

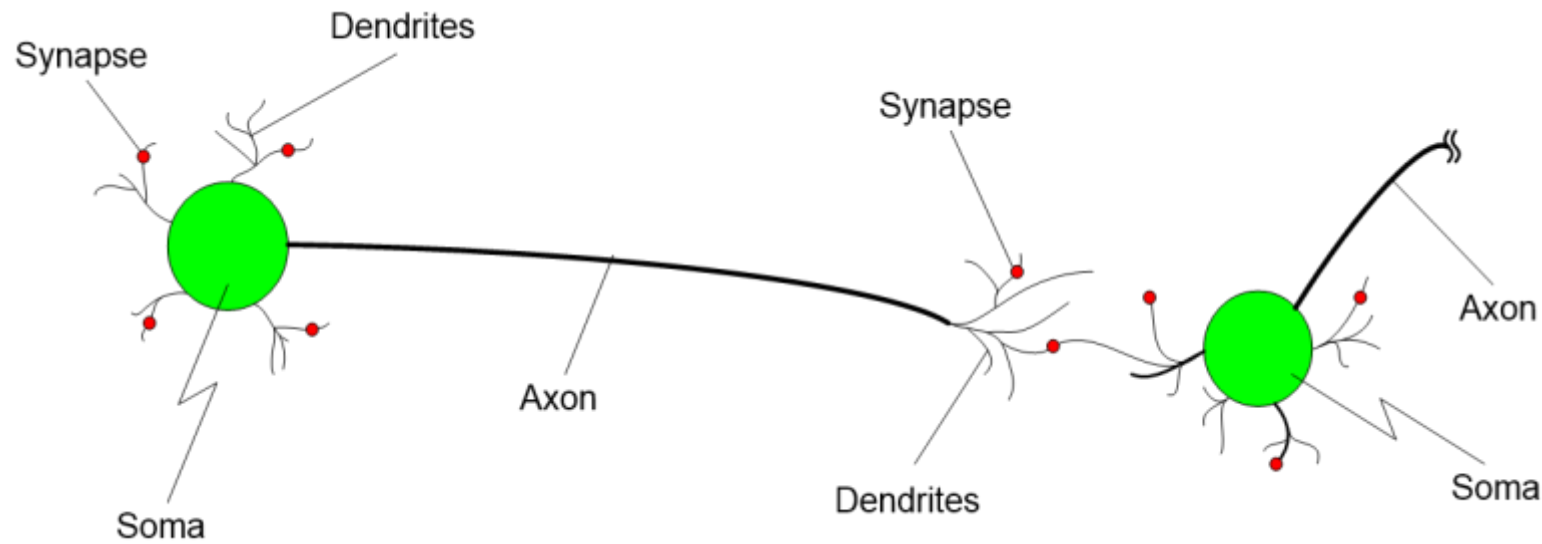
COSC 3337 : Data Science I



N. Rizk

College of Natural and Applied Sciences
Department of Computer Science
University of Houston

Artificial Neural Networks for Data Mining



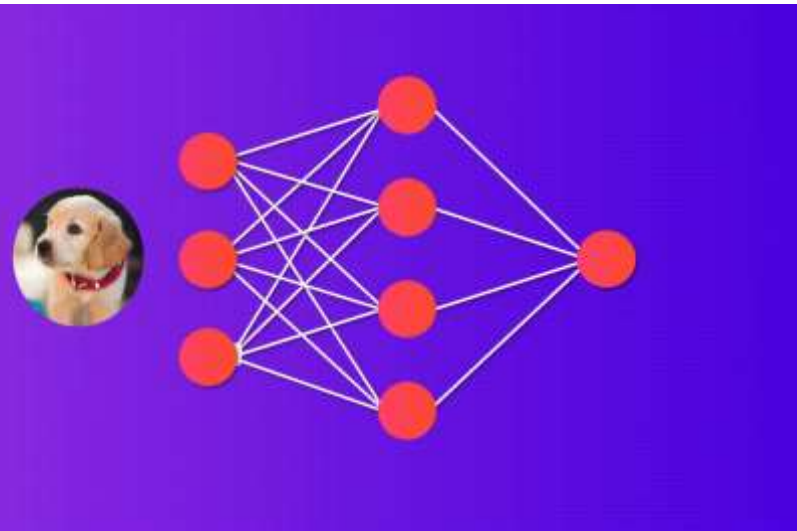
Two interconnected brain cells (neurons)
Biological Neural Networks

Opening Vignette:

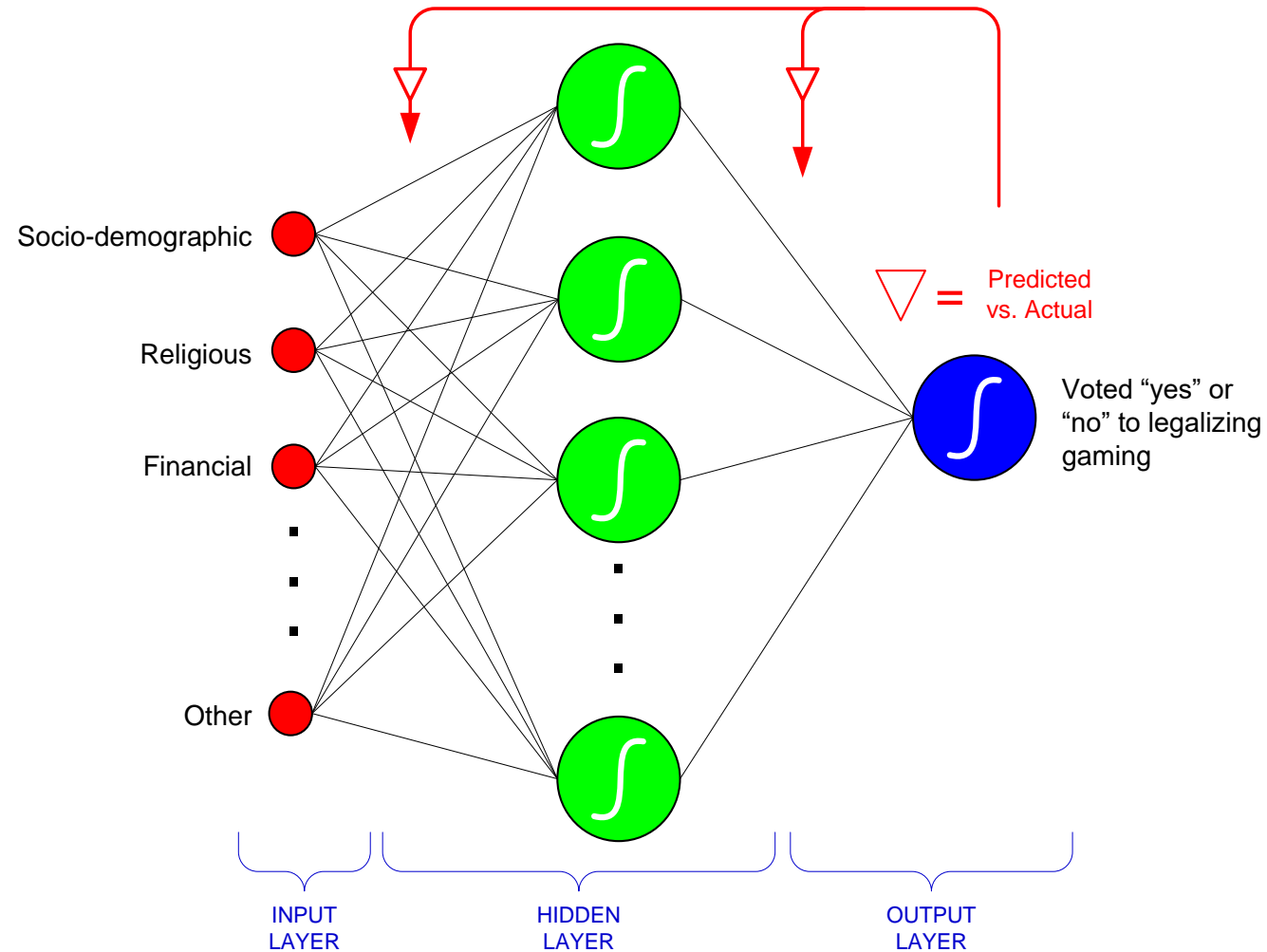


“Predicting Gambling Referenda with Neural Networks”

- Decision situation
- Proposed solution
- Results
- Answer and discuss the case questions



Opening Vignette: Predicting Gambling Referenda...

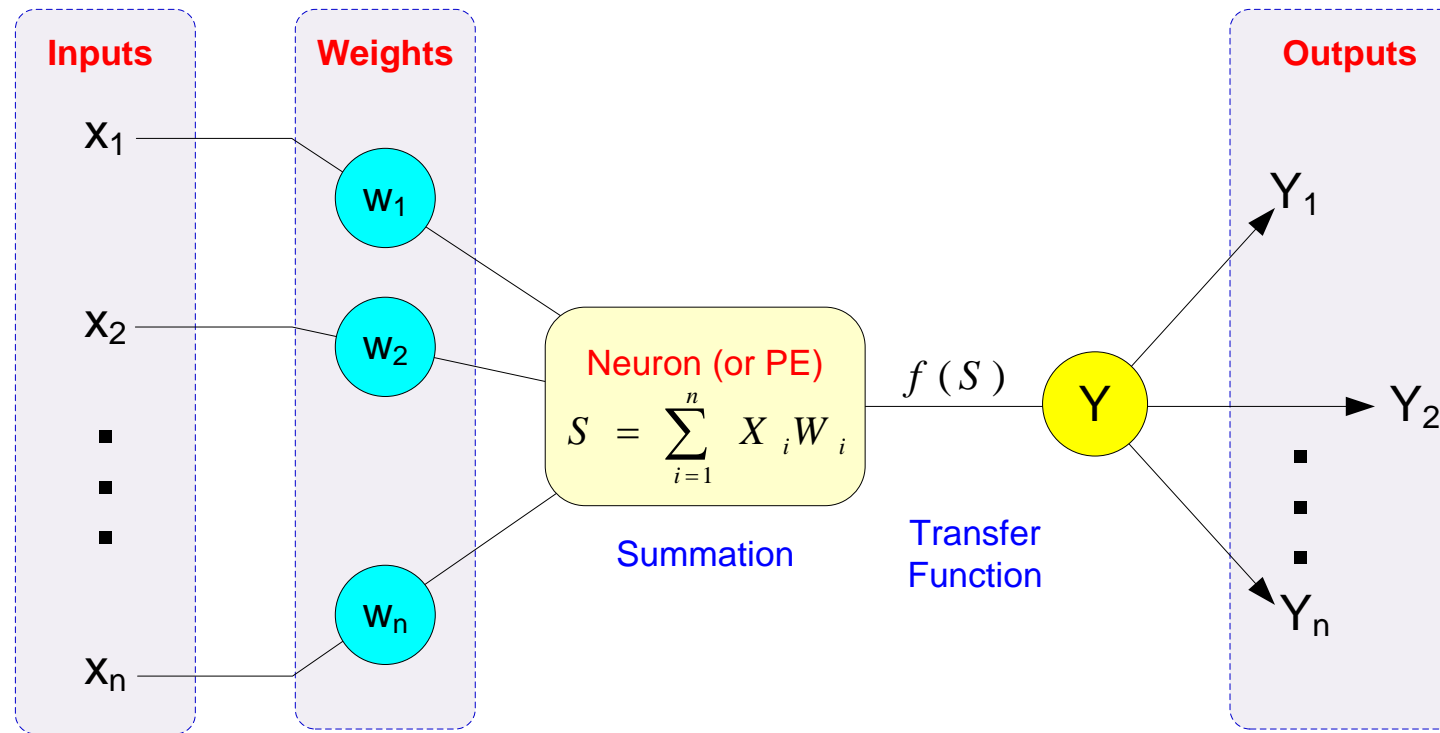


Neural Network Concepts

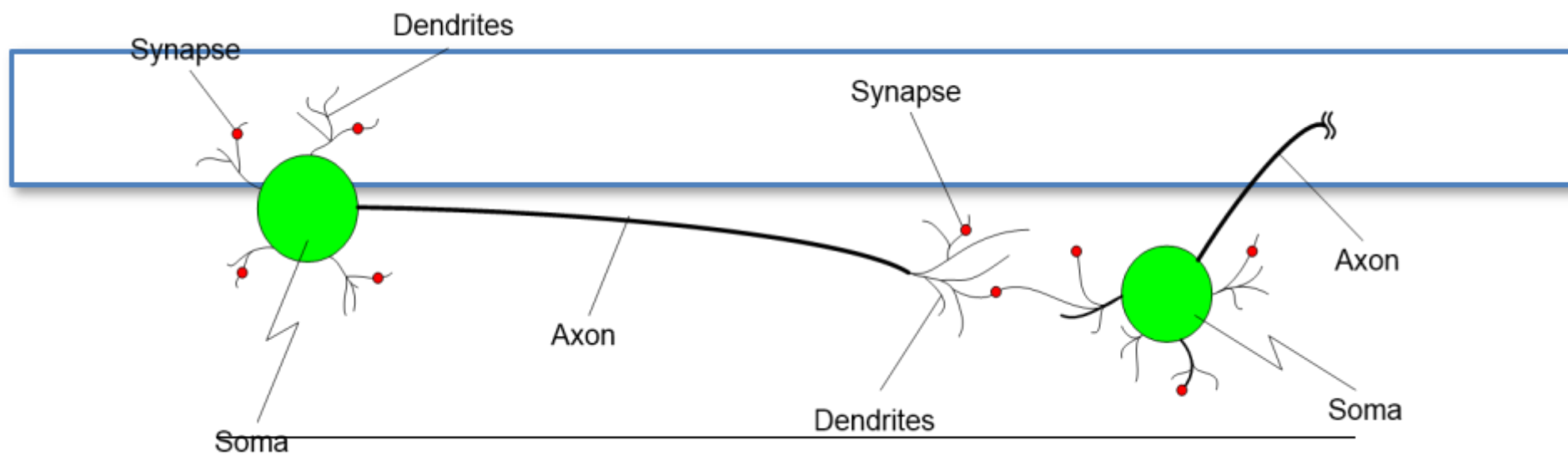


- Neural networks (NN): a brain metaphor for information processing
- Neural computing
- Artificial neural network (ANN)
- Many uses for ANN for
 - pattern recognition, forecasting, prediction, and classification
- Many application areas
 - finance, marketing, manufacturing, operations, information systems, and so on

