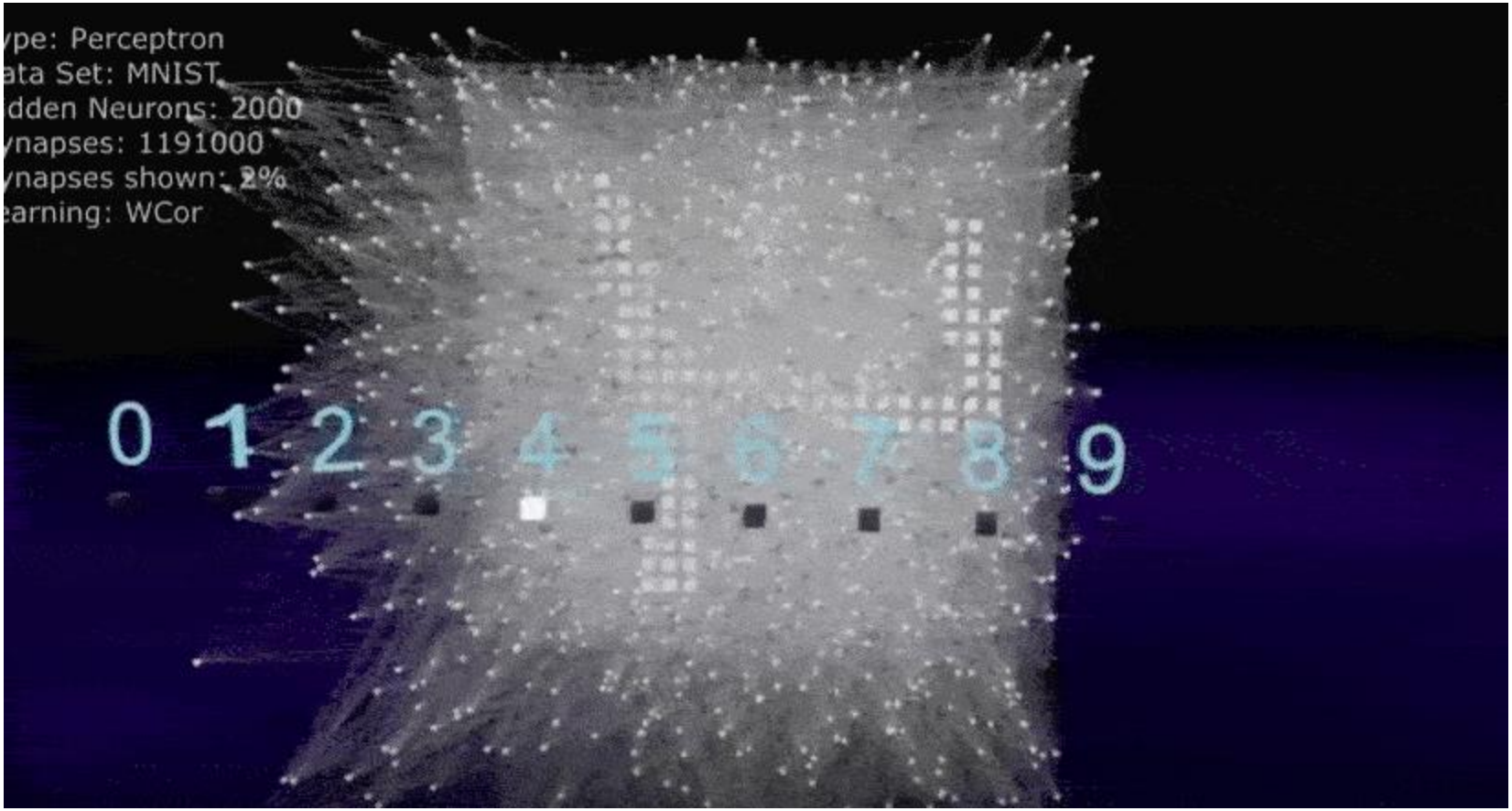
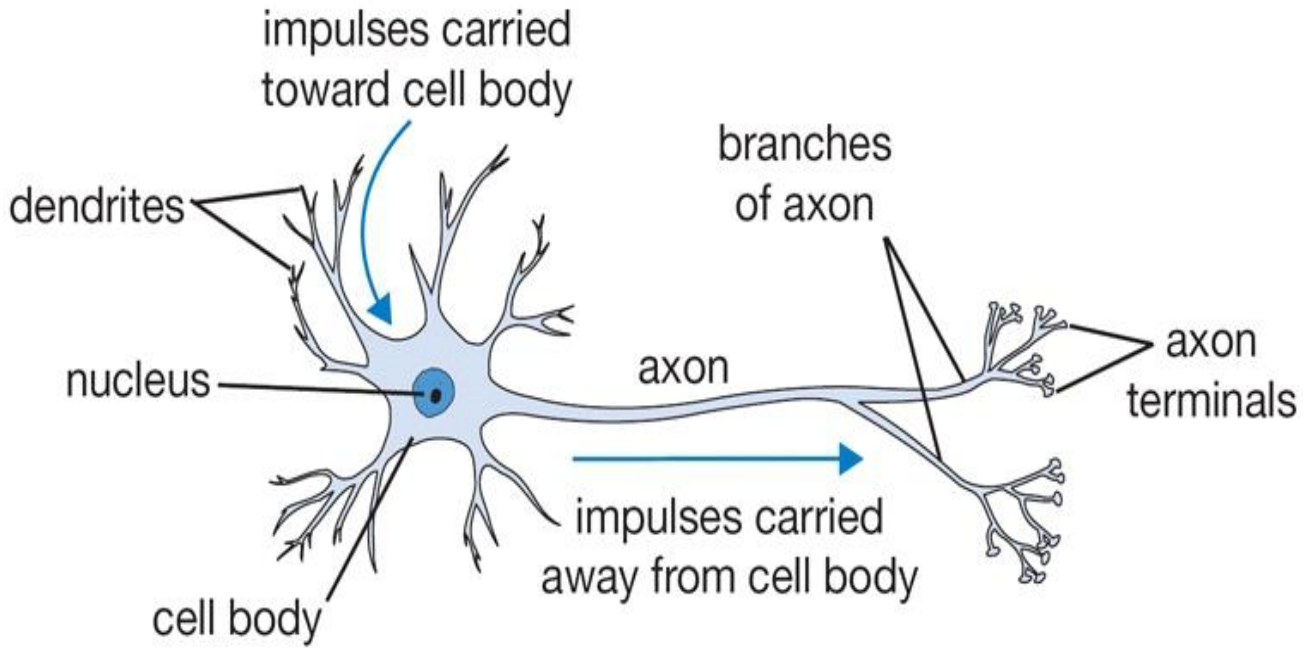

ype: Perceptron
ata Set: MNIST
dden Neurons: 2000
ynapses: 1191000
ynapses shown: 2%
earning: WCor
0 1 2 3 4 5 6 7 8 9

Artificial Neural Networks (ANNs)

An **Artificial Neural Network** (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information.

The neuron, is the **fundamental functional unit** of all nervous system tissues, including the brain. Each neuron consists of a cell body (soma), nucleus, dendrites, axon and synapses.



