

# Neural Networks I

Mark A. Austin

austin@umd.edu

*ENCE 688P, Spring Semester 2026  
University of Maryland*

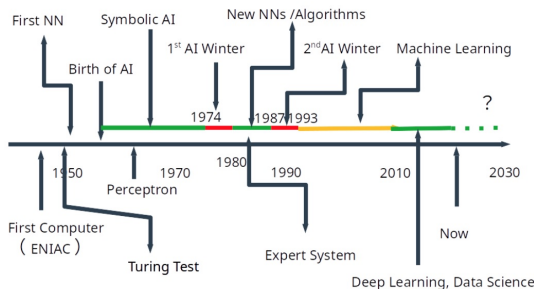
May 26, 2026

# Overview

- 1 Introduction to Neural Networks
- 2 Universal Approximation Theorem
- 3 The Perceptron (1943-1958)
- 4 Training a Single Perceptron Model
- 5 Single-Layer Perceptron Examples
  - Example 1: Using Python + NumPy
  - Example 2: Using Deeplearning4J

# Quick Review

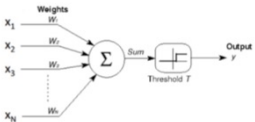
# A Brief History



- 1943: First neural networks invented (McCulloch and Pitts)
- 1958-1969: Perceptrons (Rosenblatt, Minsky and Papert).
- 1980s-1990s: CNN, Back Propagation.
- 1990s-2010s: SVMs, decision trees and random forests.
- 2010s: Deep Neural Networks and deep learning.

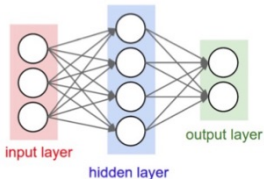
# Machine Learning Capabilities (1980-1990)

## Expressive Power of a Neural Network

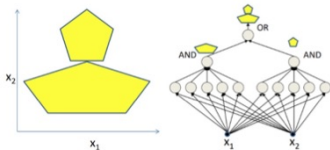
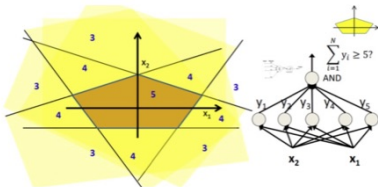


$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^d w_i x_i \geq T \\ 0 & \text{else} \end{cases}$$

## Neural Network with Single Hidden Layer



## Approximation of Functions / Boolean Logic



# Introduction to Neural Networks

# Purpose and Grand Vision (Aggarwal, 2018)

## Purpose

Neural networks are a development to **simulate** the **human nervous system** for machine learning tasks by **teaching** the **computational units** in the model similar to **human neurons**.

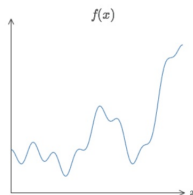
## Grand Vision

Create AI by **building machines** whose **architecture** simulates **computations** in the **human nervous system**.

# Why Neural Networks?

Reasons to use Neural Networks:

- Neural networks are **universal function approximators**, no matter how complex:



- Neural **network architectures** are **highly scalable** and **flexible**.

Caveat:

- Very large neural networks may be close to impossible to train and generalize correctly.

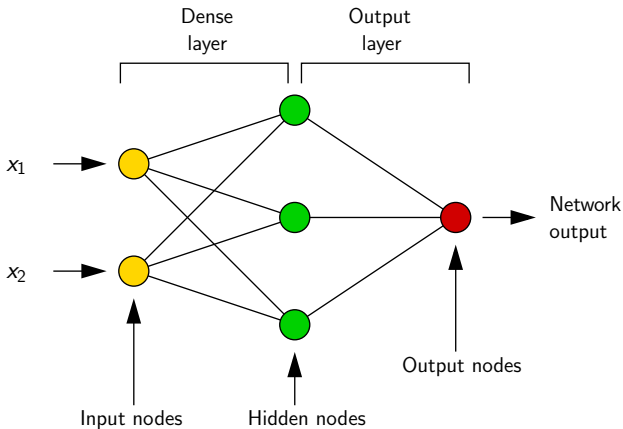
# Basic Neural Network Architecture

## Learning Procedure:

- Convert the **neural network training problem** into an **optimization problem**.
- Define a **loss function** and then optimize the parameter values to minimize its value.
- The loss function is the beginning of back propagation.

# Basic Neural Network Architecture

## Feed Forward Neural Network with One Hidden Layer:



# Basic Neural Network Architecture

## Network Layers:

- **Input Layer:** Receives input as a vector of numbers.
- **Hidden Layer(s):** Composed of weights  $w$  and biases  $b$ . Transforms the input using multiplications, addition and nonlinear (activation function) transformations.
- **Output Layer:** Outputs the final vector representing the model prediction. Two types of output: regression and classification.

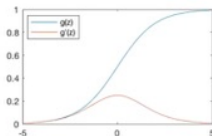
## Activation Functions:

- Nonlinear activation functions are applied to the result of the weight multiplication and bias addition.
- Allow the network to learn complex patterns in the data.