Processing Information in ANN



- A single neuron (processing element – PE) with inputs and outputs



Biological versus Artificial NNs

Soma

Node

Dendrites

Input

Axon

Output

Synapse

Weight

Slow

Fast

Many neurons (10^9)

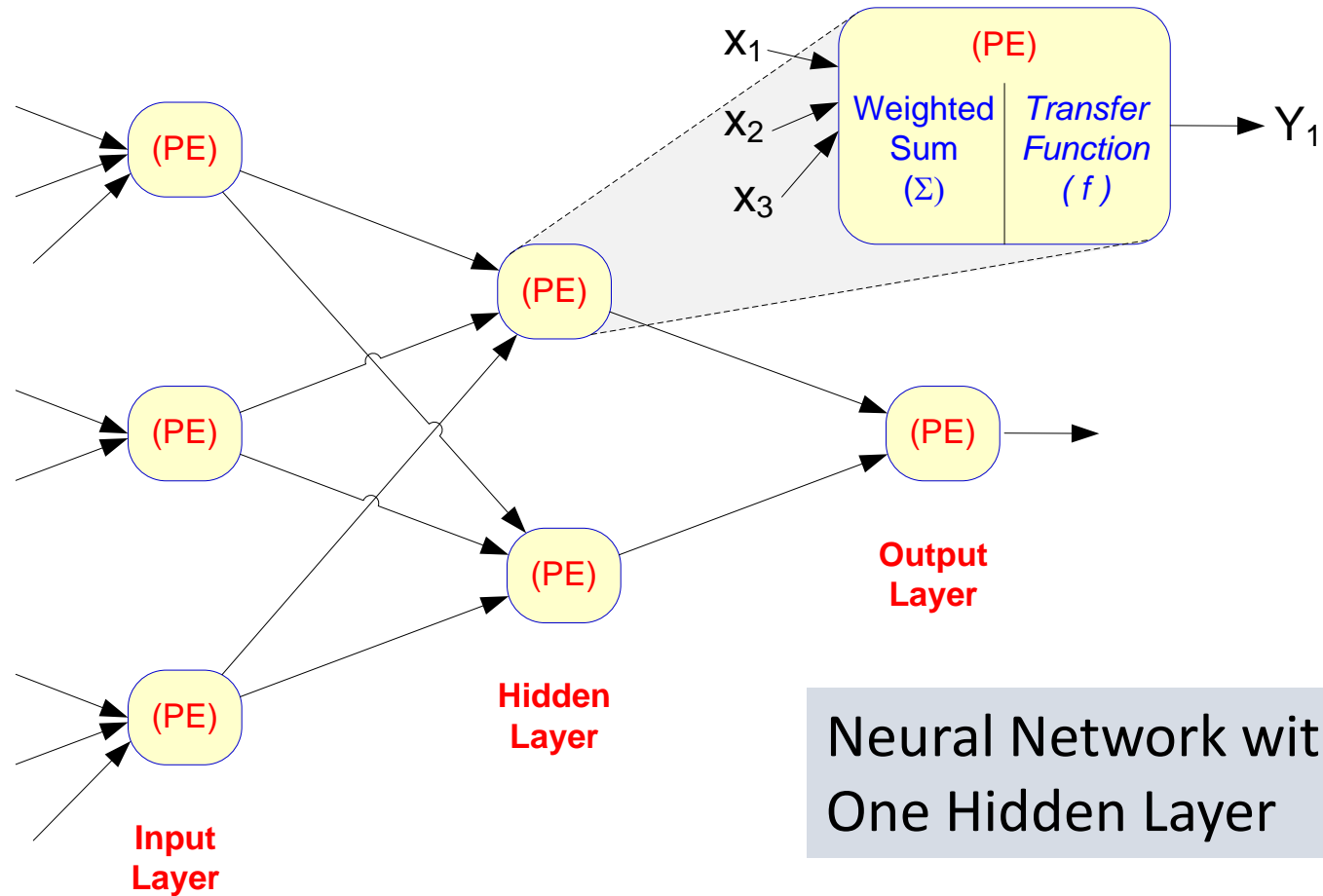
Few neurons (~ 100 s)

Elements of ANN



- Processing element (PE)
- Network architecture
 - Hidden layers
 - Parallel processing
- Network information processing
 - Inputs
 - Outputs
 - Connection weights
 - Summation function

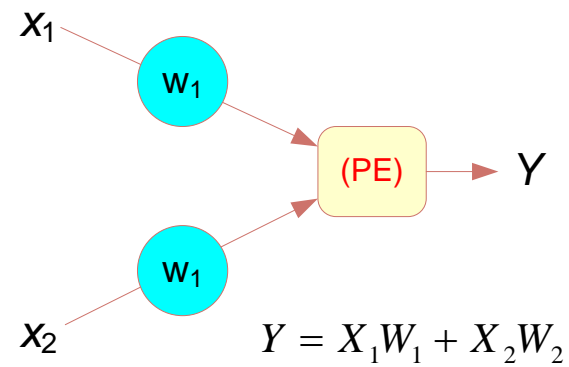
Elements of ANN



Elements of ANN



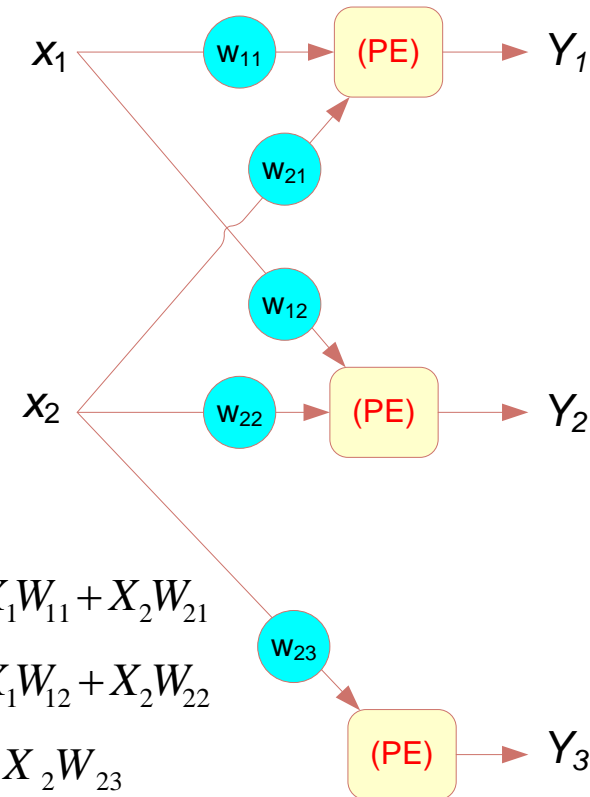
(a) Single neuron



PE: Processing Element (or neuron)

Summation Function for a
Single Neuron (a) and Several
Neurons (b)

(b) Multiple neurons

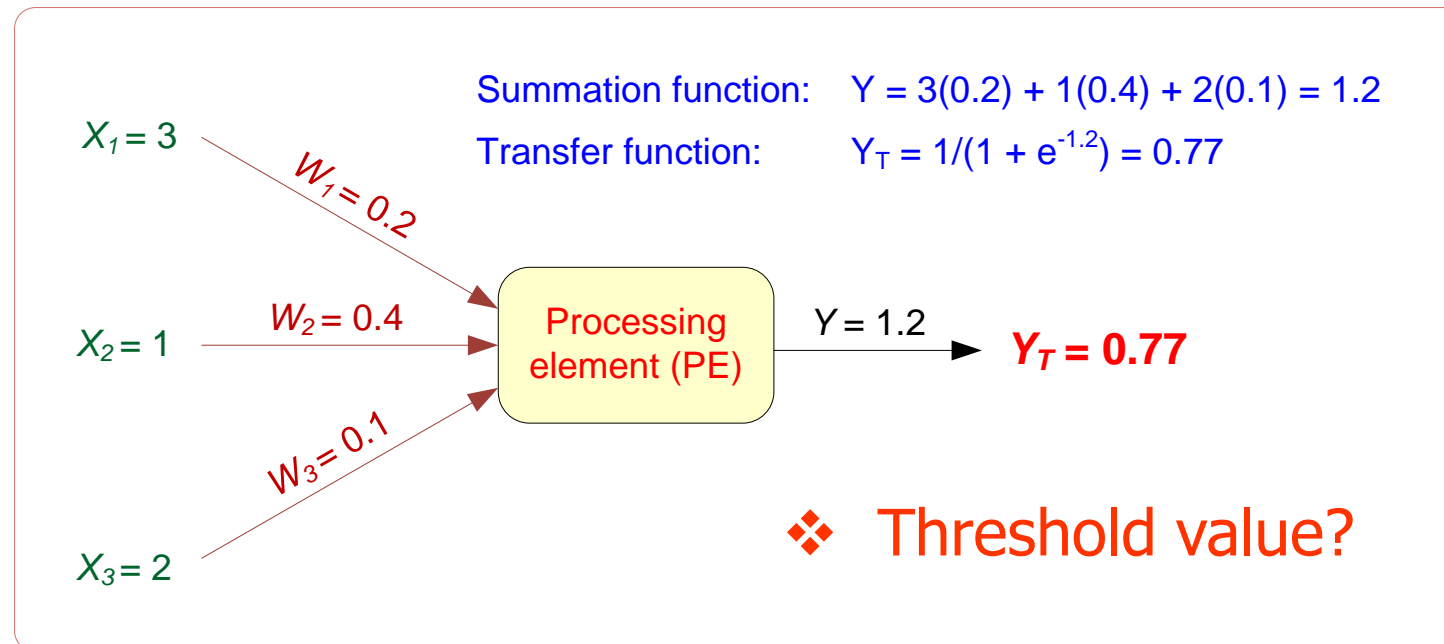


Elements of ANN



- Transformation (Transfer) Function

- Linear function
- Sigmoid (logical activation) function [0 1]
- Tangent Hyperbolic function [-1 1]



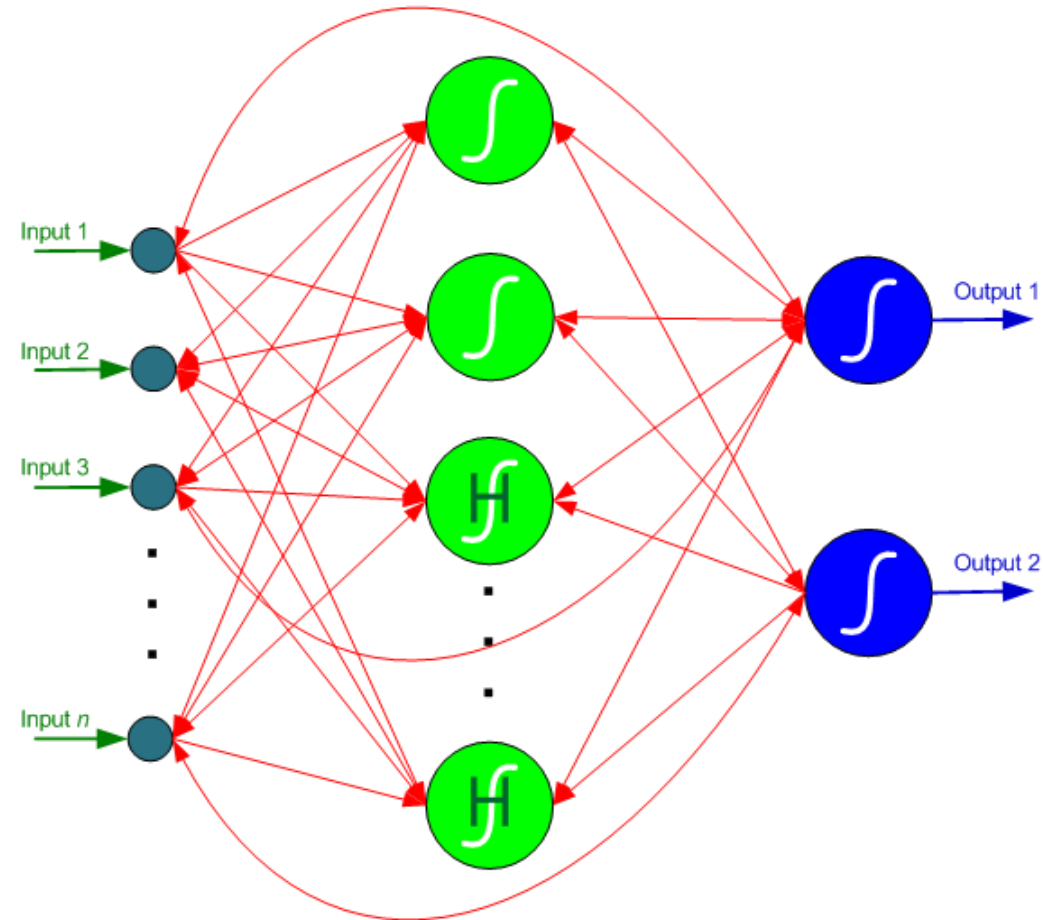
Neural Network Architectures



- Several ANN architectures exist
 - Feedforward
 - Recurrent
 - Associative memory
 - Probabilistic
 - Self-organizing feature maps
 - Hopfield networks
 - ... many more ...

