COMP701 Nature Inspired Computing

Artificial Neural Networks: Feedforward Networks

Previous Lecturers

- Evolution Algorithms (biology-inspired)

  Genetic Algorithm

  — Crossover, Mutation, Selection, Fitness

  Evolution Strategy

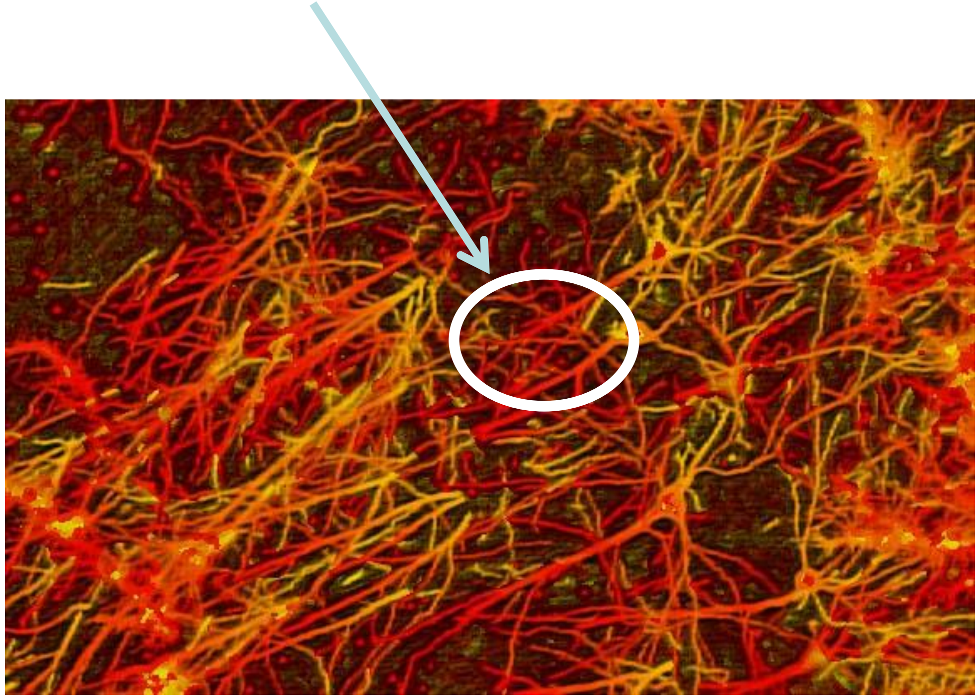
- Swarm Algorithms (social-inspired)

  Particle Swarm Optimisation

  Ant Colony Optimisation

Brain-Inspired

- Our brain is organized as layers of interconnected nerve cells (neurons) and tissue
- About 10^{11} neurons
  each with 10^4 connections



Neurons

- Neurons are basic building blocks of the central nervous system

- Connected by <u>dendrites</u> and <u>axons</u>
- Communicates through <u>electro-chemical</u> signals
- <u>Threshold</u> output *firing* of signals

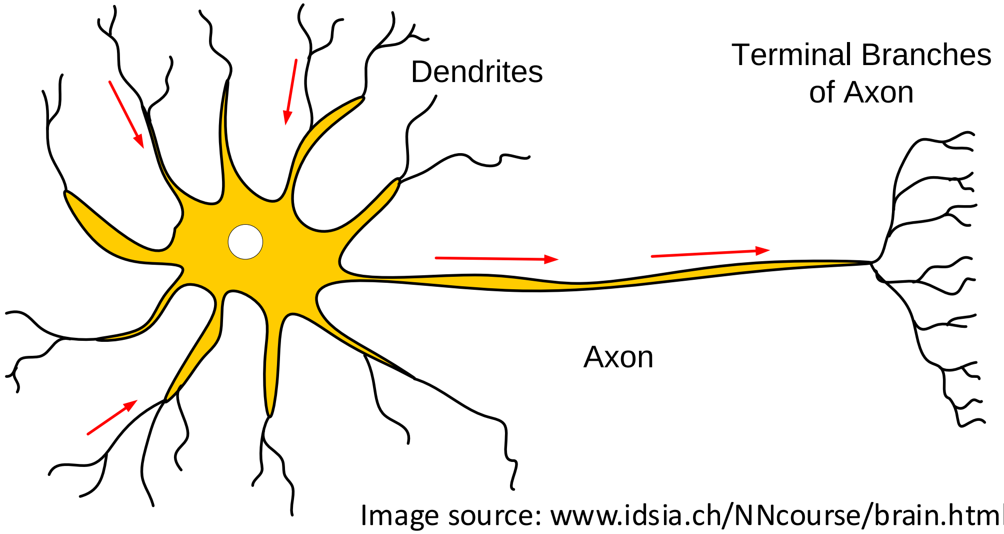
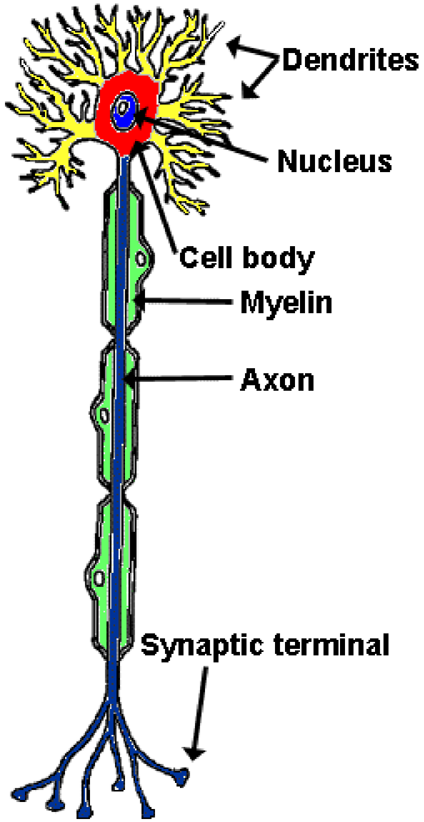


