

PATTERN RECOGNITION AND MACHINE LEARNING SYSTEMS DAY 1B

Dr Zhu Fangming
Institute of Systems Science
National University of Singapore
fangming@nus.edu.sg

Not be reproduced in any form or by any means, without the written permission of ISS, NUS, other than for the purpose for which it has been supplied.

1.2

Neural Network Basics

- Introduction to neural networks
- Biological neuron and artificial neuron
- General architecture of neural networks
- Single-layer perceptron
- Multilayer perceptron (MLP)
- Backpropagation learning
- Important issues in training

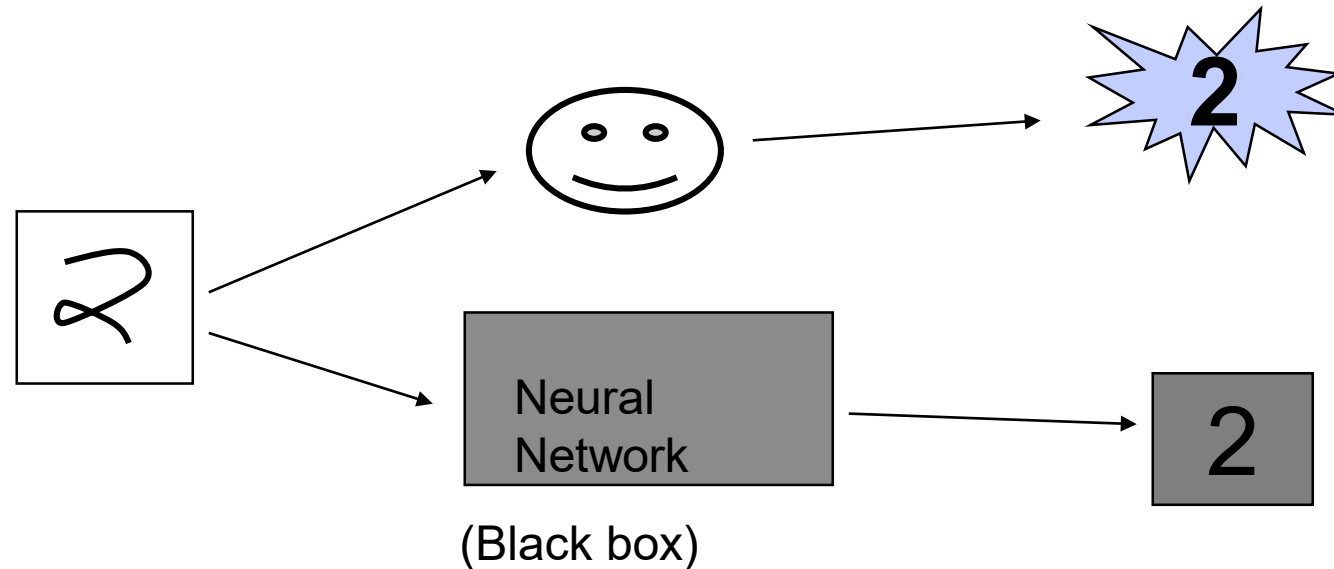
Introduction to Neural Networks

- In many real-world applications, computers are expected to perform complex pattern recognition tasks.
- Complex pattern recognition involves thinking & learning
- Learning involves both memorising and generalising
- Recognition of complex patterns needs parallel processing
 - human can easily handle
 - conventional computing paradigms are not suitable to solve this type of problems
- We therefore borrow features from physiology of brain as the basis for our new processing models — *Artificial Neural Networks (ANN) or Neural Networks (NN)*

Example: Character Recognition

- **Example:**

Character Recognition by Human Brain and NN



Some Definitions of Artificial Neural Networks

- ... a neural network is a system composed of many simple processing elements operating in parallel whose function is determined by network structure, connection strengths, and the processing performed at computing elements or nodes.

-----*DARPA Neural Network Study (1988)*

- A neural network is a massively parallel distributed processor that has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:
 - Knowledge is acquired by the network through a learning process.
 - Inter neuron connection strengths known as synaptic weights are used to store the knowledge.

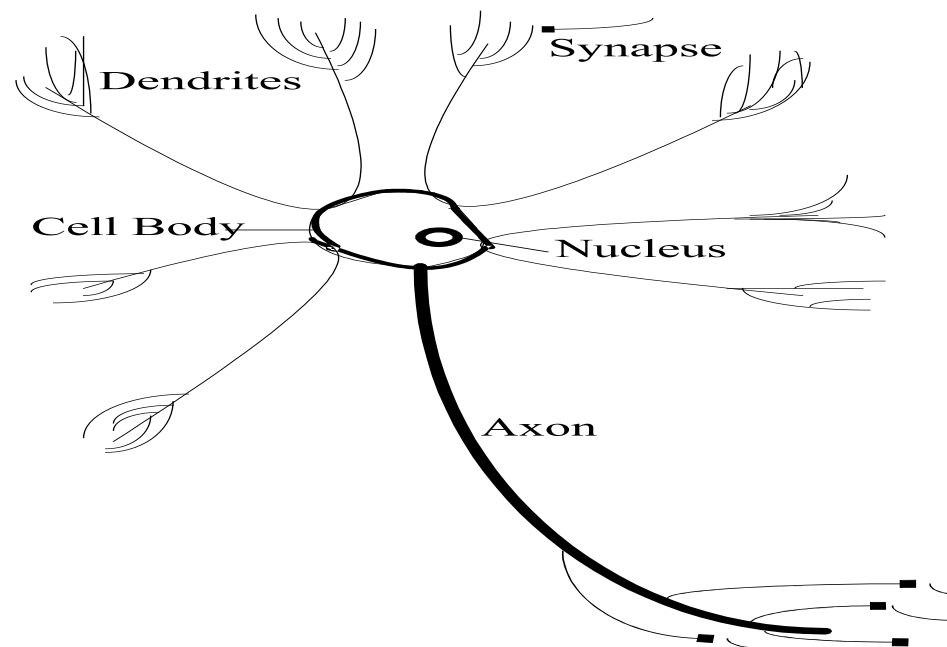
----*Haykin (1994)*

From Biological Neuron to Artificial Neuron

— Biological Inspiration for Artificial Neural Networks

- The basic biological computing element — *the neuron*

This extremely small computer is a multiple signal processor based on electrochemical processing principles



From Biological Neuron to Artificial Neuron ...

- A biological neuron is the basic biological computing element
- A neuron is a small cell that receives electrochemical stimuli from multiple sources and responds by generating electrical impulses that are transmitted to other neurons or effector cells
- There are something like 10^{10} to 10^{12} neurons in the human nervous system and each is capable of storing several bits of “information”
- About 10% of the neurons are input and output, the remaining 90% are interconnected with other neurons which store information or perform various transformations on the signals being propagated through the network

From Biological Neuron to Artificial Neuron — An Artificial Neuron

An artificial neuron — A Single Neural Computing Element

- **N input signals each weighted for its importance**
- **Signals are added to produce a cumulative (net) input**
- **The neuron transforms the net input to produce an output**

- **$\text{net} = \sum_i x_i w_i$**

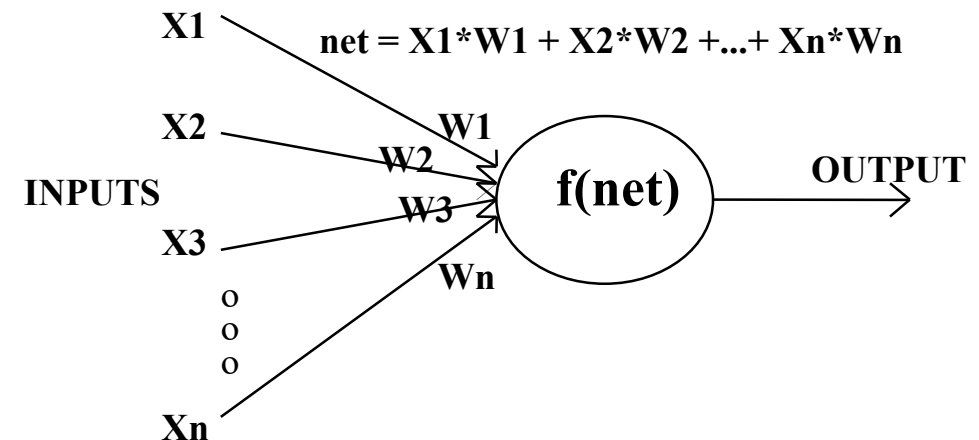
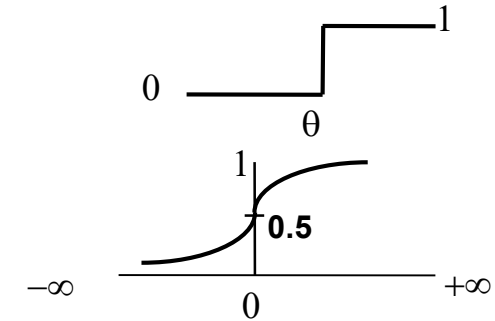
- **Activation / transfer function**

- **hard-limiting function**

$$f(\text{net}) = \begin{cases} 0 & \text{if } \text{net} < \text{threshold} \\ 1 & \text{otherwise} \end{cases}$$

- **sigmoid function**

$$f(\text{net}) = \frac{1}{1 + e^{-\text{net}}}$$



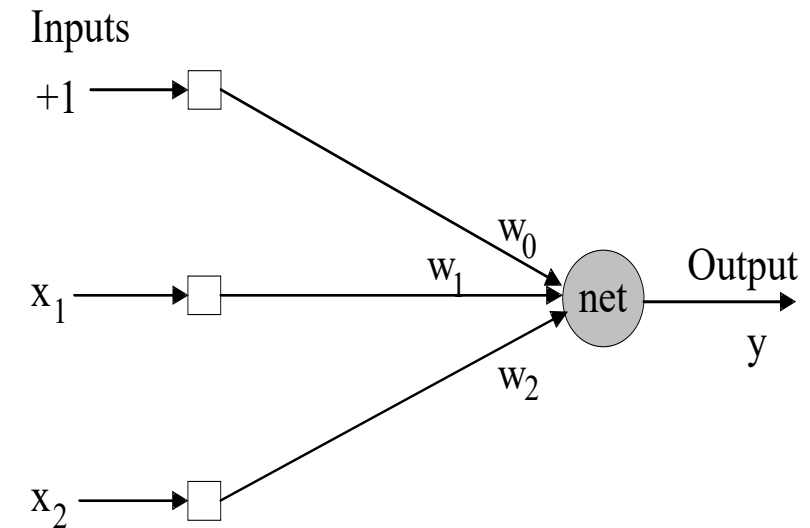
An Example of Simple NN

— two inputs & bias —

- The bias input is a fixed positive signal. The weight w_0 on the bias input is also adjustable
- $y = f(\text{net}) = f(w_0 + x_1w_1 + x_2w_2)$
- Suppose the activation function is a hard-limiting function:

$$f(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

- Instances:
 - $x_1 = 1, x_2 = 0, w_0 = -3, w_1 = 2, w_2 = 6,$
 $\text{net} = -1, y = 0$
 - $x_1 = 1, x_2 = 0, w_0 = -2, w_1 = 3, w_2 = 5,$
 $\text{net} = 1, y = 1$