Neural Network Architectures

Recurrent Neural Networks



*H: indicates a "hidden" neuron without a target output

Neural Network Architectures



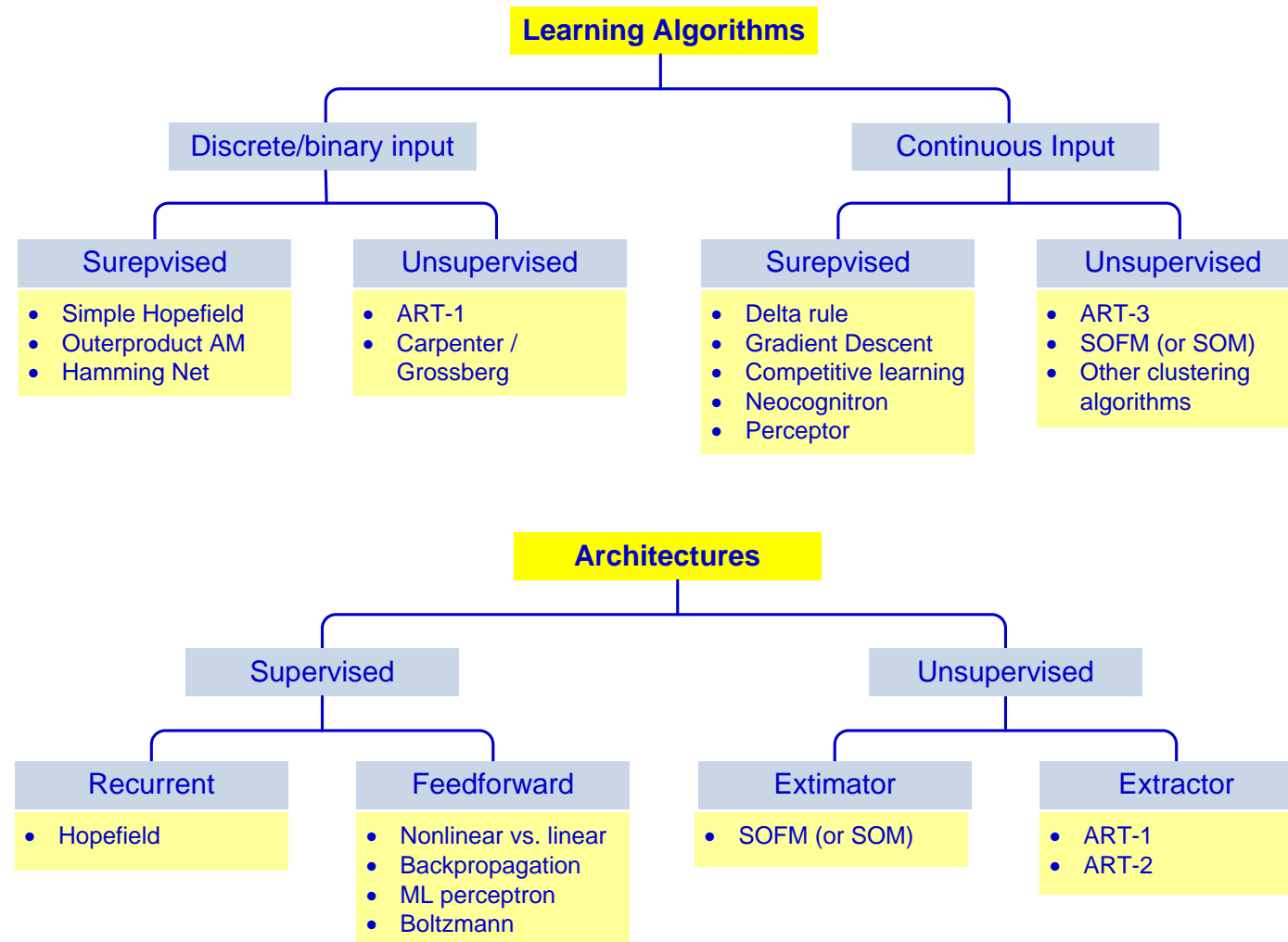
- Architecture of a neural network is driven by the task it is intended to address
 - Classification, regression, clustering, general optimization, association,
- **Most popular architecture:** Feedforward, multi-layered perceptron with backpropagation learning algorithm
 - Used for both classification and regression type problems

Learning in ANN

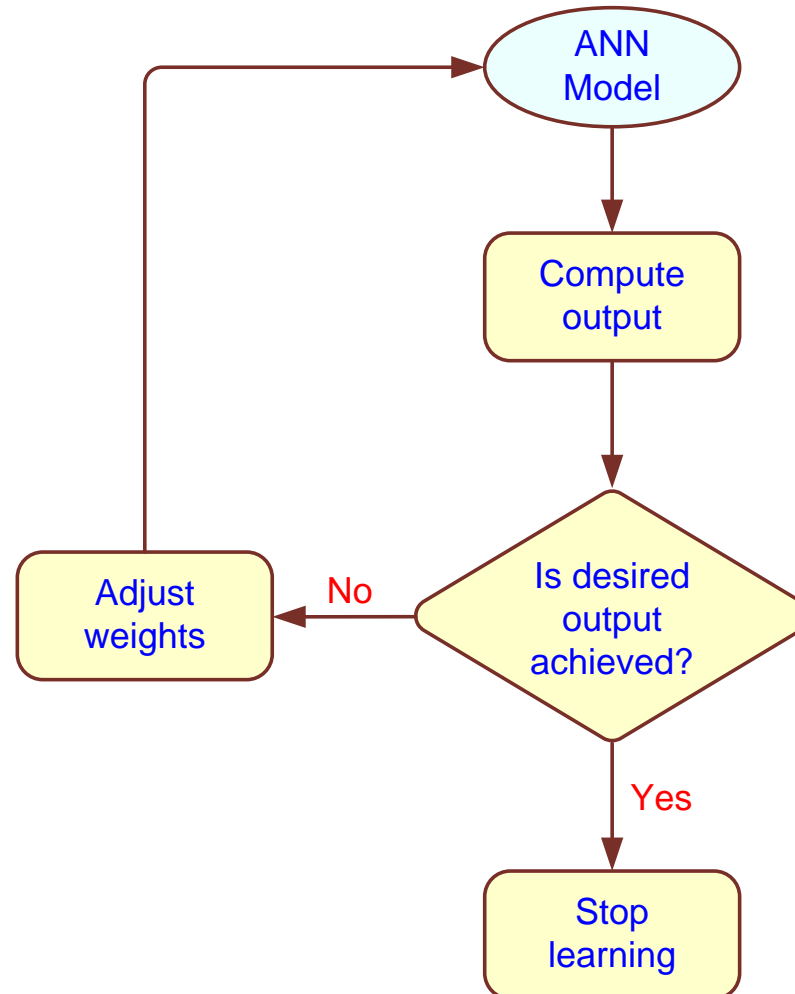


- A process by which a neural network learns the underlying relationship between input and outputs, or just among the inputs
- **Supervised learning**
 - For prediction type problems
 - E.g., backpropagation
- **Unsupervised learning**
 - For clustering type problems
 - Self-organizing
 - E.g., adaptive resonance theory

A Taxonomy of ANN Learning Algorithms

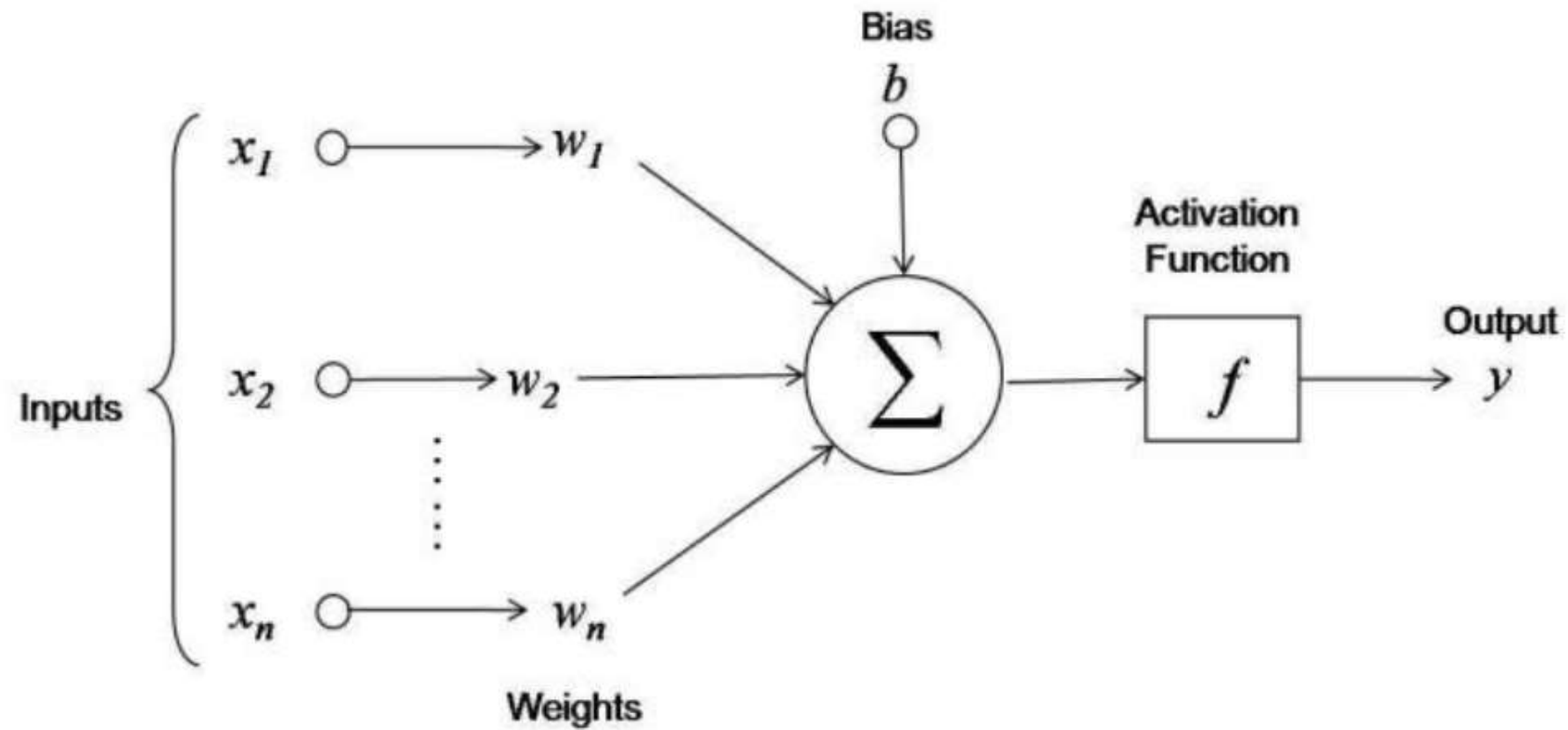


A Supervised Learning Process



Three-step process:

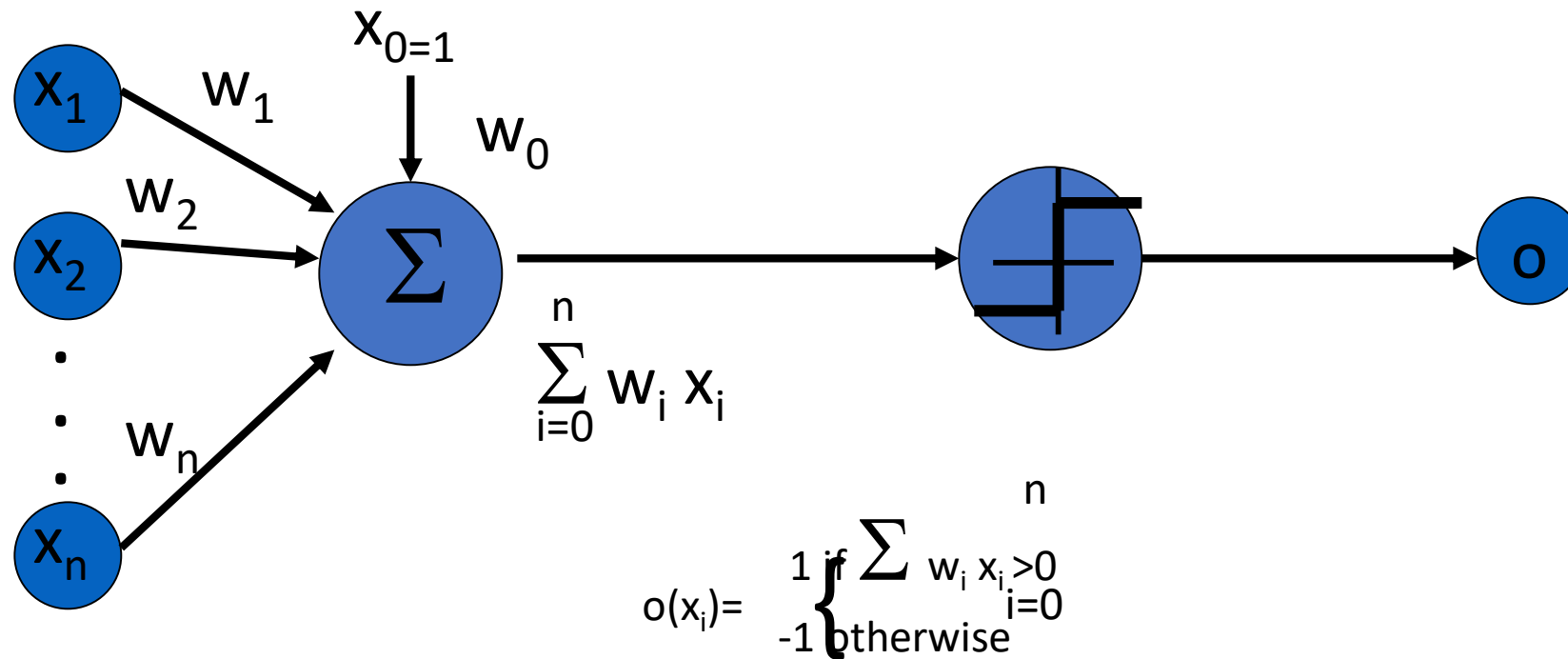
1. Compute temporary outputs
2. Compare outputs with desired targets
3. Adjust the weights and repeat the process