Image source: www.idsia.ch/NNcourse/brain.html

From Biology to Computing

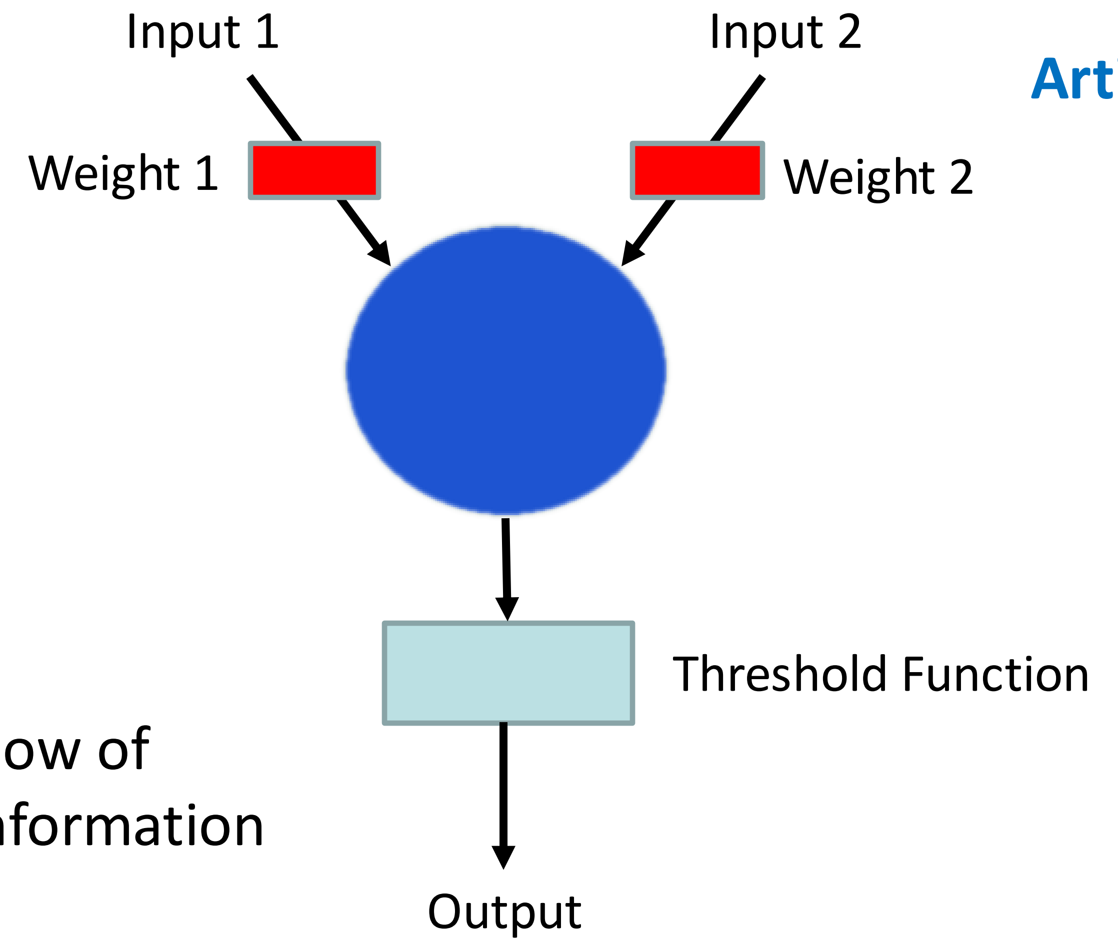
Biological



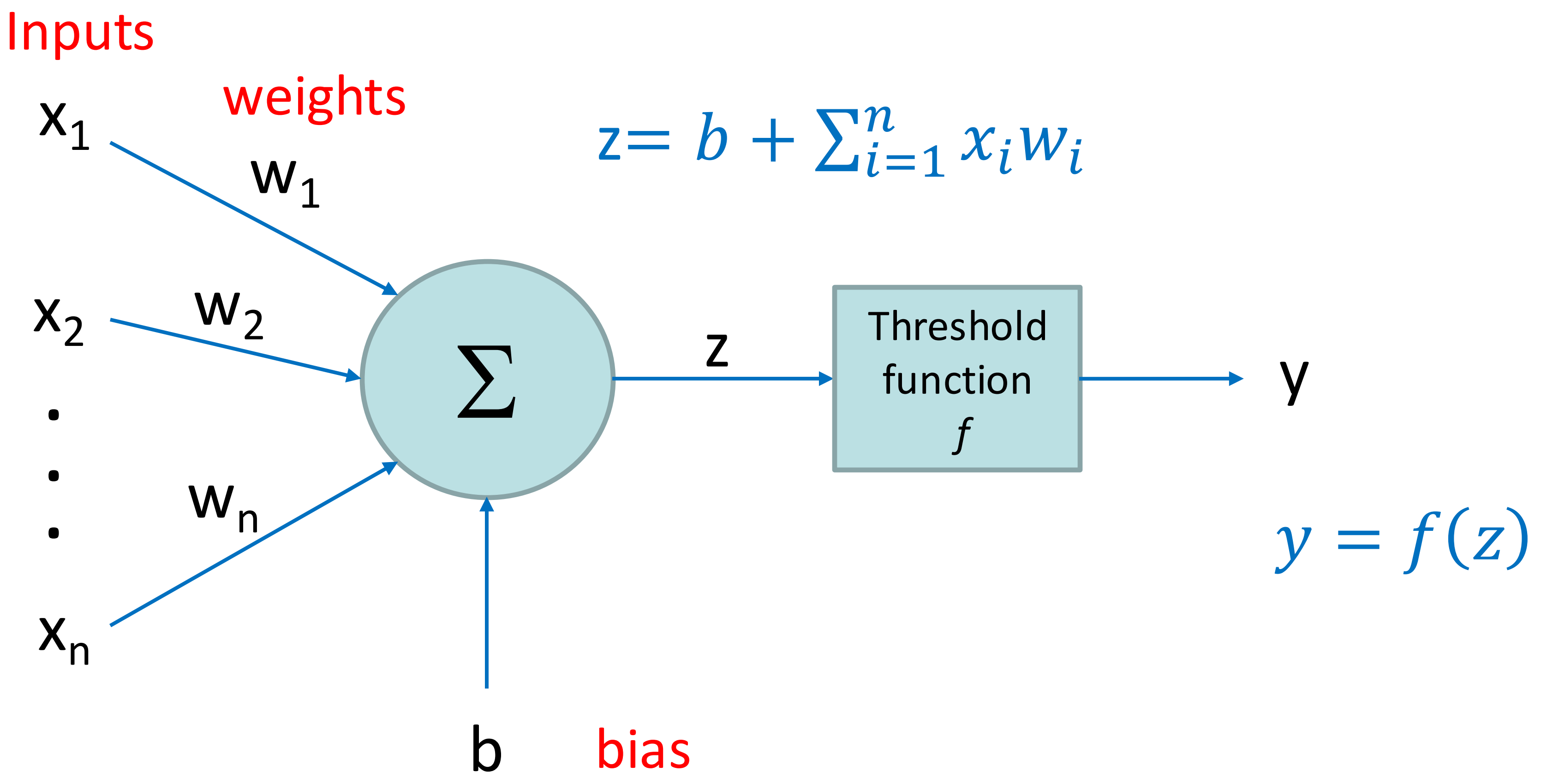
http://faculty.washington.edu/chudler/color/pic1an.gif

Flow of Information

Artificial



Perceptron

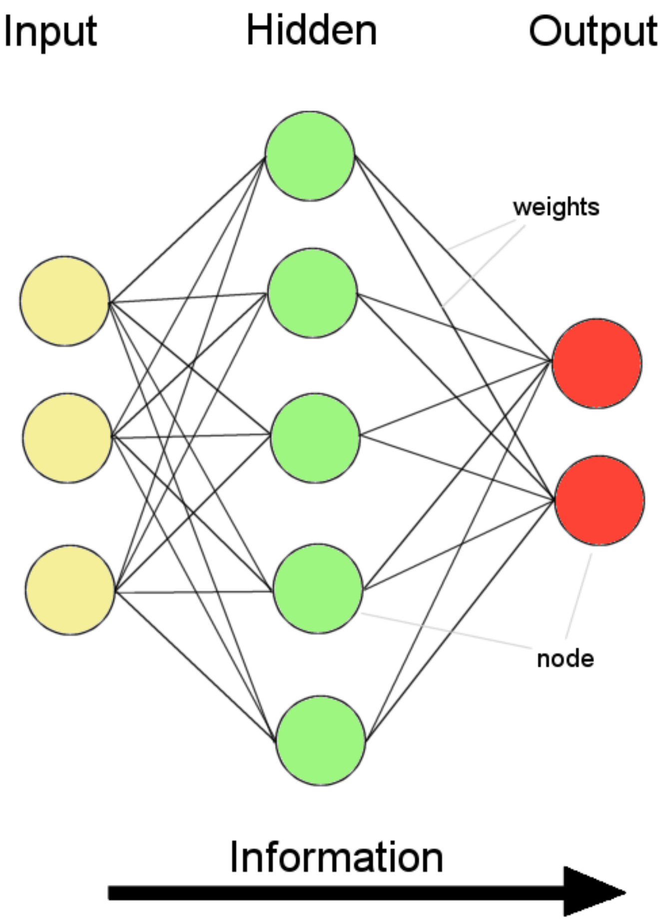


Inputs
weights
x_1
w_1
x_2
w_2
w_n
x_n
Σ
z
Threshold
function
f
y
b
bias
$z = b + \sum_{i=1}^{n} x_i w_i$
$y = f(z)$

Neural Networks

Artificial neurons are connected together to form an *artificial neural network*

The *architecture* of an artificial neural network is the way the layers are organized



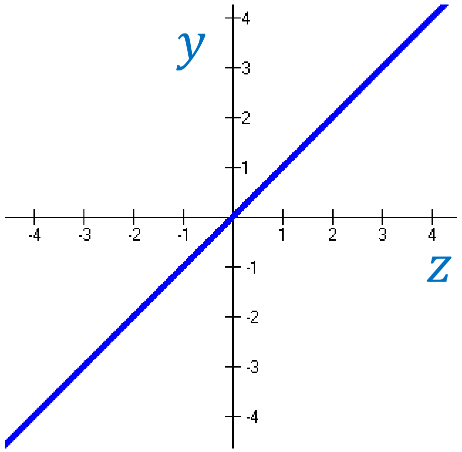
Linear Neuron

- A linear function

$$y = z$$

So,

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

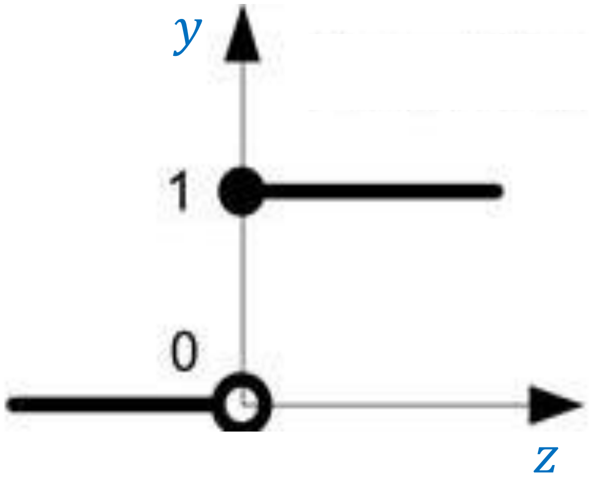


The network is reduced to a linear function

Binary Threshold Neuron

- McCulloch-Pitts (1943)