Note:

I. Not all synaptic links are of the same strength.

II. The strength of a signal passing through a synapse is proportional to the strength of the synaptic link.

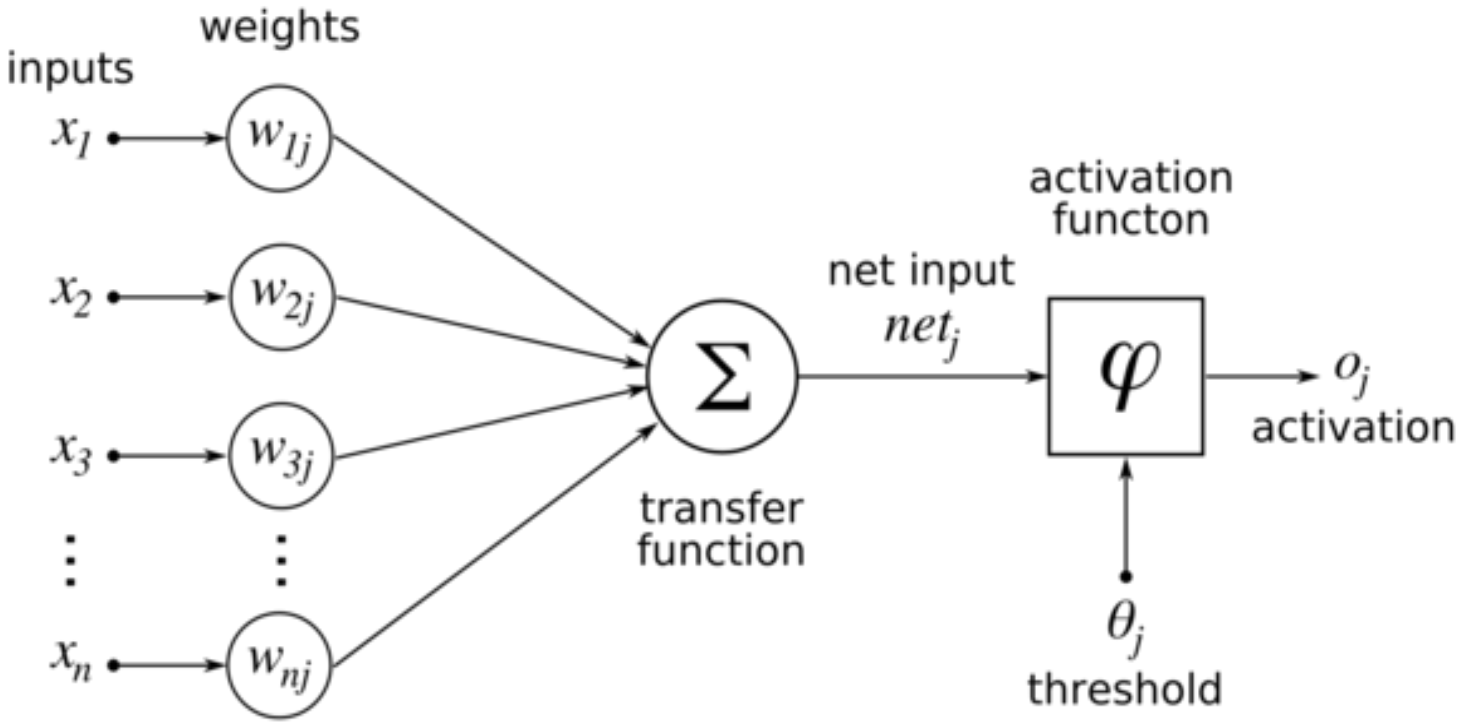
Human brain v/s Computer

	Brain	Computer
Number of Processing Units	$\approx 10^{11}$	$\approx 10^{9}$
Type of Processing Units	Neurons	Transistors
Form of Calculation	Massively Parallel	Generally Serial
Data Storage	Associative	Address-based
Response Time	$\approx 10^{-3}$s	$\approx 10^{-9}$s
Processing Speed	Very Variable	Fixed
Potential Processing Speed	$\approx 10^{13}$ FLOPS [14]	$\approx 10^{18}$ FLOPS
Real Processing Speed	$\approx 10^{12}$ FLOPS	$\approx 10^{10}$ FLOPS
Resilience	Very High	Almost None
Power Consumption per Day	20W	300W [15]

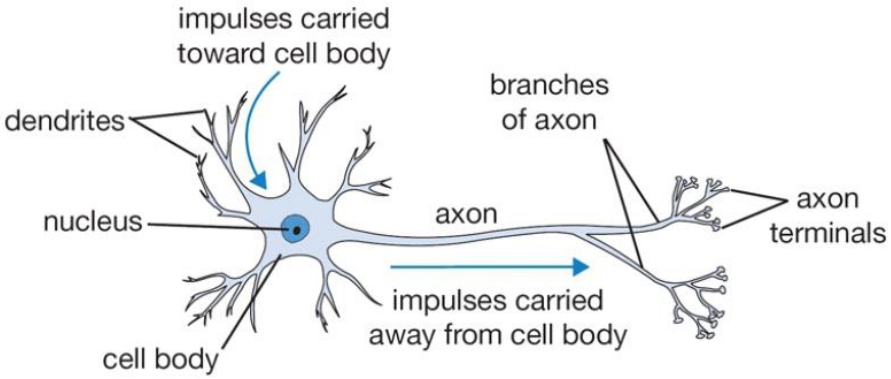
Characteristics and **capabilities** of artificial neural networks:

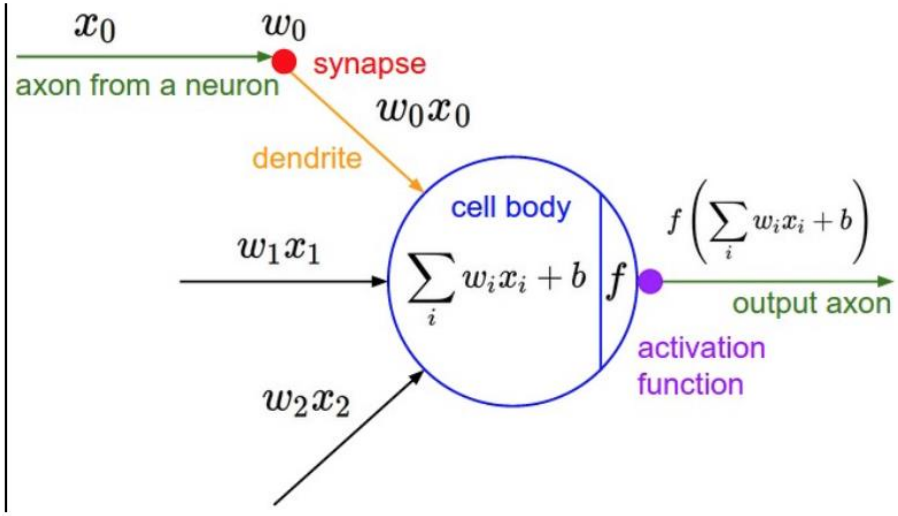
I. Non-linearity.
II. Input-Output mapping.
III. Adaptability.
IV. Evidential response.
V. Fault tolerance.
VI. VLSI implementability.
VII. Neurobiological analogy.

Electrical equivalent of the biological neuron:

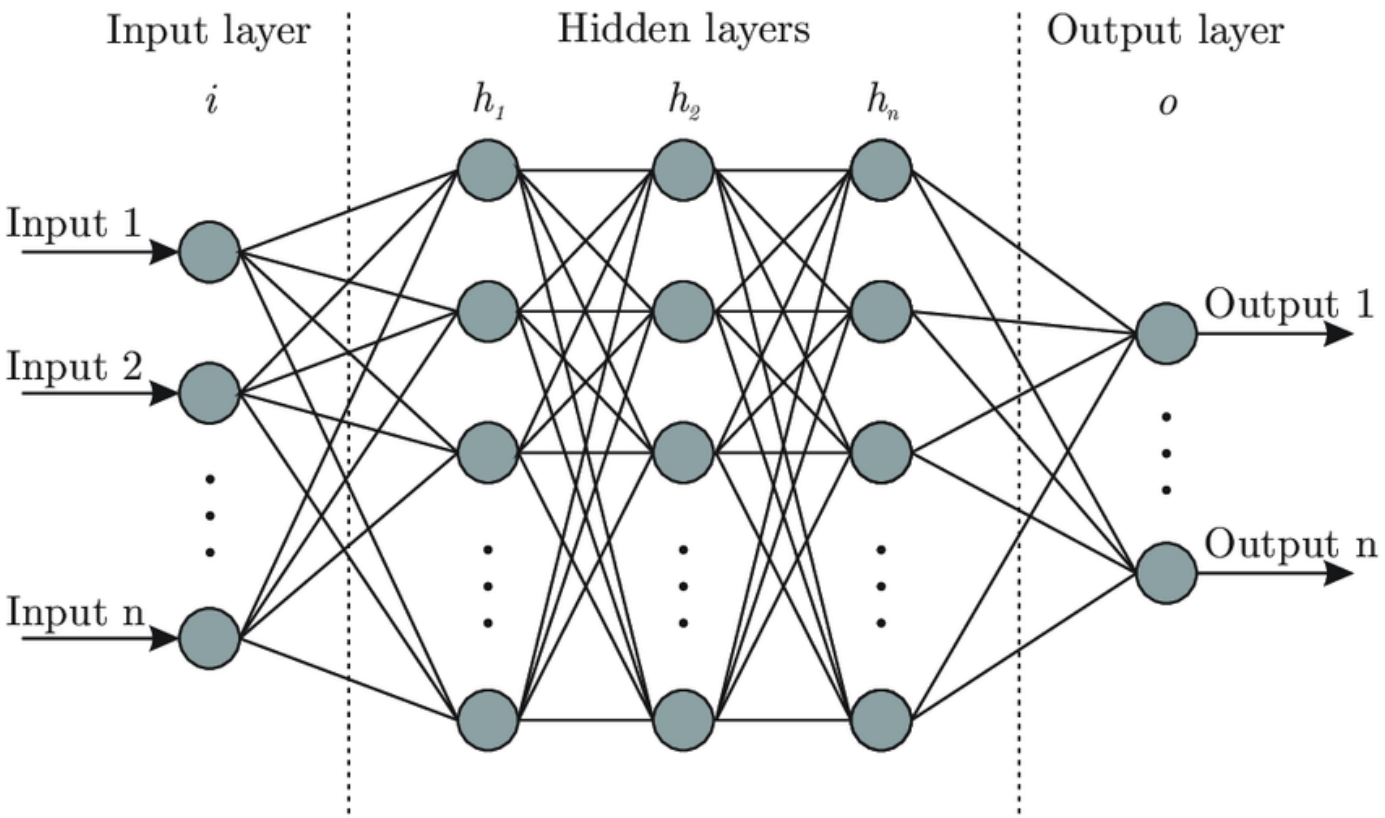


BNN v/s ANN:


impulses carried
toward cell body
dendrites
branches
of axon
nucleus
axon
axon
terminals
impulses carried
away from cell body
cell body


x_0
w_0
synapse
axon from a neuron
$w_0 x_0$
dendrite
cell body
$f\left(\sum_i w_i x_i + b\right)$
$w_1 x_1$
$\sum_i w_i x_i + b$
f
output axon
activation
function
$w_2 x_2$

An Artificial Neural Network:



Layers in artificial neural networks:

I. The input layer.

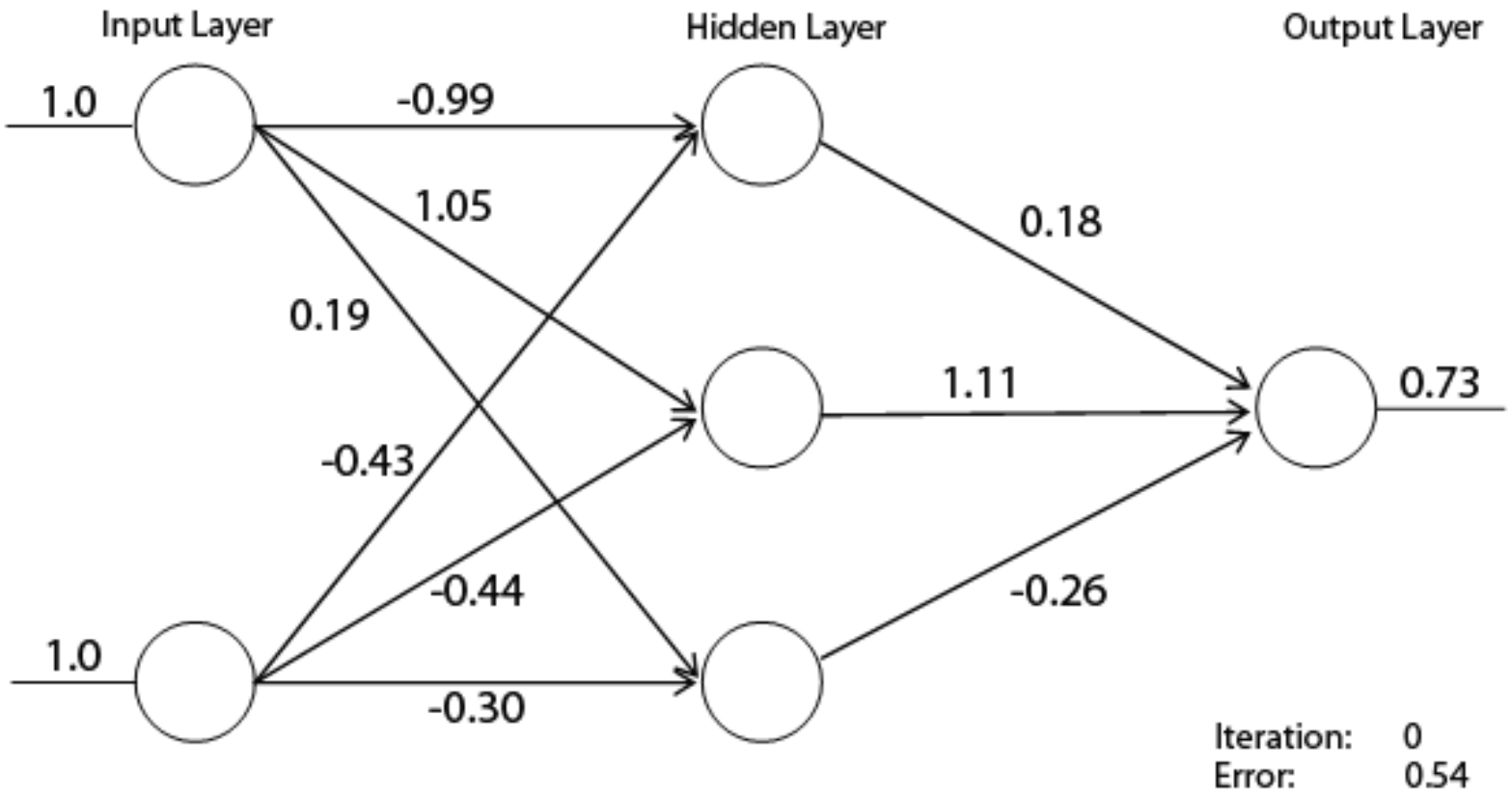
I. Introduces input values into the network.
II. No activation function or other processing.

II. The hidden layer(s).

I. Perform classification of features.
II. Two hidden layers are sufficient to solve any problem.
III. More layers may be better.

III. The output layer.

I. Functionally is similar to that of the hidden layers.
II. Outputs are passed on to the world outside the neural network.


Input Layer
Hidden Layer
Output Layer
1.0
1.0
-0.99
1.05
0.19
-0.43
-0.44
-0.30
0.18
1.11
-0.26
0.73
Iteration: 0
Error: 0.54

Activation Functions:

The activation function of a node defines the output of that node or neuron, given an input or a set of inputs.

Specifically in an ANN we compute the sum of products of inputs(**X**) and their corresponding Weights(**W**), and apply an Activation function **f(x)** to it, to get the output of that layer and feed it as an input to the next layer.