Perceptron



- Linear threshold unit (LTU)



Perceptron Learning Rule



$$w_i = w_i + \Delta w_i$$

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

$t=c(x)$ is the target value

o is the perceptron output

η is a small constant (e.g. 0.1) called *learning rate*

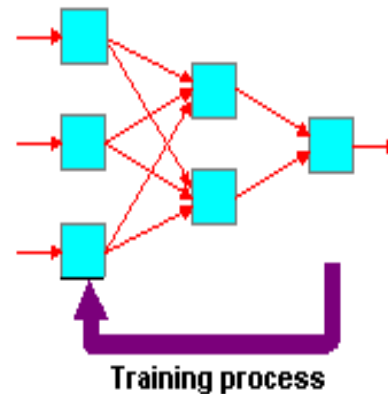
- If the output is correct ($t=o$) the weights w_i are not changed
- If the output is incorrect ($t \neq o$) the weights w_i are changed such that the output of the perceptron for the new weights is *closer* to t .
- The algorithm converges to the correct classification
 - if the training data is linearly separable
 - and η is sufficiently small

Supervised Learning



- Training and test data sets
- Training set; input & target

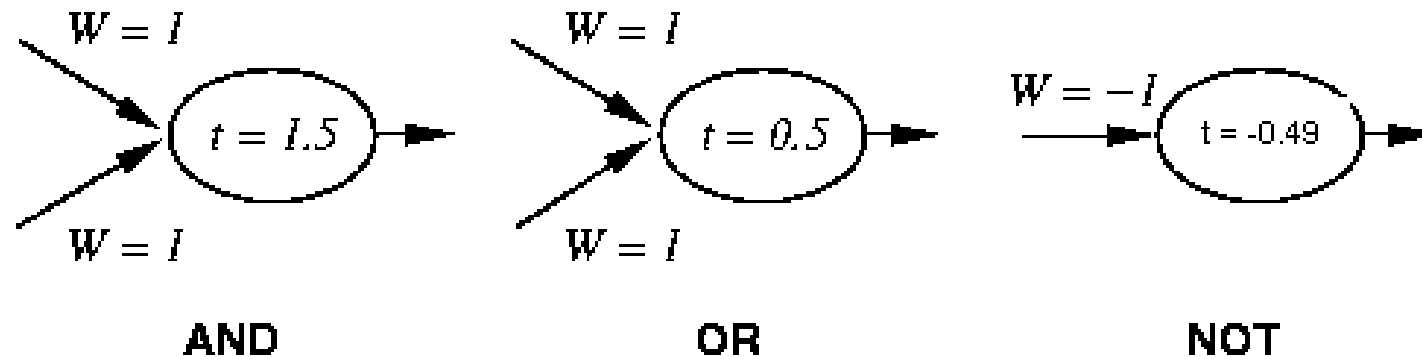
Input Data	Example Outputs
≡	≡
≡	≡
≡	≡
≡	≡
≡	≡
≡	≡
≡	≡



Results
≡
≡
≡
≡
≡

Sepal length	Sepal width	Petal length	Petal width	Class
5.1	3.5	1.4	0.2	0
4.9	3.0	1.4	0.2	2
4.7	3.2	1.3	0.2	0
4.6	3.1	1.5	0.2	1

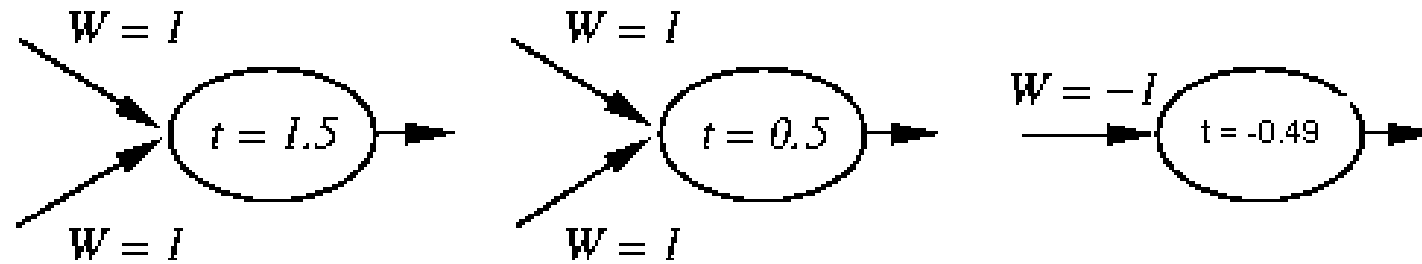
Perceptron Training



$$\text{Output} = \begin{cases} 1 & \text{if } \sum_{i=0} w_i x_i > t \\ 0 & \text{otherwise} \end{cases}$$

- Linear threshold is used.
- W - weight value
- t - threshold value

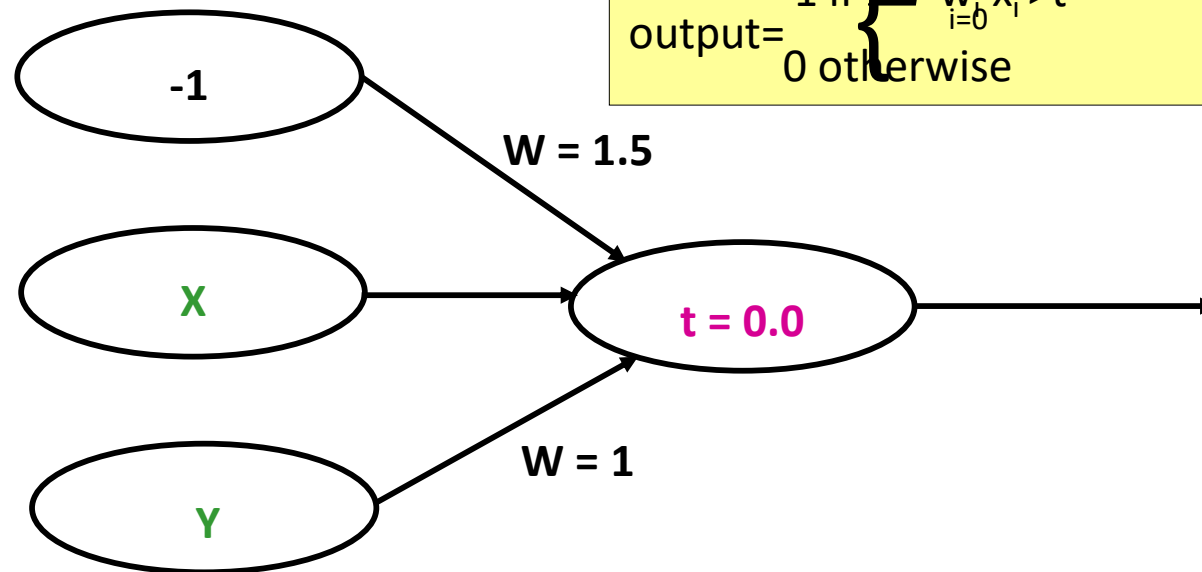
Simple network



AND

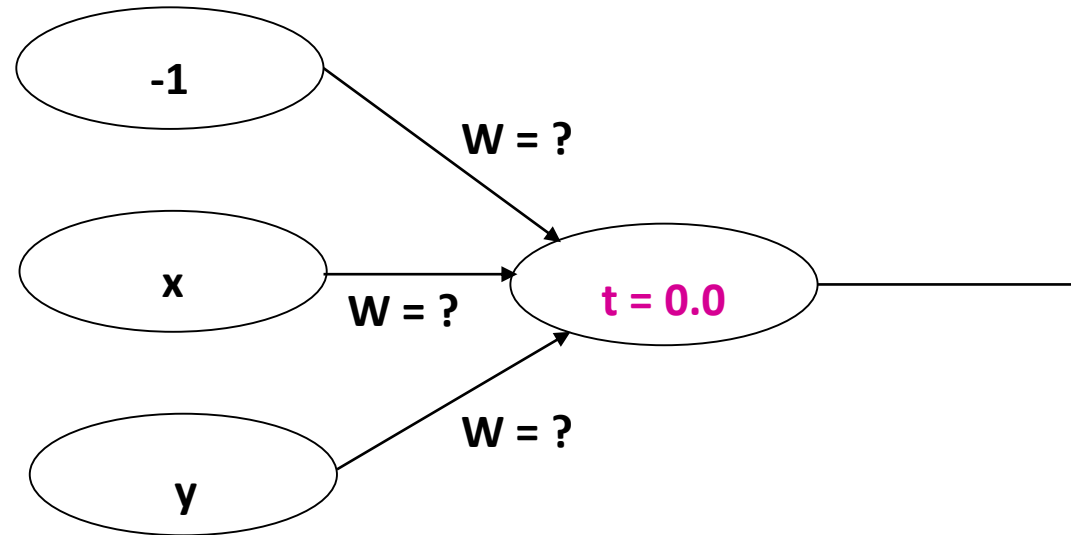
OR

NOT



$$\text{output} = \begin{cases} 1 & \text{if } \sum_{i=0} w_i x_i > t \\ 0 & \text{otherwise} \end{cases}$$

Training Perceptrons

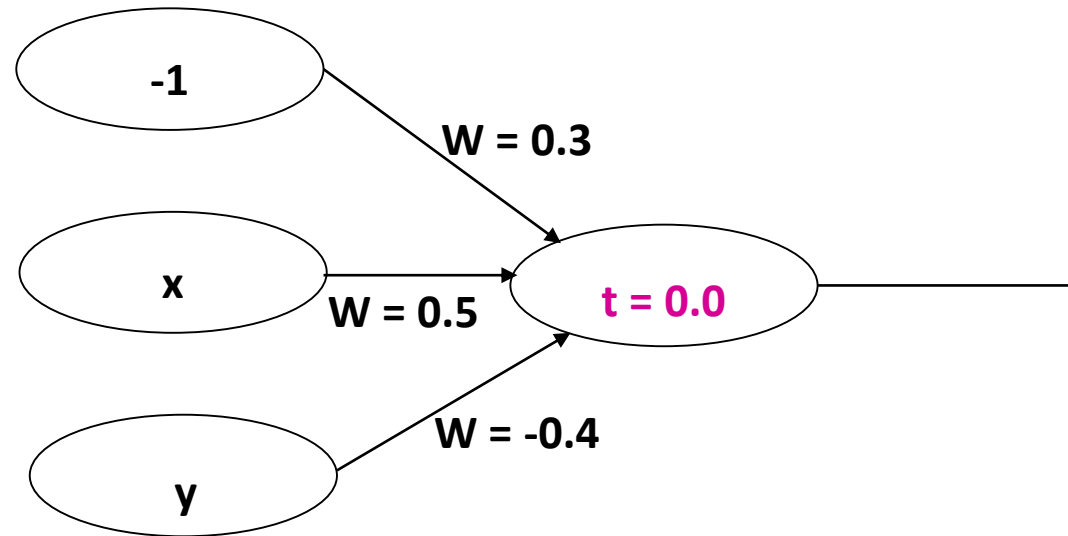


For AND

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

- What are the weight values?
- Initialize with random weight values

Training Perceptrons



For AND

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

I_1	I_2	I_3	Summation	Output
-1	0	0	$(-1 \cdot 0.3) + (0 \cdot 0.5) + (0 \cdot -0.4) = -0.3$	0
-1	0	1	$(-1 \cdot 0.3) + (0 \cdot 0.5) + (1 \cdot -0.4) = -0.7$	0
-1	1	0	$(-1 \cdot 0.3) + (1 \cdot 0.5) + (0 \cdot -0.4) = 0.2$	1
-1	1	1	$(-1 \cdot 0.3) + (1 \cdot 0.5) + (1 \cdot -0.4) = -0.2$	0

Learning algorithm



Epoch : Presentation of the entire training set to the neural network.

In the case of the AND function an epoch consists of four sets of inputs being presented to the network (i.e. $[0,0]$, $[0,1]$, $[1,0]$, $[1,1]$)

Error: The error value is the amount by which the value output by the network differs from the target value. For example, if we required the network to output 0 and it output a 1, then $\text{Error} = -1$

Learning algorithm



Target Value, T : When we are training a network we not only present it with the input but also with a value that we require the network to produce. For example, if we present the network with $[1,1]$ for the AND function the training value will be 1

Output, O : The output value from the neuron

I_j : Inputs being presented to the neuron

W_j : Weight from input neuron (I_j) to the output neuron

LR : The learning rate. This dictates how quickly the network converges. It is set by a matter of experimentation. It is typically 0.1

How a Network Learns



- **Example:** single neuron that learns the inclusive OR operation

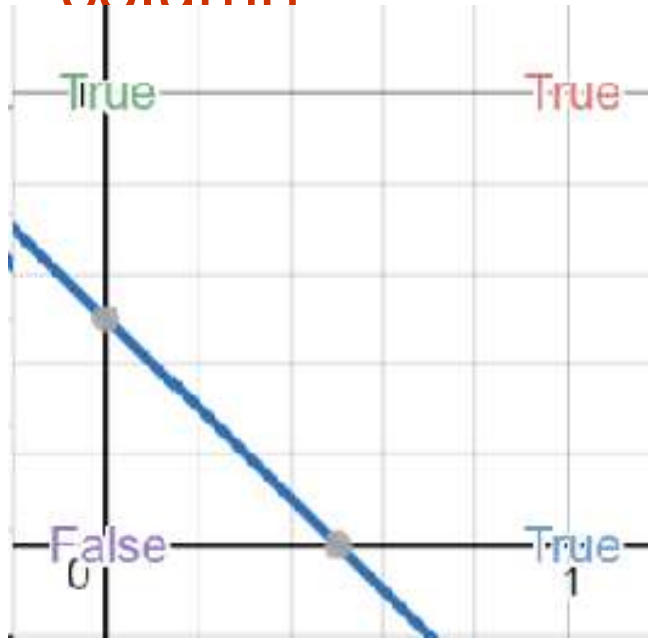
Inputs			
Case	X_1	X_2	Desired Results
1	0	0	0
2	0	1	1 (positive)
3	1	0	1 (positive)
4	1	1	1 (positive)

Learning parameters:

- Learning rate
- Momentum

classifier to distinguish 2 groups of output 0 or 1

Perceptron is the first neural network ever created. It consists on 2 neurons in the inputs column and 1 neuron in the output column



- if A is true and B is true, then A or B is true.
- if A is true and B is false, then A or B is true.
- if A is false and B is true, then A or B is true.
- if A is false and B is false, then A or B is false.