$$y = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases}$$

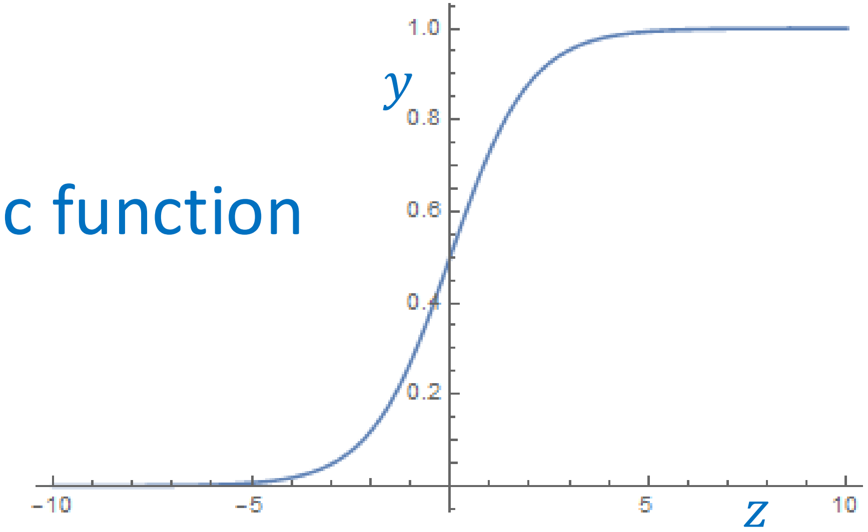


Sigmoidal Neurons

- Threshold function is a smooth bounded function of the combined input

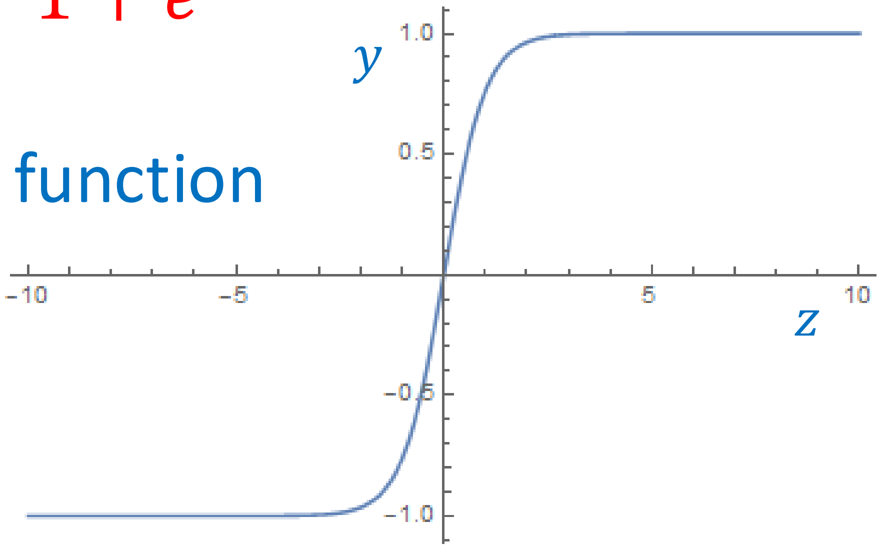
$$y = \frac{1}{1 + e^{-z}}$$

logistic function



$$y = \frac{1 - e^{-2z}}{1 + e^{-2z}}$$

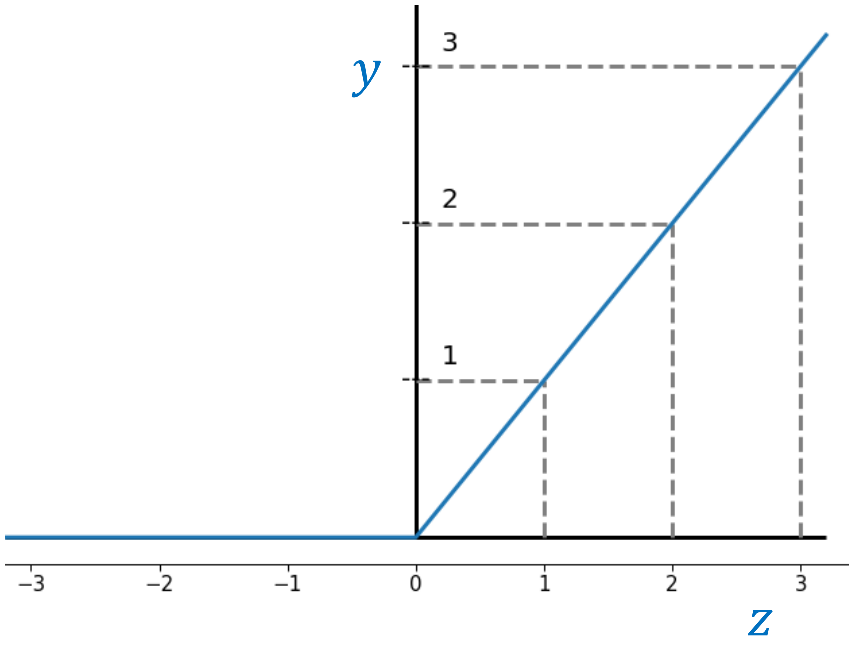
tanh function



Rectified Linear Neuron

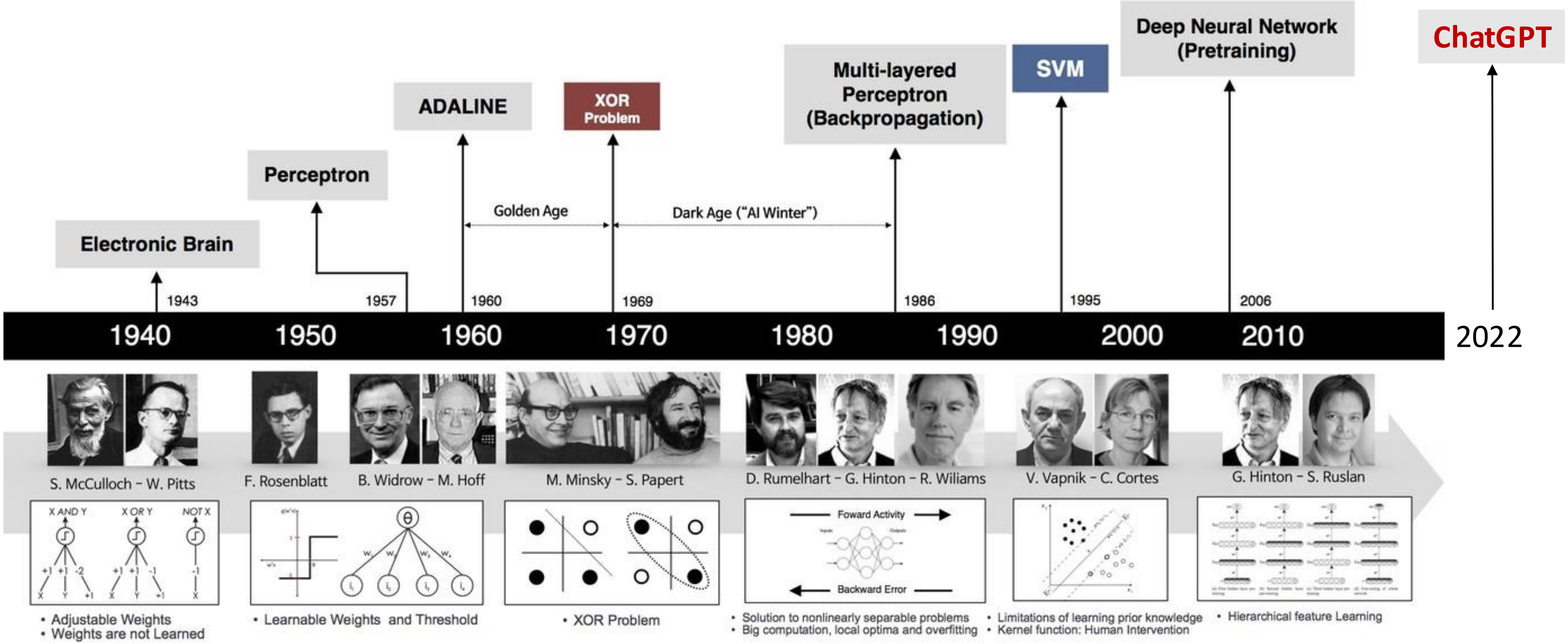
- Also called linear threshold neurons

$$y = \begin{cases} z, & z \geq 0 \\ 0, & z < 0 \end{cases}$$



Any Question so far?

