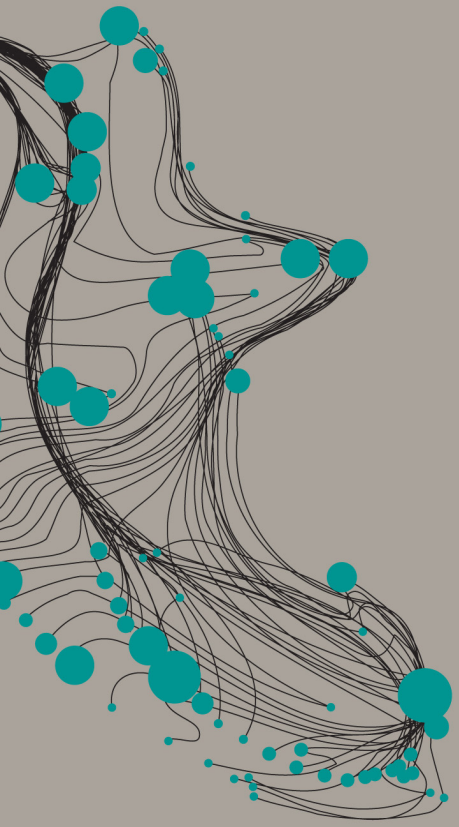
UNIVERSITY OF TWENTE.



AN INTRODUCTION TO ARTIFICIAL NEURAL NETWORKS

Mahdi KHODADADZADEH

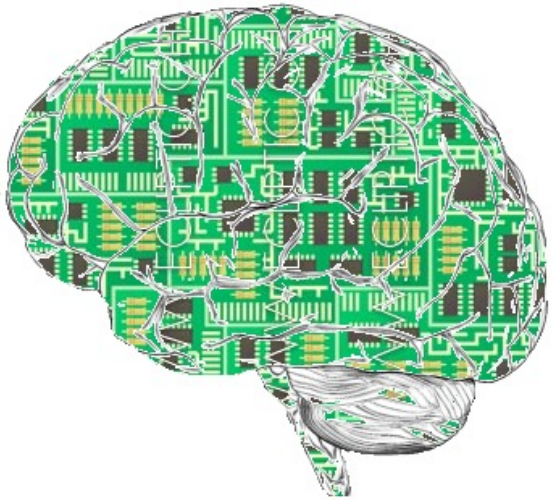
October 2021

Some materials from:
https://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15381-s06/www/nn.pdf

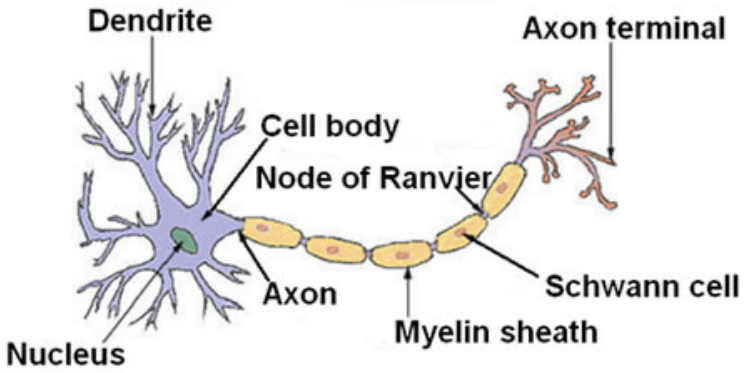
ITC FACULTY OF GEO-INFORMATION SCIENCE AND EARTH OBSERVATION

Artificial neural networks (ANNs)

ANNs are machine learning models designed to imitate the human brain.

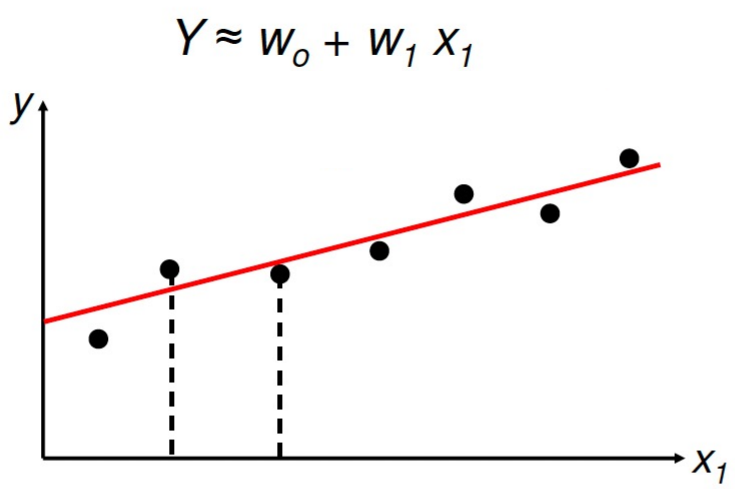


Inspired by the central nervous system and the neurons (and their axons, dendrites and synapses)



Linear Regression

- Relationship between variables is described by a Linear function
- The change of one variable causes the other variable to change
- We want to find the parameters that predict the output Y from the data X in a linear fashion

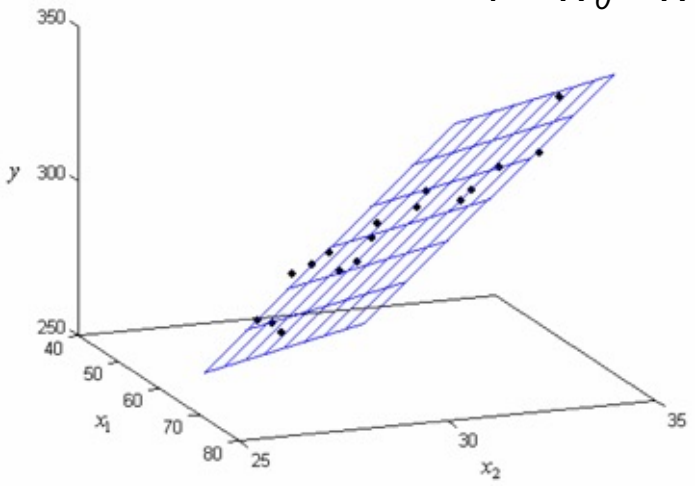


X_1	Y

Linear Regression

- Vector of attributes (feature vector) for each training data point
- We seek a vector of parameters such that we have a linear relation between prediction Y and attributes X

$$Y \approx W_0 + W_1X_1 + W_2X_2$$



X_1	X_2	Y

Linear Regression

- Vector of attributes (feature vector) for each training data point
- We seek a vector of parameters such that we have a linear relation between prediction Y and attributes X

X_1	…	X_M	Y
	…		
	…		
	…		
	…		
	…		
	…		
	…		
	…		

$$\mathbf{w} = [w_o, .., w_M]$$

$$y \approx w_o x_o + w_1 x_1 + \cdots + w_M x_M = \sum_{i=0}^{M} w_i x_i = \mathbf{w} \cdot \mathbf{x}$$

fake attribute for the input data: $x_0 = 1$

Linear Regression

- Gradient Descent

$$y \approx w_o x_o + w_1 x_1 + \cdots + w_M x_M = \sum_{i=0}^{M} w_i x_i = \mathbf{w} \cdot \mathbf{x}$$

$$E = \sum_{k=1}^{N} \left(y^k - (w_o x_o^k + w_1 x_1^k + \cdots + w_M x_M^k) \right)^2$$

$$= \sum_{k=1}^{N} \left(y^k - \mathbf{w} \cdot \mathbf{x}^k \right)^2$$

$$= \sum_{k=1}^{N} \delta_k^2 \qquad \delta_k = y^k - \mathbf{w} \cdot \mathbf{x}^k$$

$$\frac{\partial E}{\partial w_i} = -2 \sum_{k=1}^{N} \left(y^k - (w_o x_o^k + w_1 x_1^k + \cdots + w_M x_M^k) \right) x_i^k$$