Why & How NN Can Learn

- From the simple NN example, we can see that given the same inputs, the output will be different when different weights are used.
- **Why NN can learn**
 - by changing weights, the behaviour of neural network can be changed
- **How NN can learn**
 - changing weights purposely to reduce the difference between the actual output and the expected output

What NN Can Do

- NNs can learn to model complex systems from examples
- NNs are good at difficult perception tasks like vision or speech understanding
- NNs can often outperform other methods such as statistical pattern recognition and regression techniques
- NNs can mimic many intelligent traits found in humans:
learning, generalisation, tolerance to missing or noisy data, associative recall
- Parallel processing is more powerful and faster than sequential processing
- NNs are tolerant to faults. They are more reliable and robust than conventional systems

Applications of Neural Networks

- **Some of the successful applications of neural networks**
 - Content addressable memories
 - Process control
 - Data compression
 - Fault diagnosis
 - Forecasting time series
 - General mapping
 - Functional approximation
 - Multi-sensor data fusion
 - Optimization
 - Pattern recognition
 - Image Processing
 - Natural language processing
 - Data mining
 - Data visualization
- **Areas**
 - » All branches of Science and Engineering
 - » Economics, Finance, Management, Social Sciences
 - » Medicine, Agriculture

Some of Pioneering Research Work of Neural Networks

- **W. McCulloch & W. Pitts (1943): Neurons as computers**
 - described a logical calculus of neural networks that united the studies of neurophysiology and mathematical logic
 - showed that a network with a sufficient number of simple units, and synaptic connections would, in principle, compute any function
 - Influenced by this idea, von Neumann used idealized switch-delay elements derived from the McCulloch-Pitts neuron for the development of first generation of computer
- **D. Hebb (1949): Neural learning theory**
 - postulate of learning: the effectiveness of a variable synapse between two neurons is increased by the repeated activation on one neuron by the other across that synapse
 - provided a source of inspiration for the development of computational models of learning and adaptive systems
- **M. Minsky (1954, 1961):**
 - Theory of Neural-Analog reinforcement system
 - neural network

Some of Pioneering Research Work of Neural Networks ...

- **W.K. Taylor (1956): Associative memory**
 - contributions also from Steinbuch, Willshaw and others
 - Anderson (1972), Kohonen (1972), Nakano (1972): correlation matrix memory
- **F. Rosenblatt (1958), M. Minsky & S. Papert (1969): Perceptrons**
 - a novel method of supervised learning
 - perceptron convergence theorem (Rosenblatt 1960)
- **B. Widrow & Hoff (1960): Least mean-square (LSM) algorithm**
 - Adaline (ADaptive LINear Element)
 - Madaline (Multiple ADALINE) (Widrow 1962): one of the earliest trainable layered neural networks
- **S. Amari (1967): Stochastic gradient method for adaptive pattern classification**
- **S. Grossberg (1970s, 1980): Adaptive Resonance Theory (ART)**
 - established a new principle of self-organization

Some of Pioneering Research Work of Neural Networks ...

- **L. Cooper et al. (1982, 1987): Reduced Coulomb Energy networks**
 - an architecture with incremental learning capability
- **J. Hopfield (1982): Recurrent Hopfield Nets**
 - the principle of storing information in dynamically stable network
- **T. Kohonen (1982): Self-organizing Maps**
 - using a one- or two-dimensional lattice structure
- **Barto, Sutton, & Anderson (1983): Reinforcement learning**
- **D. Rumelhart & et al. (1986): back-propagation algorithm (BP)**
- **R. Hecht-Nielsen (1987): counter propagation neural network**
- **Jang, Pal and Group, Keller and others (1992)**
 - Fuzzy Neural Nets, Neuro-Fuzzy Systems etc

Some of Pioneering Research Work of Neural Networks ...

- **Broomhead & Lowe (1988), Moody & Darken (1989): Radial Basis Function (RBF) network**
 - provided an alternative to multilayer perceptrons
- **P. Baldi and K. Hornik (1989): Auto associative neural networks**
- **D.F. Specht (1991): General Regression Neural Networks (GRNN)**
- **A. M. Kramer (1991): Nonlinear principal component neural networks**
- **Nikola Kasabov (1998): Evolving connectionist systems (ECOS) for real time data mining**
- **Geoffrey Hinton, Yann LeCun, et al. Since 2000,**

Deep Learning Neural Network, Convolutional Neural Networks (CNN), RNN, LSTM, GAN

- **A neural network has a parallel-distributed architecture**
 - with a large number of nodes and connections
 - each connection points from one node to another and is associated with a weight
- **Construction of a neural network involves the following tasks:**
 - determine the network properties:
 - the network topology (framework as well as interconnection scheme)
 - the type of connections
 - the range of weight
 - determine the node properties:
 - the activation range
 - the activation (transfer) function
 - determine the system dynamics:
 - the weight initialisation scheme
 - the activation-calculating formula
 - the learning rule (algorithm)

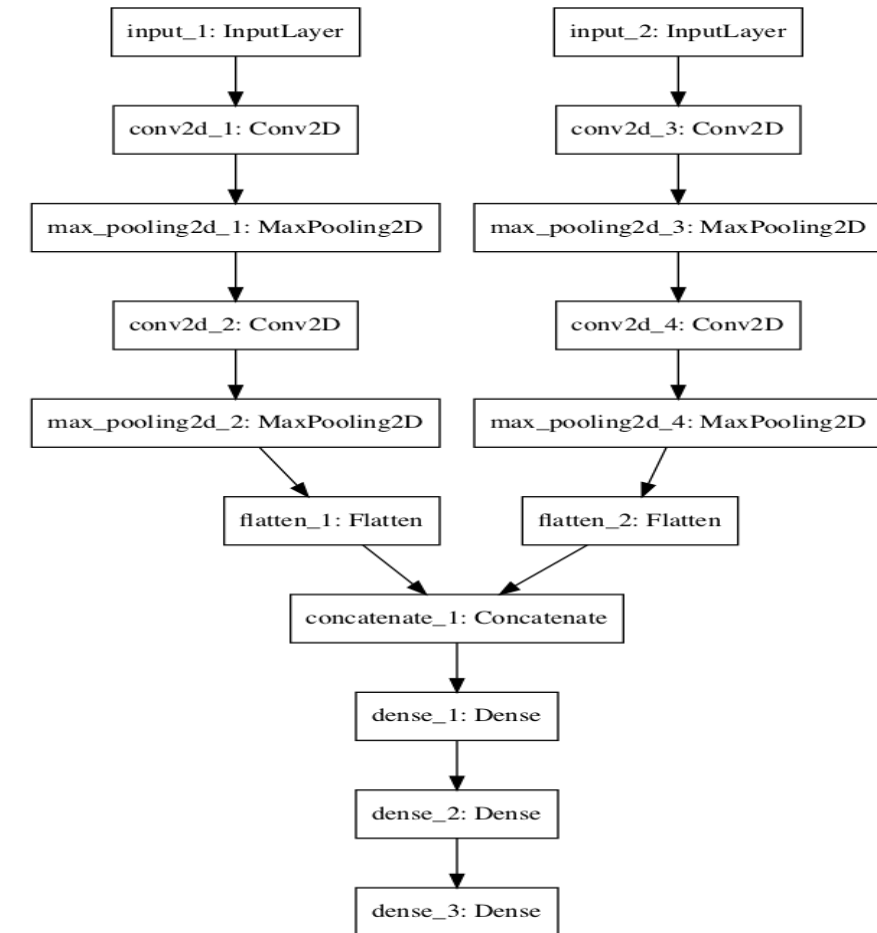
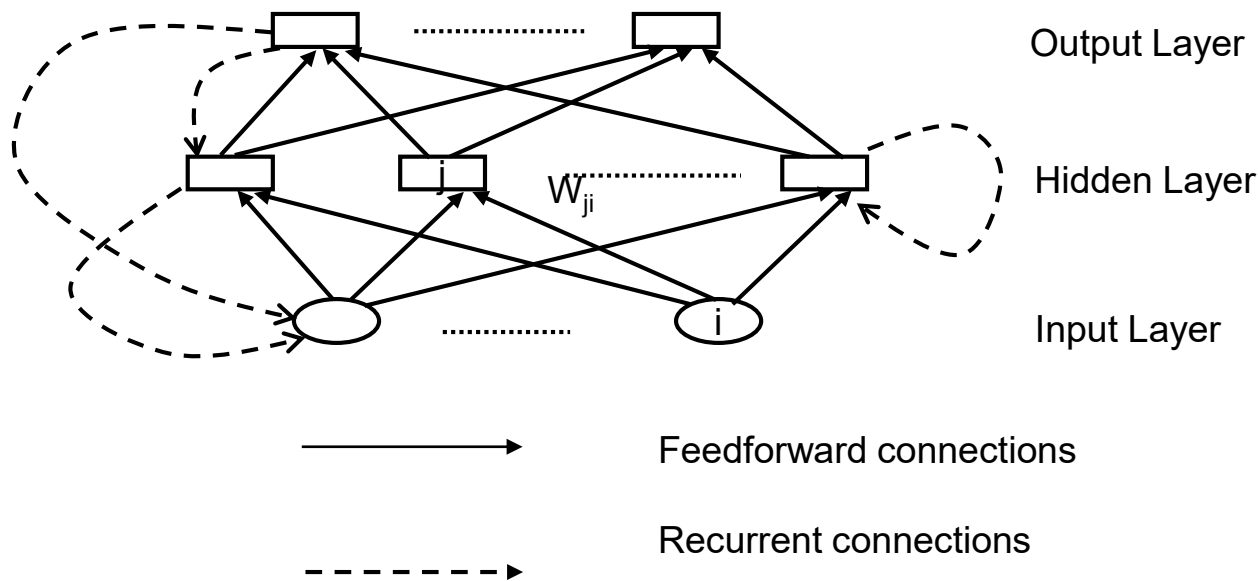
- **Framework (in general, but not for all NNs)**
 - the input layer — input units (nodes)
 - encode the instance presented to the network for processing
 - example: attribute values of an object
 - the hidden layer — hidden units (nodes)
 - are not directly observable
 - provide non-linearity for the network
 - an NN may have more than one hidden layers
 - the output layer — output units (nodes)
 - encode possible concepts (or values) to be assigned to the instance under consideration
 - example: a class of objects

General Architecture of Neural Networks (cont.)

— Network Properties —

- **Interconnection scheme**

- feedforward networks
 - all connections point in one direction (input→ output)
- recurrent networks
 - there are feedback connections or loops



Source: <https://machinelearningmastery.com>

General Architecture of Neural Networks (cont.)

