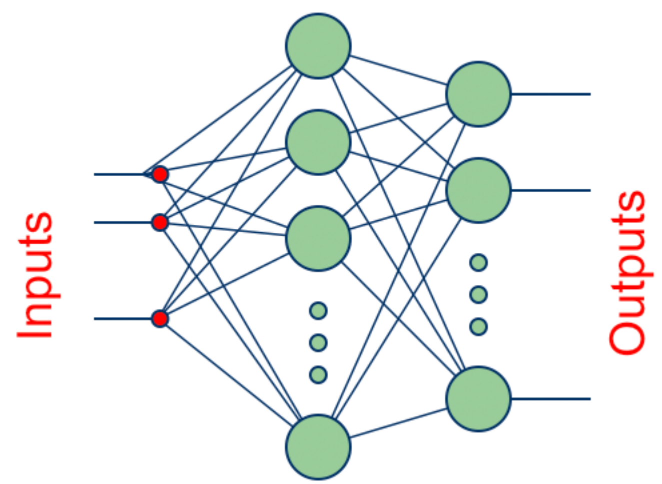
     $$\delta \leftarrow o_h(1-o_h)\sum_{k\in outputs} w_{h,k}\delta_k$$

  4. Update each network weight $w_{i,j}$

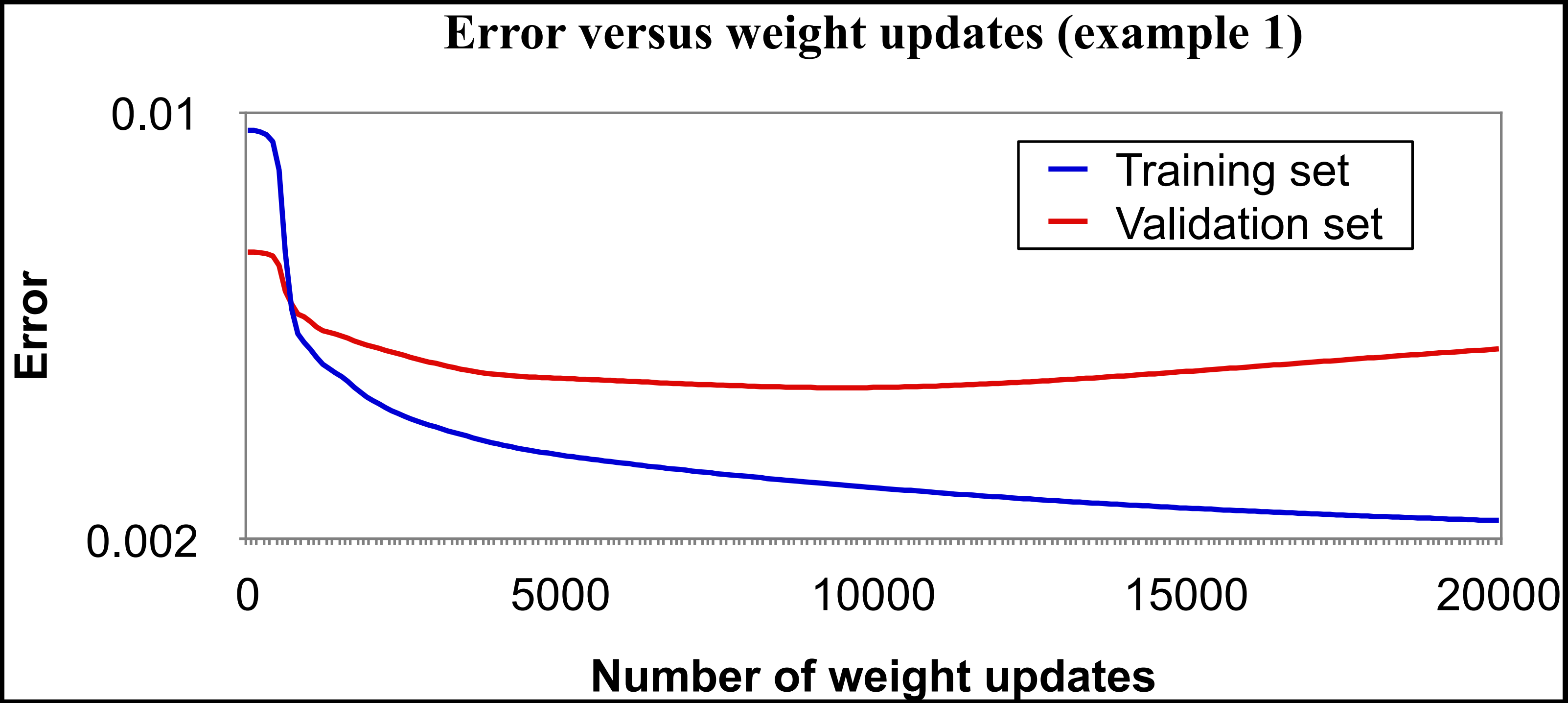
     $$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