$$= -2 \sum_{k=1}^{N} \left(y^k - \mathbf{w} \cdot \mathbf{x}^k \right) x_i^k$$

$$= -2 \sum_{k=1}^{N} \delta_k x_i^k$$

$$w_i \leftarrow w_i - \alpha \frac{\partial E}{\partial w_i}$$

Linear Regression

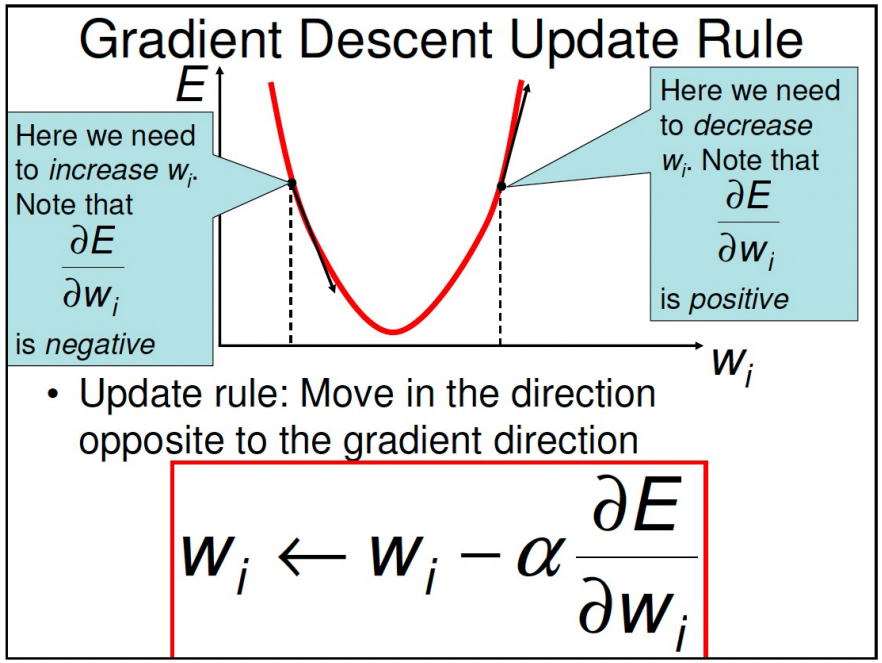
- Gradient Descent

$$y \approx w_o x_o + w_1 x_1 + \cdots + w_M x_M = \sum_{i=0}^{M} w_i x_i = \mathbf{w} \cdot \mathbf{x}$$

$$E = \sum_{k=1}^{N} \left(y^k - (w_o x_o^k + w_1 x_1^k + \cdots + w_M x_M^k) \right)^2$$

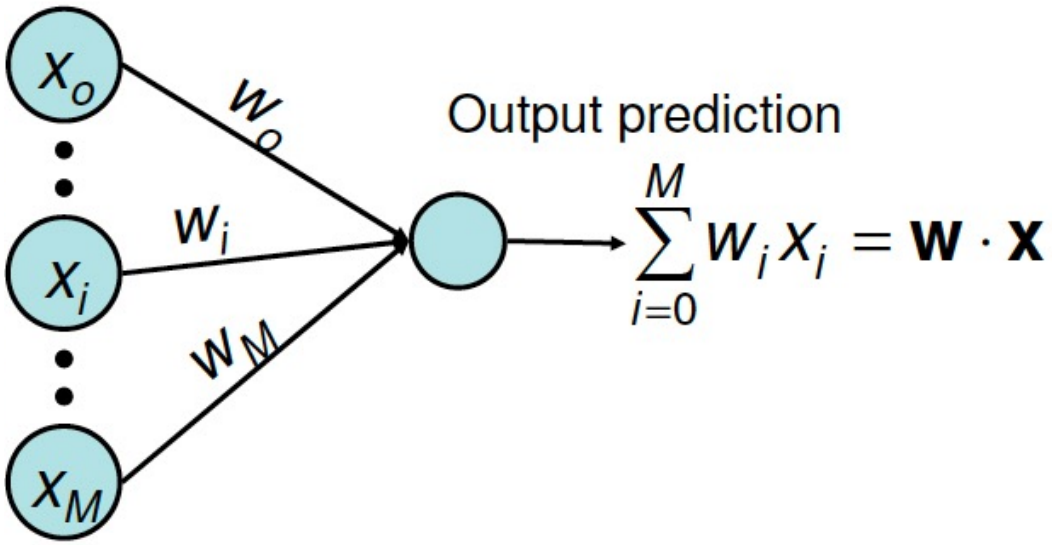
$$= \sum_{k=1}^{N} \left(y^k - \mathbf{w} \cdot \mathbf{x}^k \right)^2$$

$$= \sum_{k=1}^{N} \delta_k^2 \qquad \delta_k = y^k - \mathbf{w} \cdot \mathbf{x}^k$$



Perceptron

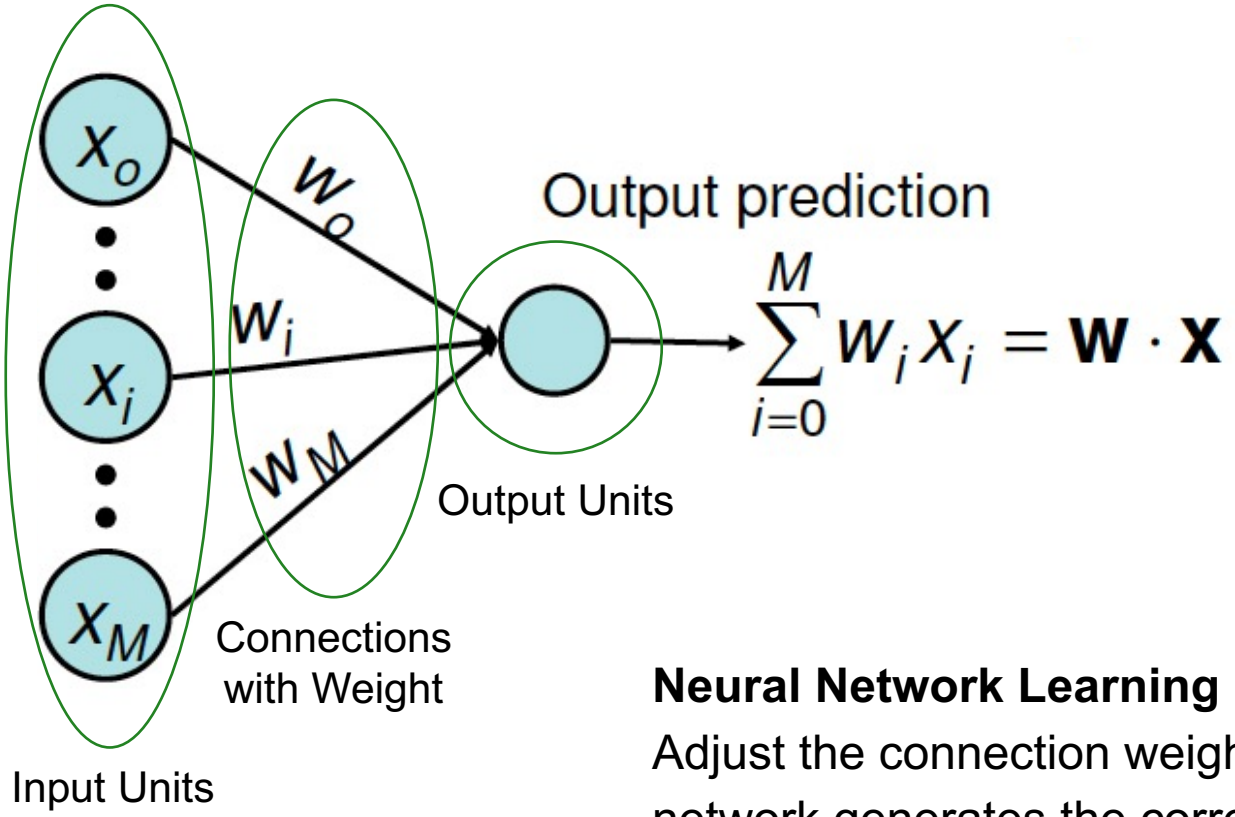
- It is also called Node or Neuron



x_1	…	x_M	y
	…		
	…		
	…		
	…		
	…		
	…		
	…		
	…		

$$y \approx w_o x_o + w_1 x_1 + \cdots + w_M x_M = \sum_{i=0}^{M} w_i x_i = \mathbf{w} \cdot \mathbf{x}$$

Perceptron



Neural Network Learning problem: Adjust the connection weights so that the network generates the correct prediction on the training data.