We are able to separate using a line (UNLIKE XOR)



```
In [1]: import numpy, random, os
lr = 1 #learning rate
bias = 1 #value of bias
weights = [random.random(),random.random(),random.random()] #weights generated in a list (3 weights in total for 2 neurons and the bias)
```

```
In [4]: #function which defines the work of the output neuron
def Perceptron(input1, input2, output) :
    outputP = input1*weights[0]+input2*weights[1]+bias*weights[2]
    if outputP > 0 : #activation function (here Heaviside)
        outputP = 1
    else :
        outputP = 0
    error = output - outputP
    weights[0] += error * input1 * lr
    weights[1] += error * input2 * lr
    weights[2] += error * bias * lr
```

```
In [5]: # repeat in this case 50 times (keep low to avoid overfitting)
for i in range(50) :
    Perceptron(1,1,1) #True or true
    Perceptron(1,0,1) #True or false
    Perceptron(0,1,1) #False or true
    Perceptron(0,0,0) #False or false
```

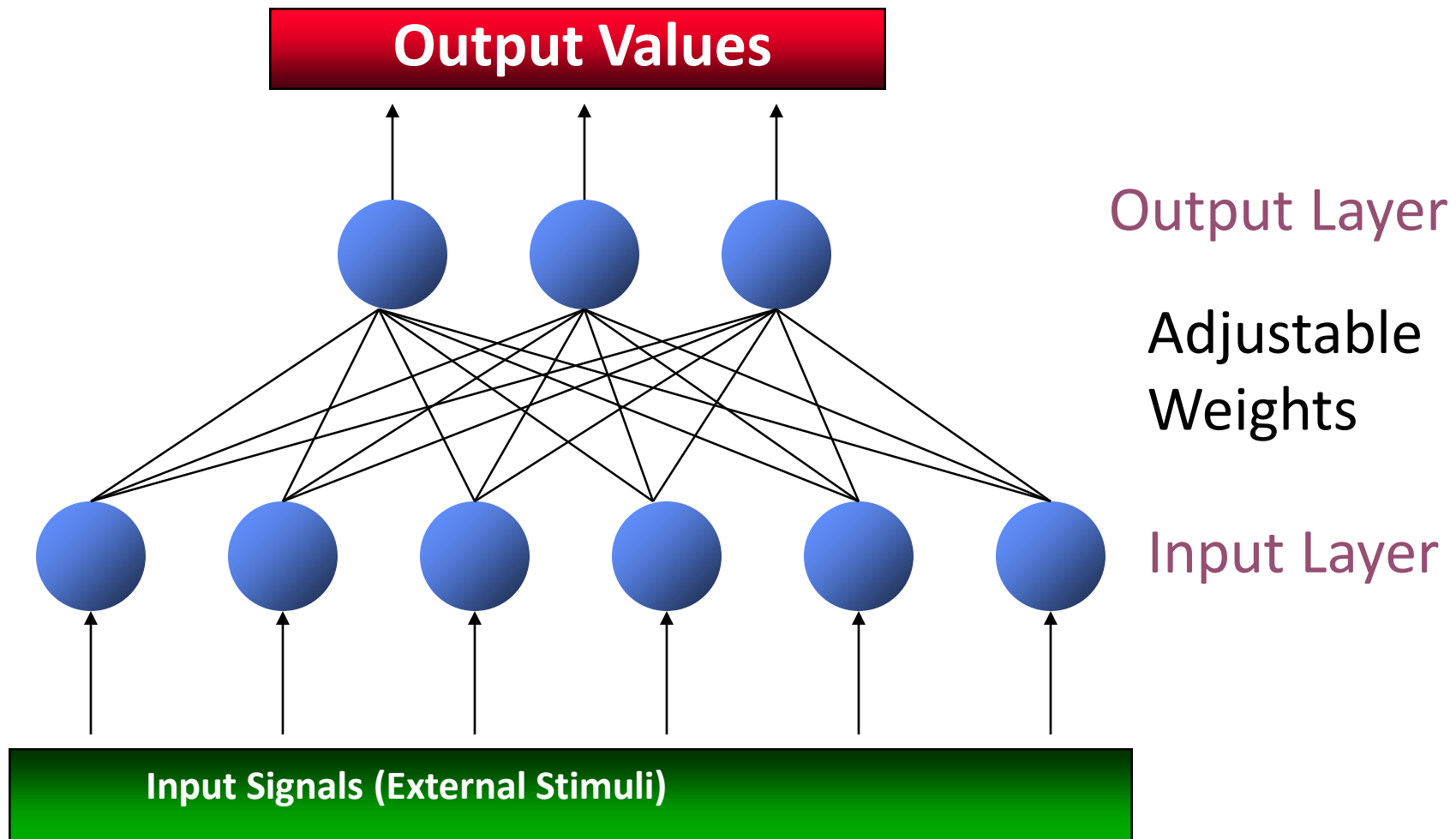
```
In [6]: x = int(input())
y = int(input())
outputP = x*weights[0] + y*weights[1] + bias*weights[2]
if outputP > 0 : #activation function
    outputP = 1
else :
    outputP = 0
print(x, "or", y, "is : ", outputP)
```

```
1
0
1 or 0 is : 1
```

```
In [7]: x = int(input())
y = int(input())
outputP = 1/(1+numpy.exp(-outputP)) #sigmoid function
print(x, "or", y, "is : ", outputP)
```

```
1
0
1 or 0 is : 0.7310585786300049
```

Multilayer Perceptron (MLP)



Types of Layers

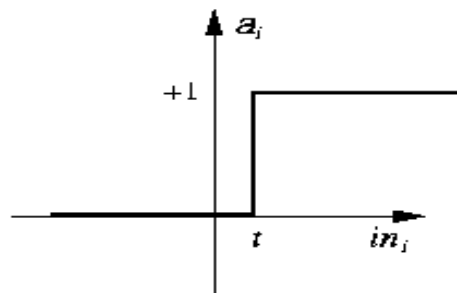


- The input layer.
 - Introduces input values into the network.
 - No activation function or other processing.
- The hidden layer(s).
 - Perform classification of features
 - Two hidden layers are sufficient to solve any problem
 - Features imply more layers may be better
- The output layer.
 - Functionally just like the hidden layers
 - Outputs are passed on to the world outside the neural network.

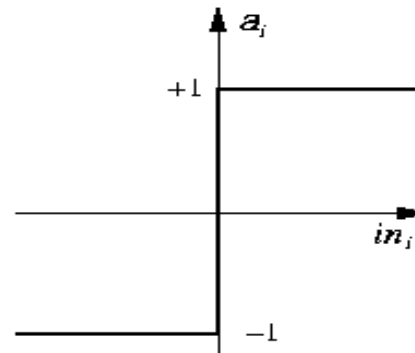
Activation functions



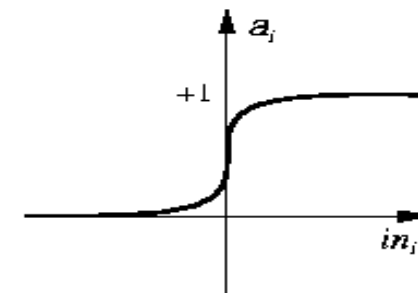
- Transforms neuron's input into output.
- Features of activation functions:
 - A squashing effect is required
 - Prevents accelerating growth of activation levels through the network.



(a) Step function



(b) Sign function




(c) Sigmoid function

Standard activation functions



- The hard-limiting threshold function
 - Corresponds to the biological paradigm
 - either fires or not
- Sigmoid functions ('S'-shaped curves)
 - The logistic function
 - The hyperbolic tangent (symmetrical)
 - Both functions have a simple differential
 - Only the shape is important


$$\phi(x) = \frac{1}{1 + e^{-ax}}$$

Training Algorithms

- Adjust neural network weights to map inputs to outputs.
- Use a set of sample patterns where the desired output (given the inputs presented) is known.
- The purpose is to learn to generalize
 - Recognize features which are common to good and bad exemplars

Back-Propagation



- A training procedure which allows multi-layer feedforward Neural Networks to be trained;
- Can theoretically perform “any” input-output mapping;
- Can learn to solve linearly inseparable problems.

Activation functions and training



- For feed-forward networks:
 - A continuous function can be differentiated allowing gradient-descent.
 - Back-propagation is an example of a gradient-descent technique.
 - Reason for prevalence of sigmoid

Gradient Descent Learning Rule



- Consider linear unit without threshold and continuous output o (not just $-1,1$)
 - $O = w_0 + w_1 x_1 + \dots + w_n x_n$
- Train the w_i 's such that they minimize the squared error
 - $E[w_1, \dots, w_n] = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$
where D is the set of training examples

Gradient Descent



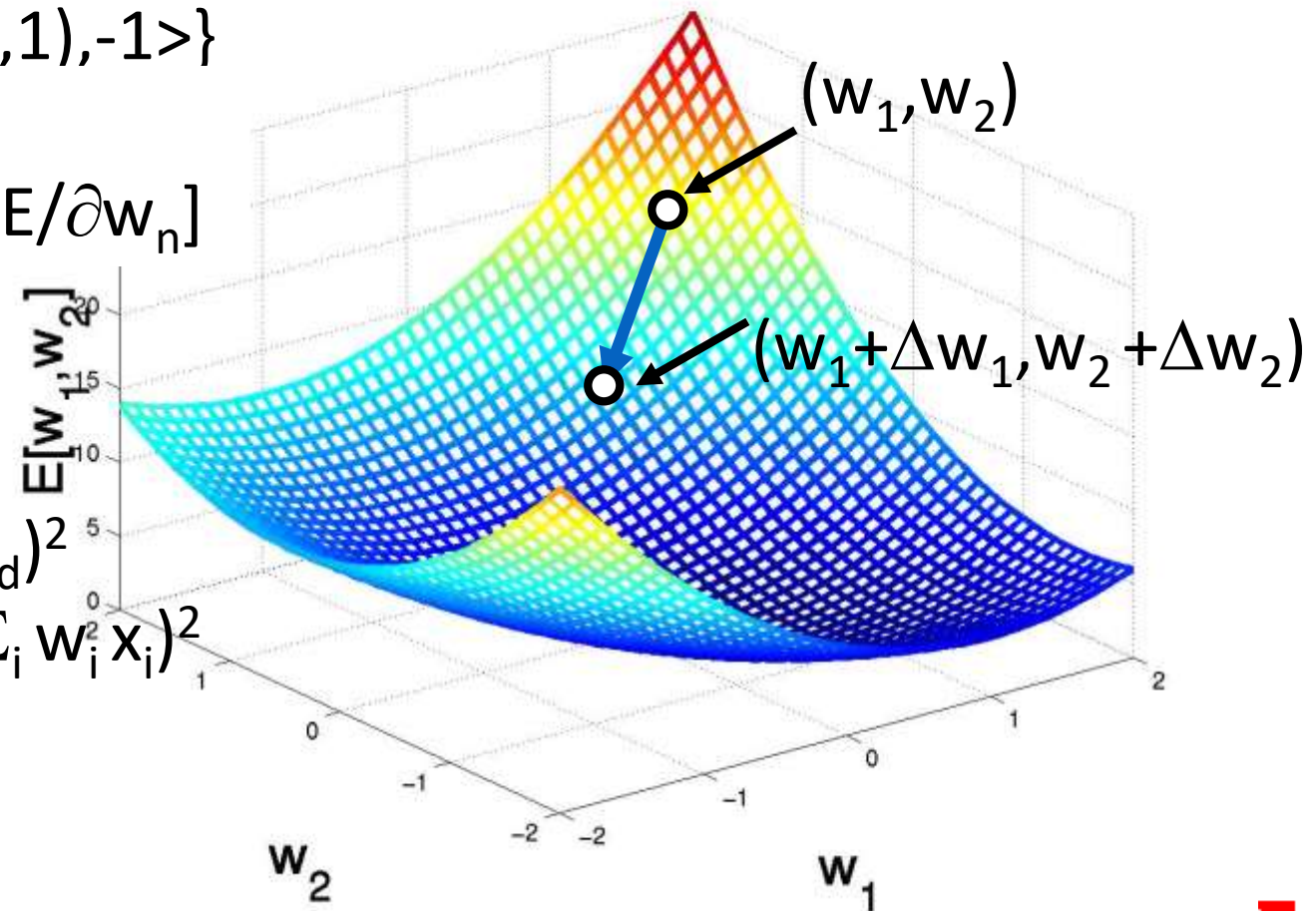
$$D=\{<(1,1),1>,<(-1,-1),1>,<(1,-1),-1>,<(-1,1),-1>\}$$

Gradient:

$$\nabla E[w]=[\partial E/\partial w_0,... \partial E/\partial w_n]$$

$$\Delta w=-\eta \nabla E[w]$$

$$\begin{aligned}\Delta w_i &= -\eta \partial E/\partial w_i \\ &= \partial/\partial w_i 1/2 \sum_d (t_d - o_d)^2 \\ &= \partial/\partial w_i 1/2 \sum_d (t_d - \sum_i w_i^2 x_i)^2 \\ &= \sum_d (t_d - o_d)(-x_i)\end{aligned}$$