# Basic Neural Network Architecture


## Common Activation Functions

Sigmoid Function

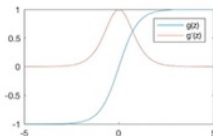


$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

 `tf.math.sigmoid(z)`

Hyperbolic Tangent



$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$


 `tf.math.tanh(z)`

Rectified Linear Unit (ReLU)



$$g(z) = \max(0, z)$$

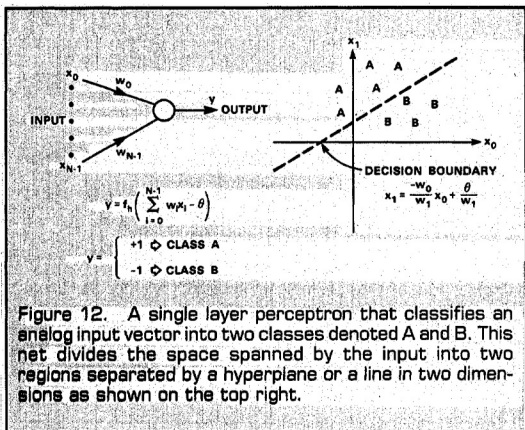
$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

 `tf.nn.relu(z)`



# Modeling Expectations

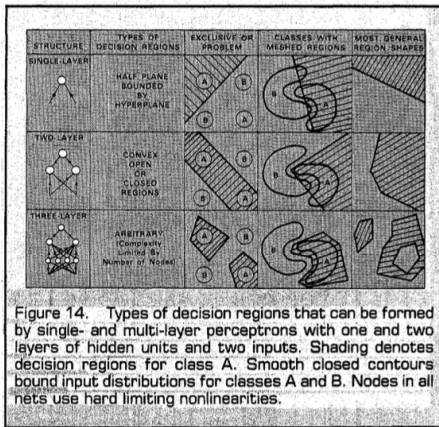
## Capabilities of a Perceptron Model:



Source: Lippman, IEEE ASSP Magazine, 1987.

# Modeling Expectations

## Neural Networks with Hidden Layers:



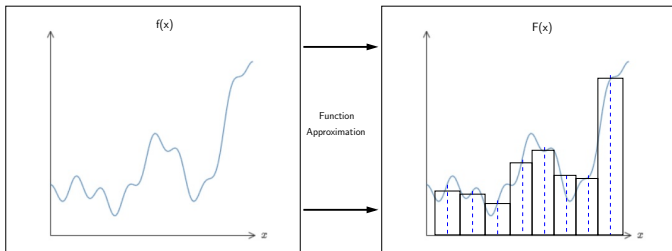
Source: Lippman, IEEE ASSP Magazine, 1987.

# Universal Approximation Theorem



# Universal Approximation Theorem

## Continuous Function to Discrete Approximation with Bumps



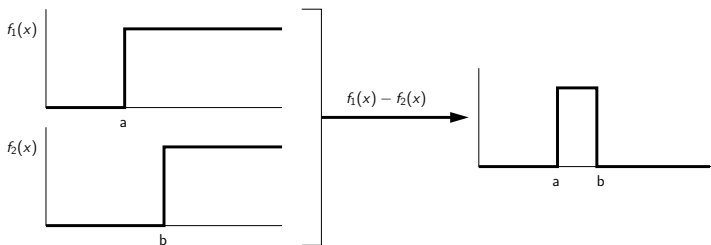
We wish to find:

$$F(x) = \sum_{i=1}^N v_i \sigma(w_i^T x + b_i) \quad (1)$$

such that  $|F(x) - f(x)| \leq \epsilon$ .

# Universal Approximation Theorem

**Strategy:** We can systematically assemble bump function (i.e., the weighting coefficients  $v_i$ ) on the interval  $[a,b]$  from pairs of steps, i.e.,

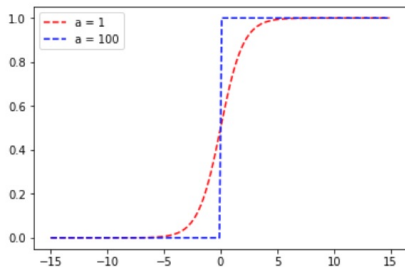


In practice, **threshold steps** are **replaced** by a **smooth approximation** to a step (e.g., sigmoid function).

# Function Approximation: Modeling Steps and Bumps

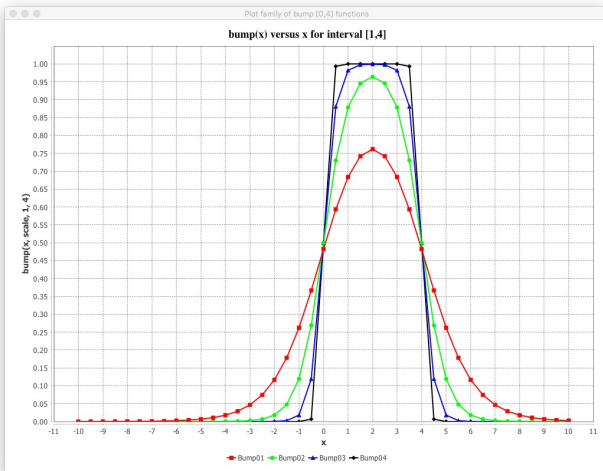
**Approximation:** Use Sigmoid Functions to Approximate Steps

Let  $\sigma(x) = \left[ \frac{1}{1+e^{-ax}} \right]$ .