— Node Properties —

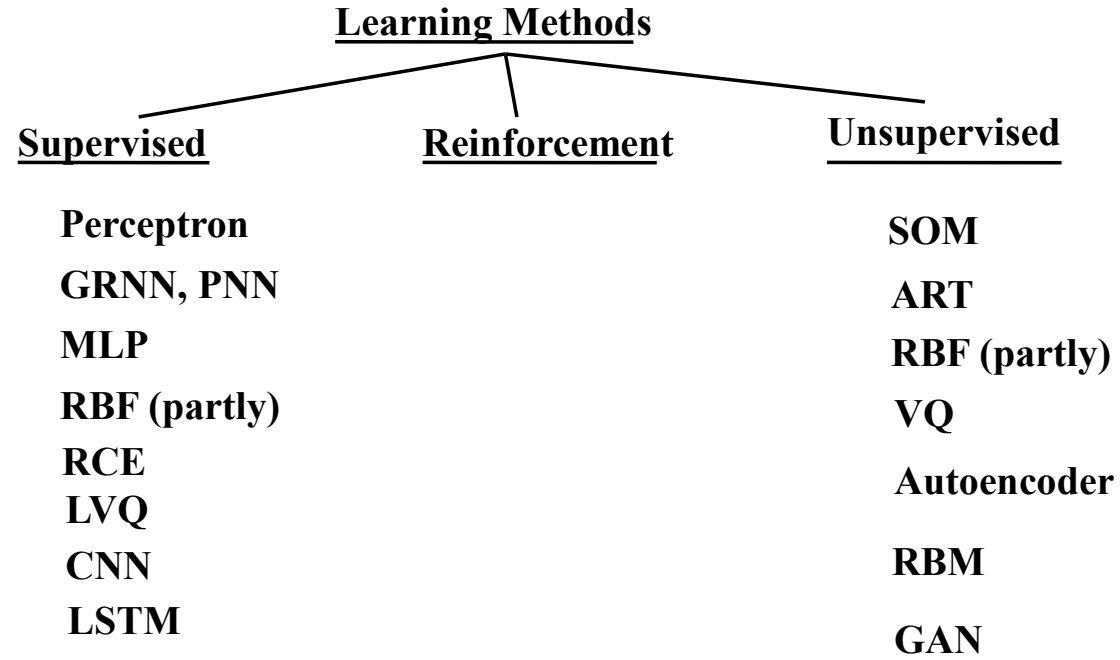
- **The activation level of node can be**
 - discrete (e.g., 0 and 1, when the activation function is hard-limiting)
 - continuous across a range (e.g., [0, 1] when a sigmoid function is used for activation function)
- **Data representation in NNs (for input layer)**
 - **Discrete feature value**
 - can be encoded by a single input unit. In this case, an activation 1 corresponds to “yes” and 0 corresponds “no”.
 - **Continuous feature value**
 - by continuous representation: each feature is encoded by an input unit, the feature value is mapped into the unit activation (normalised to the interval [0, 1])
 - by discrete representation: the range of continuous values is divided into multiple intervals, each interval encoded by an input unit
- **Description of training data**
 - input vector (feature vector) e.g. [0, 1, 0, ... , 1, 0]
 - output vector

General Architecture of Neural Networks (cont.)

— System Dynamics —

- **The weight initialisation scheme**
 - it is specific to the particular neural network model chosen
 - in many cases, initial weights are randomised to small real numbers
- **The learning rule**
 - one of the most important features to specify for a neural network
 - determine how to adapt connection weights in order to optimise the network performance
 - indicate how to calculate the weight adjustment during each training cycle
 - is suspended after training is completed
- **The activation calculation**
 - the important inference behaviour of a NN — how to compute the activation levels across the network
 - training of a NN also involves the same calculation — weight adjustment based on the errors (differences between the actual activation and the desired activation)

Category of Neural Network Learning — by Learning Methods —



MLP – Multi-layer Perceptron

RCE — Reduced Coulomb Energy

GRNN — General (Generalized) Regression Neural Network

PNN — Probabilistic Neural Network

RBF — Radial Basis Function

SOM — Self-Organizing Map

ART — Adaptive Resonance Theory

VQ — Vector Quantization

LVQ — Learning Vector Quantization

CNN — Convolutional Neural Network

LSTM — Long Short Term Memory

GAN — Generative Adversarial Networks

RBM — Restricted Boltzmann Machines

Single-layer Perceptron

- The activation function employed is a hard-limiting function

- $I = \sum_i x_i w_i$
- hard-limiting function

$$f(I) = \begin{cases} 0 & \text{if } I < \theta \text{ (threshold)} \\ 1 & \text{otherwise} \end{cases}$$

- Perceptron learning algorithm as a formula

$$\Delta W_i = \alpha (T - O) X_i = \alpha \times \text{error} \times \text{input-}i$$

$$W_i(t+1) = W_i(t) + \Delta W_i$$

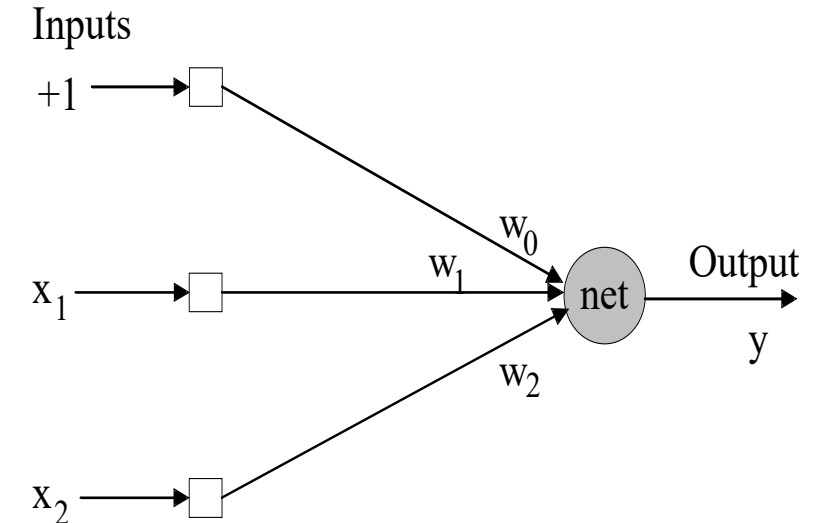
where $W_i(t)$ — the weight at time t (or t -th iteration) ΔW_i — the change made to weight W_i

X_i — the i -th input

α — learning step ($0 < \alpha < 1$)

T — target output

O — perceptron output



- **Perceptron learning rule**

- If the output is ONE and should be ONE or if the output is ZERO and should be ZERO (*no error*), do nothing (no change to weights).
- If the output is ZERO (inactive) and should be ONE (active), increase the weight values on all active input links.
- If the output is ONE (active) and should be ZERO (inactive), decrease the weight values on all active input links.

Example: Learning in Single Perceptron

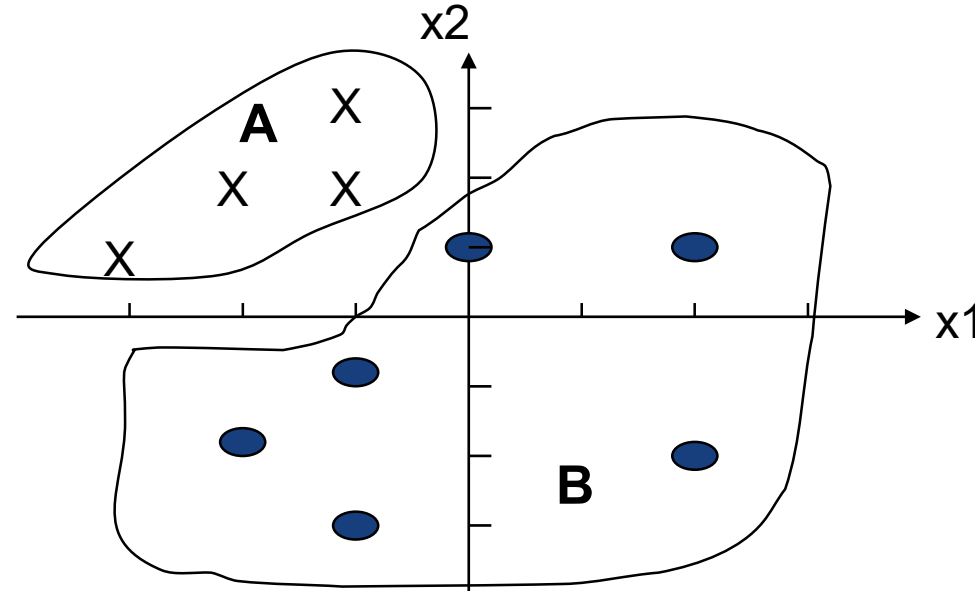
- An example of pattern classification on x_1 - x_2 space

Class A with four patterns given

$(-3, 1)$, $(-2, 2)$, $(-1, 2)$, $(-1, 3)$

Class B with six patterns given

$(-1, -1)$, $(-2, -2)$, $(-1, -3)$, $(2, -2)$, $(2, 1)$, $(0, 1)$



Example: Learning in Single Perceptron ...

- **Single perceptron**

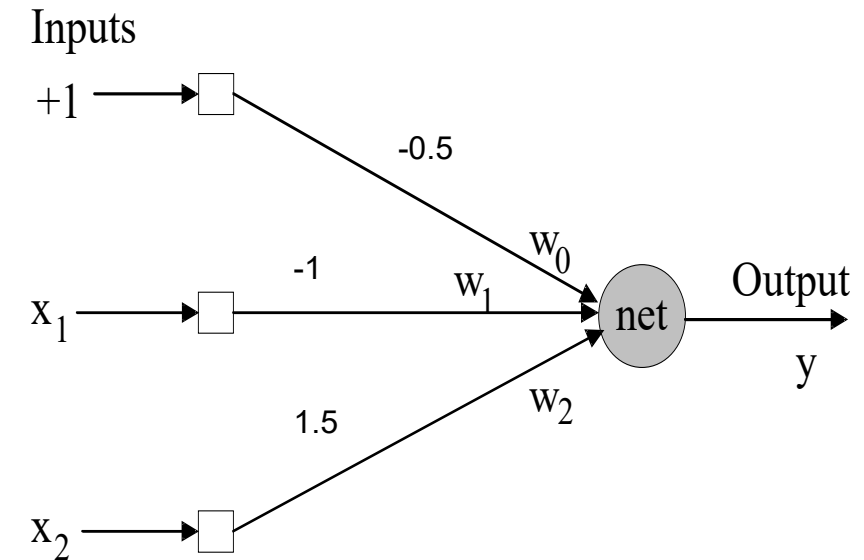
- $I = \sum_i x_i w_i$
- hard-limiting function

$$y = f(I) = \begin{cases} 0 & \text{if } I \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

- $y = 1$ indicate class A
- $y = 0$ indicate class B
- let initial weights be

$w_0 = -0.5, w_1 = -1, w_2 = 1.5$ learning step $\alpha = 0.5$

- using the 10 given patterns for training (assuming the sequence as appearance)



Example: Learning in Single Perceptron ...