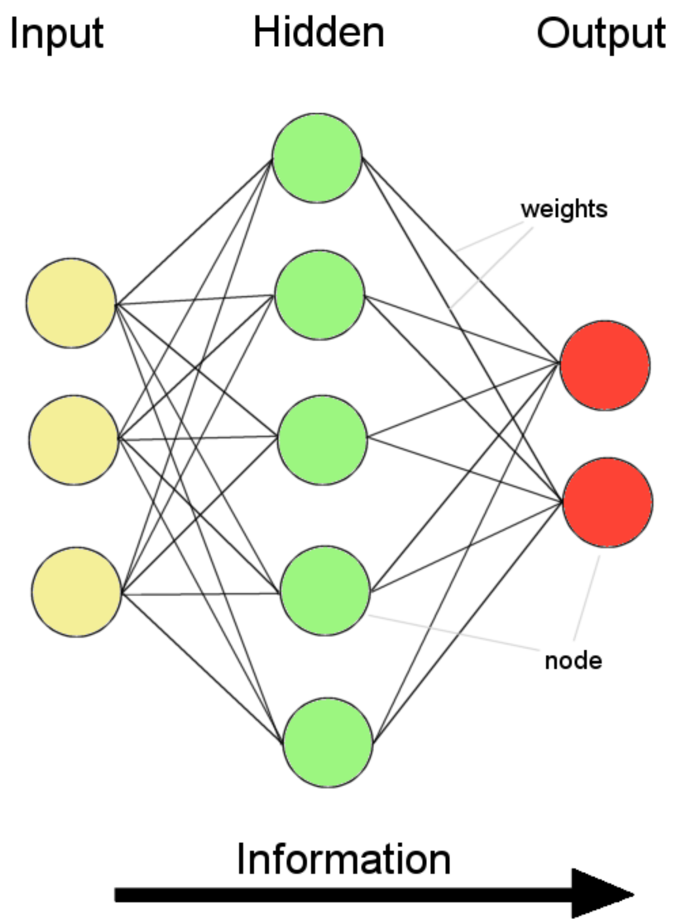
Brief History of Neural Network



Source: datascience.ibm.com

Feed-forward Networks

- Information flow is uni-directional:
  - Data is presented at the input layer
  - Passed to hidden layer(s)
  - Passed to output layer
- Information is distributed
- Information processing is parallel



Learning

- The weights need to be learned (from data) in some way so that an ANN can perform certain functions (such as classification)
  - *Supervised* learning
  - *Unsupervised* learning
  - *Reinforcement* learning (not covered here)

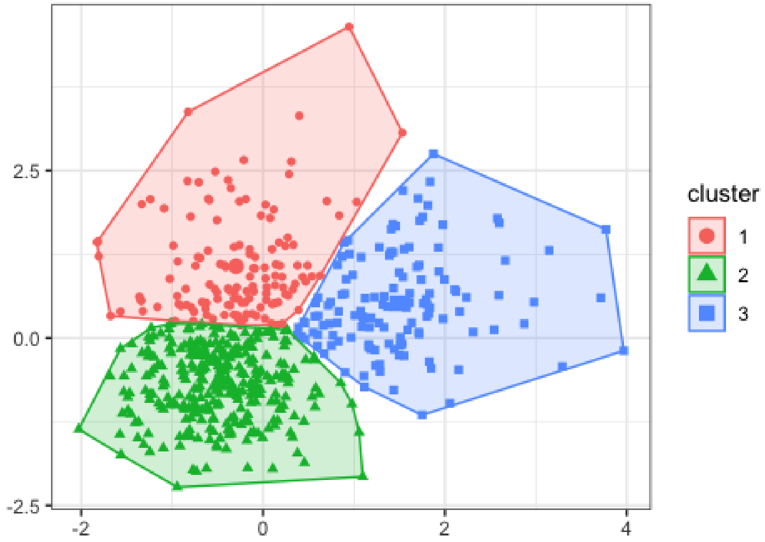
Supervised Learning

- Weights are learned from a set of training samples with inputs $\boldsymbol{x}$ and their corresponding target output(s) $\boldsymbol{y}$

- Regression – Target outputs are **real numbers**
  - Prediction based on past values, *e.g.* stock market index on Monday next week
  - x: *past values*, y: *future values*
- Classification – Target output is a **class label**
  - E.g. image recognition y=1, 2, 3 … *etc*
  - x: *an image*, y: *category of the image* (cat, dog, banana, kiwi, *etc*)

Unsupervised Learning

- Samples consists of <u>only inputs with no target</u>



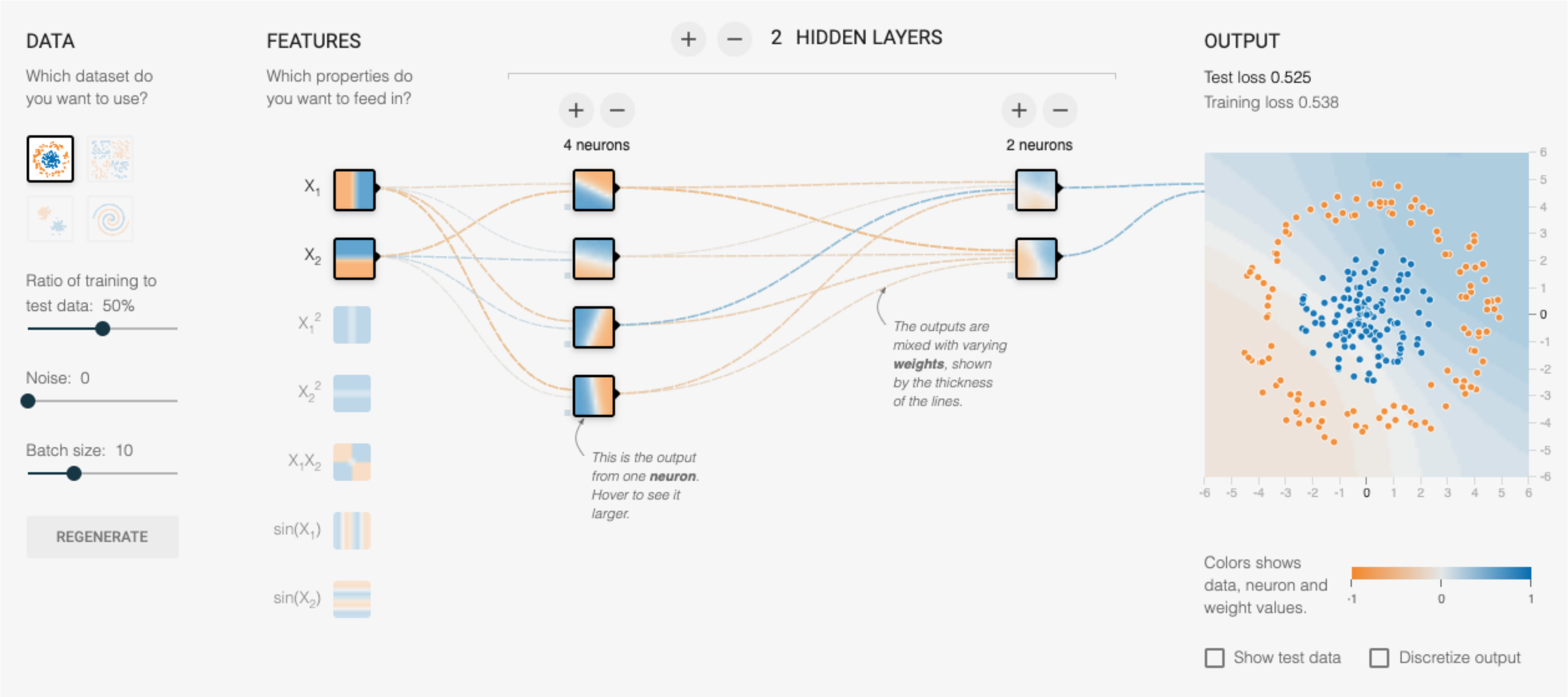
- *Clustering* – find a partition of the input data
  —— k-Means

- Low-dimensional Representation – provide a low-dimensional representation of high-dimensional inputs, *e.g.*, images