The blue dashed curve is a good approximation of step function.

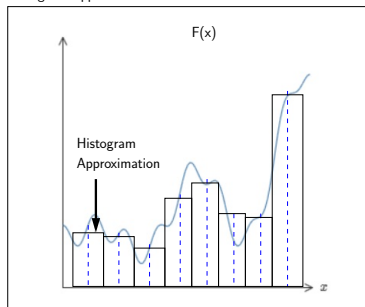
# Function Approximation: Modeling Steps and Bumps



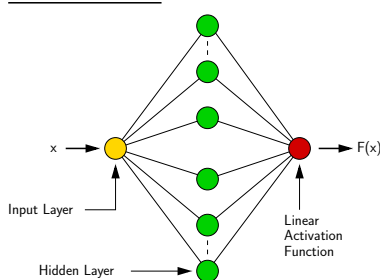
# Function Approximation: Neural Network Model

## Neural Network Model Approximation of Histogram

Histogram Approximation



Neural Network Model



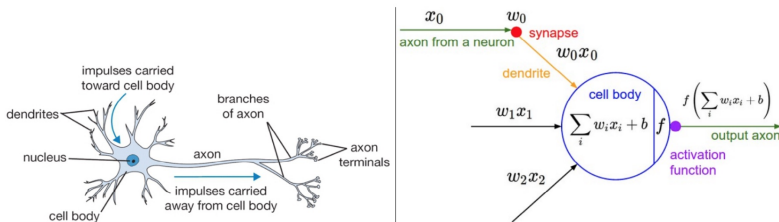
**Note:** Use **linear activation function** to get combined **sum** of **individual bump functions**.

# The Perceptron

Building Block of Machine Learning

# A Little History / Biological Inspiration

Neural networks originally began as **computational models** of the **brain** (i.e., models of cognition).

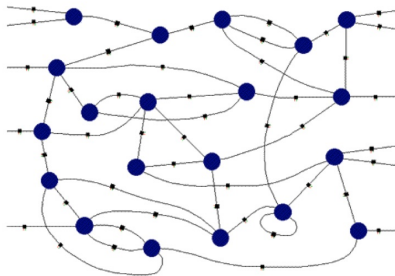


A cartoon drawing of a biological neuron (left) and its mathematical model (right).

- Early models were based on **association relationships**.
- More recent models of brain are **connectionist** – neurons **connect to neurons**.

# Connectionist Models

Present-day neural network models are connectionist machines.

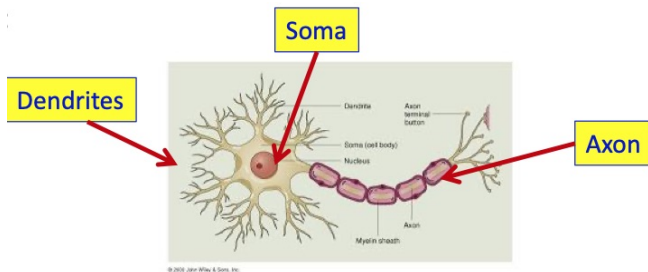


That is:

- Network of processing elements.
- Knowledge is stored in the connections between the elements.
- We need a model for these computational units.

# Mathematical Model of a Single Neuron

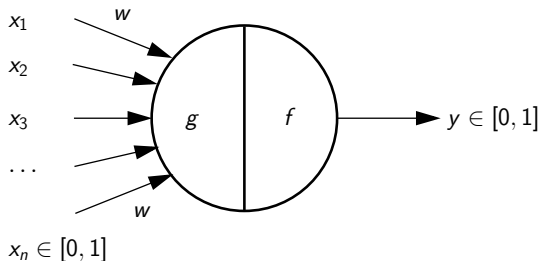
**Modelling the Brain.** Basic units are neurons:



- Signals come in through the dendrites into Soma.
- A signal exits via the axon to other neuron (only one axon per neuron).
- Neurons do not undergo cell division.

# Mathematical Model of a Single Neuron

McCulloch and Pitts Model for a Single Neuron (1943):



First artificial neural network:

- Assumes boolean input (i.e.,  $x \in [0, 1]$ ).
- A neuron fires when its activation is 1, otherwise its activation is 0 (i.e.,  $y \in [0, 1]$ ).