- **for class A:** **No error for all four patterns**
 $(-3, 1), (-2, 2), (-1, 2), (-1, 3)$
- **for class B:** **No error for five patterns**
 $(-1, -1), (-2, -2), (-1, -3), (2, -2), (2, 1)$
- **there is an error for the pattern $(0, 1)$ of class B**
- **learning**

$$\Delta W_0 = \alpha (T - O) X_0 = 0.5 \times (0 - 1) \times 1 = -0.5$$

$$\Delta W_1 = \alpha (T - O) X_1 = 0.5 \times (0 - 1) \times 0 = 0$$

$$\Delta W_2 = \alpha (T - O) X_2 = 0.5 \times (0 - 1) \times 1 = -0.5$$

- **modified weights**

$$W'_0 = -0.5 + (-0.5) = -1$$

$$W'_1 = -1$$

$$W'_2 = 1.5 + (-0.5) = 1$$

- **with the new set of weights, all ten patterns are classified correctly (verify by yourself)**

Single Perceptron and Linear Separability

— An Example in 2-dimensional Space —

- Equation for straight line

$$x_2 = a \times x_1 + b$$

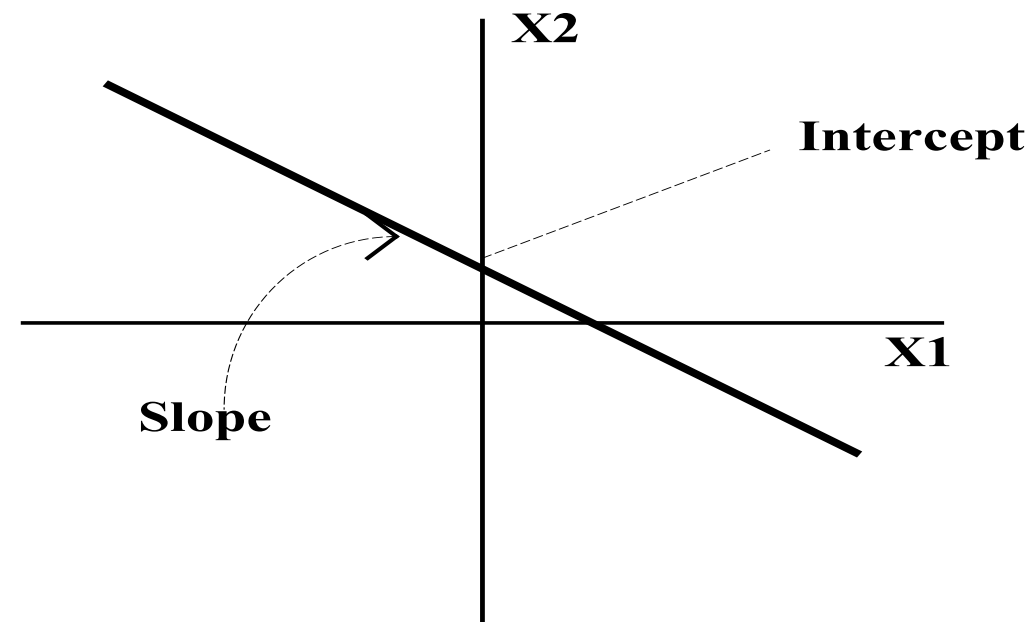
- Weighted sum for single perceptron

$$w_0 + w_1 \times x_1 + w_2 \times x_2 = 0$$

$$x_2 = - (w_1/w_2) x_1 - (w_0/w_2)$$

$- (w_0/w_2)$ — **intercept**

$- (w_1/w_2)$ — **slope**



Single Perceptron and Linear Separability...

For previous example

- **N_0 , the original neuron**

$w_0 = -0.5, w_1 = -1, w_2 = 1.5$ slope = $2/3$, intercept = $1/3$

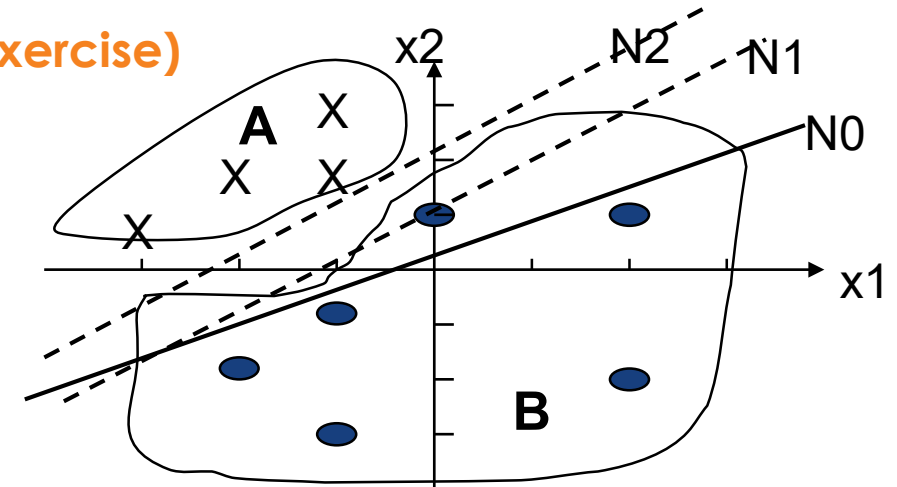
- **N_1 , the neuron after learning**

$w_0 = -1, w_1 = -1, w_2 = 1$ slope = 1, intercept = 1

- another set of weights represent the same line:

$w_0 = -2, w_1 = -2, w_2 = 2$

- **N_2 , a neuron can also solve the same problem (for your exercise)**



Single Perceptron and Linear Separability ...

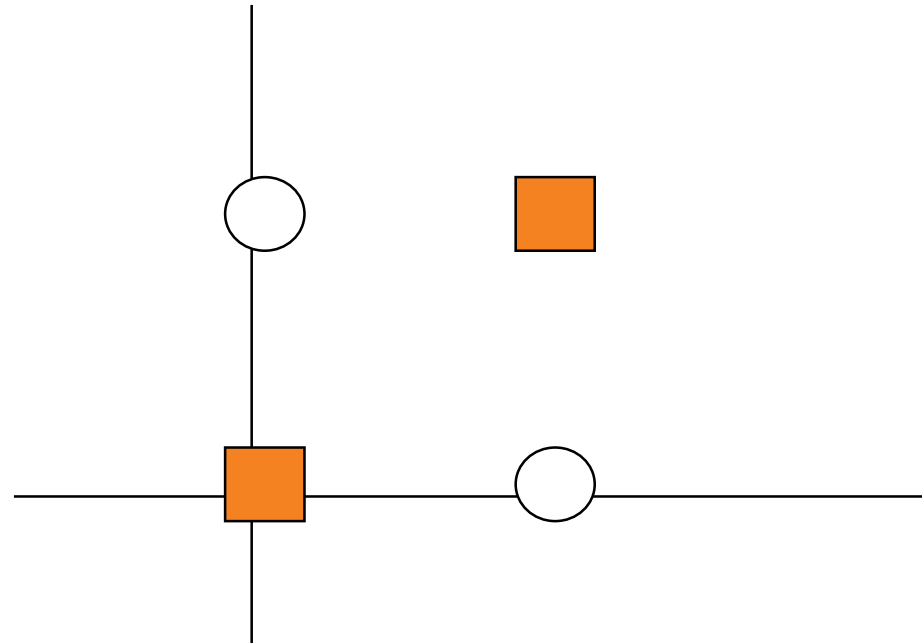
XOR Problem

- **four patterns**

- $(0, 0) \rightarrow 0$ (false)
- $(0, 1) \rightarrow 1$ (true)
- $(1, 0) \rightarrow 1$ (true)
- $(1, 1) \rightarrow 0$ (false)

- **initial weights**

- $w_0 = 1, w_1 = 1, w_2 = 1$
- learning step $\alpha = 0.5$



- **Can we train a perceptron to solve this problem?
Why?**

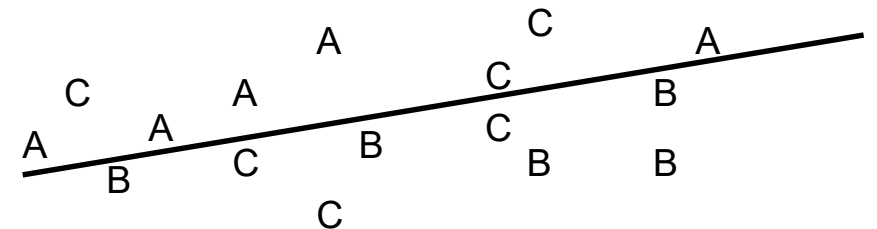
Single Perceptron and Linear Separability ...

- In the activation function

$$\sum_{i=1}^n w_i x_i = \theta$$

forms a hyperplane in the n-dimensional space, dividing the space into two halves. Using θ as a threshold to produce output value (1 or 0) means classifying the instances to two classes. When $n = 2$, the hyperplane becomes a line.

- **Linear separability**
 - A data set is called “linearly separable”, when a linear hyper plane exists to place the instances of one class on one side and those of the other class on the other side.
- A single perceptron can only solve a classification problem when it is linearly separable.
- Many classification problems are not linearly separable.
 - Sets A and B are linearly separable
 - Sets A and C, sets B and C are not linearly separable



If

- given a set of input vectors, each of them with a desired output, and
- each training case is presented to the network with positive probability

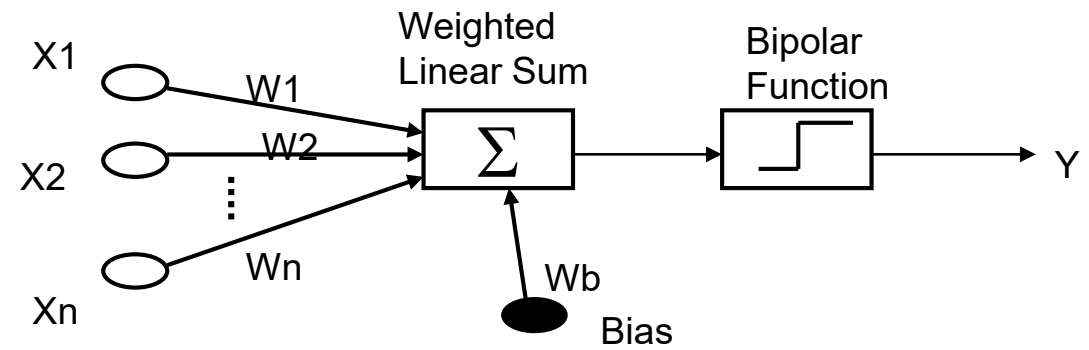
Then

- there is a procedure guaranteed to find a set of weights that give correct outputs if-and-only-if a set of weights exist for the task