Gradient Descent



Gradient-Descent(*training_examples*, η)

Each training example is a pair of the form $\langle (x_1, \dots, x_n), t \rangle$ where (x_1, \dots, x_n) is the vector of input values, and t is the target output value, η is the learning rate (e.g. 0.1)

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
 - Initialize each Δw_i to zero
 - For each $\langle (x_1, \dots, x_n), t \rangle$ in *training_examples* Do
 - Input the instance (x_1, \dots, x_n) to the linear unit and compute the output o
 - For each linear unit weight w_i Do
 - $\Delta w_i = \Delta w_i + \eta (t - o) x_i$
 - For each linear unit weight w_i Do
 - $w_i = w_i + \Delta w_i$

Incremental Stochastic Gradient Descent



- **Batch mode** : gradient descent

$w = w - \eta \nabla E_D[w]$ over the entire data D

$$E_D[w] = 1/2 \sum_d (t_d - o_d)^2$$

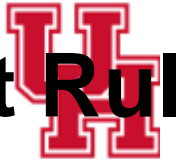
- **Incremental mode**: gradient descent

$w = w - \eta \nabla E_d[w]$ over individual training examples d

$$E_d[w] = 1/2 (t_d - o_d)^2$$

Incremental Gradient Descent can approximate Batch Gradient Descent arbitrarily closely if η is small enough

Comparison Perceptron and Gradient Descent Rule



Perceptron learning rule guaranteed to succeed if

- Training examples are linearly separable
- Sufficiently small learning rate η

Linear unit training rules uses gradient descent

- Guaranteed to converge to hypothesis with minimum squared error
- Given sufficiently small learning rate η
- Even when training data contains noise
- Even when training data not separable by H

```

In [ ]: ▶ #Multi-Layer Perceptron model
import pandas as pd
wine = pd.read_csv('wine_data.csv', names = ["Cultivator", "Alchol", "Malic_Acid", "Ash", "Alcalinity_of_Ash", "Ma

In [ ]: ▶ wine.head()

In [ ]: ▶ wine.describe().transpose()

In [ ]: ▶ wine.shape

In [ ]: ▶ #set you Label
X = wine.drop('Cultivator',axis=1)
y = wine['Cultivator']

In [ ]: ▶ from sklearn.model_selection import train_test_split

In [ ]: ▶ #X_train, X_test, y_train, y_test = train_test_split(X, y)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)

In [ ]: ▶ #preprocessing
from sklearn.preprocessing import StandardScaler

In [ ]: ▶ from sklearn import preprocessing
scaler = preprocessing.StandardScaler().fit(X_train)

```

```
In [ ]:  MLPClassifier(hidden_layer_sizes=(13,13,13),max_iter=500)

In [ ]:  mlp.fit(X_train,y_train)

In [ ]:  MLPClassifier(activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9,
    beta_2=0.999, early_stopping=False, epsilon=1e-08,
    hidden_layer_sizes=(13, 13, 13), learning_rate='constant',
    learning_rate_init=0.001, max_iter=500, momentum=0.9,
    nesterovs_momentum=True, power_t=0.5, random_state=None,
    shuffle=True, solver='adam', tol=0.0001, validation_fraction=0.1,
    verbose=False, warm_start=False)

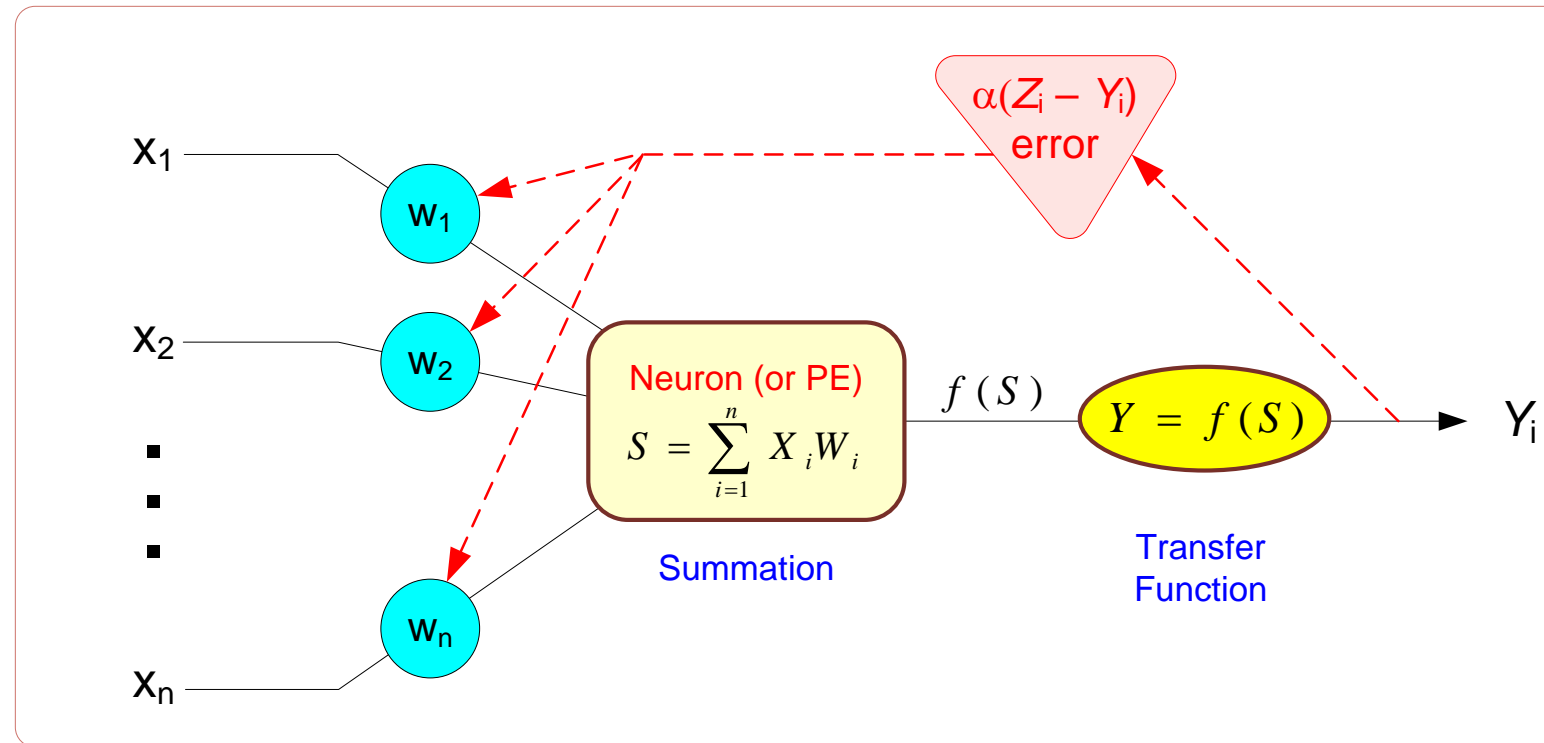
In [ ]:  #predictions
    predictions = mlp.predict(X_test)

In [ ]:  from sklearn.metrics import classification_report,confusion_matrix

In [ ]:  print(confusion_matrix(y_test,predictions))

In [ ]:  print(classification_report(y_test,predictions))
```

Backpropagation Learning



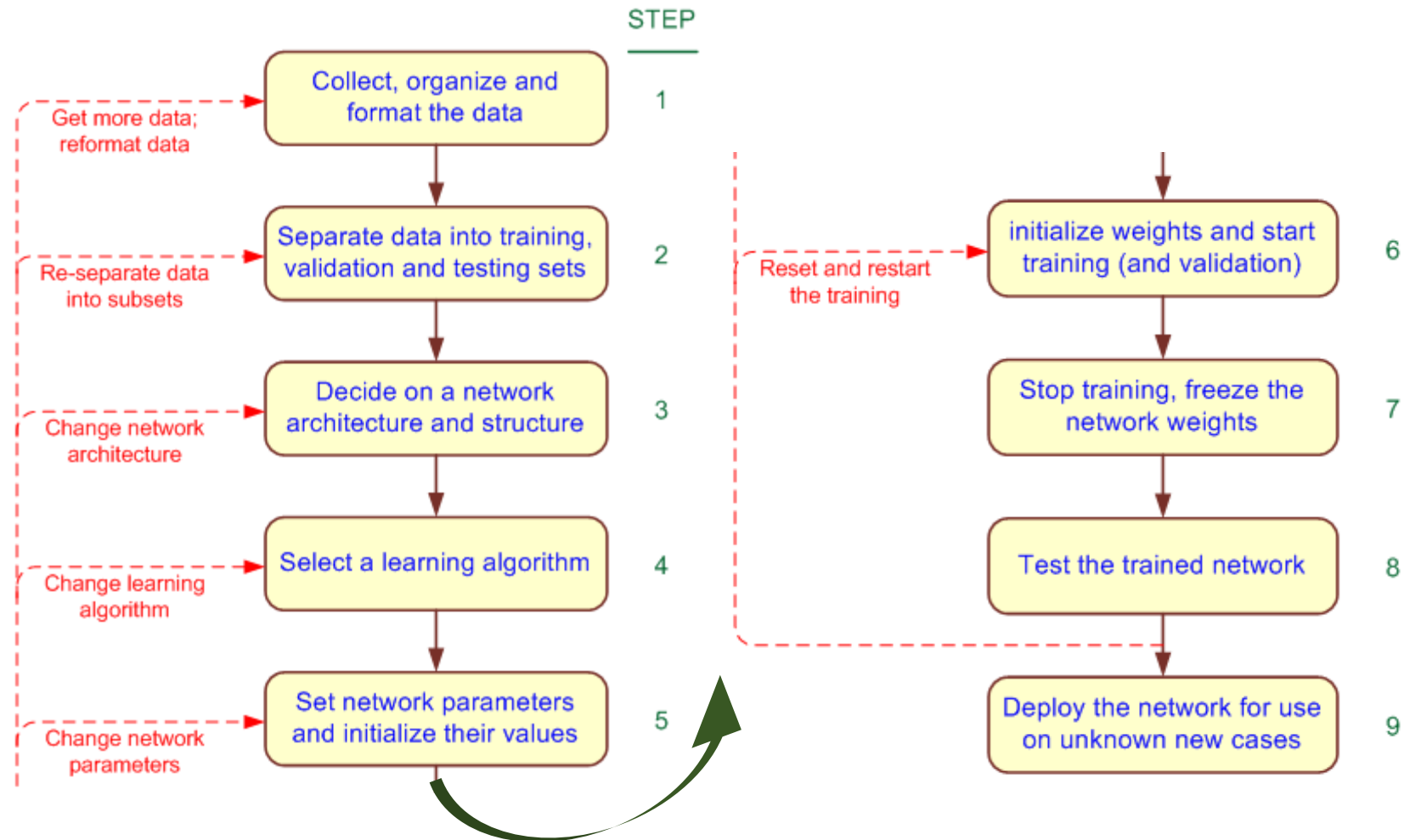
- Backpropagation of Error for a Single Neuron

Backpropagation Learning

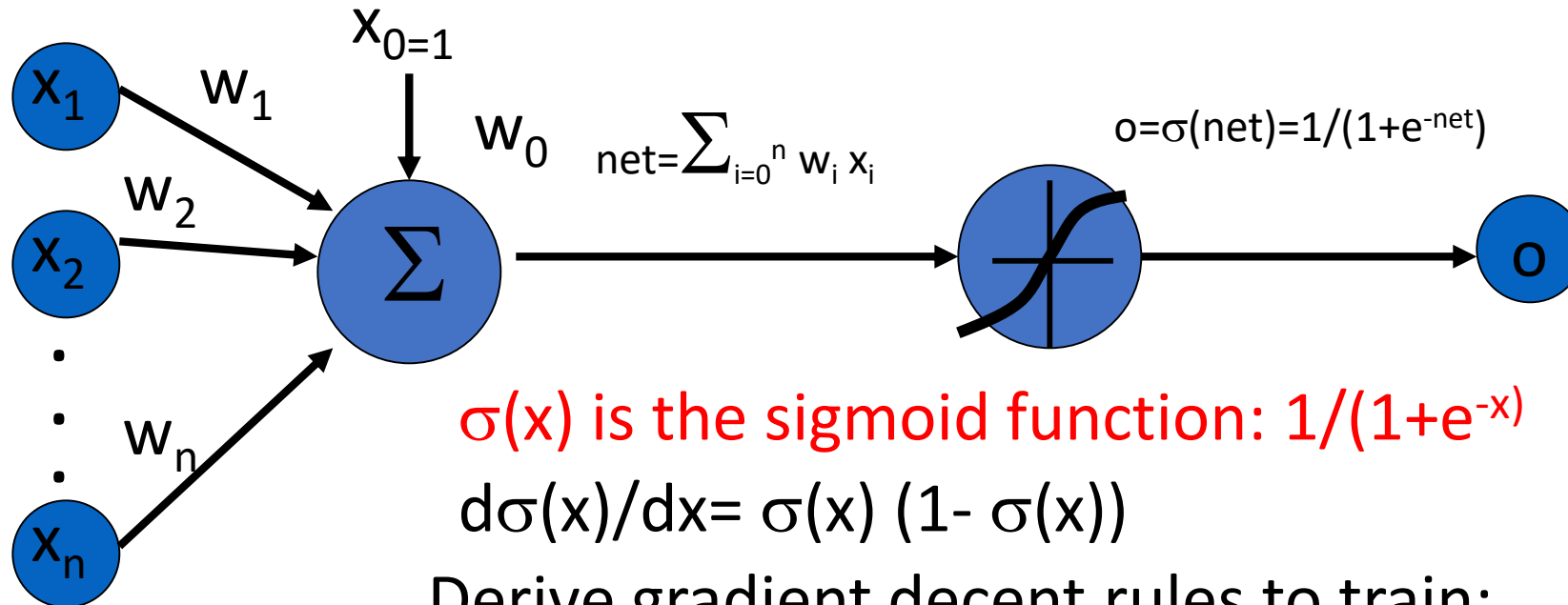


- The learning algorithm procedure:
 1. Initialize weights with random values and set other network parameters
 2. Read in the inputs and the desired outputs
 3. Compute the actual output (by working forward through the layers)
 4. Compute the error (difference between the actual and desired output)
 5. Change the weights by working backward through the hidden layers
 6. Repeat steps 2-5 until weights stabilize