# Mathematical Model of a Single Neuron

## Mathematical Model:

- All **incoming connections** have the **same weight**.
- Function  $g()$  aggregates the inputs, i.e.,

$$g(x_1, x_2, \dots, x_n) = g(x) = \sum_{i=1}^n x_i \quad (2)$$

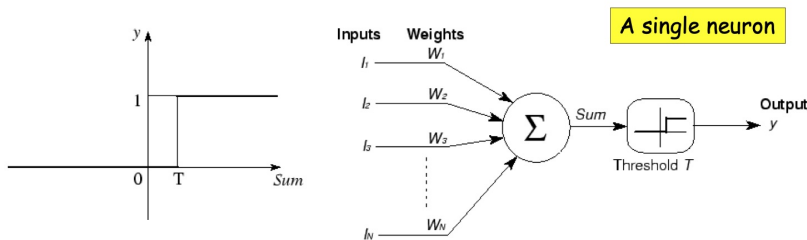
- Function  $f()$  takes a decision based on this aggregation.  $y = 0$  if any input  $x_i$  is inhibitory. Otherwise:

$$\begin{aligned} y = f(g(x)) &= 1 \text{ if } g(x) \geq \theta. \\ &= 0 \text{ if } g(x) < \theta. \end{aligned}$$

- $\theta$  is called the **threshold parameter**.

# Mathematical Model of a Single Neuron

Behavior of a Simple Neuron Unit:



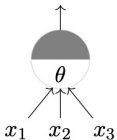
Criticisms:

- Claimed their machine could emulate a Turing machine.
- Did not provide a learning mechanism.

# Mathematical Model of a Single Neuron

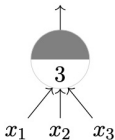
## Simplified Modeling of Boolean Gates:

$$y \in \{0, 1\}$$



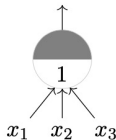
A McCulloch Pitts unit

$$y \in \{0, 1\}$$



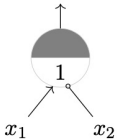
AND function

$$y \in \{0, 1\}$$



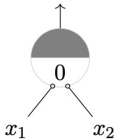
OR function

$$y \in \{0, 1\}$$



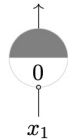
$x_1$  AND  $\neg x_2$ \*

$$y \in \{0, 1\}$$



NOR function

$$y \in \{0, 1\}$$

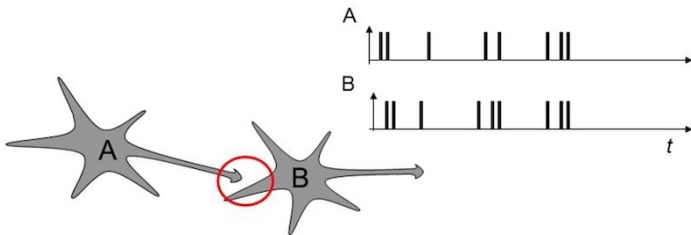


NOT function

# Mathematical Model of a Single Neuron

## Hebbian Learning (Donald Hebb, 1949)

When an axon of cell A excites cell B and repeatedly or persistently takes part in firing it, some growth processes or metabolic change takes place in one or both cells so that A's **efficiency** is **increased**.



**Observation:** In other words, neurons that fire together wire together!

# Mathematical Model of a Single Neuron

## Principles of Hebbian Learning

- Neurons that fire together wire together!
- If neuron  $x_i$  repeatedly triggers neuron  $y$ , the synaptic knob connecting  $x_i$  to  $y$  gets larger.
- Mathematically, we can write:

$$w_i = w_i + \eta x_i y \quad (3)$$

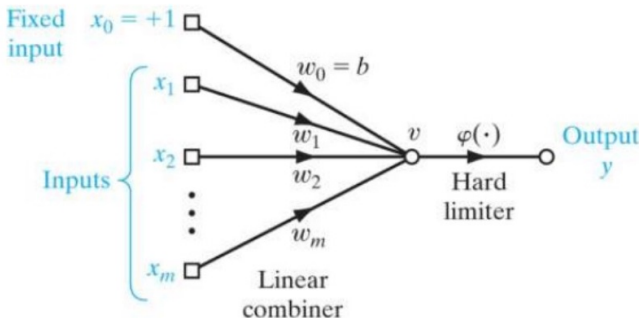
- Here,  $w_i$  is the weight of the  $i$ -th neuron's input to output neuron  $y$ .
- This [simple formula](#) is actually the [basis](#) of many [learning algorithms](#) in [machine learning](#).

# Mathematical Model of a Single Perceptron

## Perceptron Model (Rosenblatt, 1958)

The simplest form of a neural network consists of a single neuron with **adjustable** synaptic **weights** and **bias**.

A nonlinear neuron consists of a linear combiner followed by a hard limiter.



# Mathematical Model of a Single Perceptron

## Perceptron Model (Rosenblatt, 1958):

- Learning algorithm:

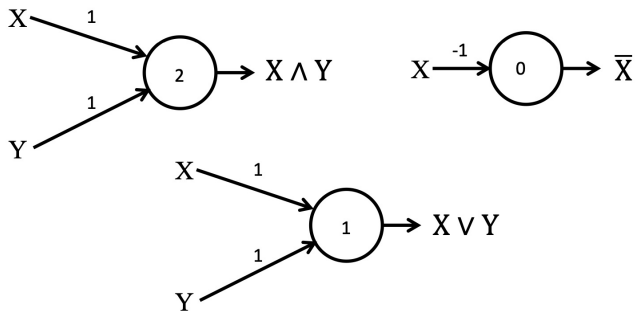
$$w(x) = w(x) + \eta (d(x) - y(x)) x. \quad (4)$$

Here:

- $\eta$  is the learning rate,
- $d(x)$  and  $y(x)$  are the desired and actual outputs in response to  $x$ .
- Update weights whenever the perceptron output is wrong.
- Proved convergence.
- Solution for OR and AND Boolean Gates.