A single perceptron will converge only when the problem is linearly separable

ADALINE & Its Learning Rule

- **ADALINE (ADaptive LINear Element) (Widrow, 1959)**
 - a single unit similar to the perceptron
 - has a single output which receives multiple inputs, takes the weighted linear sum of the inputs and passes it to a bipolar function (which produces either +1 or -1, depending on the polarity of the sum)
- **MADALINE (Multiple ADALINE)**
 - a network of ADALINES
- **Learning rule of ADALINE**
 - Widrow-Hoff rule (LMS, least mean square)



Widrow-Hoff Delta Rule

- Linear Outputs:

i^{th} output for pattern p $O_i^p = \sum_k W_{ik} X_k^p$

- Target Output:

$$T_i^p$$

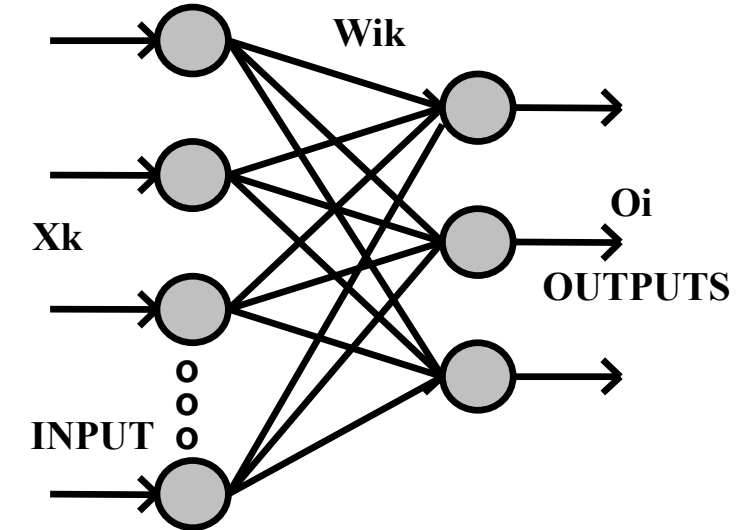
- Error Measure:

$$E(W) = \frac{1}{2} \sum_{ip} (T_i^p - O_i^p)^2 = \frac{1}{2} \sum_{ip} (T_i^p - \sum_k W_{ik} X_k^p)^2$$

- Learning algorithm:

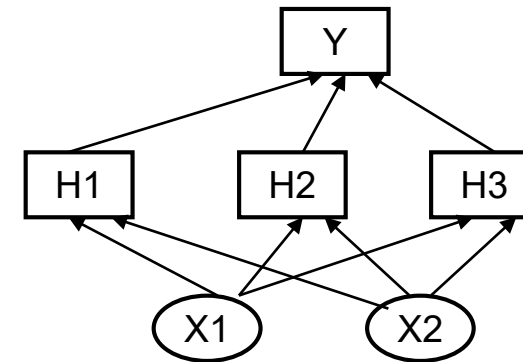
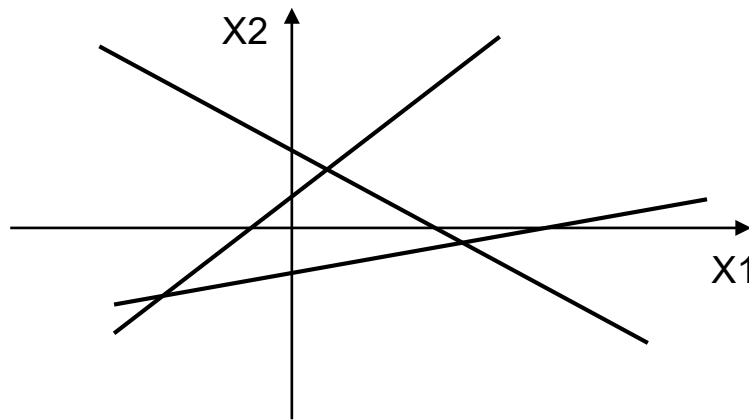
- A form of gradient descent learning: change weight W_{ik} proportional to the negative derivative of error. η is learning rate.

$$\Delta W_{ik} = -\eta \frac{\partial E}{\partial W_{ik}} = \eta (T_i^p - O_i^p) X_k^p = \eta \delta_i^p X_k^p$$



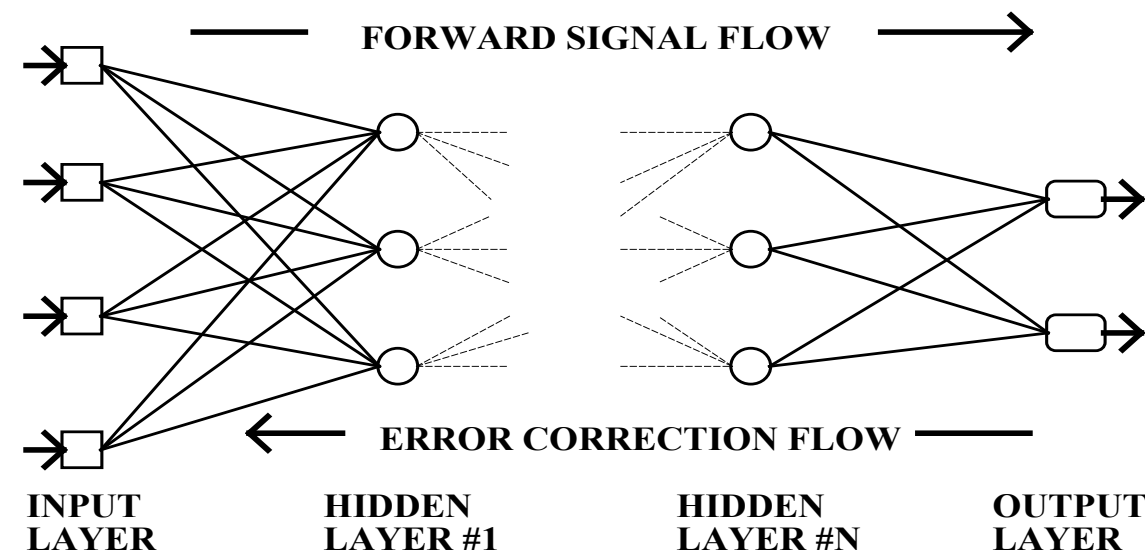
Multilayer Perceptron

- Multilayer perceptron is a feedforward neural network with at least one hidden layer
- It can form more complex decision regions to solve nonlinear classification problem
- Each node in the first layer (above the input layer) can create a hyperplane. Each node in the second layer can combine hyperplanes to create convex decision regions. Each node in the third layer can combine convex regions to form concave regions.
- The delta rule does not apply to training a multilayer network since the error of a hidden unit is not known.
- *The backpropagation learning method can overcome this difficulty*



Multilayer Perceptron and Backpropagation Learning

- Fully connected units
- Feedforward signals only
- One input-layer, one or more hidden-layers and one output layer
- Nonlinear differentiable activation functions
- Weights in hidden layers are adjusted to reduce aggregate errors in the output layer
- A two-stage process:
 - Propagate signals forward and then errors backward



Steps of Backpropagation Algorithm

1. Initialize the weights to small random numbers
2. Randomly select a training pattern pair (x_p , t_p) and present the input pattern x_p to the network. Compute the corresponding network output pattern z_p
3. Compute the error E_p for pattern (x_p , t_p)
4. Backpropagate the errors according to the BP weight adjustment formulas (given later)

5. Test the mean square error (MSE) over P training patterns:
- $$E = \frac{1}{P} \sum_{p=1}^P E^p$$

If the MSE is below the required threshold, stop. Otherwise, repeat steps 2-5.

6. Test for generalization performance if appropriate

BP Errors and Weight Updates

- Total error over all training patterns p and m output units:

$$E_{\text{tot}} = \sum_{p=1}^P E^p$$

Total error, all training patterns

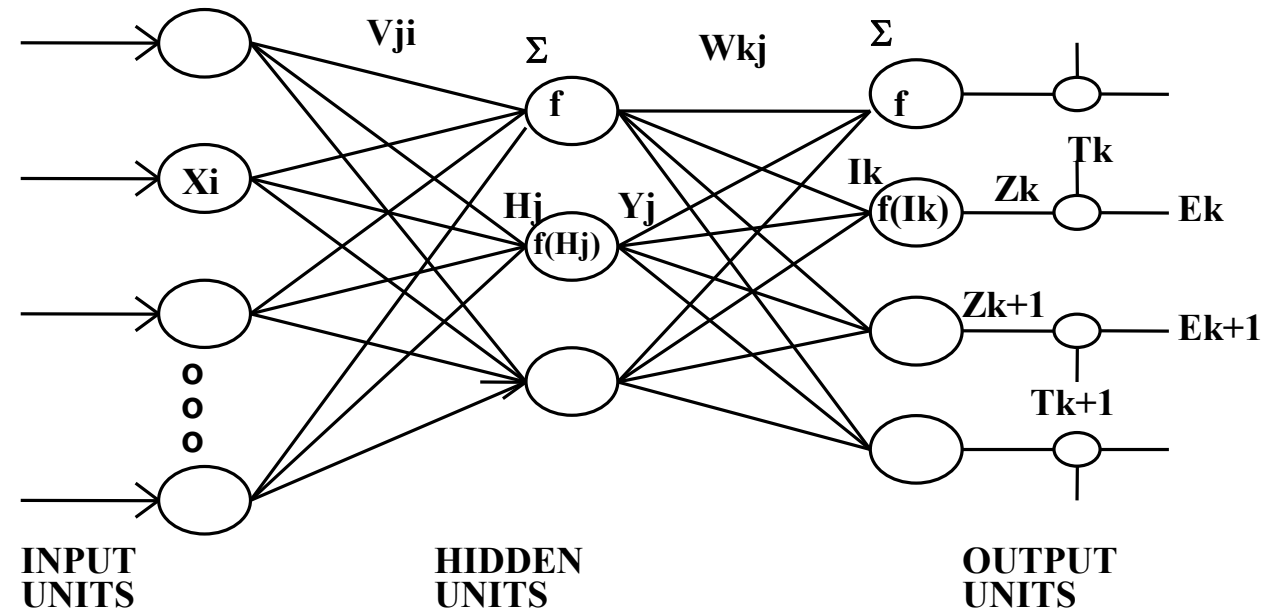
$$E^p = \frac{1}{2} \sum_{k=1}^m (t_k - z_k)^2$$

Error for pattern p

$$\begin{aligned} z_k &= f(I_k) = f\left(\sum_j w_{kj} y_j\right) = f\left(\sum_j w_{kj} f(H_j)\right) = \\ &= f\left(\sum_j w_{kj} f\left(\sum_i v_{ji} x_i\right)\right) \end{aligned}$$

z_k — computed output

t_k — target output



BP Errors and Weight Updates (cont.)