The need for Activation Functions:

If we do not apply an activation function then the output signal would simply be a simple linear function. A linear function is just a polynomial of one degree.

$$Y = f(X) = X1.W1 + X2.W2 + \ldots + Xn.Wn$$

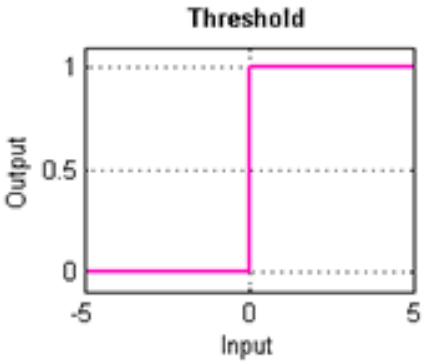
Now, a linear equation is easy to solve but they are limited in their complexity and have less power to learn complex functional mappings from data.

We want our neural network to not just learn and compute a linear function, but something more complicated than that. Also without the activation function, our neural network will not be able to learn and model other complicated kinds of data such as images, videos, audios, etc.

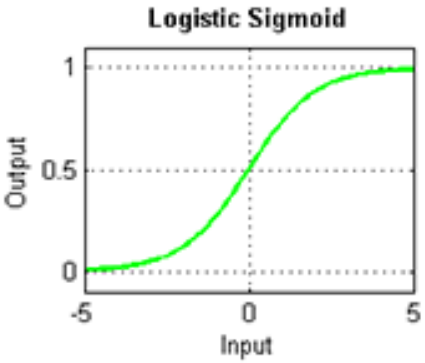
Y = f(X) = X1.W1 + X2.W2 + … + Xn.Wn
(Output of a linear summer.)

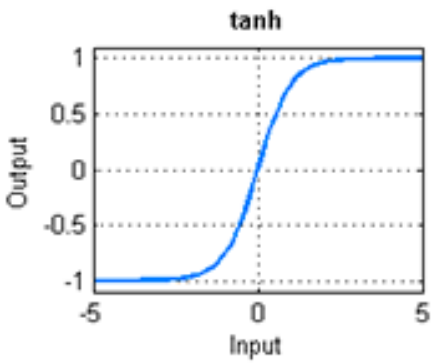
A(Y) = Actual Output.
(Activation function applied over the output of the linear summer.)

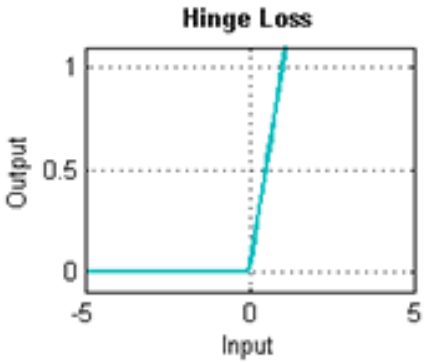
Types of Activation Functions:



Undefined gradient
Cant be used







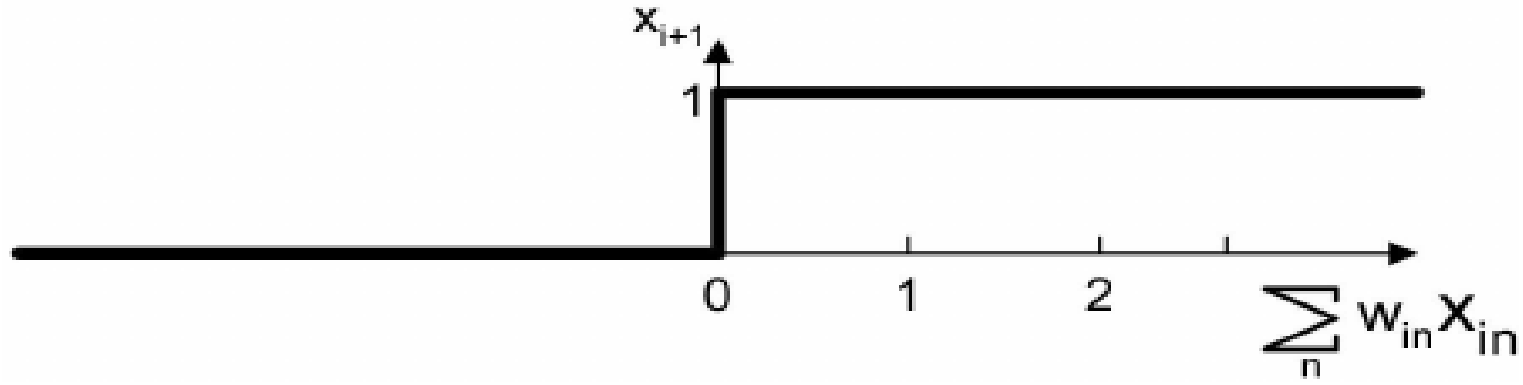
New default activation

Choosing the right activation function:

Depending upon the properties of the problem under consideration, we can make a better choice of an activation function, for easy and quicker convergence of the network.

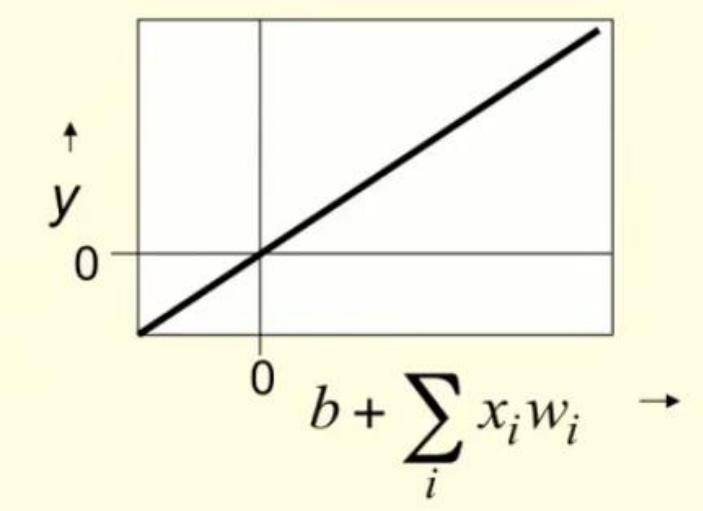
Artificial Neuron Models and Linear Regression:

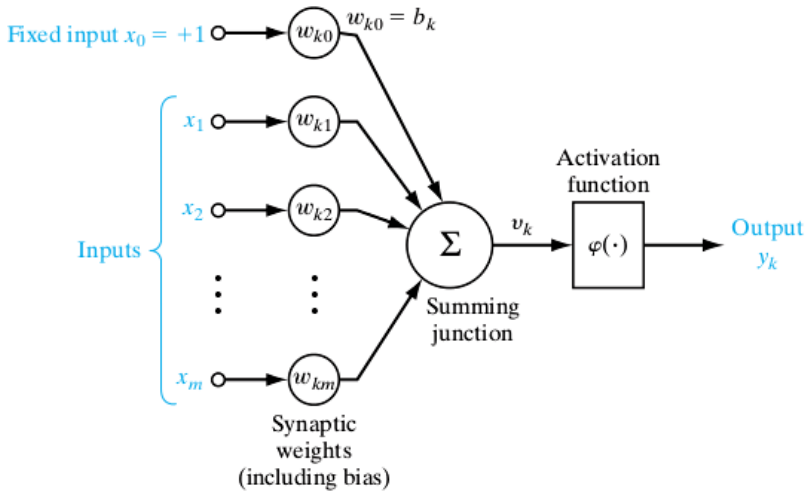
The McCulloch and Pitt's model or the threshold function :



The early model of an artificial neuron is introduced by Warren McCulloch and Walter Pitts in 1943. The McCulloch-Pitts neural model is also known as linear threshold gate. The output is hard limited and the response of this model is absolutely binary in nature.