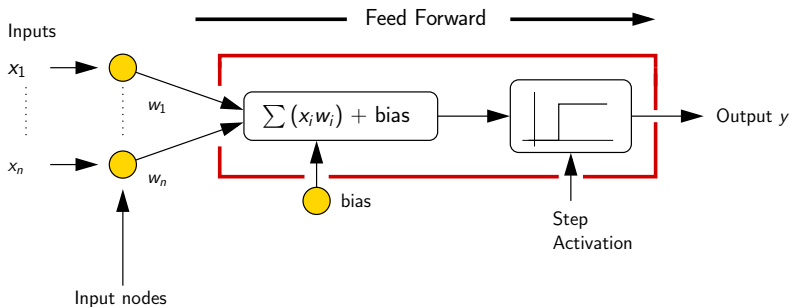
# Mathematical Model of a Single Perceptron

## Perceptron Model for OR and AND Boolean Gates



Individual elements are weak – **no solution** for **XOR problem**.  
Networked elements are required.

# The Perceptron Model: Forward Propagation



Here:

- Inputs  $x_1, x_2, x_3, \dots, x_n$  are real valued.
- Weights  $w_1, w_2, w_3, \dots, w_n$  are real valued.
- The output  $y$  can also be real valued.

# The Perceptron Model: Forward Propagation

**Step 1:** Linear combiner:

$$z = g(x) = \sum_{i=1}^n w_i x_i + \text{bias.} \quad (5)$$

**Step 2:** Step activation:

$$y = f(z) = \begin{cases} 0, & z < \theta, \\ 1, & z \geq \theta. \end{cases} \quad (6)$$

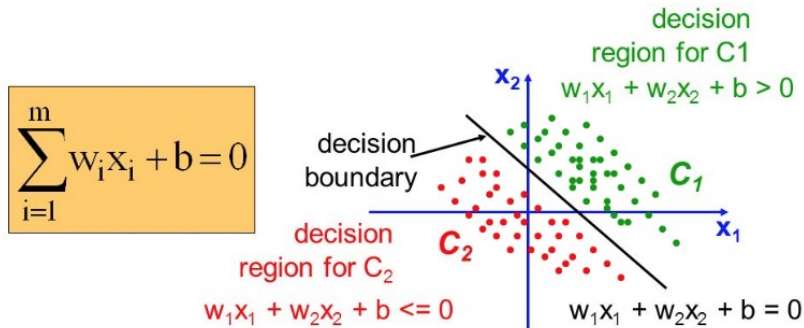
Here,  $\theta$  is the **threshold parameter**.

Composition of steps 1 and 2:

$$y = f(g(x)) \quad (7)$$

# Perceptron Model as a Linear Classifier

Perceptron operating on real-valued vectors is a linear classifier:

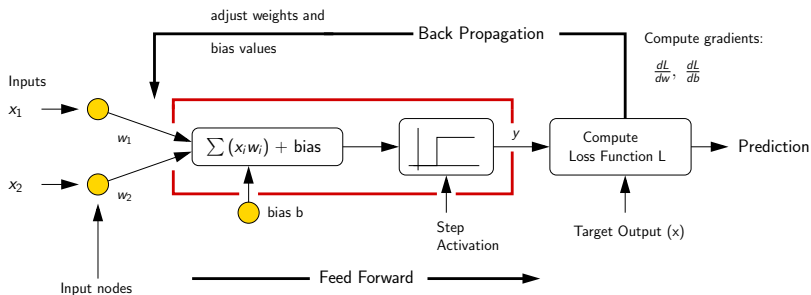


Addition of bias values expands modeling capability. No bias value  
→ decision boundary constrained to pass through the origin.



# Training a Single Perceptron Model

**Network Architecture:** Perceptron model with two input streams, weights and a bias, and step activation.



**Input Data:**  $(x_{11}, x_{21}), (x_{12}, x_{22}), \dots (x_{1n}, x_{2n})$ .

**Target Data:** target  $(x_1), \dots$  target  $(x_n)$ .

# Training Perceptron Models

## Training Objective

Find weight and bias ( $w_1, w_2, b$ ) values to minimize difference between predictions and target values.

## Sample Loss Function

Define a quadratic loss function  $L$ ,

$$L(y, target(x)) = \frac{1}{2} \sum_{i=1}^n (y_i - target(x_i))^2 \quad (8)$$

Here,  $y_i$  is the network prediction for input  $x_i$  and  $target(x_i)$  is the target value for learning.