Perceptron Training

- Given input training data x^k with corresponding value y^k
- Compute error:

$$\delta_k \leftarrow y^k - \mathbf{w} \cdot \mathbf{x}^k$$

- Update NN weights:

$$w_i \leftarrow w_i + \alpha \delta_k x_i^k$$

α is the learning rate.
α too small: May converge slowly and may need a lot of training examples
α too large: May change **w** too quickly and spend a long time oscillating around the minimum.

Perceptron Training

- Given input training data x^k with corresponding value y^k
- Compute error:

$$\delta_k \leftarrow y^k - \mathbf{w} \cdot \mathbf{x}^k$$

- Update NN weights:

$$w_i \leftarrow w_i + \alpha \delta_k x_i^k$$

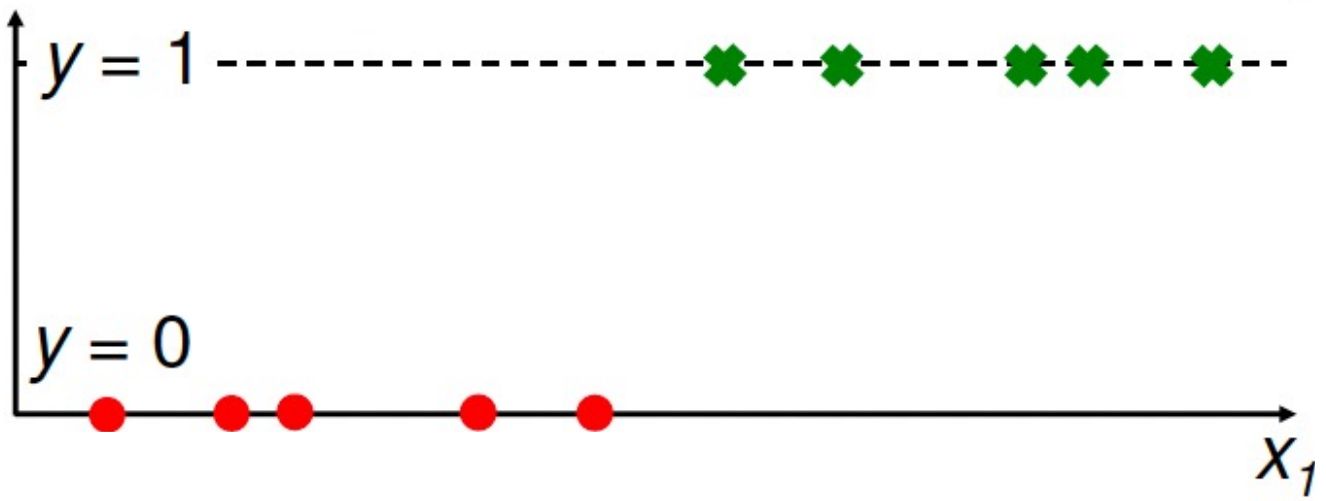
α is the learning rate.
α too small: May converge slowly and may need a lot of training examples
α too large: May change **w** too quickly and spend a long time oscillating around the minimum.

Perceptron for Classification

- Suppose that we have one attribute X_1
- Suppose that the data is in two classes (red dots and green dots)
- Given an input value X_1, we wish to predict the most likely class