- For output units $k = 1, 2, \dots, m$, adjust the weights to reduce the error for each pattern p (Gradient descent)

$$\Delta w_{kj} = -\eta \frac{\partial E^p}{\partial w_{kj}} = -\frac{\eta}{2} \sum_{k=1}^m \frac{\partial (t_k^p - z_k^p)^2}{\partial w_{kj}}$$

Dropping superscripts p

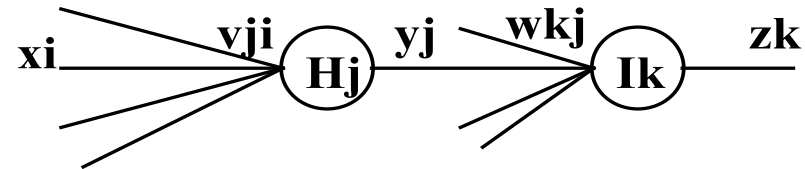
$$\frac{\partial (t_k - z_k)^2}{\partial w_{kj}} = \frac{\partial (t_k - f(I_k))^2}{\partial I_k} \frac{\partial I_k}{\partial w_{kj}} = -2(t_k - z_k) f'(I_k) y_j$$

So

$$\begin{aligned} \Delta w_{kj} &= -\eta \frac{\partial E^p}{\partial w_{kj}} = -\frac{\eta}{2} \sum_{k=1}^m \frac{\partial (t_k^p - z_k^p)^2}{\partial w_{kj}} \\ &= \eta (t_k - z_k) f'(I_k) y_j = \eta \delta_k y_j \end{aligned}$$

- For hidden units, use the chain rule, get

$$\Delta v_{ji} = \eta f'(H_j) x_i \sum_k \delta_k w_{kj}$$



Summary of Backpropagation Algorithm

- **Weights initialization**

Set all weights to random numbers following Uniform distribution in the range $(-1,1)$

- **Calculation of activation**

- the activation level of an input unit is determined by the instance presented to the network
- the activation level O_j of a hidden and output unit is determined by

$O_j = f(\sum W_{ji}O_i - \theta_j)$, where W_{ji} is the weight from an input O_i , θ_j is the node threshold, f is the transfer function.

- **Weight Updating**

- 1) start at the output nodes and work backward to the hidden layers recursively, adjust weights by
 $W_{ji}^{(t+1)} = W_{ji}^{(t)} + \Delta W_{ji}$
where $W_{ji}^{(t)}$ is the weight from unit i to j at time t (or t -th iteration), ΔW_{ji} is the weight adjustment

Summary of Backpropagation Algorithm (cont.)

(Assume a sigmoid function $f(a) = 1/[1 + e^{-a}]$ is used)

- **Weight Updating (cont.)**

2) The weight change is computed by $\Delta W_{ji} = \eta \delta_j O_i$
where η is a trial-independent learning rate ($0 < \eta < 1$),

δ_j is the error gradient at unit j

3) The error gradient

3a) for the output units: $\delta_j = O_j(1 - O_j)(T_j - O_j)$

where T_j is the desired (target) output activation and O_j is the actual output activation at output unit j




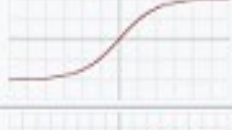
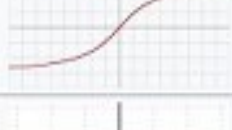

3b) for the hidden units: $\delta_j = O_j(1 - O_j) \sum_k \delta_k W_{kj}$

where δ_k is the error gradient at unit k to which a connection points from hidden unit j

- Repeat iterations until convergence in terms of the selected error criterion. An iteration includes presenting an instance, calculating activations, and modifying weights.

Typical Activation Functions

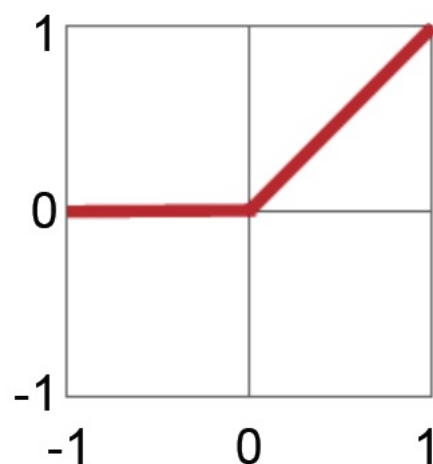
**SIGMOID
FUNCTION** ==

Name ⇅	Plot ⇅	Equation ⇅	Derivative (with respect to x) ⇅
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a. Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
Arc Tan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Softsign [7][8]		$f(x) = \frac{x}{1 + x }$	$f'(x) = \frac{1}{(1 + x)^2}$

<https://www.codeproject.com/Articles/1200392/Neural-Network>

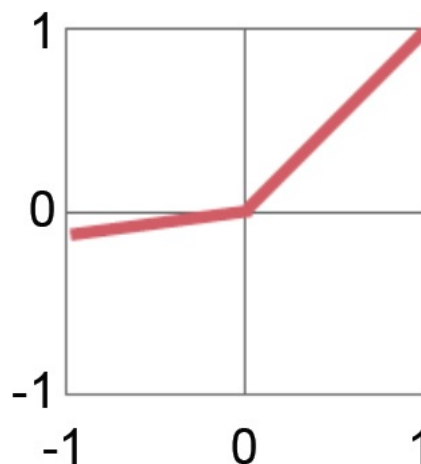
Typical Activation Function

**Rectified Linear Unit
(ReLU)**



$$y = \max(0, x)$$

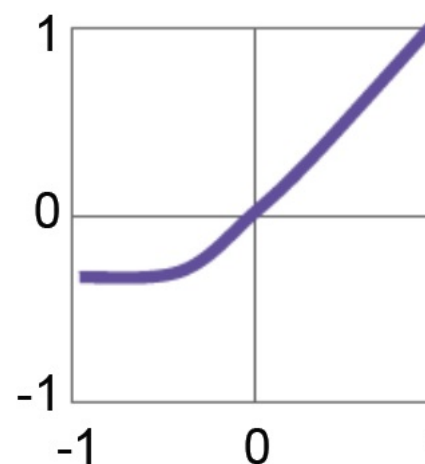
Leaky ReLU



$$y = \max(\alpha x, x)$$

α = small const. (e.g. 0.1)

Exponential LU



$$y = \begin{cases} x, & x \geq 0 \\ \alpha(e^x - 1), & x < 0 \end{cases}$$

Source: Caffe Tutorial

Typical Activation Functions

- Normalizing the output vector with a softmax function:

$$\sigma : \mathbb{R}^K \rightarrow (0, 1)^K$$
$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \quad \text{for } j = 1, \dots, K.$$

- They serve as a multinomial probability distribution.
- The softmax function is often used in the final layer of a neural network-based classifier.

An Example of Backpropagation — XOR problem —

- Initial weights

$$W_{12} = -0.02, W_{13} = 0.02, W_{14} = 0.03, W_{1b} = -0.01, W_{23} = 0.01, W_{24} = 0.02, W_{2b} = -0.01$$

- | Input | Output | Input | Output |
|--------|--------|--------|--------|
| (1, 1) | 0 | (0, 1) | 1 |
| (0, 0) | 0 | (1, 0) | 1 |

- Calculation of activation (assume $b=1.0$ and sigmoid activation function used)

$$O_3 = O_4 = 1$$

$$O_2 = 1/[1+e^{-(1 \times W_{23} + 1 \times W_{24} + 1 \times W_{2b})}] = 1/[1+e^{-(1 \times 0.01 + 1 \times 0.02 - 1 \times 0.01)}] = 0.505$$

$$O_1 = 1/[1+e^{-(0.505 \times W_{12} + 1 \times W_{13} + 1 \times W_{14} + 1 \times W_{1b})}] = 1/[1+e^{-(0.505 \times (-0.02) + 1 \times 0.02 + 1 \times 0.03 - 1 \times 0.01)}] = 0.508$$

- Weight training (assume that the learning rate $\eta = 0.3$)

$$\delta_1 = 0.508(1 - 0.508)(0 - 0.508) = -0.127$$

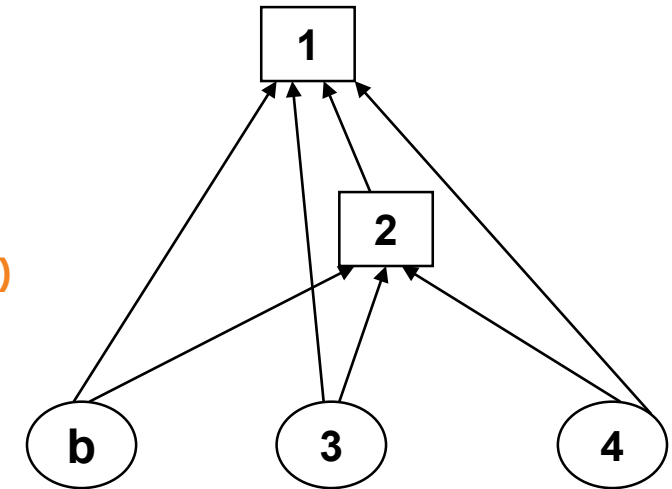
$$\Delta W_{13} = 0.3 \times (-0.127) \times 1 = -0.038$$

$$\delta_2 = 0.505(1 - 0.505)[(-0.127) \times (-0.02)] = 0.0006$$

$$\Delta W_{23} = 0.3 \times 0.0006 \times 1 = 0.00018 \quad (\text{omit the rest})$$