# Training Perceptron Models

**Numerical Strategy:** Use **gradient descent algorithm** to compute sequence of weight approximations, i.e.,

$$w_{n+1} = w_n - \eta \nabla L. \quad (9)$$

Here,  $w$  = matrix of network weights and  $\eta$  = learning rate.

**Chain Rule:** Network predictions  $y$  are a **composition** of the **linear combiner** + **activation function**.

Mathematically,  $L$  is related to  $x$ ,  $w$  and  $b$  as follows:

$$L = L(f(g(x, w, b))) \rightarrow \frac{dL}{dw} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial w}. \quad (10)$$

# Training Perceptron Models

## Two problems with Step Activation:

- Can change weights without affecting  $L$ ,
- Function is not continuously differentiable.

Hence, replace step activation with sigmoid function:

$$y = f(z) = \begin{cases} 0, & z < 0, \\ 1, & z \geq 0. \end{cases} \quad \rightarrow \quad y = \sigma(z) = \left[ \frac{1}{1 + e^{-z}} \right]. \quad (11)$$

Derivative of sigmoid is easy:

$$\frac{dy}{dz} = \frac{d}{dz} \sigma(z) = \sigma(z) [1 - \sigma(z)]. \quad (12)$$

# Back Propagation in Feed Forward Models

**Minimize L.** Here,  $L = L(y)$ , where  $y = \sigma(z)$  and  $z = g(x, w, b)$ .

Use chain rule to find derivative of  $L$  with respect to  $w$ :

$$\frac{dL}{dw} = \frac{\partial L}{\partial y} \cdot \frac{dy}{dz} \cdot \frac{\partial z}{\partial w}. \quad (13)$$

First term,

$$\frac{dL}{dy} = \frac{\partial}{\partial y} \left[ \sum_{i=1}^n (y_i - \text{target}(x_i))^2 \right] = \sum_{i=1}^n (y_i - \text{target}(x_i)). \quad (14)$$

# Back Propagation in Feed Forward Models

Second term,

$$\frac{dy}{dz} = \frac{d}{dz} \sigma(z) = \sigma(z) [1 - \sigma(z)]. \quad (15)$$

Third term,

$$\frac{\partial z}{\partial w} = \frac{\partial}{\partial w} \left[ \sum_{i=1}^n w_i \cdot x_i + b \right]. \quad (16)$$

Collecting terms,

$$\nabla L = \left[ \frac{\partial L}{\partial w_1} \quad \frac{\partial L}{\partial w_2} \quad \frac{\partial L}{\partial b} \right]^T. \quad (17)$$

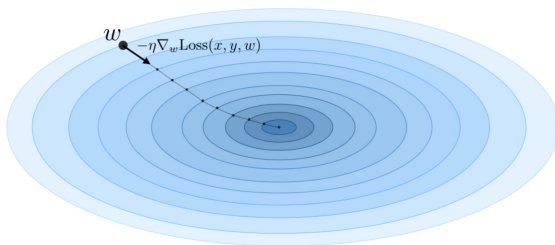
**Update Weights:** Plug equation 17 into equation 9. Repeat.

# Iterations of Learning in Feed Forward Models

## Stochastic Gradient Descent

Strategies of neural network learning are iterative.

$$w_{n+1} = w_n - \eta \nabla_w \text{Loss}(x, y, w). \quad (18)$$



They employ training datasets to update the model.

# Single-Layer Perceptron Examples

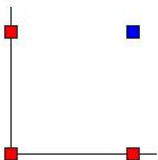
# Modeling Boolean Gates

## Problem Description:



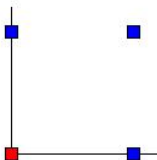
AND

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



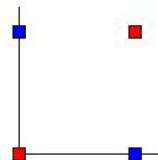
OR

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



XOR

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



# Example 1: Modeling an OR Boolean Gate

## Python + NumPy Code: Step-by-step solution (pg 1).

```
1 # =====
2 # TestNeural-BooleanORGate.py: Perceptron model for boolean OR gate:
3 #
4 # Reference: Shukla, et al., Neural Networks from Scratch with
5 # Python Code and Math in Detail, Towards AI, 2020
6 #
7 # Modified by: Mark Austin                                October, 2020
8 # =====
9
10 import math
11 import matplotlib
12 import matplotlib.pyplot as plt
13 import numpy as np
14
15 # Define Sigmoid function:
16
17 def sigmoid(x):
18     return 1/(1+np.exp(-x))
19
20 # Define derivative of Sigmoid function:
21
22 def sigmoid_der(x):
23     return sigmoid(x)*(1-sigmoid(x))
24
25 # main method ...
```

# Example 1: Modeling an OR Boolean Gate

## Python + NumPy Code: Step-by-step solution (pg 2) ...

```
27 def main():
28     print("--- Enter TestNeuralNetwork01.main()      ... ");
29     print("--- ===== ... ");
30
31     input_features = np.array( [[0,0],[0,1],[1,0],[1,1]] )
32
33     print (input_features.shape)
34     print (input_features)
35
36     # Define target output:
37
38     target_output = np.array([[0,1,1,1]])
39
40     # Reshaping target output into vector:
41
42     target_output = target_output.reshape(4,1)
43     print (target_output)
44
45     weights = np.array([[1.0],[2.0]])
46     print(weights.shape)
47     print (weights)
48
49     bias = 0.3          # Bias weight:
50     lr   = 0.05         # Learning Rate:
```