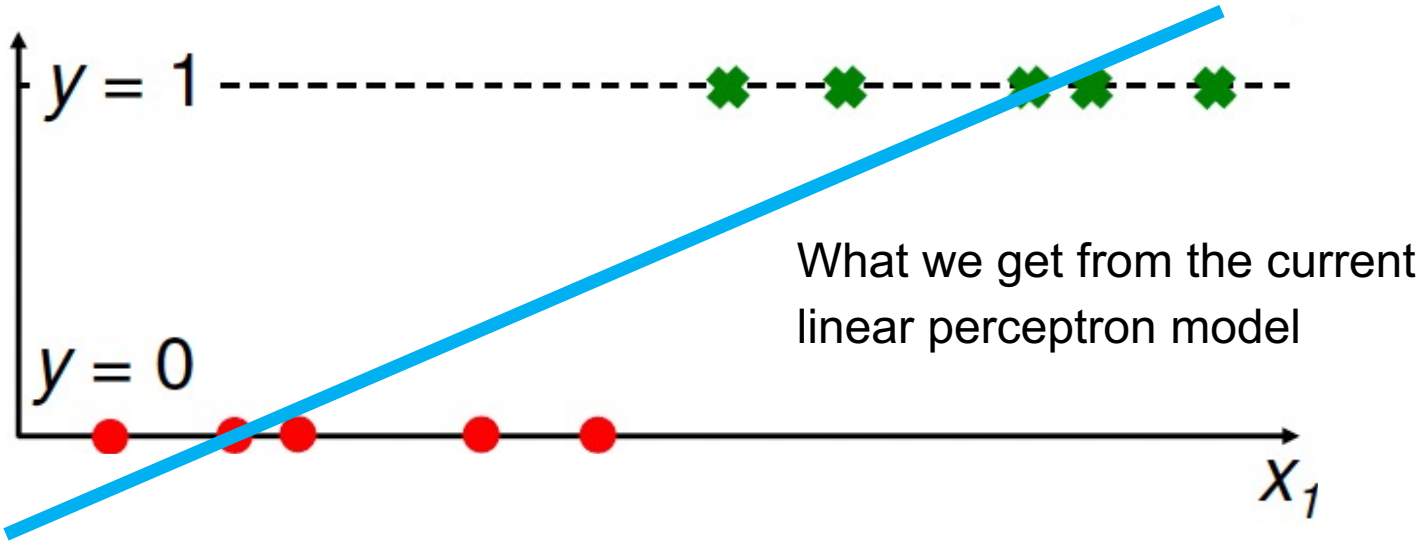
Perceptron for Classification

- We could convert it to a problem similar the regression problem



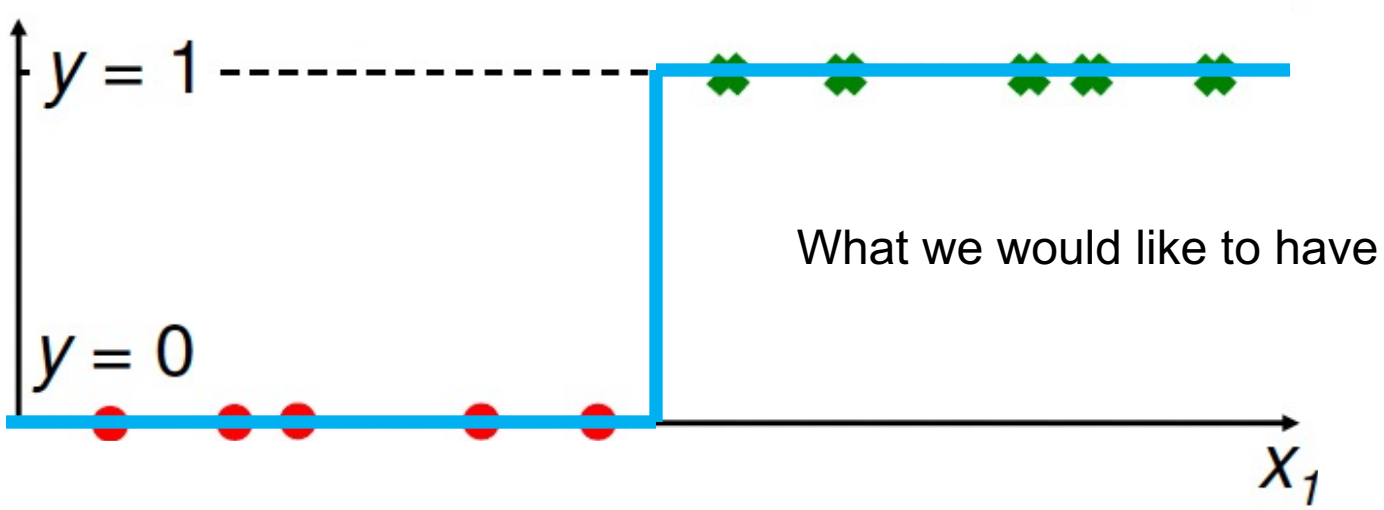
Perceptron for Classification

- We could convert it to a problem similar the regression problem



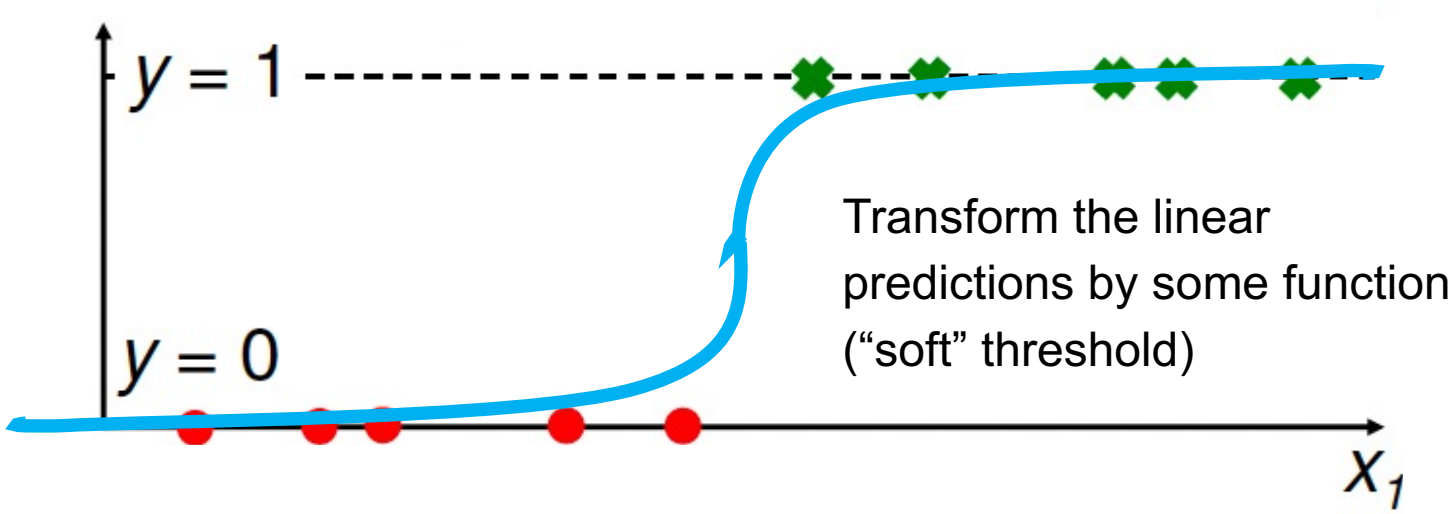
Perceptron for Classification

- We could convert it to a problem similar the regression problem



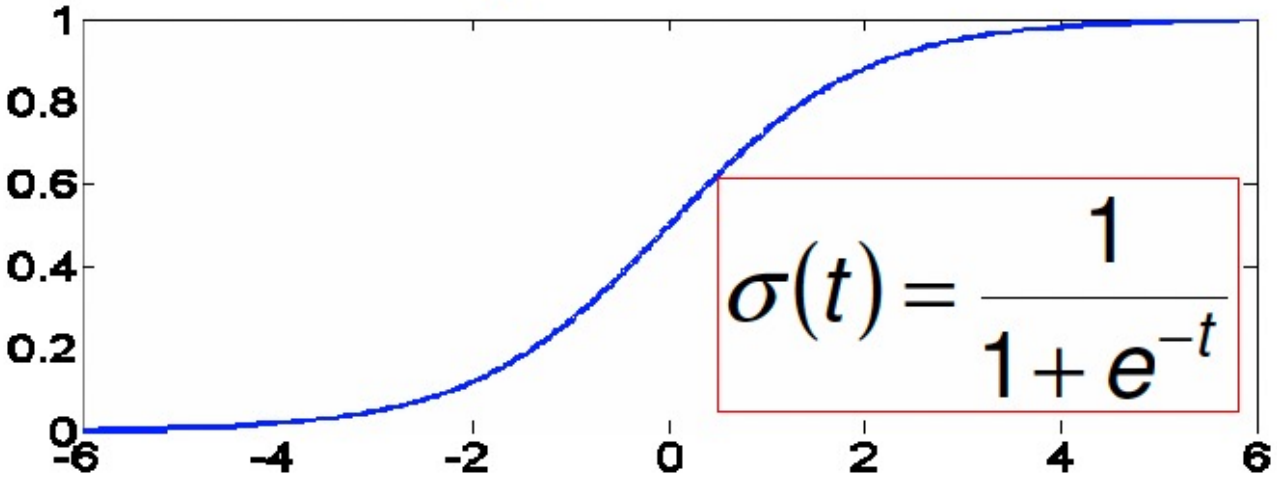
Perceptron for Classification

- We could convert it to a problem similar the regression problem



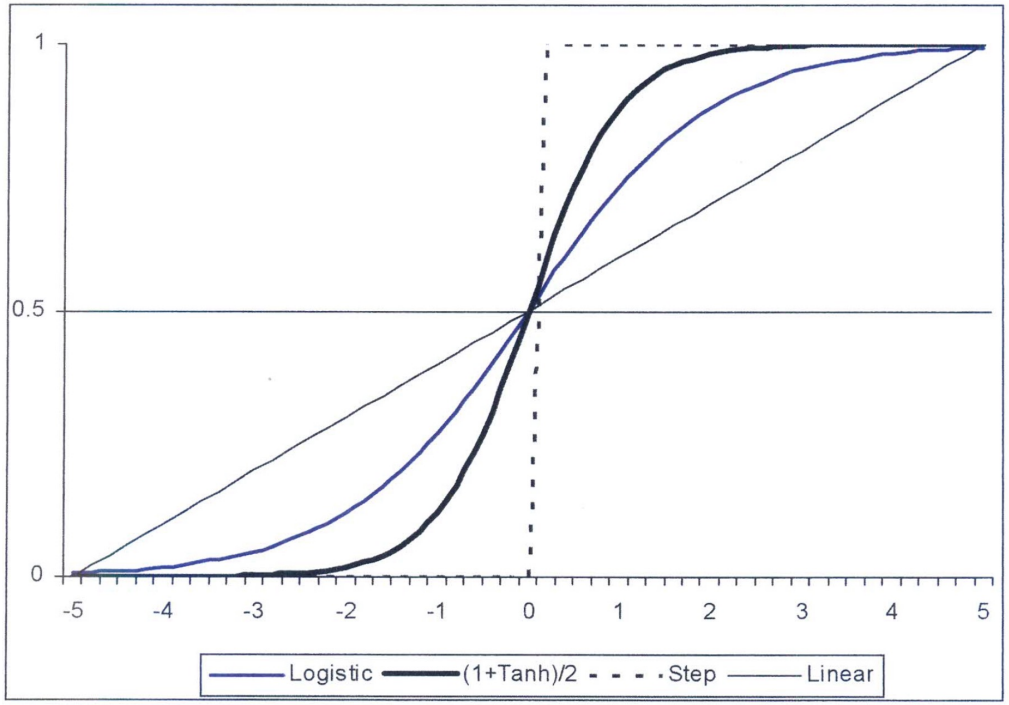
Perceptron for Classification

- The Sigmoid Function
- It approximates a hard threshold function at x = 0



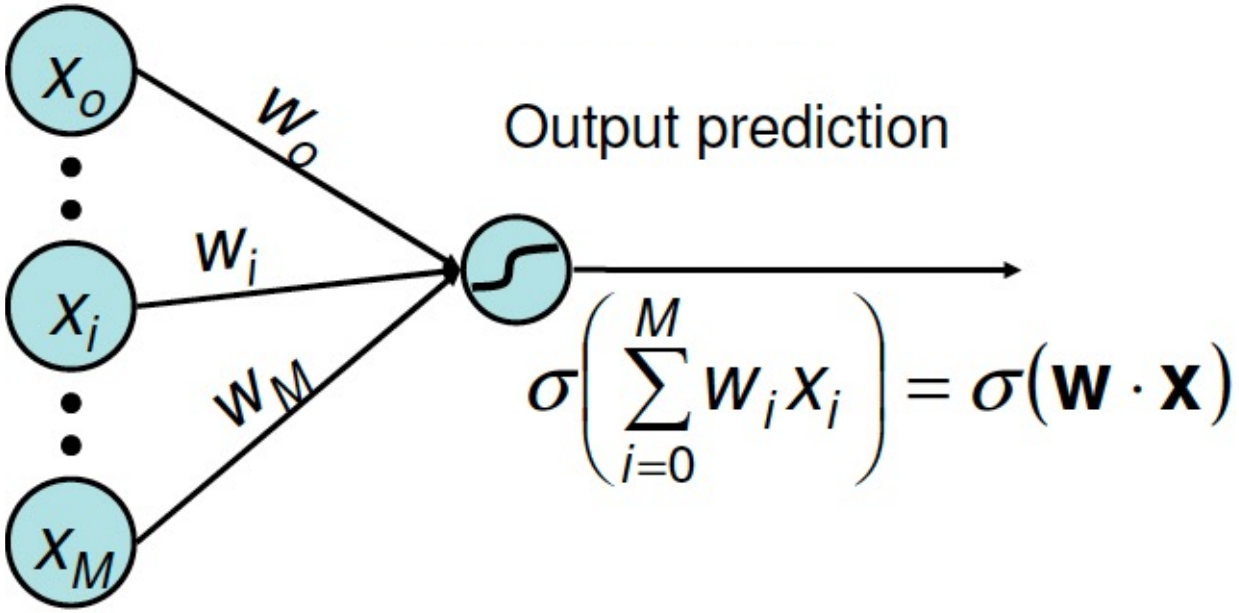
Perceptron for Classification

- Several activation functions are possible



Perceptron for Classification

- Single-layer Perceptron
- Input data are weighted, then added up and finally transformed using an activation function



Perceptron for Classification: Training

- Given input training data x^k with corresponding value y^k