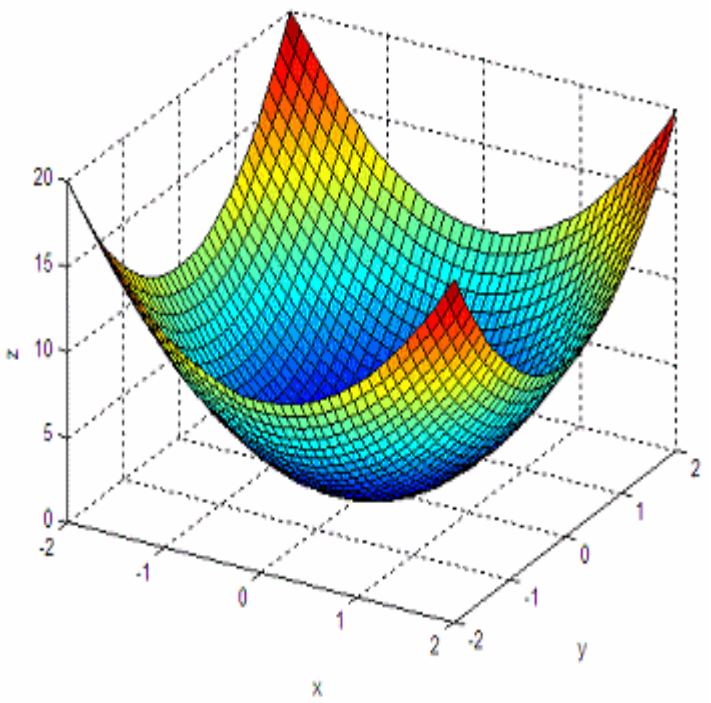
The linear neuron model:

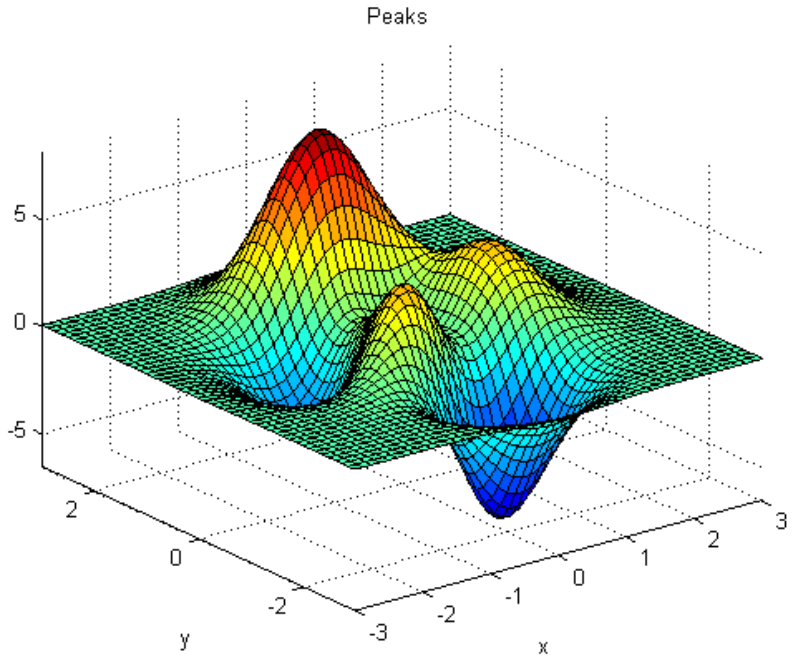




To obtain the output i.e. V, as it is, for a particular range and domain, the linear neuron model was created. In this model, the bias neuron isn't considered separately but as one of the inputs. Here, we hard limit the function to use it in practical applications since signal can't be processed outside a particular range.

Error functions:





The Gradient Descent Algorithm:

Gradient descent is a first-order iterative optimization algorithm. To find a local minimum of a function using gradient descent, one takes steps proportional to the *negative* of the gradient (or of the approximate gradient) of the function at the current point. If instead one takes steps proportional to the *positive* of the gradient, one approaches a local maximum of that function; the procedure is then known as **gradient ascent**.

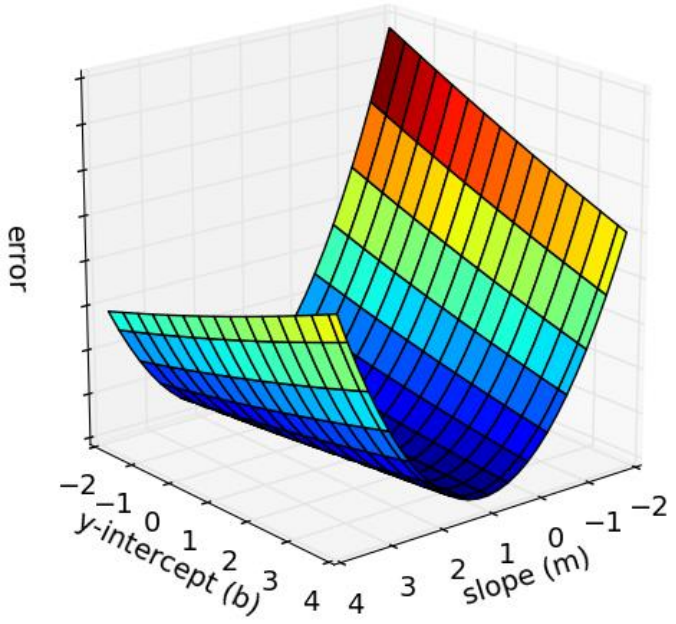
Gradient descent is also known as **steepest descent**, or the **method of steepest descent**. Gradient descent should not be confused with the method of steepest descent for approximating integrals.

At a theoretical level, gradient descent is an algorithm that minimizes functions. Given a function defined by a set of parameters, gradient descent starts with an initial set of parameter values and iteratively moves toward a set of parameter values that minimize the function. This iterative minimization is achieved using calculus, taking steps in the negative direction of the function gradient.

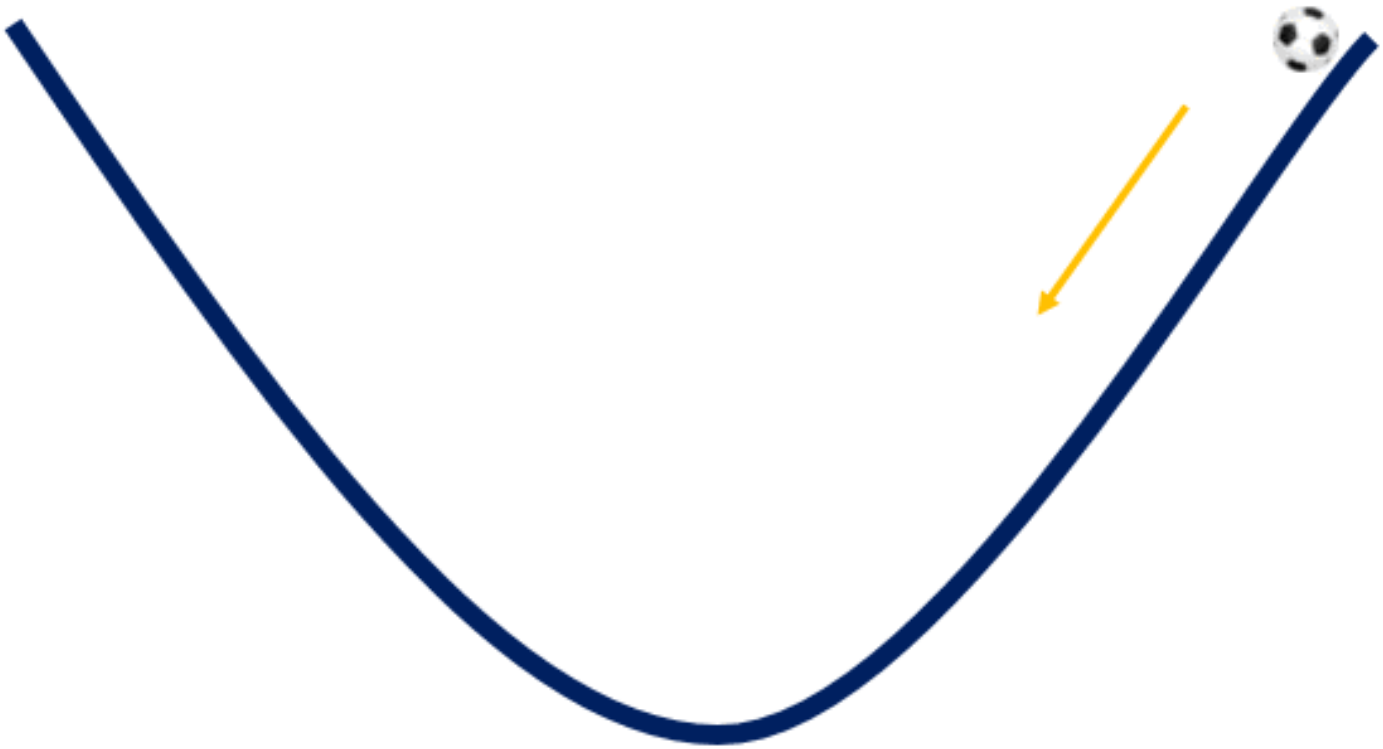
$$\text{Error}_{(\text{m,b})} = \frac{1}{N}\sum_{i=1}^{N}(y_i - (mx_i + b))^2$$