  —— Principal Component Analysis (PCA) is a commonly used linear method

ANN Playground

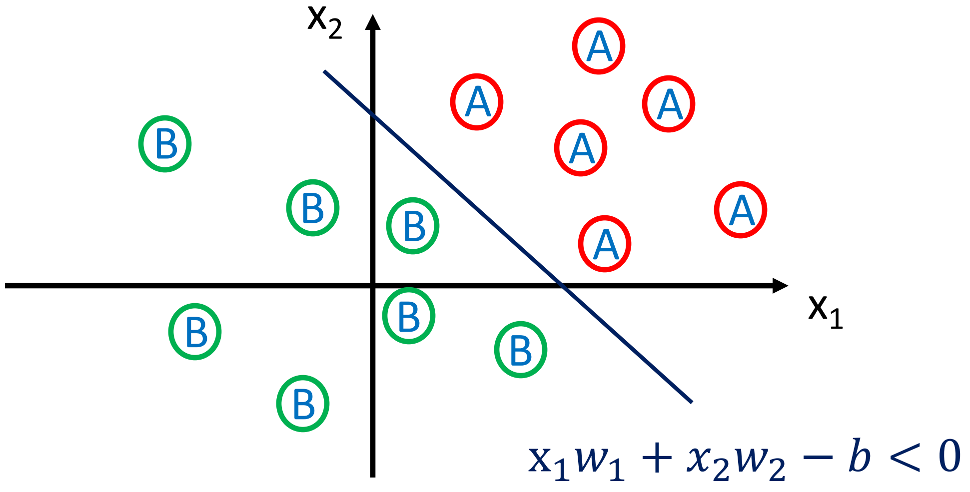
- [ANN Playground Interactions]



An example: Perceptron

- No hidden layer, having a single artificial neuron
- Based on a model of the retina
- Created by Rosenblatt
- A two-input perceptron:

$$y = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases} \quad \begin{array}{l} \Rightarrow x_1 w_1 + x_2 w_2 - b \geq 0 \\ \Rightarrow x_1 w_1 + x_2 w_2 - b < 0 \end{array}$$



Patterns must be linearly separable

Perceptron Learning Problem

Find the weights (w_1, w_2, b) so that the output error is minimized

Supervised Learning

Delta Learning Rule

- Rule for weight adjustment:

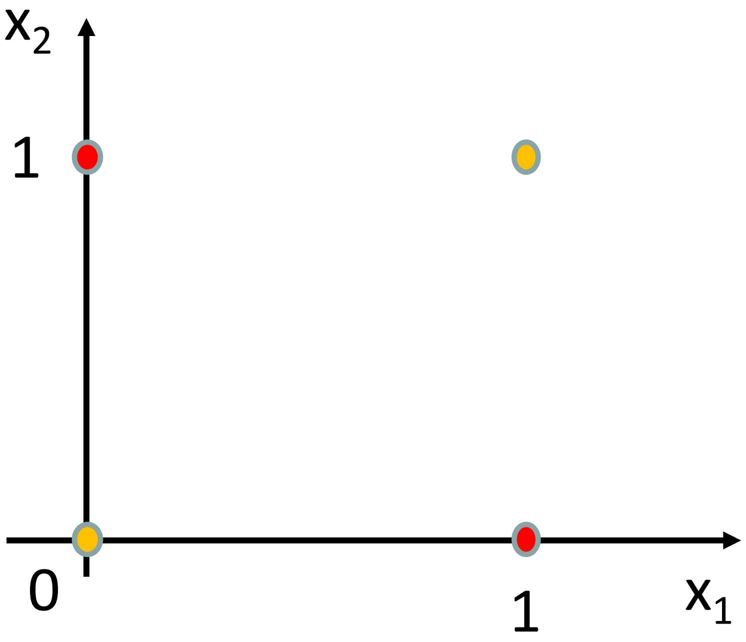
$$w_i^{\text{current}} = w_i^{\text{previous}} + \overset{\text{learning rate}}{\eta}\left(\underbrace{\underset{k\text{-th target output}}{d_i^{(k)}} - \underset{\text{computed output}}{y}}_{\text{error}}\right)\overset{k\text{-th training input}}{x_i^{(k)}} \longrightarrow \boxed{\Delta w_i}$$

- Same adjustment rule applies to the bias
- Learning rate must be carefully selected – $\boxed{0 < \eta < 1}$
  - too high leads to instability in training (does not converge)
  - too low leads to slow convergence

Is one single layer enough?

Exclusive OR

- The logical exclusive-OR (XOR) function:

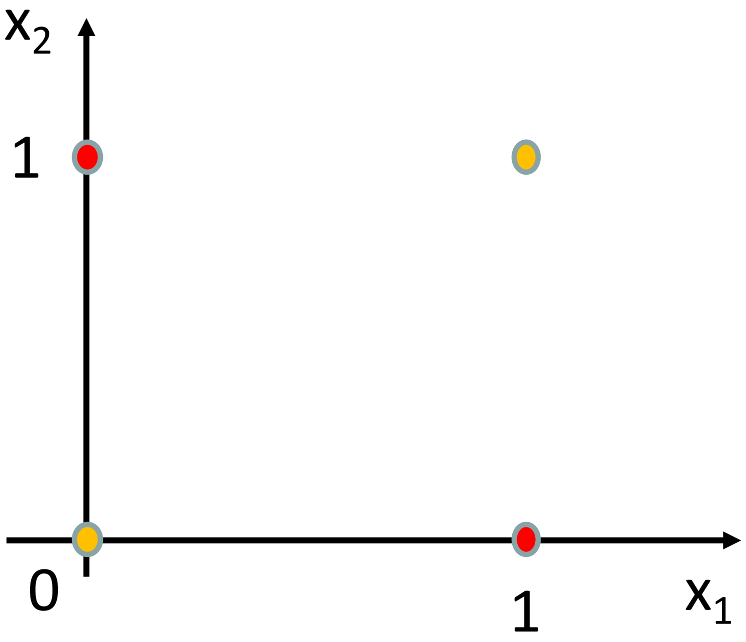


x_1	x_2	y
0	1	1
1	0	1
0	0	0
1	1	0

Truth table

Exclusive OR

- The logical exclusive-OR (XOR) function:



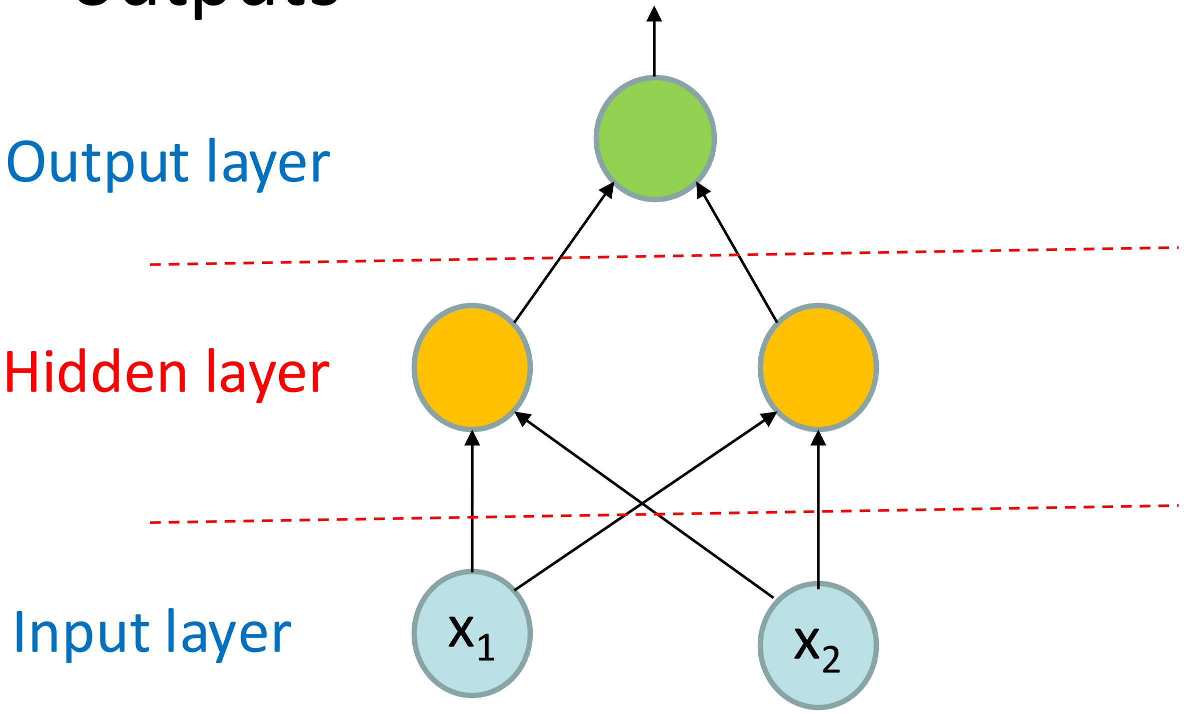
 and 

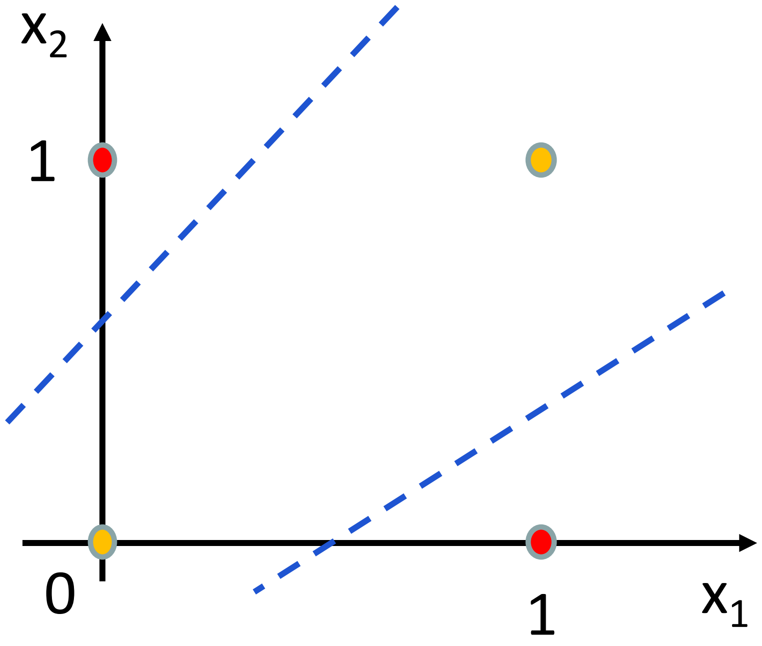
cannot be separated
by a single straight line

=> Perceptron cannot learn
this function

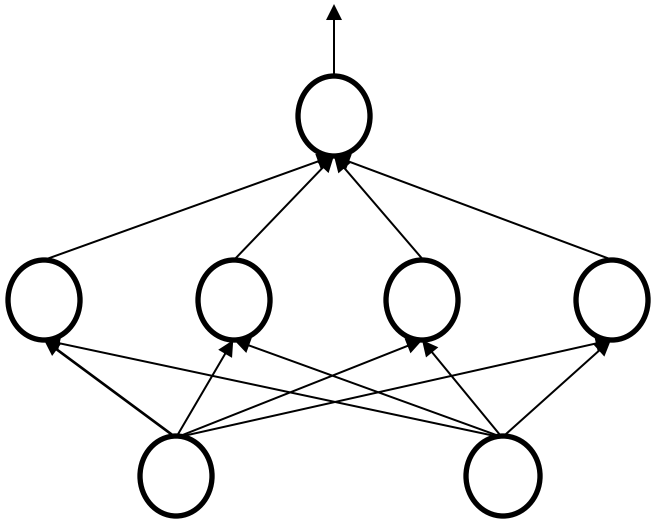
Multilayer Perceptron

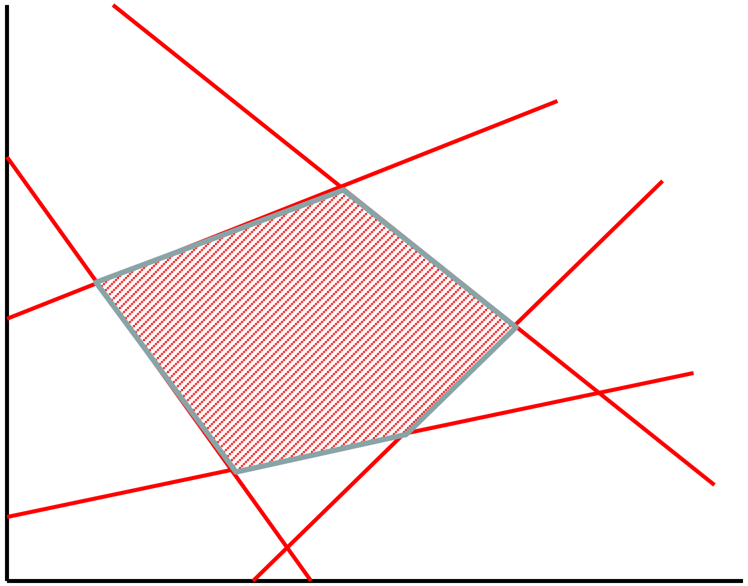
- Introduce *hidden layers* – layers in-between inputs and outputs