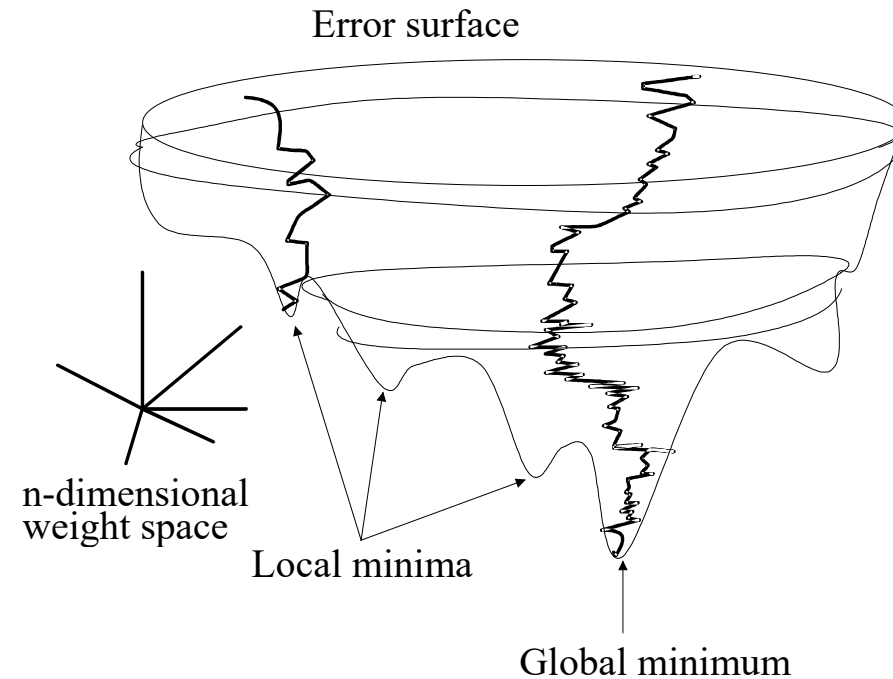
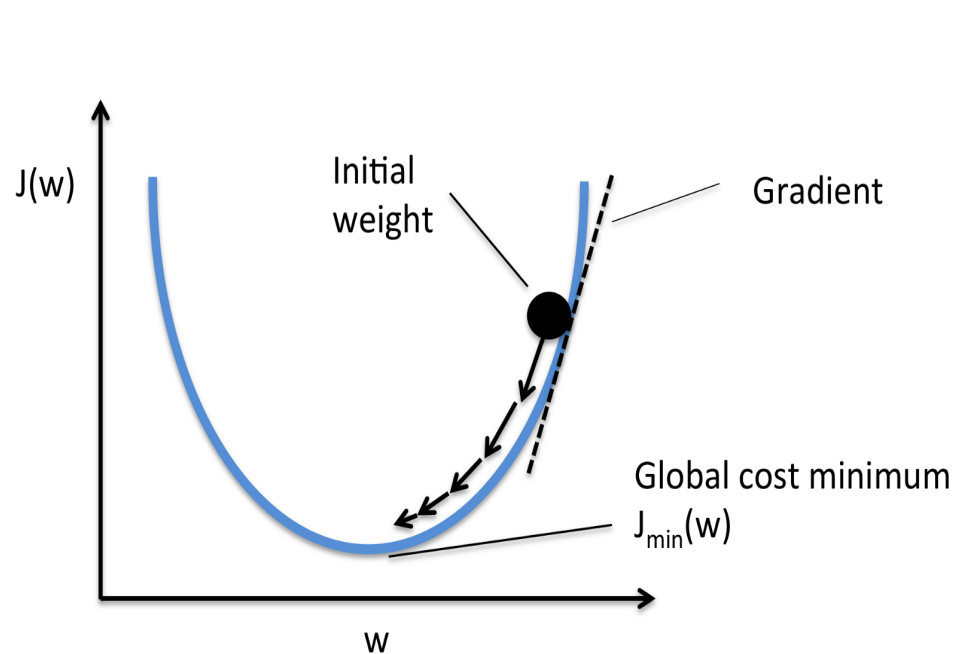
- After many iterations, a set of final weights give the mean squared error of less than 0.01:

$$W_{12} = -11.30, W_{13} = 4.98, W_{14} = 4.98, W_{1b} = -2.16, W_{23} = 5.62, W_{24} = 5.62, W_{2b} = -8.83$$



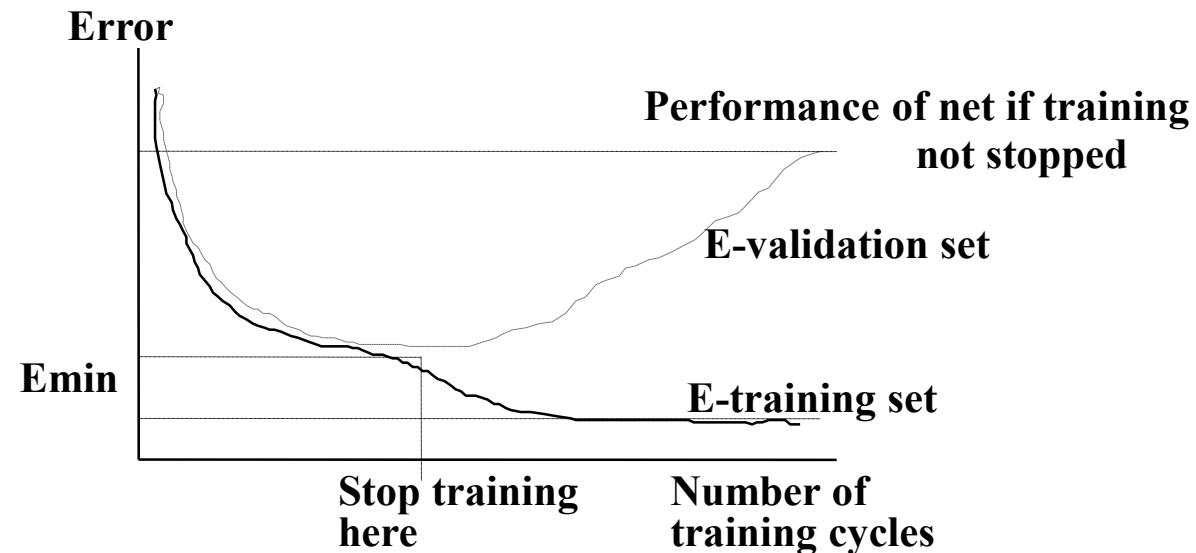
Issues of Training: Local Minimum

- Error surface: one global minimum, but many local minima
- BP is never assured of finding a global minimum.



Issues of Training: Generalization, Overtraining

- **Generalization** is the ability of a network to correctly classify a pattern it has not seen (not been trained on). NNs generalize when they recall full patterns from partial or noisy input patterns, when they recognize patterns not previously trained on or when they predict new outcomes from past behaviors.
- Networks can be **overtrained**. It means that they memorize the training set and are unable to generalize well.



Avoid Overtraining & Achieving Good Generalization

- **Overtraining can usually be avoided by:**
 - choosing a large training set when possible
 - selecting the patterns randomly during training
 - stopping the training process before excessive training occurs (monitoring the training process and stopping after the error drops to a fairly low level)
 - introducing noise directly into the training patterns
 - eliminate unnecessary hidden-layer nodes & weights
 - Adding regularization terms such as L2 regularization
 - using a combination of the above
 - Using a global optimization based training algorithm, e.g. genetic algorithm (GA)
- **Generalization is influenced by the size of training set, the architecture of the network, and the physical complexity of the problem at hand. In practice, we need the size of the training set, N , satisfy the condition**

$$N = O\left(\frac{W}{\varepsilon}\right)$$

where W is the total number of free parameters (i.e., weights and biases) in the network, ε denotes the fraction of error rate permitted on the test set, O denotes the order of quantity enclosed within. Eg., with an error of 10%, N should be about 10 times the number of free parameters in the network.

- BP training is based on the gradient descent computations. This process iteratively searches for a set of weights W^* that minimize the error function E over all training pattern pairs, that is

$$E(W^*) = \min_W \left\{ \sum_{p=1}^P E^p(W, x^p) \right\}$$

- For such an optimization problem, a cost function is usually defined to be minimized with respect to a set of variables. In this case, the network variables that optimize the error function E over the training set are the weight values.

Convergence Properties...

- The chosen error measure (or LOSS function) should be differentiable and tends to zero as the collective differences between the target and computed patterns decrease over the entire training set.

- The MSE measure is one of the acceptable functions for this purpose [cross-entropy (preferred for classification), absolute error, mean error are others].

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \tilde{y}_i)^2$$

- cross-entropy cost function (error measure):

$$C = -\frac{1}{n} \sum_x \sum_j \left[y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right]$$

- The search process depends on the shape of the error surface as well as learning algorithm and the training set.

Improving the Rate of Convergence

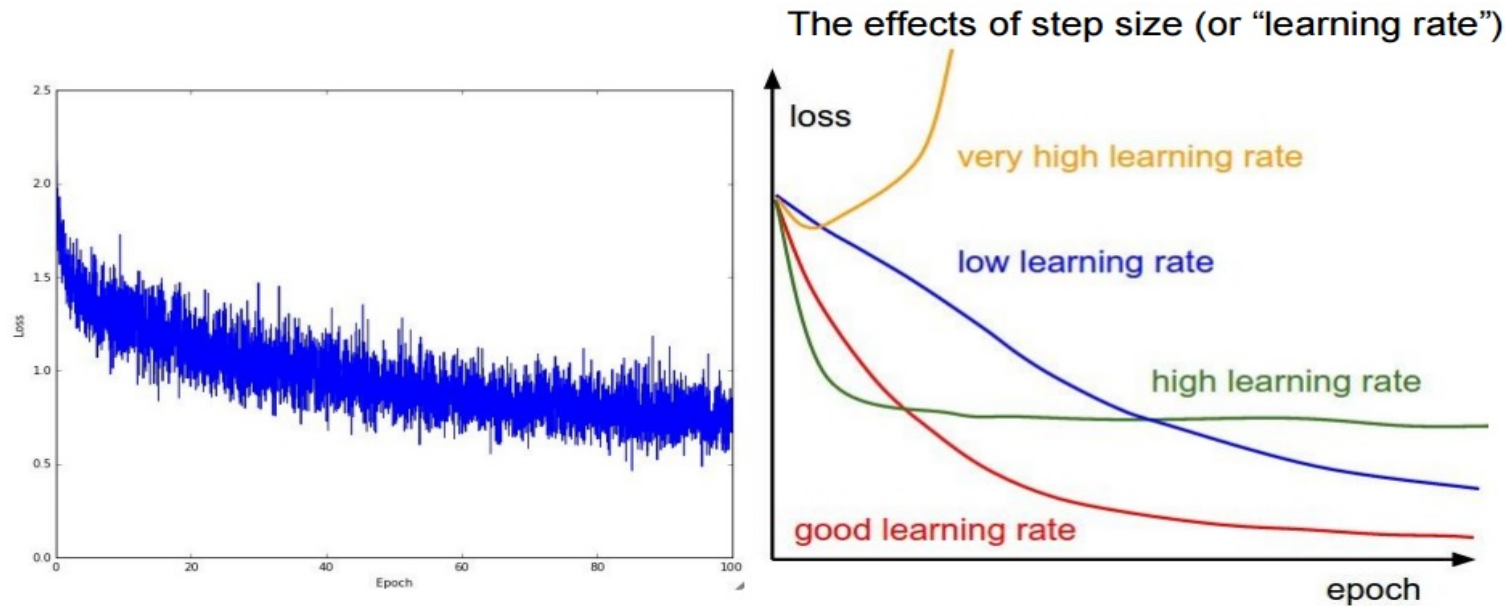
- **Weight initialization**

- It has been shown that setting the initial weights to small (but not too small) random values, say between -1 and +1, is an effective way to avoid shallow troughs and possible entrapment at the start of the training process.
- Estimating initial values
 - Estimate values that are near a minimum solution rather than assigning random values
- LeCun suggested to draw values from a zero-mean Gaussian distribution with standard deviation $1/\sqrt{N}$, N is the number of connections feeding into the node.

- **Choosing learning rate coefficient η**

- it determines the size of the weight adjustment made at each iteration and hence influences the rate of convergence. It should not be constant throughout the learning process for best results.

Improving the Rate of Convergence



Source: Stanford Course

Learning rate can decay over time:

- ❑ step decay: e.g. decay learning rate by half every few epochs.
- ❑ exponential decay: $\eta = \eta_0 * e^{-kt}$
- ❑ 1/t decay: $\eta = \eta_0 / (1+kt)$

Improving the Rate of Convergence (cont.)