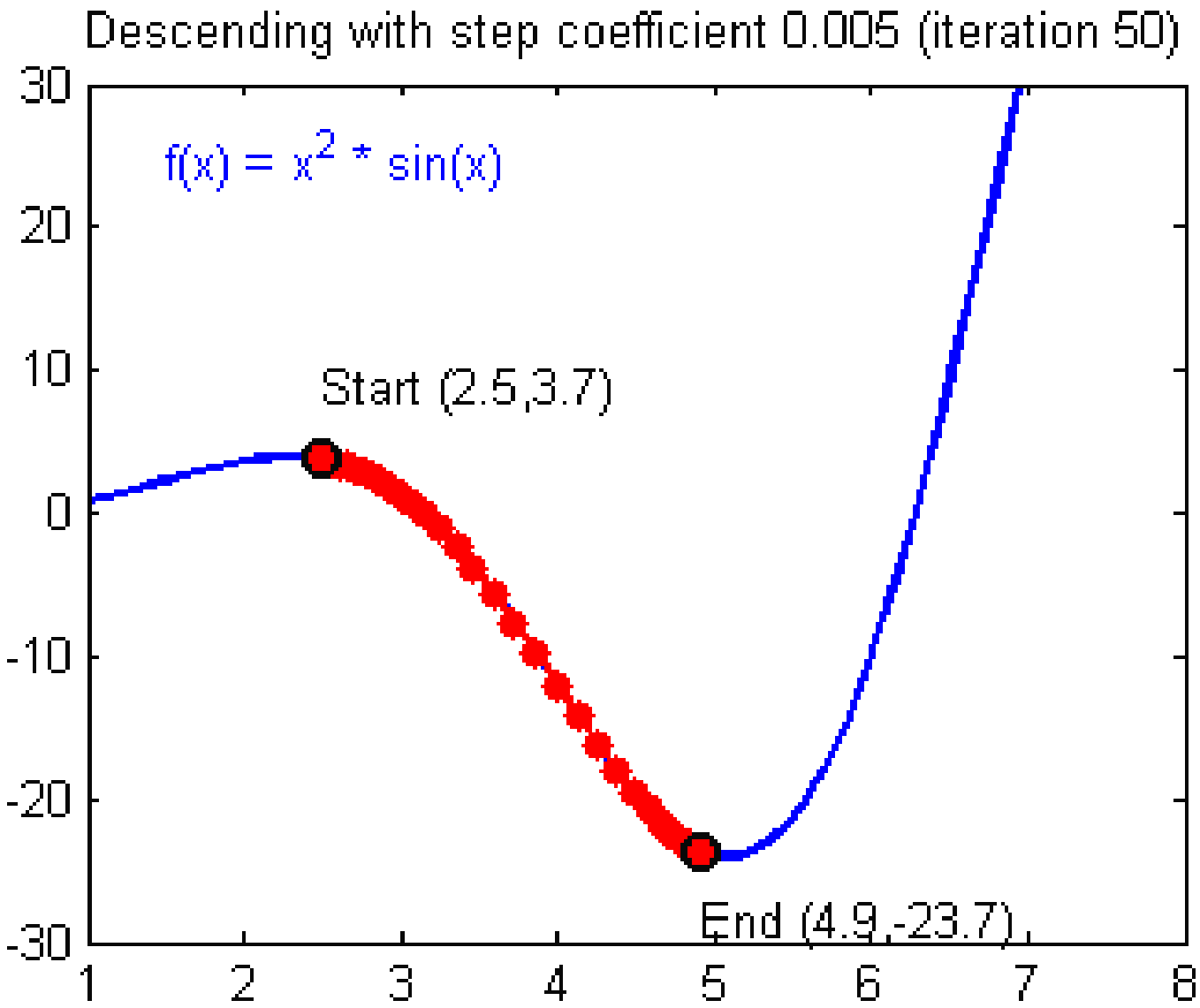
$$\frac{\partial}{\partial \text{m}} = \frac{2}{N}\sum_{i=1}^{N} - x_i(y_i - (mx_i + b))$$

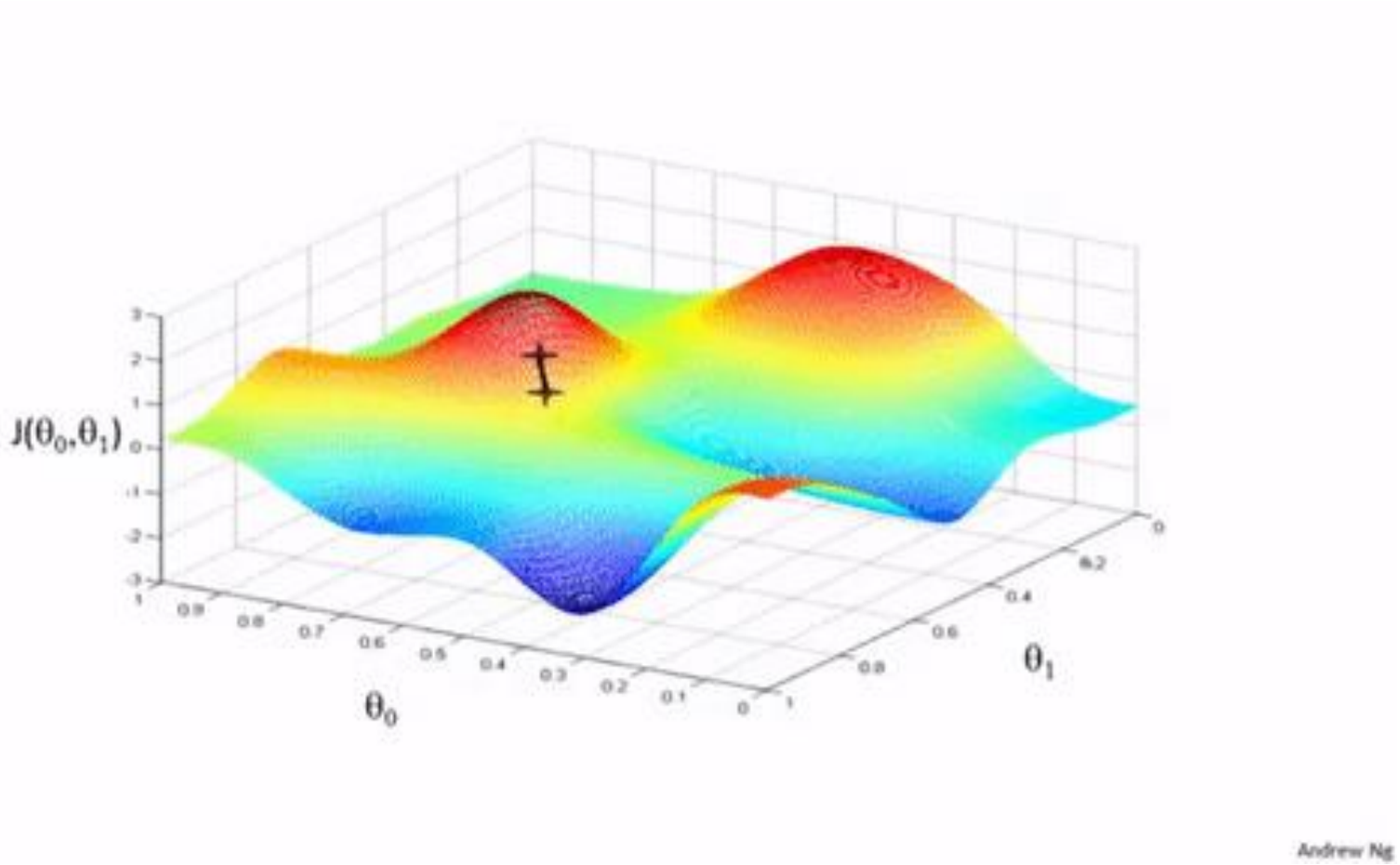
$$\frac{\partial}{\partial \text{b}} = \frac{2}{N}\sum_{i=1}^{N} - (y_i - (mx_i + b))$$