Development Process of an ANN



Example using Sigmoid Unit



$\sigma(x)$ is the sigmoid function: $1/(1+e^{-x})$

$$d\sigma(x)/dx = \sigma(x) (1 - \sigma(x))$$

Derive gradient decent rules to train:

- one sigmoid function

$$\partial E / \partial w_i = -\sum_d (t_d - o_d) o_d (1 - o_d) x_i$$

- Multilayer networks of sigmoid units
backpropagation:

Backpropagation Algorithm

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
 - For each training example $\langle (x_1, \dots, x_n), t \rangle$ Do
 - Input the instance (x_1, \dots, x_n) to the network and compute the network outputs o_k
 - For each output unit k
 - $\delta_k = o_k(1-o_k)(t_k-o_k)$
 - For each hidden unit h
 - $\delta_h = o_h(1-o_h) \sum_k w_{h,k} \delta_k$
 - For each network weight w_{ij} Do
 - $w_{i,j} = w_{i,j} + \Delta w_{i,j}$ where

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

Backpropagation

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
 - in practice often works well (can be invoked multiple times with different initial weights)
- Often include weight *momentum* term

$$\Delta w_{i,j}(t) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(t-1)$$
- Minimizes error training examples
 - Will it generalize well to unseen instances (over-fitting)?
- Training can be slow typical 1000-10000 iterations
 - (use Levenberg-Marquardt instead of gradient descent)

Convergence of Backprop

Gradient descent to some local minimum

- Perhaps not global minimum
- Add momentum
- Stochastic gradient descent
- Train multiple nets with different initial weights

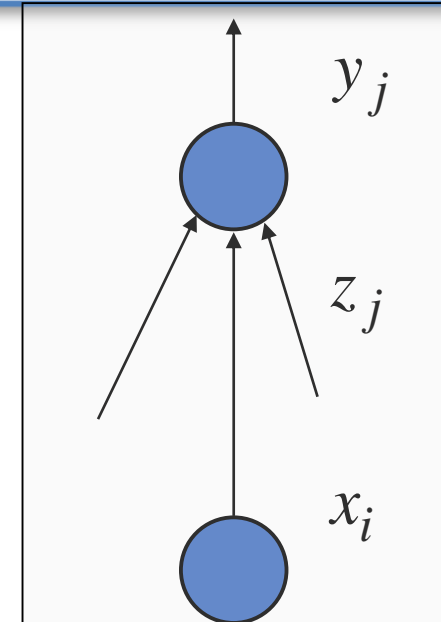
Nature of convergence

- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly non-linear functions possible as training progresses

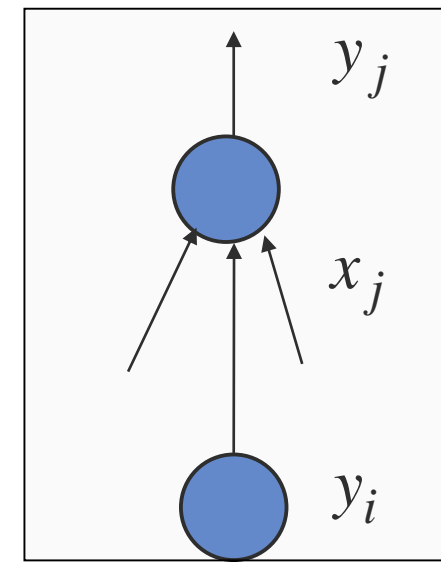
A change of notation



- For simple networks we use the notation
 - x for activities of input units
 - y for activities of output units
 - z for the summed input to an output unit

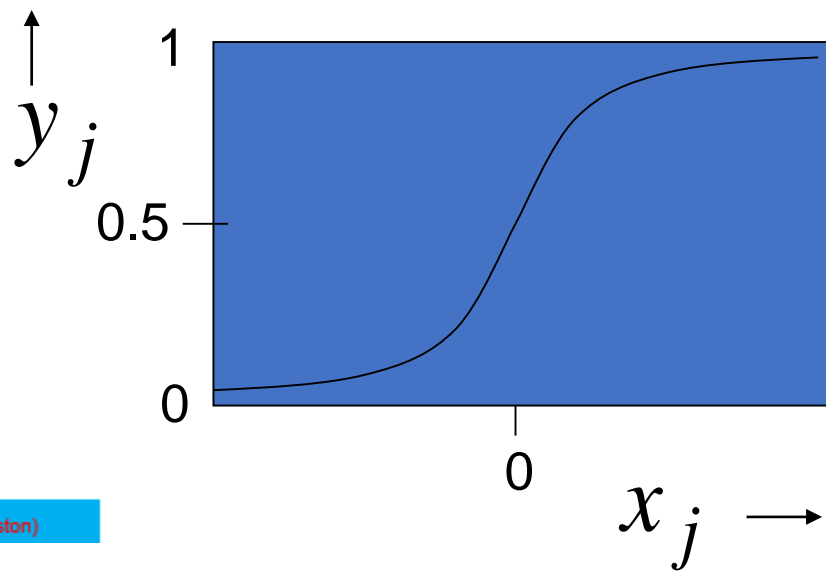


- For networks with multiple hidden layers:
 - y is used for the output of a unit in any layer
 - x is the summed input to a unit in any layer
- The index indicates which layer a unit is in.



Non-linear neurons with smooth derivatives

- For backpropagation, we need neurons that have well-behaved derivatives.
 - Typically they use the logistic function
 - The output is a smooth function of the inputs and the weights.



$$x_j = b_j + \sum_i y_i w_{ij}$$

$$y_j = \frac{1}{1 + e^{-x_j}}$$

$$\frac{\partial x_j}{\partial w_{ij}} = y_i \quad \frac{\partial x_j}{\partial y_i} = w_{ij}$$

$$\frac{dy_j}{dx_j} = y_j (1 - y_j)$$

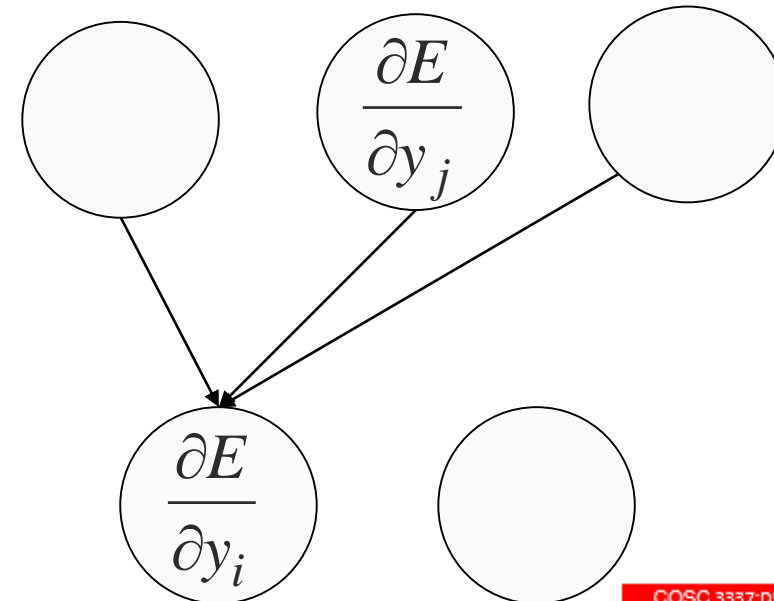
Its odd to express it
in terms of y .

Sketch of the backpropagation algorithm on a single training case

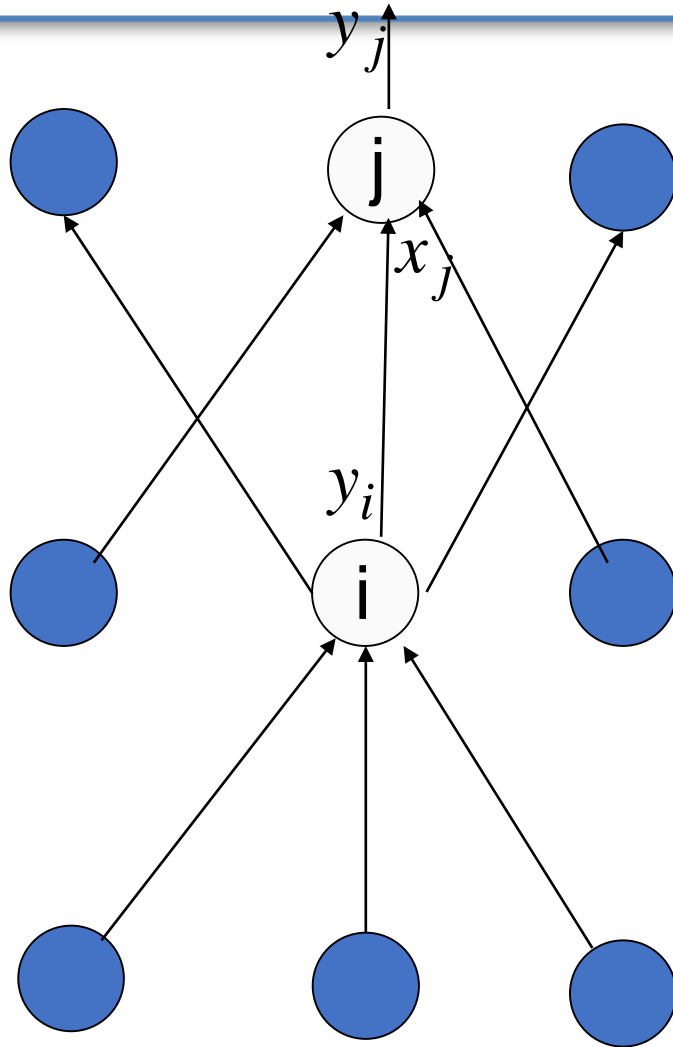
- First convert the discrepancy between each output and its target value into an error derivative.
- Then compute error derivatives in each hidden layer from error derivatives in the layer above.
- Then use error derivatives w.r.t. activities to get error derivatives w.r.t. the weights.

$$E = \sum_j \frac{1}{2} (y_j - d_j)^2$$

$$\frac{\partial E}{\partial y_j} = y_j - d_j$$



The derivatives



$$\frac{\partial E}{\partial x_j} = \frac{dy_j}{dx_j} \frac{\partial E}{\partial y_j} = y_j (1 - y_j) \frac{\partial E}{\partial y_j}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial x_j}{\partial w_{ij}} \frac{\partial E}{\partial x_j} = y_i \frac{\partial E}{\partial x_j}$$

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{dx_j}{dy_i} \frac{\partial E}{\partial x_j} = \sum_j w_{ij} \frac{\partial E}{\partial x_j}$$

Momentum



- Sometimes we add to ΔW_{ji} a momentum factor α . This allows us to use a high learning rate, but prevent the oscillatory behavior that can sometimes result from a high learning rate.

$$W_{ji} \leftarrow W_{ji} + \eta \times a_j \times \delta_i$$

Add to this a momentum α times the weight update from the last iteration, i.e., add α times the previous value of $\eta \times a_j \times \delta_i$ where $0 \leq \alpha < 1$ and often $\alpha = 0.9$

Momentum keeps it going in the same direction.