- Compute error:

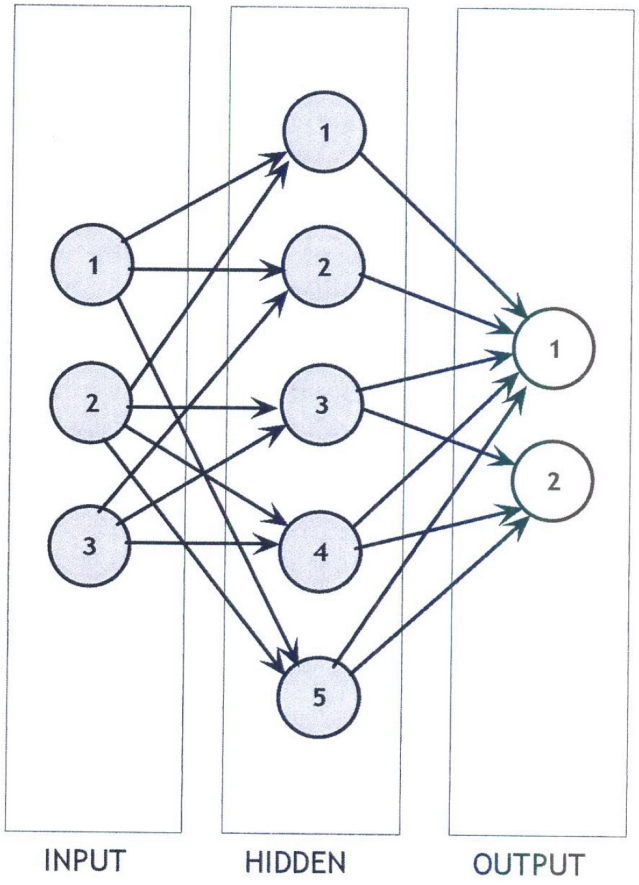
$$\delta_k \leftarrow y^k - \sigma\left(\mathbf{w} \cdot \mathbf{x}^k\right)$$

- Update NN weights:

$$w_i \leftarrow w_i + \alpha \delta_k x_i^k \sigma'\left(\mathbf{w} \cdot \mathbf{x}^k\right)$$

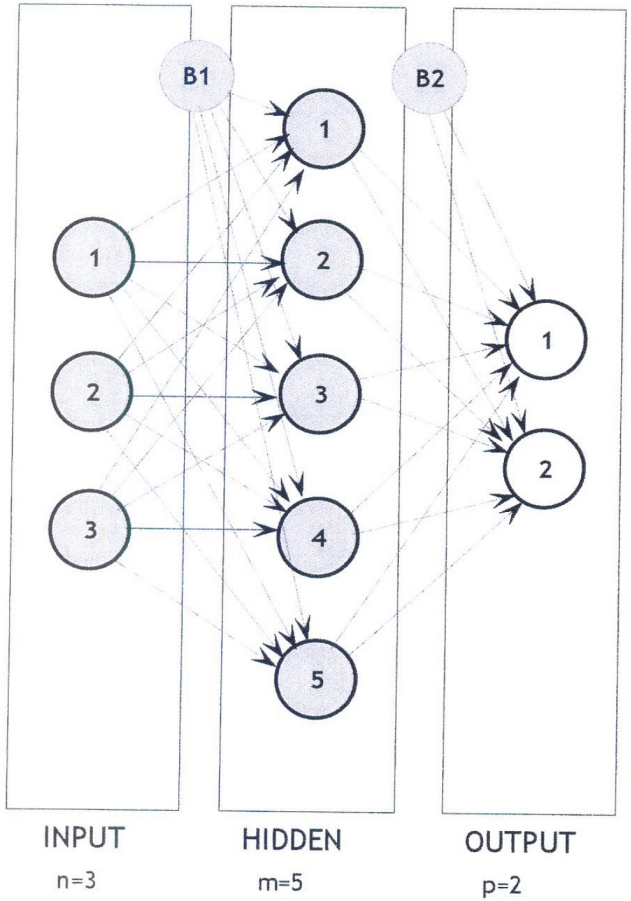
Feedforward Neural Networks

- Sequential layers
- Each layer is a set of functions
- Each layer outputs a set of vectors that serve as input to the next layer
- There are three types of layers:
  - **Input layer**: the raw input data
  - **Hidden layer(s)**: functions to apply to either inputs or outputs of previous hidden layers
  - **Output layer**: final function or set of functions.



Multi-layer perceptron (MLP)

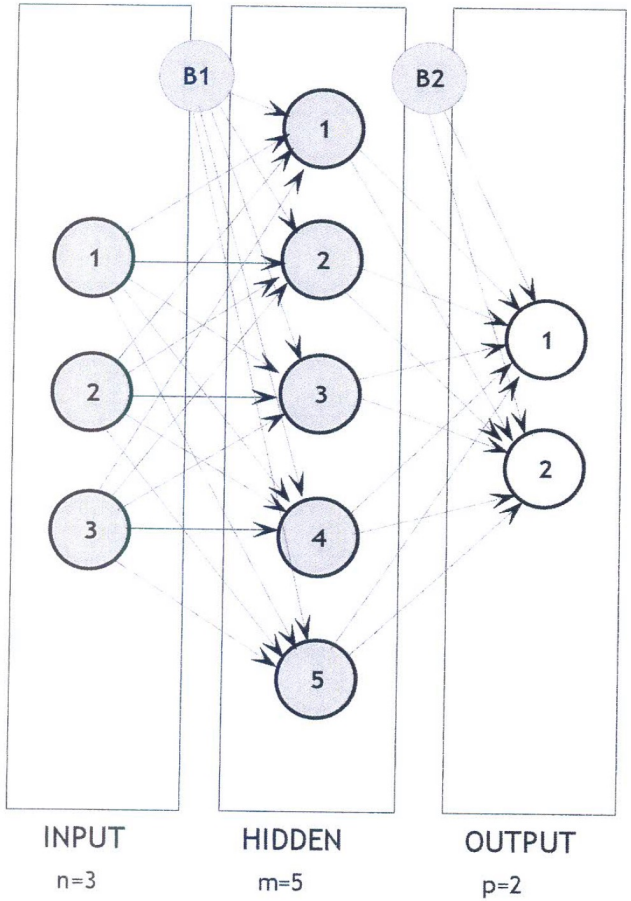
- A special case of a feedforward neural network where every layer is a **fully connected layer**
- In **MLP** each node is connected to every node in the next layer
- Nodes in the hidden and output layers are connected to a bias node (feeding a constant value ~ constant in regression models)