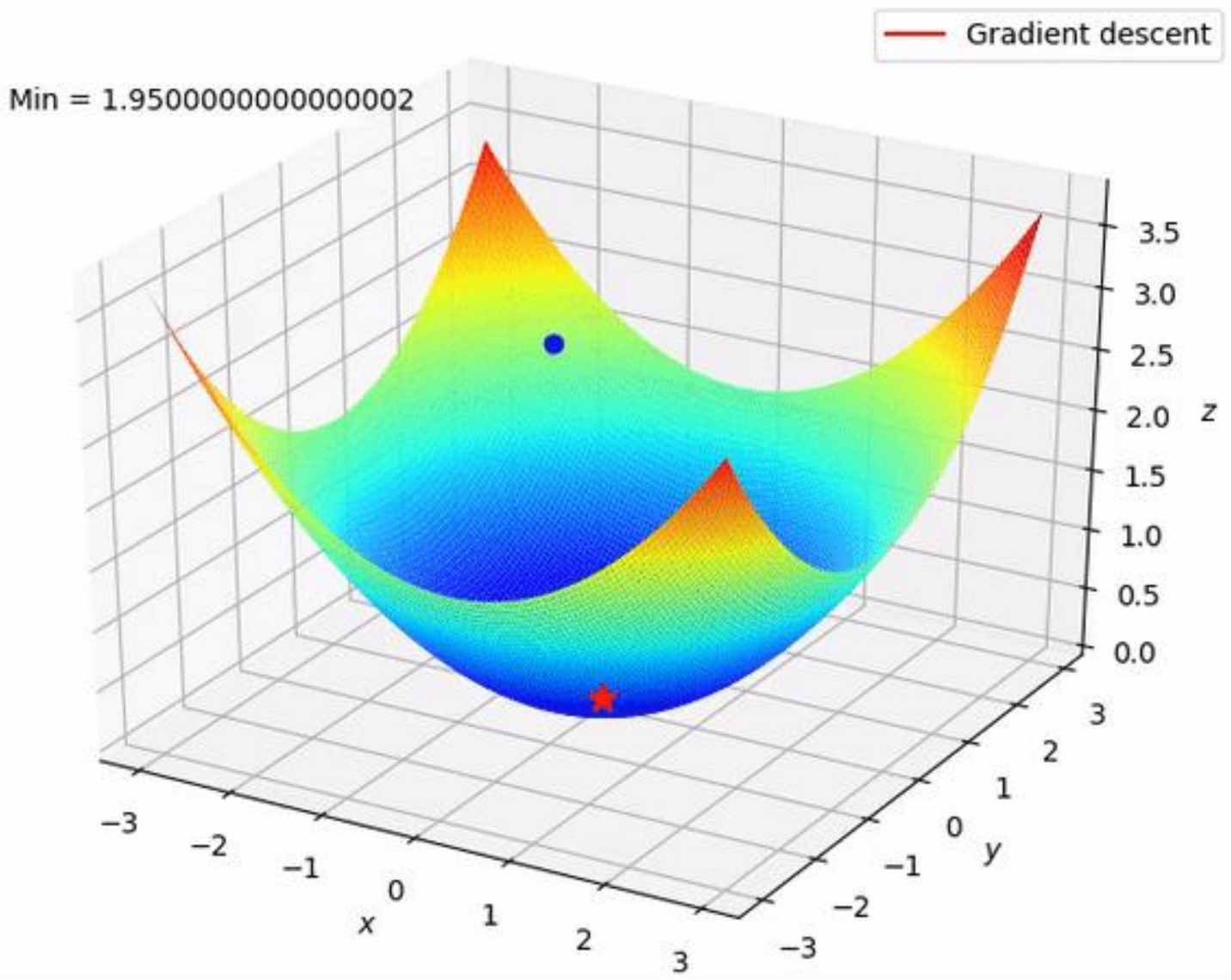


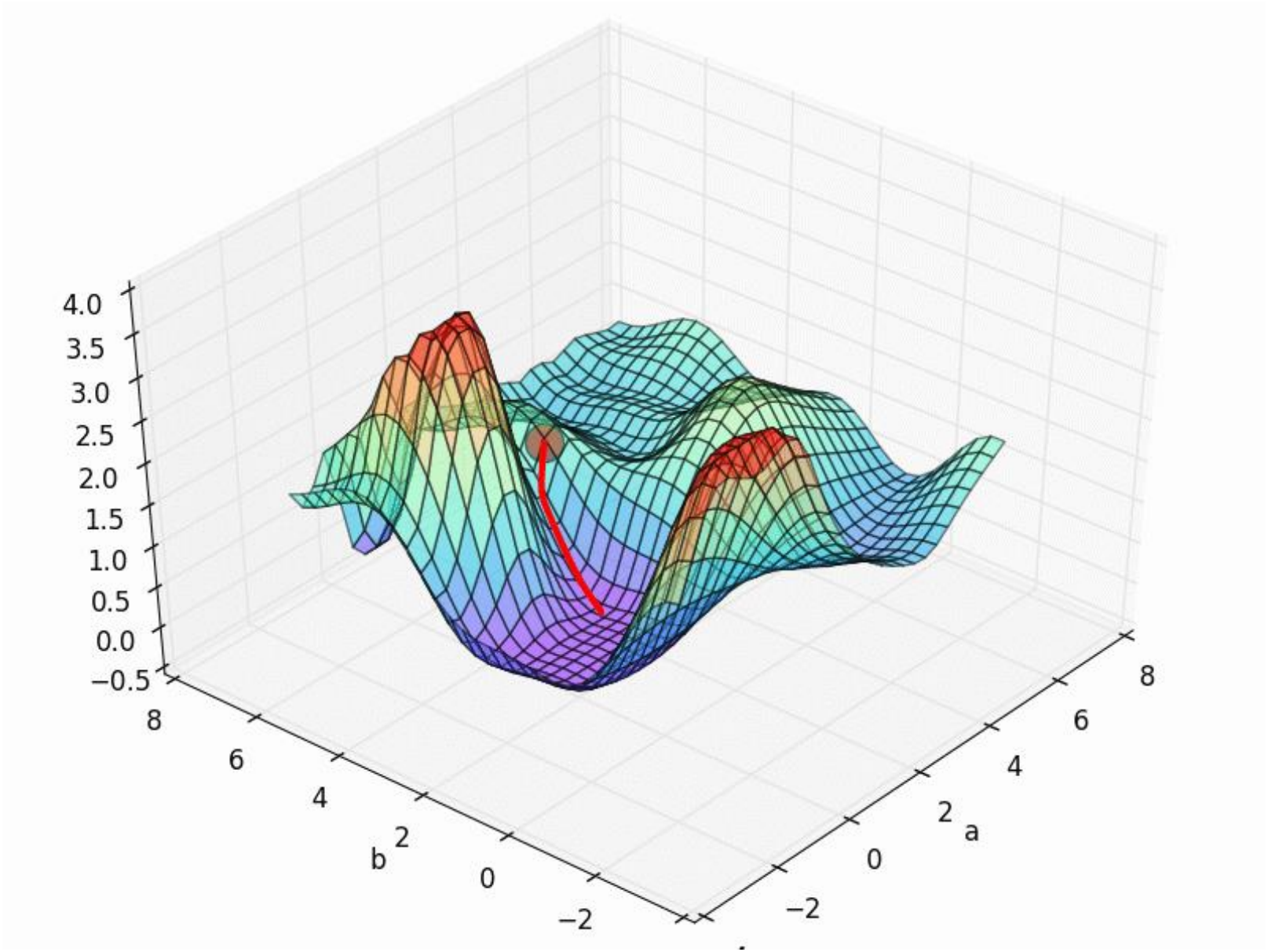
Gravity, $g = 9.8\ m/s^2$




Descending with step coefficient 0.005 (iteration 50)
f(x) = x^2 * sin(x)
Start (2.5,3.7)
End (4.9,-23.7)
30
20
10
0
-10
-20
-30
1
2
3
4
5
6
7
8


$J(\theta_0,\theta_1)$
θ_0
θ_1
Andrew Ng


Gradient descent
Min = 1.9500000000000002
3.5
3.0
2.5
2.0
z
1.5
1.0
0.5
0.0
−3
−2
−1
0
1
2
3
x
−3
−2
−1
0
1
2
3
y


4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
8
6
4
b
2
0
−2
8
6
4
2
a
0
−2

Learning rate of a Neural Network:

In order for Gradient Descent to work we must set the λ (learning rate) to an appropriate value. This parameter determines how fast or slow we will move towards the optimal weights. If the λ is very large we will skip the optimal solution. If it is too small we will need too many iterations to converge to the best values. So using a good λ is crucial.

Another good technique is to adapt the value of λ in each iteration. The idea behind this is that the farther you are from optimal values the faster you should move towards the solution and thus the value of λ should be larger. The closer you get to the solution the smaller its value should be.

Unfortunately since you don't know the actual optimal values, you also don't know how close you are to them in each step. To resolve this you can use a technique called **Bold Driver**.

Learning in Artificial Neural Networks

Learning process is a method or a mathematical logic which improves the artificial neural network's performance and usually this rule is applied repeatedly over the network.