# Example 1: Modeling an OR Boolean Gate

## Python + NumPy Code: Step-by-step solution (pg 3) ...

```
52     # Main loop for training network ...
53
54     for epoch in range(20000):
55
56         # feedforward input, feedforward output, back propogation ...
57
58         inputs = input_features
59         in_o   = np.dot(inputs, weights) + bias
60         out_o  = sigmoid(in_o)
61
62         # Calculate error in computed output ...
63
64         error = out_o - target_output
65
66         # Calculate derivative:
67
68         derror_douto = error
69         douto_dino   = sigmoid_der(out_o)
70
71         # Multiplying individual derivatives:
72
73         deriv = derror_douto * douto_dino
74
75         # Finding the transpose of input_features:
76
77         inputs = input_features.T
78         deriv_final = np.dot(inputs, deriv)
```

# Example 1: Modeling an OR Boolean Gate

## Python + NumPy Code: Step-by-step solution (pg 4) ...

```
80         # Update the weights values:
81
82         weights -= lr * deriv_final
83         for i in deriv:
84             bias -= lr * i #Check the final values for weight and biasprint (weights)
85
86     # Print summary of results ...
87
88     print("--- Weights:");
89
90     print (weights)
91
92     print("--- Bias: %f ... \n" %(bias))
93
94     print("--- Use trained network to predict values ... ");
95
96     print("--- Verify input [1,0] --> 1 ... ");
97
98     single_point = np.array([1,0]) #1st step:
99     result1 = np.dot(single_point, weights) + bias #2nd step:
100    result2 = sigmoid(result1) #Print final result
101
102    print("--- Result 1: %f ..." %(result1))
103    print("--- Result 2: %f ..." %(result2))
104
105    print("--- Verify input [0,1] --> 1 ... ");
```

# Example 1: Modeling an OR Boolean Gate

## Python + NumPy Code: Step-by-step solution (pg 5) ...

```
107     single_point = np.array([0,1]) #1st step:
108     result1 = np.dot(single_point, weights) + bias #2nd step:
109     result2 = sigmoid(result1) #Print final result
110
111     print("--- Result 1: %f ..." %(result1))
112     print("--- Result 2: %f ..." %(result2))
113
114     print("--- Verify input [1,1] --> 1 ... ");
115
116     single_point = np.array([1,1]) #1st step:
117     result1 = np.dot(single_point, weights) + bias #2nd step:
118     result2 = sigmoid(result1) #Print final result
119
120     print("--- Single input point [1,1] ...")
121     print("--- Result 1: %f ..." %(result1))
122     print("--- Result 2: %f ..." %(result2))
123
124     print("--- Verify input [0,0] --> 0 ... ");
125
126     single_point = np.array([0,0]) #1st step:
127     result1 = np.dot(single_point, weights) + bias #2nd step:
128     result2 = sigmoid(result1) #Print final result
129
130     print("--- Single input point [0,0] ...")
131     print("--- Result 1: %f ..." %(result1))
132     print("--- Result 2: %f ..." %(result2))
```

# Example 1: Modeling an OR Boolean Gate

## Python + NumPy Code: Step-by-step solution (pg 6) ...

```
133
134     print("--- ===== ... ");
135     print("--- Finished TestNeuralNetwork01.main() ... ");
136
137     # call the main method ...
138
139     main()
```

## Python + NumPy Code: Abbreviated Results ...

```
--- Summary of weights and biases ...
```

```
--- Weights:
```

```
  [ [8.46406006]
    [8.46563981] ]
```

```
--- Bias: -3.886041 ...
```

# Example 1: Modeling an OR Boolean Gate

## Python + NumPy Code: Abbreviated Results ...

```
--- Use trained network to predict values ...
```

```
--- Verify input [1,0] --> 1 ...
```

```
--- Result 1: 4.578019, result 2: 0.989829 ...
```

```
--- Verify input [0,1] --> 1 ...
```

```
--- Result 1: 4.579599, result 2: 0.989845 ...
```

```
---
```

```
--- Verify input [1,1] --> 1 ...
```

```
--- Result 1: 13.043659, result 2: 0.999998 ...
```

```
---
```

```
--- Verify input [0,0] --> 0 ...
```

```
--- Result 1: -3.886041, result 2: 0.020114 ...
```

## Example 2. Modeling an OR Boolean Gate

### DL4J: Create training dataset:

```
1 // Create matrix of input values ...
2
3 double[][] matrixDouble = new double[][]{ {0.0, 0.0}, {1.0, 0.0},
4                                             {0.0, 1.0}, {1.0, 1.0}};
5 INDArray input01 = Nd4j.create(matrixDouble);
6
7 // Create vector of expected output values ....
8
9 double[] vectorDouble = new double[]{0,1,1,1};
10 INDArray output01 = Nd4j.create(vectorDouble).transpose();
11
12 DataSet ds = new DataSet(input01, output01 );
```