Multi-layer perceptron (MLP)

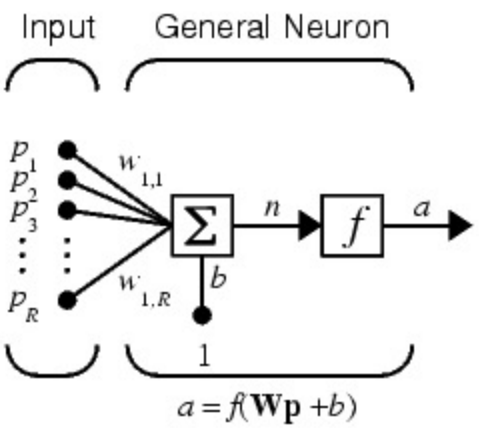
Total number of parameters to be fitted:

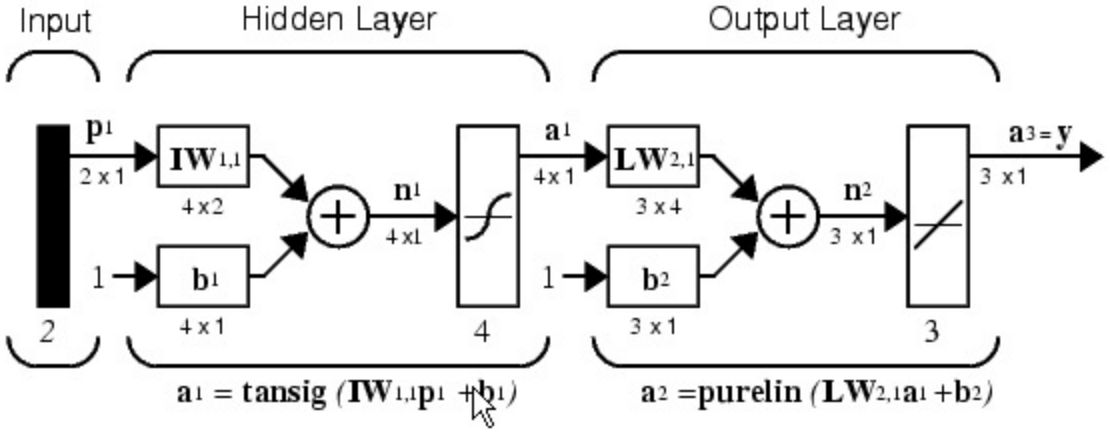
20+12 = 32

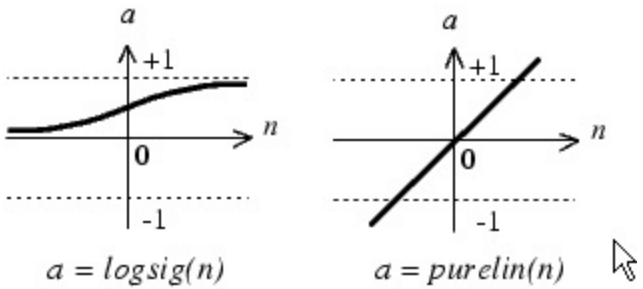


ANNs: terminology

Inputs, weights, transfer functions and epochs?



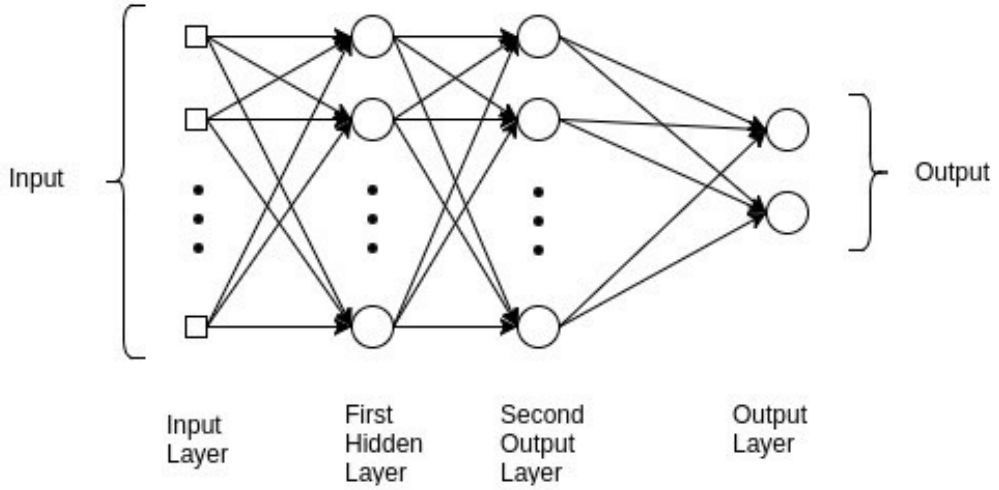




Training is done iteratively.
Every loop is known as a **epoch**

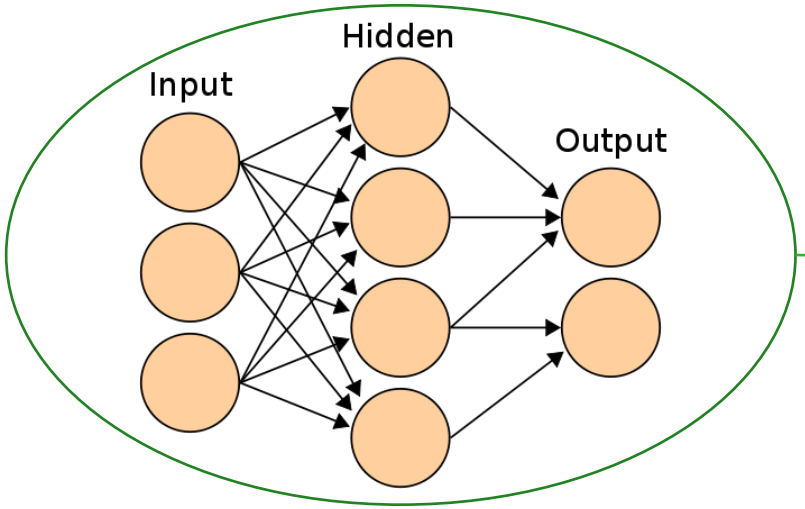
ANNs: terminology

- **Size**: The number of nodes in the model.
- **Width**: The number of nodes in a specific layer.
- **Depth**: The number of layers in a neural network.
  - The input layer is often not counted
- **Architecture**: The specific arrangement of the layers and nodes in the network.



ANNs: terminology

- The structure of an MLP can be summarized using a simple notation.
- From the input layer to the output layer: The number of nodes in each layer separated by a forward-slash character (“/”).

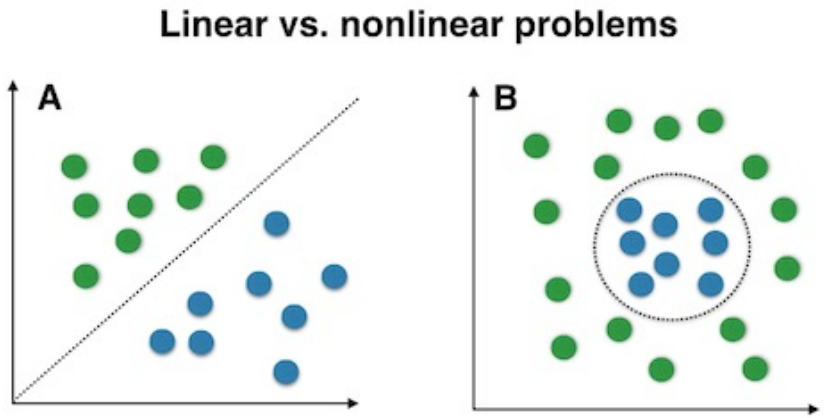


3/4/2

What about **2/8/1**?

Why Multiple Layers?