- Adding a momentum term to the weight update rule

- It has the effect of smoothing the descent down the error curve by adding an averaging of the weight adjustments (exponential smoothing). This can prevent overshooting off the minimum.

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

where α is the momentum coefficient ($0 < \alpha < 1$).

- Separate momentum terms can be added to each set of the layer weights using different values of α for each layer to achieve even better results.
- The most important parameter, the number of hidden nodes, is empirically chosen as a number around or close to

$$\sqrt{n_{in} \times n_{out}}$$

where n_{in} and n_{out} are respectively the number of input and output nodes (Timothy Masters)

Control Overtraining/ Overfitting

- **Weight Decay (Regularization)**

- To regularize the cost function so that the model parameters will not be tuned to fit the training data too well .

L2 regularization

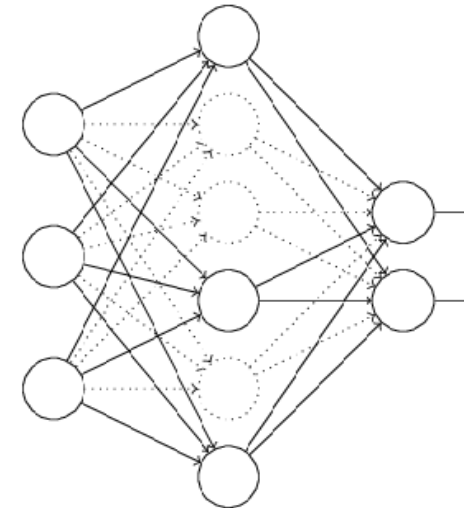
$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2$$

L1 regularization

$$C = C_0 + \frac{\lambda}{n} \sum_w |w|$$

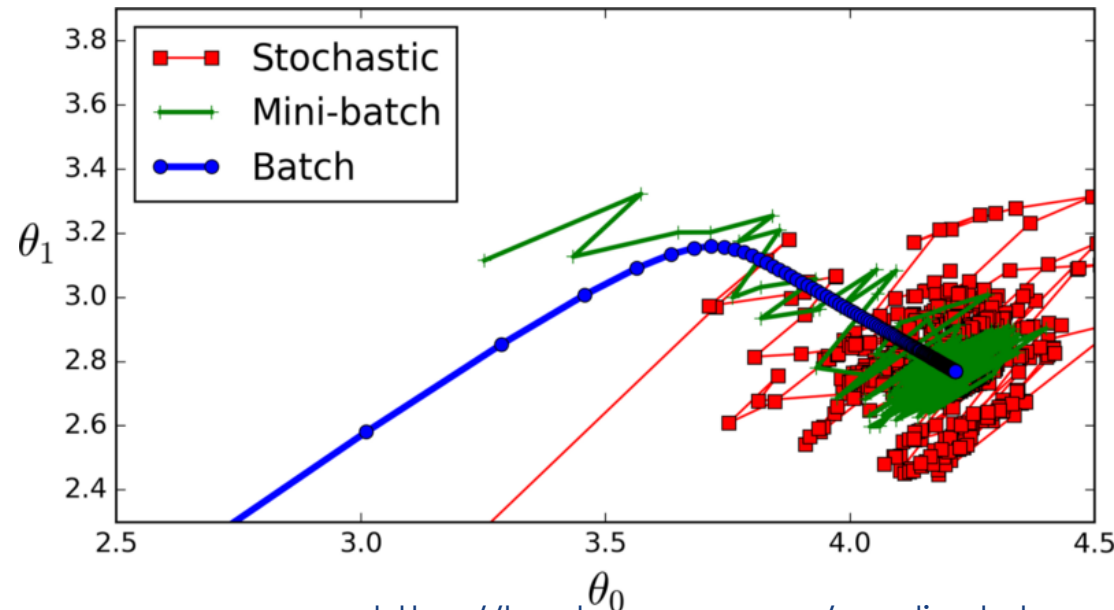
- **Dropout**

- Randomly omit a certain percentage of the hidden neurons during training.



Different Types of Gradient Descent

- **Stochastic GD (SGD):** Just use one single training example with each iteration and feedback error directly.
- **Batch GD:** Loop over all training examples, accumulate errors and then feedback.
- **Mini Batch GD:** Process n training examples at once. This is a good choice for very large data sets.



<https://hackernoon.com/gradient-descent-aynk-7cbe95a778da>

General Design Guidelines

- Carefully select the proper network architecture for the given task (number of hidden nodes) and a good choice of activation functions. Experimentation is usually helpful. More number of hidden nodes than what is required causes “over-fitting” – making its choice a gamble
- Choose a large, reliable training set and divide it into appropriate training, testing and validation sets for use in the corresponding operations
- Pre-process and analyse the training data for maximum utility and effectiveness (such as data smoothing, data normalization, etc.)
- Select good initial training parameter values (η and α) and decide on changes to be made (dynamically) during training (experiment with the use of different values). Consider using modified BP training methods (e.g. Adam, AdaGrad, RMSProp, etc.) when the network is very large or the problem complex
- Once acceptable performance has been realized by the network, optimize it by pruning nodes and weights

1.3

Workshop: Building Multi-Layer Perceptron Neural Networks using Weka and Python

1. Background information

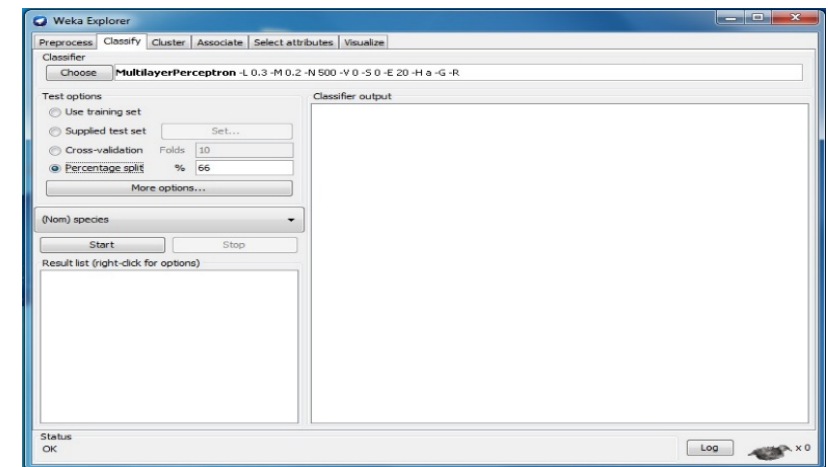
- We wish to train neural networks to predict the class of a flower in the well-known iris flower classification benchmark problem. We are given a data set of 150 samples (patterns) of Iris flowers each with 4 different feature variables representing petal length, petal width, sepal length and sepal width. Each pattern falls into one of the three classes.

2. What to do

- Construct neural network models. Use the given data file “iris.csv” to train and test the network. Note that the last column represents the ‘class label’ as the output in the given data set.
- Check the NN architecture, the number of training iterations, and network performance.

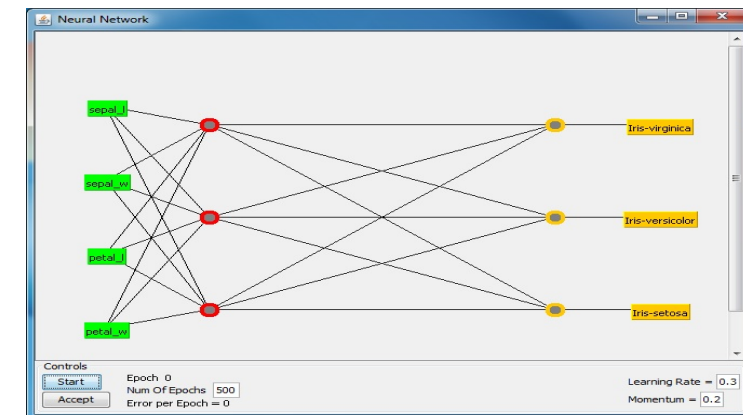
Workshop – Weka - MLP

- **Download and install Weka from**
<http://www.cs.waikato.ac.nz/ml/weka/downloading.html>
- ◆ Launch Weka Explorer
- ◆ Open file... ..iris.csv
- ◆ Check the information about instances, attributes, attribute type, etc.
- ◆ Click “Classify” tab, Choose the classifier as `weka.classifiers.functions.MultiLayerPerceptron`
- ◆ Click the classifier chosen to launch the properties window
- ◆ Mouse over each option to understand its definition or click “More” button to get more explanation
- ◆ Change GUI to “True” -- (to bring up a GUI interface)
- ◆ Click “OK” to accept the changes
- ◆ Choose “Test Options” as “Percentage split 66%”
- ◆ Click “Start” to train the MLP



Workshop – Weka - MLP

- ◆ A BP network is auto-built according to the architecture setting
- ◆ Modify the network if necessary, such as adding/removing nodes and connections
- ◆ Click “Accept” to accept the network architecture
- ◆ Click “Start” to train the NN
- ◆ After training is done, click “Accept”, the NN model information and testing results is shown in “classifier output”
- ◆ Browse the results to find out model accuracy, confusion matrix, etc.
- ◆ Experiment with different parameters such as hidden layers, learning rate, epoch, etc.
- ◆ Compare the model performance
- ◆ Right click on the model item in the Result List and select “save model” to save the model



- **Installation Instructions:**
- **Download and install latest Anaconda for Python 3.7:**
<https://www.anaconda.com/download/>
- **Start Menu -> Anaconda Prompt:**
 - `conda create -n prmls python=3.7`
 - `conda activate prmls`
 - `conda install numpy matplotlib jupyter pandas scikit-learn keras pydot pydotplus`
 - `pip install neupy`
 - Navigate to your working directory, eg. “d:\myfolder”
 - Run “jupyter notebook”
- Now you can open .ipynb files in the jupyter notebook within your browser
- If you need an IDE to edit and run a python program, install and run spyder in Anaconda Prompt.

Problem Description: Diabetes Prediction

- This dataset is originally from the National Institute of Diabetes and Digestive and Kidney Diseases. The objective of the dataset is to diagnostically predict whether or not a patient has diabetes, based on certain diagnostic measurements included in the dataset.
- The datasets consists of several medical predictor variables and one target variable, Outcome. Predictor variables includes the number of pregnancies the patient has had, their BMI, insulin level, age, and so on.

Smith, J.W., Everhart, J.E., Dickson, W.C., Knowler, W.C., & Johannes, R.S. (1988). [Using the ADAP learning algorithm to forecast the onset of diabetes mellitus](#). In *Proceedings of the Symposium on Computer Applications and Medical Care* (pp. 261--265). IEEE Computer Society Press.

Workshop- Python- Scikit-Learn & Keras

- Open the iPython notebook provided for this workshop.
- As you go through the notebook, make sure you understand how the NN models are built. (you can save notes as markdown in the notebook).
- Check and compare the model performance.
- Experiment with different parameter settings.
- Save your notebook with all output as HTML file and upload it to LumiNUS.