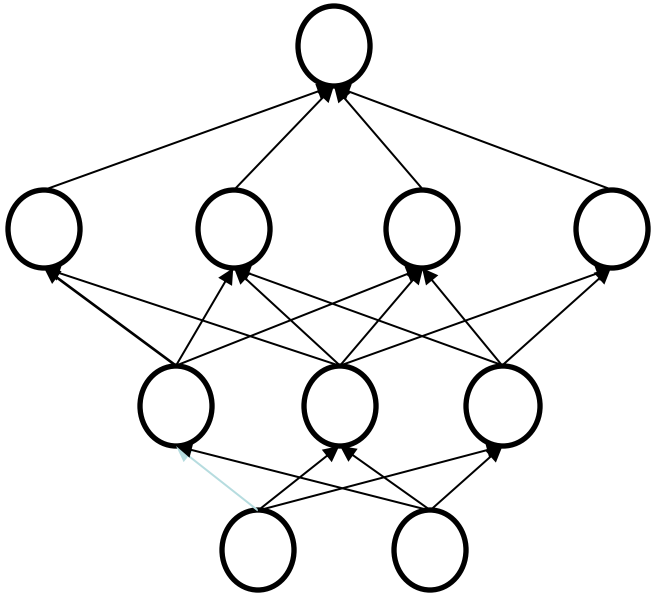
1 Hidden Layer

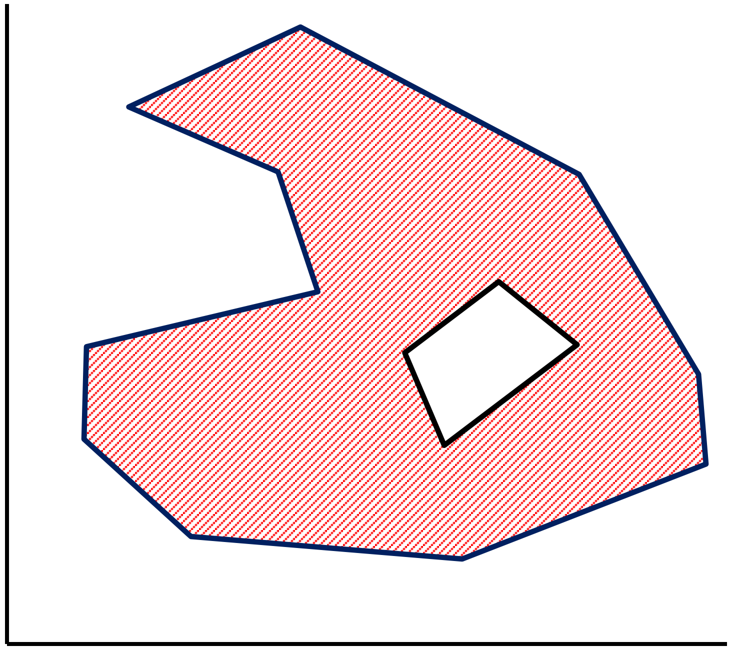




Convex polygonal region

2 Hidden Layers



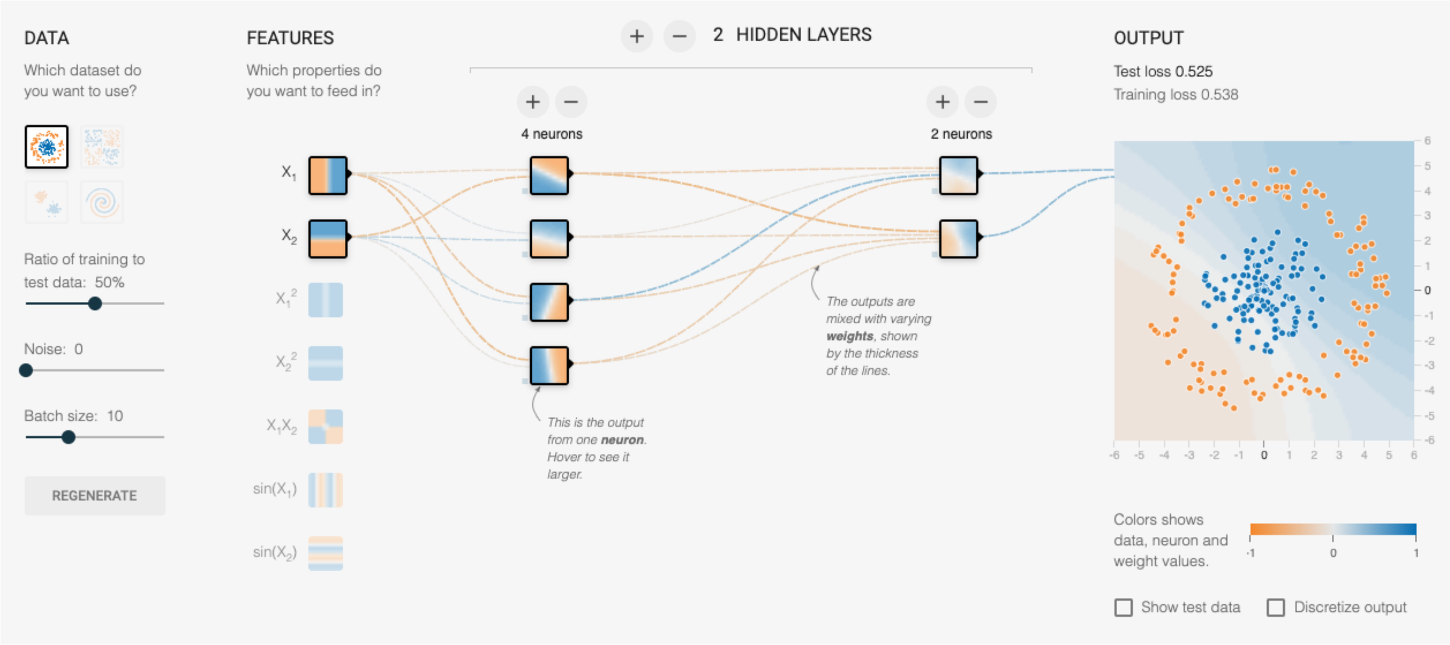


Composition of polygons

ANN Playground

- [ANN Playground Interactions]()

- Non-linear Activation
- Multiple NN Layers

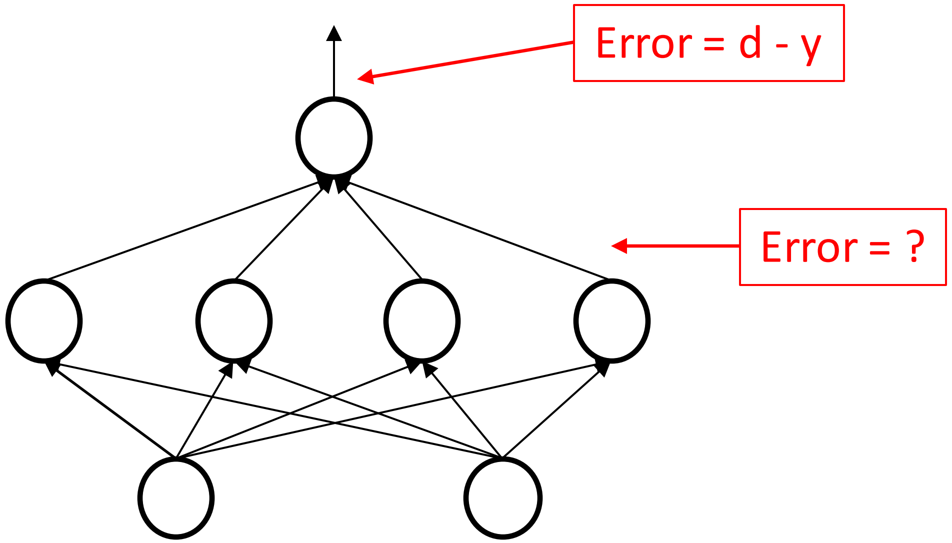


Universal Approximation Theorem

- Multilayer Feedforward Networks are universal approximators
  - Homik, Stinchcombe and White (1989)
  - 1 hidden layer is sufficient to represent any Boolean function and to approximate all bounded continuous functions
  - 2 hidden layers allow arbitrary number of labelled clusters

How to Train a Multilayer Network?

Cannot use perceptron training algorithm – Why?



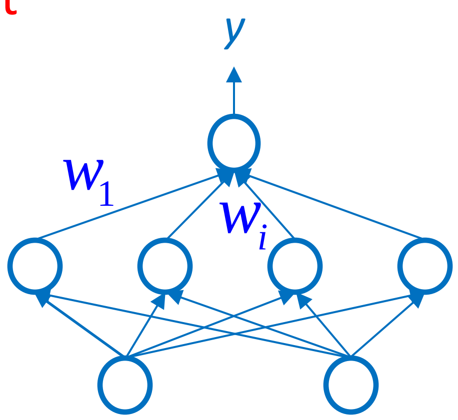
Don’t have the target outputs for the hidden units

Gradient Descent Algorithm

Squared error at the output:

$$E = \frac{1}{2}(d-y)^2 \quad \Rightarrow \frac{dE}{dy} = (d-y)$$

Slope *w.r.t.* output



Recall the Delta learning rule: $\Delta w_i = \eta(d-y)x_i$

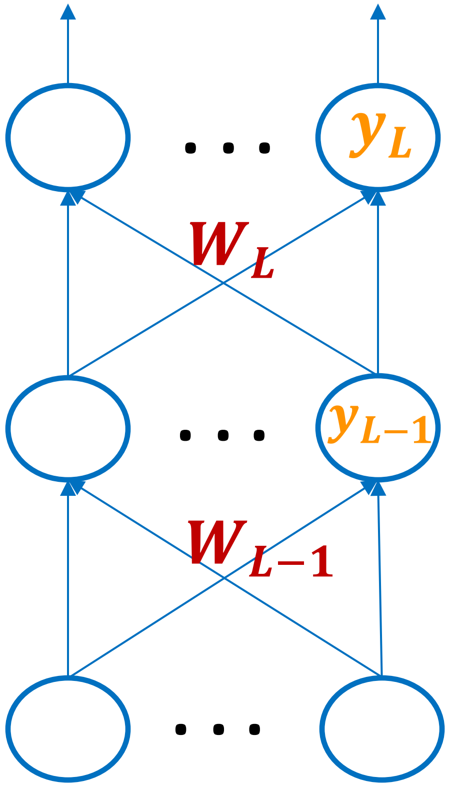
$$y = \sum_i w_i x_i \quad \Rightarrow \frac{\partial y}{\partial w_i} = x_i \qquad \text{and} \qquad \frac{\partial E}{\partial w_i} = \frac{dE}{dy} \cdot \frac{\partial y}{\partial w_i} = (d-y)x_i$$

Slope of output
w.r.t. each weight

$$\boxed{\Delta w_i = -\eta \frac{\partial E}{\partial w_i}}$$

Extend to Hidden Layer

$$E_L = \frac{1}{2}(d - y_L)^2$$



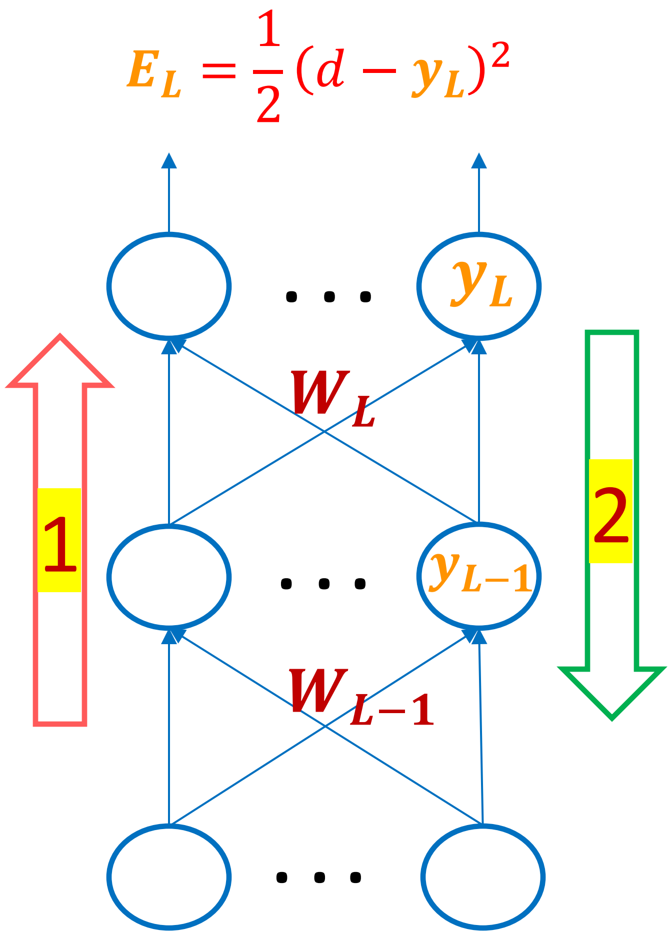
$$\frac{\partial E_L}{\partial W_L} = \frac{dE_L}{dy_L} \cdot \frac{\partial y_L}{\partial W_L} = (d - y_L)y_{L-1}$$