- If your problem is relatively simple, perhaps a single layer network would be sufficient.
- Most problems that we are interested in solving are not linearly separable.
- Theoretically, with one hidden layer, an MLP can approximate any function that we require



From: https://sebastianraschka.com/Articles/2014_kernel_pca.html

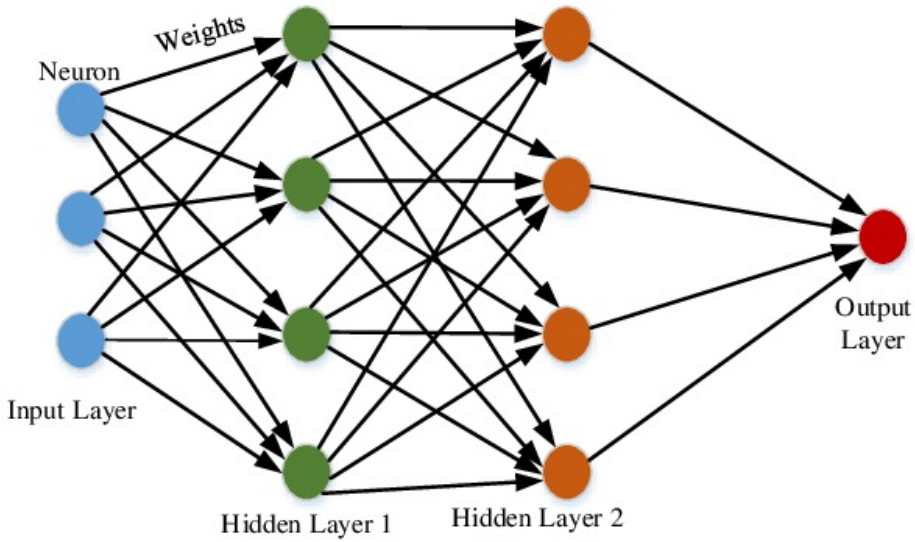
How Many Layers and Nodes to Use?

- The input layer nodes take the values of the input features
- The number of neurons comprising **the input layer is equal to the number of features (columns)** in your data
  - Some configurations add one additional node for a bias term

- The **output layer has one node for each output**
  - A single node in the case of regression or binary classification
  - K nodes (with *softmax*) in the case of K-class classification

Question

- Is the below architecture has been designed for classification or regression?
- How many input features are in the data?



Question

- We have applied an Artificial Neural Network (e.g., Multi-layer Perceptron) to a hypothetical dataset with 4 features (predictor variables) and obtained the following results.

- What would be the possible architecture (i.e., number of layers and neurons in each layer) of the applied algorithm?

Actual (True) Labels	1	1	1	2	2	2	2	3	3	3
Predicted Labels	1	2	1	1	2	2	2	3	3	1

How Many Layers and Nodes to Use?

- For the hidden layers, use systematic experimentation to discover what works best for your specific dataset!
- In general, you cannot analytically calculate the number of layers and nodes

  → **hyperparameters optimization**

- Intuition and experience
- Read the literature
- First try other classifiers (e.g., Random Forest)!

How Many Layers and Nodes to Use?

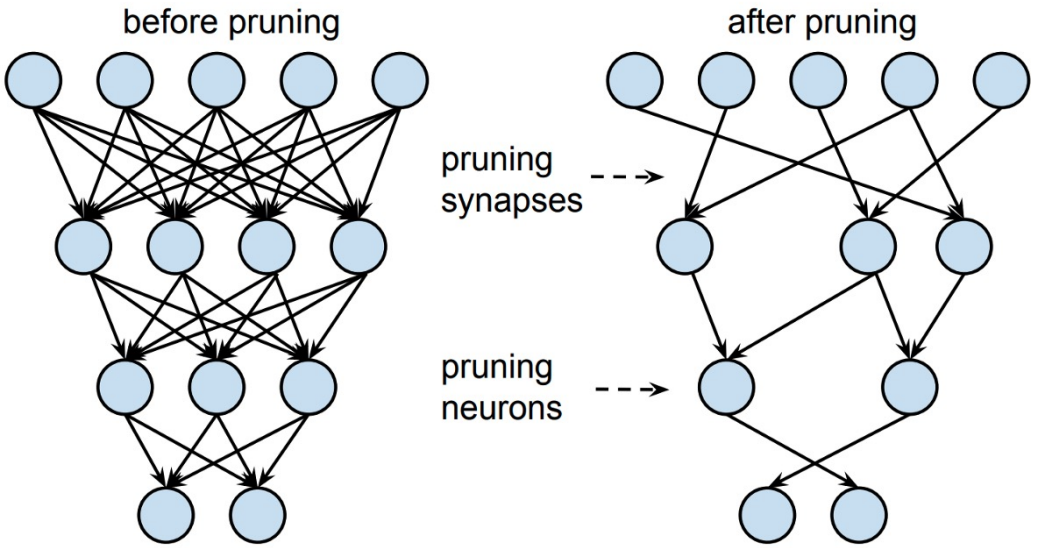
- There are some rules of thumb that may help you, e.g.:
  - The average of input and output neurons
  - Using the following equation:

$$N_h = \frac{N_s}{(\alpha * (N_i + N_o))}$$

N_i = number of input neurons.
N_o = number of output neurons.
N_s = number of samples in training data set.
α = an arbitrary scaling factor usually 2-10.

Other Concepts

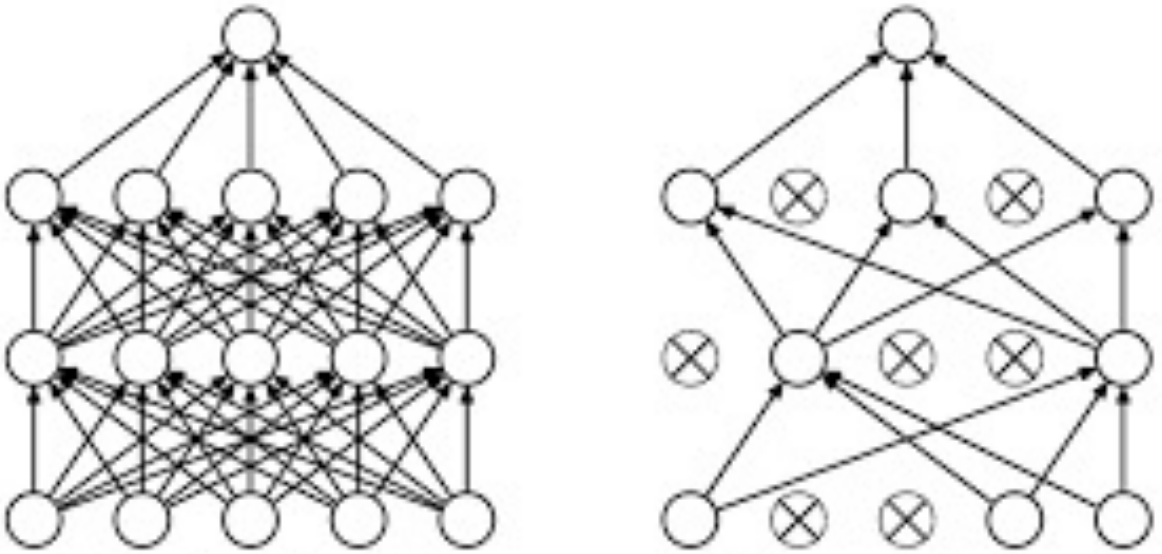
- **Pruning** → Trim network size (by nodes not layers) to improve performance
- Removes the nodes which add little predictive power
- More neurons if you add a pruning!