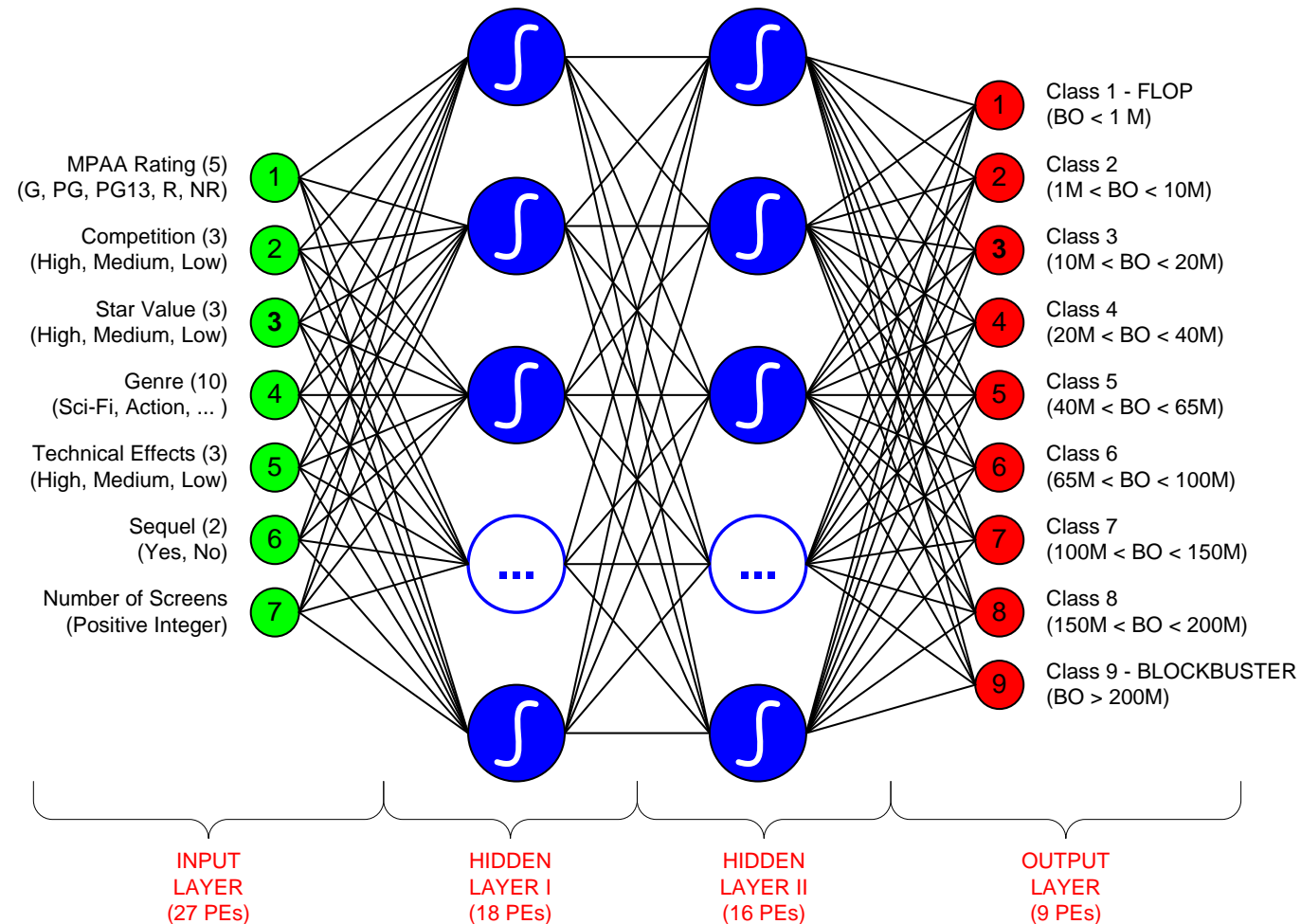
More on backpropagation



- Performs gradient descent over the entire network weight vector.
- Will find a local, not necessarily global, error minimum.
- Minimizes error over training set; need to guard against overfitting just as with decision tree learning.
- Training takes thousands of iterations (epochs) --- slow!

An MLP ANN Structure for Prediction Problem

the Box-Office



Testing a Trained ANN Model



- Data is split into three parts
 - Training (~60%)
 - Validation (~20%)
 - Testing (~20%)
- k -fold cross validation
 - Less bias
 - Time consuming

```
In [ ]: ▶ import pandas as pd
wine = pd.read_csv(r'wine_data.csv', names = ["Cultivator", "Alchol", "Malic_Acid", "Ash", "Alcalinity_of_Ash", "M
X = wine.drop('Cultivator',axis=1) #input
y = wine['Cultivator']
```

```
In [ ]: ▶ from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.25, random_state=1)
```

```
In [ ]: ▶ from sklearn.model_selection import KFold
kf = KFold(n_splits = 5, shuffle = True, random_state = 2)

for train_index, test_index in kf.split(X):
    X_tr_va, X_test = X.iloc[train_index], X.iloc[test_index]
    y_tr_va, y_test = y[train_index], y[test_index]
    X_train, X_val, y_train, y_val = train_test_split(X_tr_va, y_tr_va, test_size=0.25, random_state=1)
    print("TRAIN:", list(X_train.index), "VALIDATION:", list(X_val.index), "TEST:", test_index)
```

Applications Types of ANN



- Classification
 - Feedforward networks (MLP), radial basis function, and probabilistic NN
- Regression
 - Feedforward networks (MLP), radial basis function
- Clustering
 - Adaptive Resonance Theory (ART) and SOM
- Association
 - Hopfield networks

Advantages of ANN



- Able to deal with (identify/model) highly nonlinear relationships
- Not prone to restricting normality and/or independence assumptions
- Can handle variety of problem types
- Usually provides better results (prediction and/or clustering) compared to its statistical counterparts
- Handles both numerical and categorical variables (transformation needed!)

Disadvantages of ANN



- They are deemed to be black-box solutions, lacking expandability
- It is hard to find optimal values for large number of network parameters
 - Optimal design is still an art: requires expertise and extensive experimentation
- It is hard to handle large number of variables (especially the rich nominal attributes)
- Training may take a long time for large datasets; which may require case sampling

ANN Software



- Standalone ANN software tool
 - NeuroSolutions
 - BrainMaker
 - NeuralWare
 - NeuroShell, ... for more (see pcai.com) ...
- Part of a data mining software suit
 - PASW (formerly SPSS Clementine)
 - SAS Enterprise Miner
 - Statistica Data Miner, ... many more ...

In order to create a neural network:

- 1- specify the number of layers within the model (e.g two layers)
- 2-Select the type of activation for each layer (e.g a sigmoid activation for the first layers)
- 3- Number of nodes in the final layer (e.g having 10 nodes)→set to return 10 probability scores
- 4- To compile the model, a loss function must be defined (how the model evaluates its own performance)
- 5- An optimizer must be determined, which is how the information from the cost function is used to change the weights and the bias of each node.

```
model = keras.Sequential([keras.layers.Flatten(input_shape (28,28)),  
keras.layers.Dense(128,activation = tf.nn.sigmoid),  
keras.layers.Dense(10,activation = tf.nn.softmax)])  
model.compile(optimizer =  
'adam',loss='sparse_categorical_crossentropy',metrics =['accuracy'])
```

Train the model , then evaluate it based on the testing data. → the need number of epochs.

Epochs determine how many iterations through the data shall be done.

```
model.fit(x_train, y_train,epochs = 5)
```



```
Epoch 1/5
60000/60000 [=====] - 6s 93us/sample - loss: 0.8991 - accuracy: 0.6946
Epoch 2/5
60000/60000 [=====] - 5s 84us/sample - loss: 0.7550 - accuracy: 0.7120
Epoch 3/5
60000/60000 [=====] - 5s 90us/sample - loss: 0.7294 - accuracy: 0.7199
Epoch 4/5
60000/60000 [=====] - 6s 92us/sample - loss: 0.7073 - accuracy: 0.7318
Epoch 5/5
60000/60000 [=====] - 5s 90us/sample - loss: 0.6969 - accuracy: 0.7340
```

```
class NeuralNetwork:
```

```
def __init__(self, x, y):  
    self.input = x  
    self.weights1 =  
        np.random.rand(self.input.shape[1],4)  
    self.weights2 = np.random.rand(4,1)  
    self.y = y  
    self.output = np.zeros(y.shape)
```

Neural Networks consist of the following components

- An input layer, x
- An arbitrary amount of hidden layers
- An output layer, \hat{y}
- A set of weights and biases between each layer, W and b
- A choice of activation function for each hidden layer, σ . Such as a Sigmoid activation function.

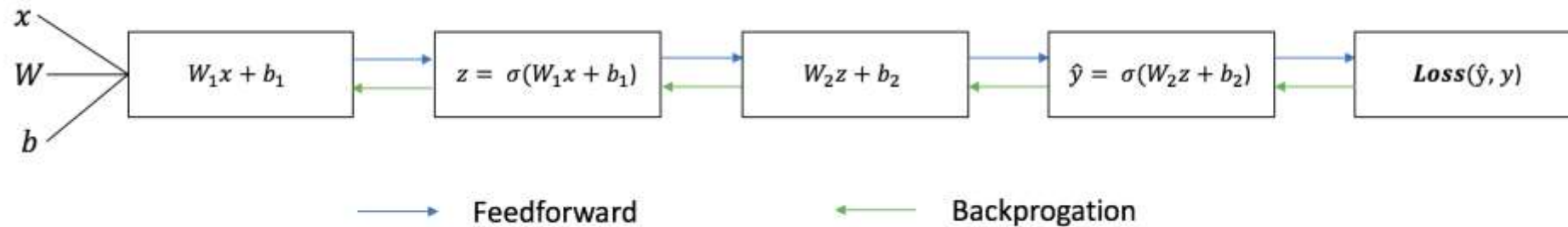
Training the Neural Network

The process of fine-tuning the weights and biases from the input data is known as training the Neural Network.

The output \hat{y} of a simple 2-layer Neural Network is:

$$\hat{y} = \sigma(W_2 \sigma(W_1 x + b_1) + b_2)$$

Each iteration of the training process consists of the following steps:



- Calculating the predicted output \hat{y} , known as feedforward
- Updating the weights and biases, known as backpropagation



```
def feedforward(self):
```

```
    self.layer1 = sigmoid(np.dot(self.input,  
    self.weights1))  
    self.output = sigmoid(np.dot(self.layer1,  
    self.weights2))
```

Evaluate the “goodness” of the predictions → loss function



$$\text{Sum of Squares Error} = \sum_{i=1}^n (y - \hat{y})^2$$

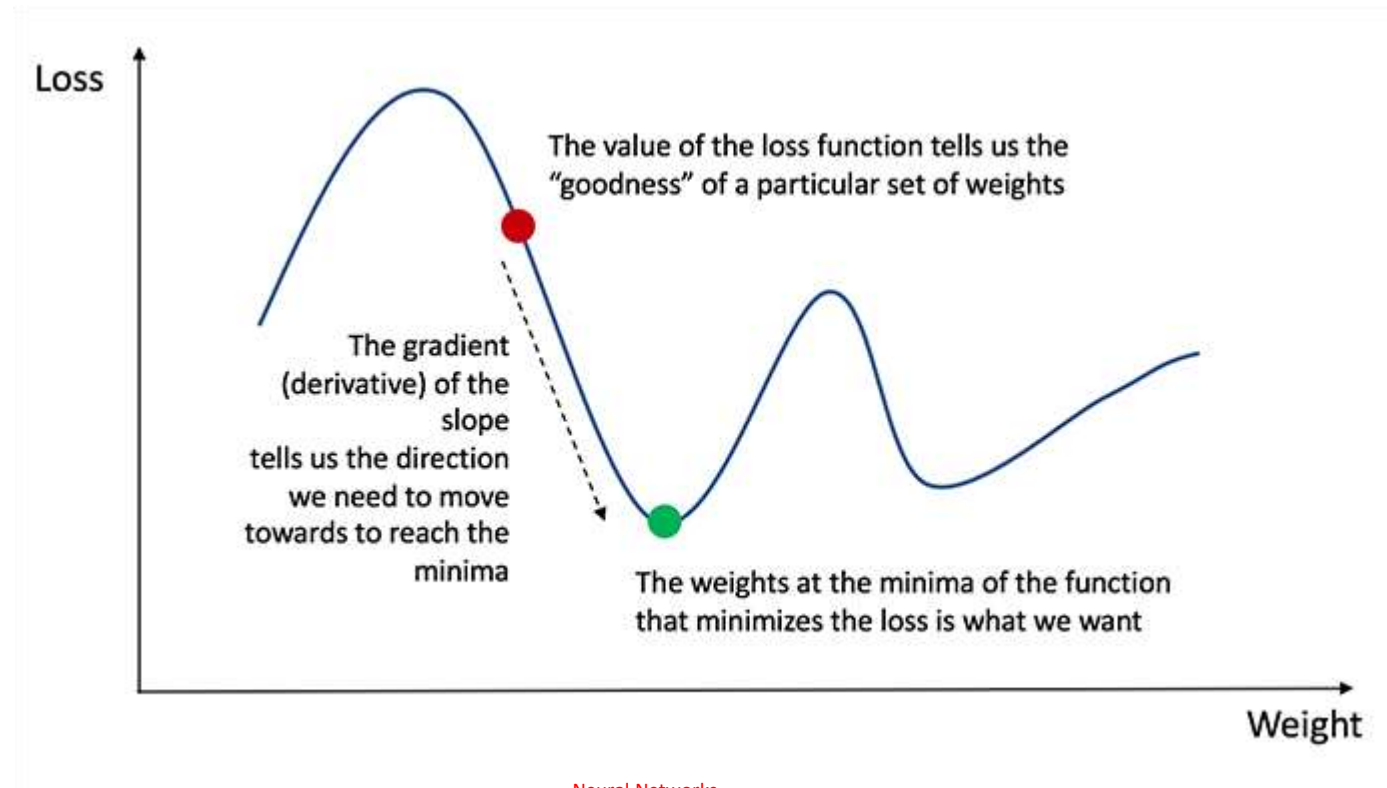
Training the model is to find the best set of weights and biases that minimizes the loss function.

Backpropagation



How to propagate the error back, and to update weights and biases?

By finding derivative of the loss function with respect to the weights and biases (the slope)



The derivative of the loss function cannot be calculated with respect to the weights and biases because the equation of the loss function does not contain the weights and biases. → the we need the chain rule

$$Loss(y, \hat{y}) = \sum_{i=1}^n (y - \hat{y})^2$$

$$\frac{\partial Loss(y, \hat{y})}{\partial W} = \frac{\partial Loss(y, \hat{y})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} * \frac{\partial z}{\partial W} \quad \text{where } z = Wx + b$$

$$= 2(y - \hat{y}) * \text{derivative of sigmoid function} * x$$

$$= 2(y - \hat{y}) * z(1-z) * x$$

```
def backprop(self):
```

```
# application of the chain rule to find
derivative of the loss function with respect
to weights2 and weights1
```

```
d_weights2 = np.dot(self.layer1.T,
(2*(self.y - self.output) *
sigmoid_derivative(self.output)))
d_weights1 = np.dot(self.input.T,
(np.dot(2*(self.y - self.output) *
sigmoid_derivative(self.output),
self.weights2.T) *
sigmoid_derivative(self.layer1)))
```

```
# update the weights with the derivative
(slope) of the loss function
self.weights1 += d_weights1
self.weights2 += d_weights2
```

Example



X1	X2	X3	Y
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0

Predictions after 1500 training iterations

Prediction	Y (Actual)
0.023	0
0.979	1
0.975	1
0.025	0



feedforward and
backpropagation algorithm
trained the Neural Network
successfully and the
predictions converged on
the true values.