How about W_{L-1}?

$$\frac{\partial E_L}{\partial W_{L-1}} = \frac{dE_L}{dy_L} \cdot \frac{\partial y_L}{\partial y_{L-1}} \frac{\partial y_{L-1}}{\partial W_{L-1}} = (d - y_L)W_L y_{L-2}$$

Chain Rule

Backpropagation (BP) Algorithm



Training Pipeline:

1. **Forward** pass
   Input x $(i.e., y_0)$ with W_1 … W_L to obtain y_1 … y_L

2. **Backward** pass
   Output y_L and E_L with the chain rule to update W_1 … W_L

Iterate 1 and 2 until E_L is very small or after a certain number of iters

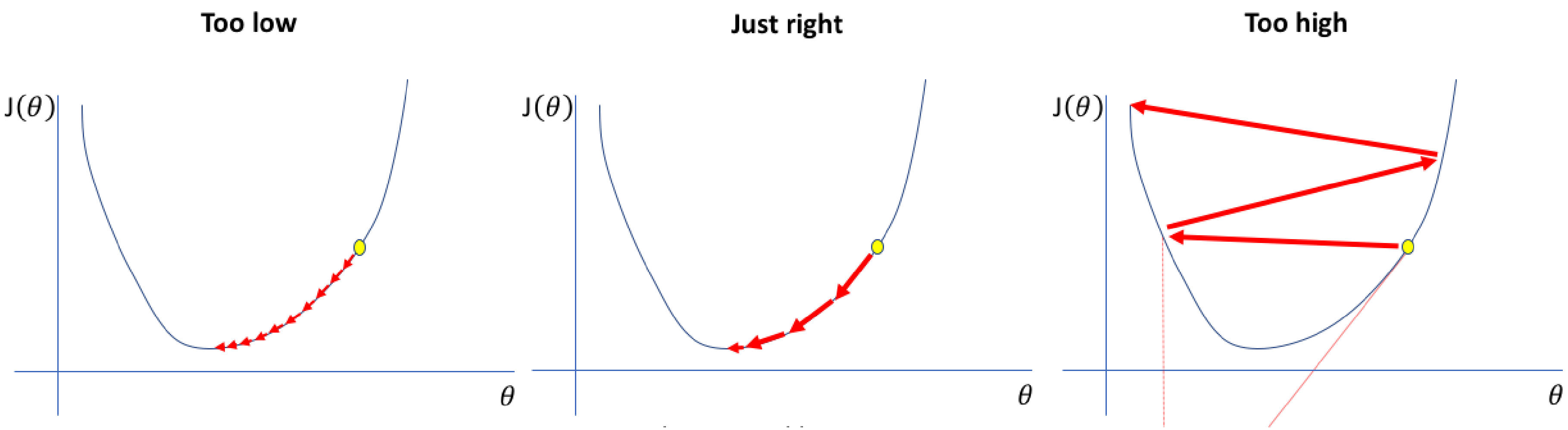
Any Question so far?

Training Practice

- Momentum
- Vanishing Gradient
- Mini-batch Updates
- Cross-entropy Loss

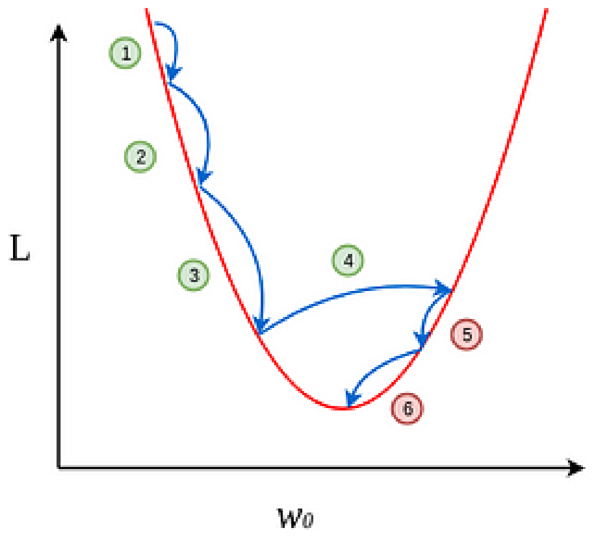
Momentum

- Convergence can be very slow
- Ways to speed up convergence:
  - Increase learning rate
    - weights may oscillate if set too high



Momentum

- Convergence can be very slow
- Ways to speed up convergence:
  - Introduce a momentum term



Keep a portion of previous change

$$\Delta w_{ij}(t) = -\eta \frac{\partial E}{\partial w_{ij}(t)} + \alpha \Delta w_{ij}(t-1)$$

Usually around 0.5

Vanishing Gradient 1

- If the weights W are large, y_j becomes large

  $f(y_j)$ approaches 1, and $f'(y_j) = \frac{\partial E_L}{\partial W_L} \rightarrow 0$

- Hence weights do not change: $W = W + \frac{\partial E_L}{\partial W_L}$

- Possible solution: Add a small constant, say 0.1, to f'

Vanishing Gradient 2

- Error signal is attenuated as it goes backwards through multiple layers
- Consequently, input-to-hidden weights learn more slowly than hidden-to-output weights

- Possible solution: use <u>different learning rates</u> for different layers

Mini-Batch Update

- Online update –
  - Weights are updated after *each* training pattern
  - Computationally demanding

- Mini-Batch update:
  - Divide training dataset into small batches (mini-batch)
  - Weights are accumulated and updated for each *mini-batch* data
  - Both efficient and more accurate

  E.g., batch_size=8, 16, …, 256

Variations of Gradient Algorithms

- Stochastic Gradient Descent (SGD) – Batch update
- Root Mean Square Propagation (RMSprop)
- Adaptive Gradient Algorithm (Adagrad)
- Adadelta – improved version of Adagrad
- Adaptive Moment Estimation (Adam) – keeps separate learning rate for each weight
- Adamax – variant of Adam

Don't worry, PyTorch has all the implementations!!!!

Cross-entropy Loss Function

- An alternative to squared error suitable for binary outputs

$$\boldsymbol{E} = \frac{1}{2}(d - \boldsymbol{y})^2$$

- Cross entropy function:

$$E = \sum_{p} \left[d^p \log \frac{d^p}{y^p} + (1 - d^p) \log \frac{1 - d^p}{1 - y^p} \right]$$

Target output

Computed output

- Heavily penalizes very wrong outputs
- Leads to faster convergence for some problems and avoids local minima

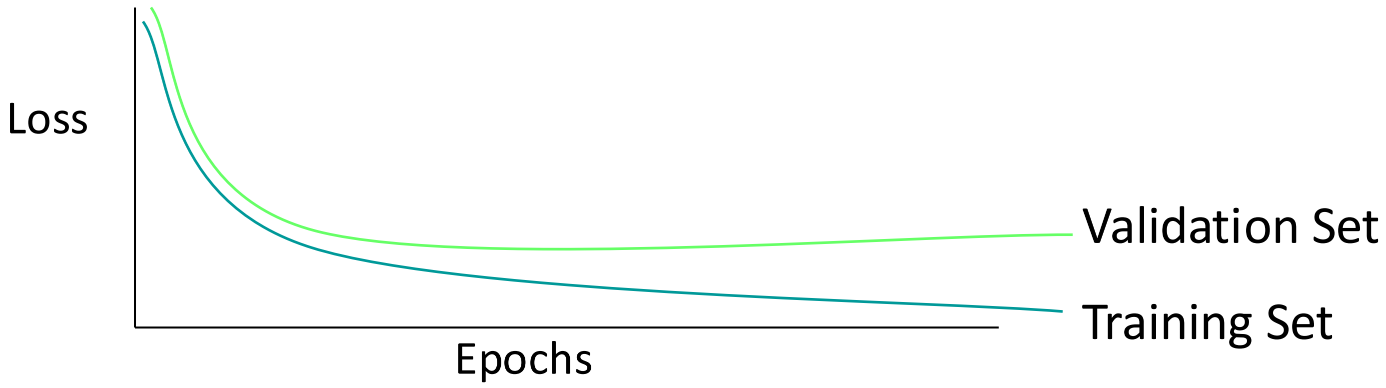
Overfitting

- The model produces very small errors for the training set but large errors for non-training (“unseen”) data

- Could be due to the model is too large for the given amount of training data

Overcome Overfitting

- Use more training data
- Use a validation set – a set of data separate from the training and test sets – stop when there is no more improvement to the validation set



Summary

- Biological VS. Artificial
- Neuron
- Multilayer Perception
- Train with Gradient Descent

  BP, Vanishing gradient, Cross-entropy
- Overfitting
- All above are implemented with PyTorch (Workshop 7)