From: https://towardsdatascience.com/pruning-neural-networks-1bb3ab5791f9

Other Concepts

- **Dropout** → A regularization technique to prevent overfitting during the training
- Nodes are randomly selected and ignored



(a) Standard Neural Net

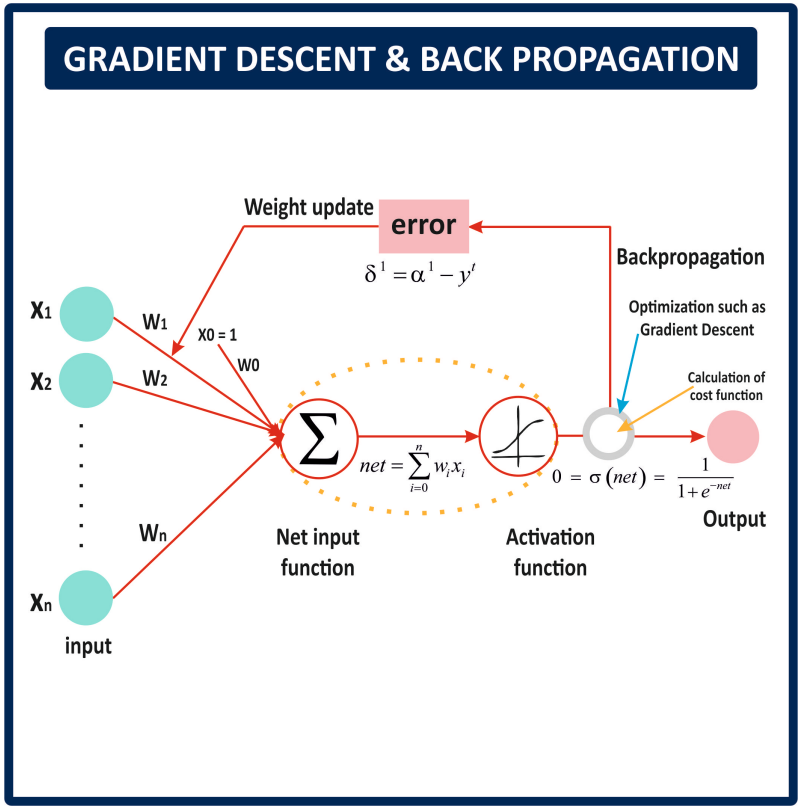
(b) After applying dropout.

Other Concepts

Backpropagation training algorithm:

Forward phase – the inputs are presented to the network

Backpropagation phase – the outputs are compared with the targets and the **weights** are adjusted



Artificial neural networks

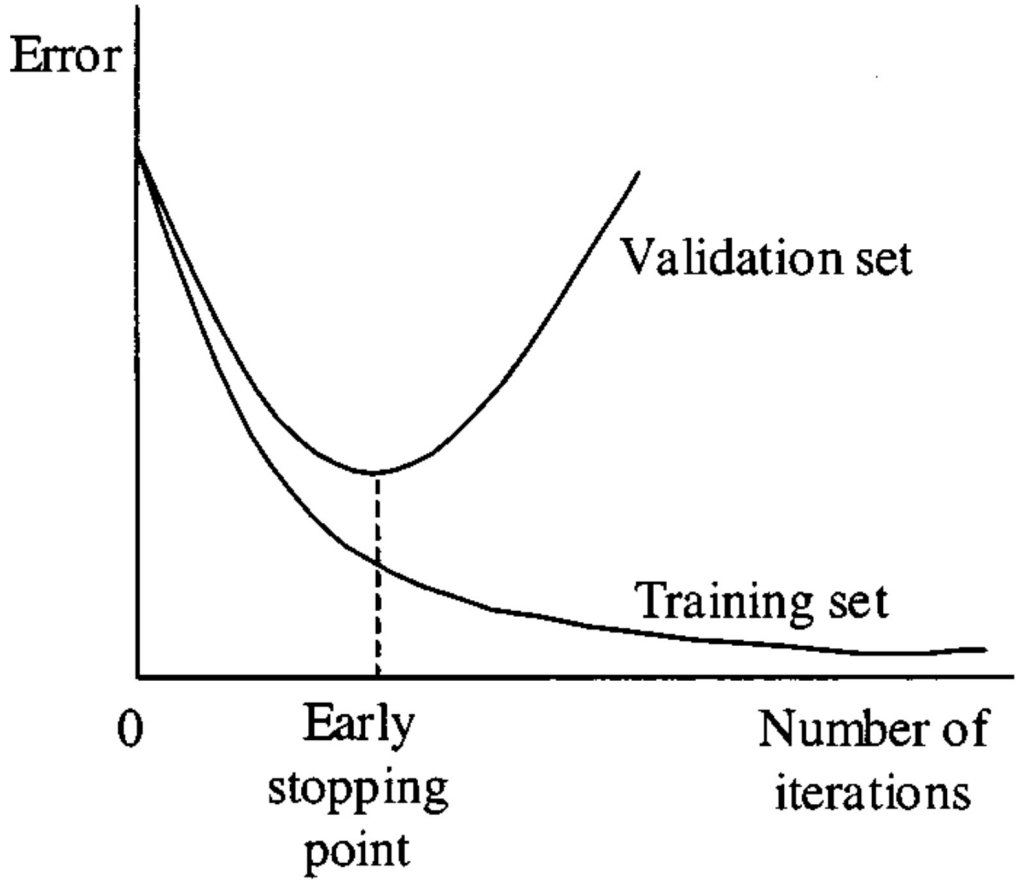
Inputs are usually scaled
This facilitates the optimization of the weights

For every input you should have an output target

The dataset is normally split into 3:

-Training → presented to the network
-Testing → early stopping to minimize the risk of "**overfitting**"
- Validation → to get some kind of error measurement (RMSE)

Artificial neural networks: overfitting



Artificial neural networks

ANN are commonly used to classify or to extract information from RS images (subtle patterns that are mostly missed when using traditional statistical tools)

ANN are non-linear so thay can handle complex problems and generalize a solution set
The selection of the parameters is the hardest part (how many nodes/layers/type of transfer function?)

Trained ANN are very fast!
ANN can take as inputs continuous, near-continuos and categorical data without violating any model assumption.

Artificial neural networks

sklearn.neural_network.MLPRegressor

class sklearn.neural_network.MLPRegressor(*hidden_layer_sizes=(100)*, *activation='relu'*, ***, *solver='adam'*, *alpha=0.0001*, *batch_size='auto'*, *learning_rate='constant'*, *learning_rate_init=0.001*, *power_t=0.5*, *max_iter=200*, *shuffle=True*, *random_state=None*, *tol=0.0001*, *verbose=False*, *warm_start=False*, *momentum=0.9*, *nesterovs_momentum=True*, *early_stopping=False*, *validation_fraction=0.1*, *beta_1=0.9*, *beta_2=0.999*, *epsilon=1e-08*, *n_iter_no_change=10*, *max_fun=15000*) [source]

Multi-layer Perceptron regressor.

This model optimizes the squared error using LBFGS or stochastic gradient descent.

New in version 0.18.

Parameters:

hidden_layer_sizes : ***tuple, length = n_layers - 2, default=(100,)***
The ith element represents the number of neurons in the ith hidden layer.

activation : ***{'identity', 'logistic', 'tanh', 'relu'}, default='relu'***
Activation function for the hidden layer.

- 'identity', no-op activation, useful to implement linear bottleneck, r
- 'logistic', the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x
- 'tanh', the hyperbolic tan function, returns f(x) = tanh(x).
- 'relu', the rectified linear unit function, returns f(x) = max(0, x)

solver : ***{'lbfgs', 'sgd', 'adam'}, default='adam'***
The solver for weight optimization.

sklearn.neural_network.MLPClassifier

class sklearn.neural_network.MLPClassifier(*hidden_layer_sizes=(100)*, *activation='relu'*, ***, *solver='adam'*, *alpha=0.0001*, *batch_size='auto'*, *learning_rate='constant'*, *learning_rate_init=0.001*, *power_t=0.5*, *max_iter=200*, *shuffle=True*, *random_state=None*, *tol=0.0001*, *verbose=False*, *warm_start=False*, *momentum=0.9*, *nesterovs_momentum=True*, *early_stopping=False*, *validation_fraction=0.1*, *beta_1=0.9*, *beta_2=0.999*, *epsilon=1e-08*, *n_iter_no_change=10*, *max_fun=15000*) [source]

Multi-layer Perceptron classifier.

This model optimizes the log-loss function using LBFGS or stochastic gradient descent.

New in version 0.18.

Parameters:

hidden_layer_sizes : ***tuple, length = n_layers - 2, default=(100,)***
The ith element represents the number of neurons in the ith hidden layer.

activation : ***{'identity', 'logistic', 'tanh', 'relu'}, default='relu'***
Activation function for the hidden layer.

- 'identity', no-op activation, useful to implement linear bottleneck, returns f(x) = x
- 'logistic', the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)).
- 'tanh', the hyperbolic tan function, returns f(x) = tanh(x).
- 'relu', the rectified linear unit function, returns f(x) = max(0, x)

solver : ***{'lbfgs', 'sgd', 'adam'}, default='adam'***
The solver for weight optimization.

Down