     where

     $$\Delta w_{i,j} = \eta\,\delta_j x_{i,j}$$



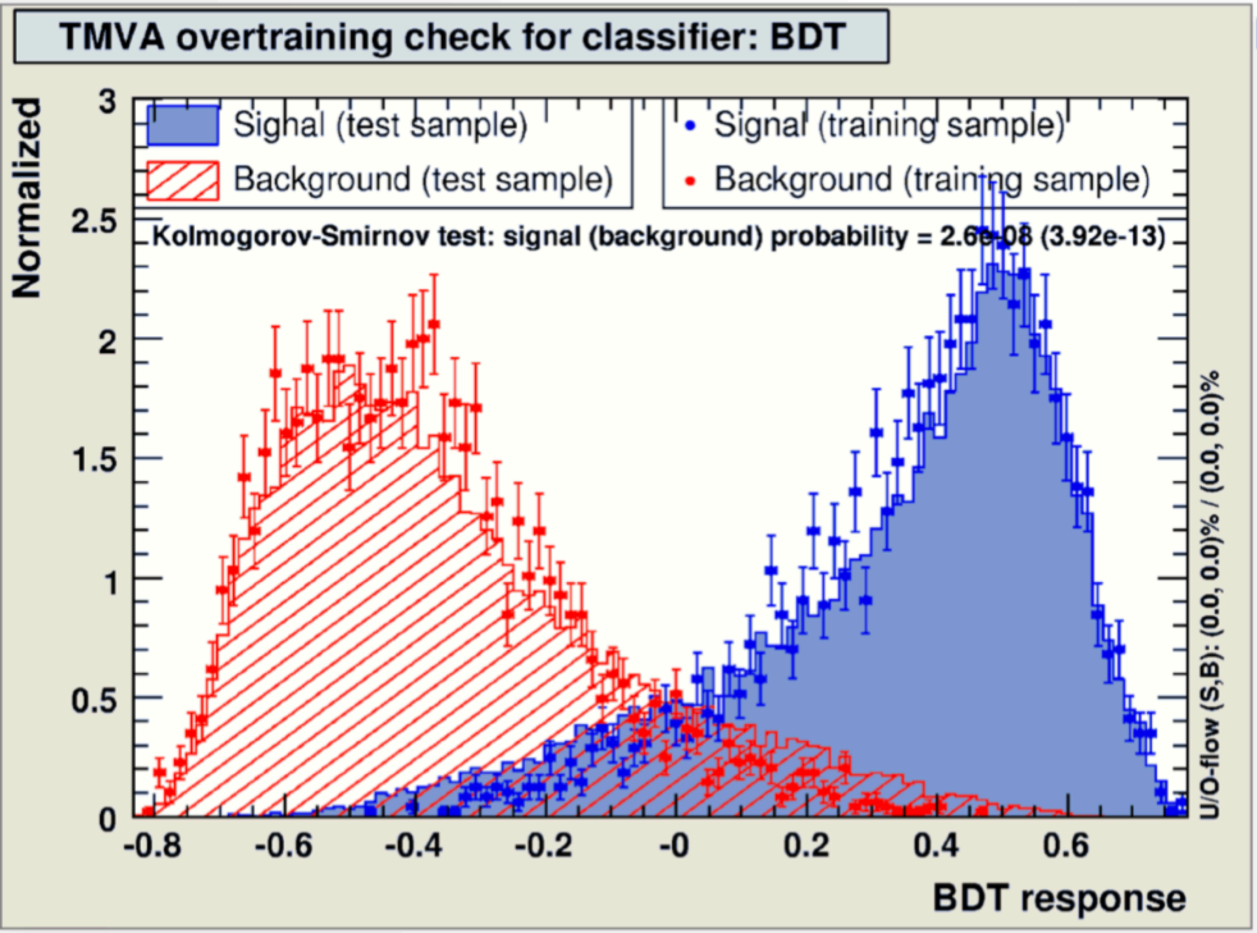
# Convergence/Overfitting in ANNs




Train on ***Training set***, check results
on independent ***Validation set*** *(or Test set)*

# Overtraining MVAs

- Check for overtraining: classifier output for test *and* training samples




- Remark on **overtraining**
  - Occurs when classifier training has too few degrees of freedom because the classifier has too many adjustable parameters for too few training events
  - Sensitivity to overtraining depends on classifier: *e.g.*, Fisher weak, BDT strong
  - Compare performance between training and test sample to detect overtraining
  - Actively counteract overtraining: *e.g.*, smooth likelihood PDFs, prune decision trees

# Let's look at some MLP examples in Scikit learn