### DL4J: Dataset values:

Matrix of input values

```
[ [ 0, 0],
  [ 1.0000, 0],
  [ 0, 1.0000],
  [ 1.0000, 1.0000] ]
```

Vector of output values

```
[ 0,
  1.0000,
  1.0000,
  1.0000 ]
```

## Example 2. Modeling an OR Boolean Gate

### DL4J: Create Network Configuration:

```
23 // Create neural network configuration builder ...
24
25 MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
26     .updater(new Sgd(0.1))
27     .seed(seed)
28     .biasInit(0)
29     .miniBatch(false)
30     .list()
31     .layer(new OutputLayer.Builder( LossFunctions.LossFunction.MSE )
32         .nIn(2)
33         .nOut(1)
34         .activation(Activation.SIGMOID)
35         .weightInit(new UniformDistribution(0, 1))
36         .build())
37     .build();
38
39 // Create multilayer network ...
40
41 MultiLayerNetwork net = new MultiLayerNetwork(conf);
42 net.init();
43 net.setListeners(new ScoreIterationListener(1000));
```

## Example 2. Modeling an OR Boolean Gate

**DL4J:** Summary of Network Model (4 nodes on hidden layer)

LayerName (LayerType)	nIn,nOut	TotalParams	ParamsShape
layer0 (OutputLayer)	2,1	3	W:{2,1}, b:{1,1}
-----			
Total Parameters:	3	Trainable Parameters:	3
=====			

**DL4J:** Train the network for 10,000 epochs:

```
52     for( int i=0; i <= 10000; i++ ) {  
53         net.fit(ds);  
54     }
```

## Example 2. Modeling an OR Boolean Gate

**DL4J:** Trained weights and bias:

```
--- Layer: layer0 ...
```

```
-----
```

```
--- Weights: [ 6.9568, 6.9568 ] ...
```

```
--- Bias:      -3.2378 ...
```

Decision boundary:

$$f(x_1, x_2) = 6.9568(x_1 + x_2) - 3.2378 = 0.0. \quad (19)$$

**DL4J:** Trained model predictions:

```
[ 0.0378,  
  0.9763,  
  0.9763,  
  1.0000 ]
```

## Example 2. Modeling an OR Boolean Gate

### DL4J: Evaluation Metrics:

```
# of classes:      2
Accuracy:          1.0000
Precision:         1.0000
Recall:            1.0000
F1 Score:          1.0000
```

Precision, recall & F1: reported for positive class (class 1 - "1") of

### DL4J: Confusion Matrix:

```
0 1
----
1 0 | 0 = 0
0 3 | 1 = 1
```

## References

- Aggarwal C.C., Neural Networks and Deep Learning, Springer, 2018.
- Bhiksha R., Introduction to Neural Networks, Lisbon Machine Learning School, June, 2018.
- Lippmann R.P., An Introduction to Computing with Neural Nets, IEEE ASSP Magazine, April 1987.

# Appendix A

Chart of Neural Network Architectures

# Neural Network Architectures

-  Backfed Input Cell
-  Input Cell
-  Noisy Input Cell
-  Hidden Cell
-  Probabilistic Hidden Cell
-  Spiking Hidden Cell
-  Output Cell
-  Match Input Output Cell
-  Recurrent Cell
-  Memory Cell
-  Different Memory Cell
-  Kernel
-  Convolution or Pool

A mostly complete chart of

## Neural Networks

©2016 Fjodor van Veen - asimovinstitute.org

Perceptron (P)



Feed Forward (FF)



Radial Basis Network (RBF)



Deep Feed Forward (DFF)



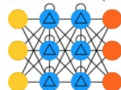
Recurrent Neural Network (RNN)



Long / Short Term Memory (LSTM)



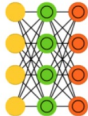
Gated Recurrent Unit (GRU)



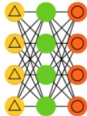
Auto Encoder (AE)



Variational AE (VAE)



Denoising AE (DAE)



Sparse AE (SAE)



# Neural Network Architectures

Markov Chain (MC)



Hopfield Network (HN)



Boltzmann Machine (BM)



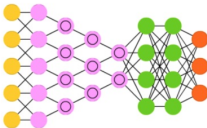
Restricted BM (RBM)



Deep Belief Network (DBN)



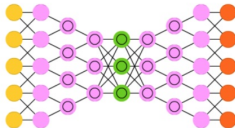
Deep Convolutional Network (DCN)



Deconvolutional Network (DN)



Deep Convolutional Inverse Graphics Network (DCIGN)



Generative Adversarial Network (GAN)



Liquid State Machine (LSM)



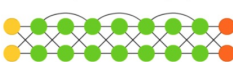
Extreme Learning Machine (ELM)



Echo State Network (ESN)



Deep Residual Network (DRN)



Kohonen Network (KN)



Support Vector Machine (SVM)



Neural Turing Machine (NTM)