It is done by updating the weights and bias levels of a network when a network is simulated in a specific data environment.

A learning rule may accept existing condition (weights and bias) of the network and will compare the expected result and actual result of the network to give new and improved values for weights and bias. Depending upon the process to develop the network there are three main models of machine learning:

I. **Unsupervised Learning**
II. **Supervised Learning**
III. **Reinforcement Learning**

The Perceptron:

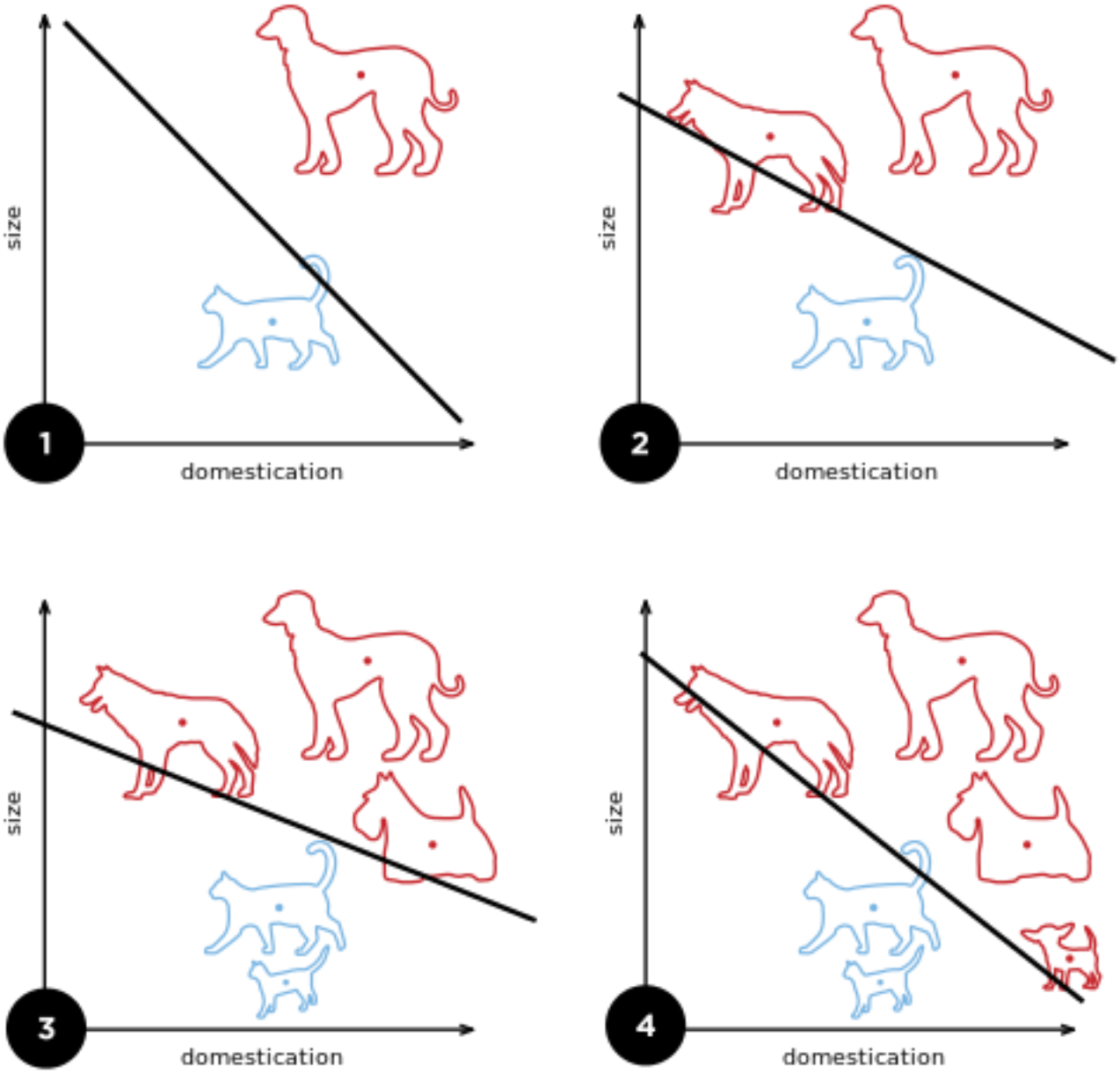
Layered feed-forward networks were first studied in the late 1950s under the name perceptrons.

Although networks of all sizes and topologies were considered, the only effective learning element at the time was for single-layered networks, so that is where most of the effort was spent. Today, the name perceptron is used as a synonym for a single-layer, feed-forward network.

In machine learning, the **perceptron** is an algorithm for supervised learning of binary classifiers (functions that can decide whether an input, represented by a vector of numbers, belongs to some specific class or not).

It is a type of **linear classifier**, i.e. a classification algorithm that makes its predictions based on a linear predictor function combining a set of weights with the feature vector.

A diagram showing a perceptron updating its linear boundary as more training examples are added.



Perceptron Convergence:

The Perceptron convergence theorem states that for any data set which is linearly separable the Perceptron learning rule is guaranteed to find a solution in a finite number of steps.

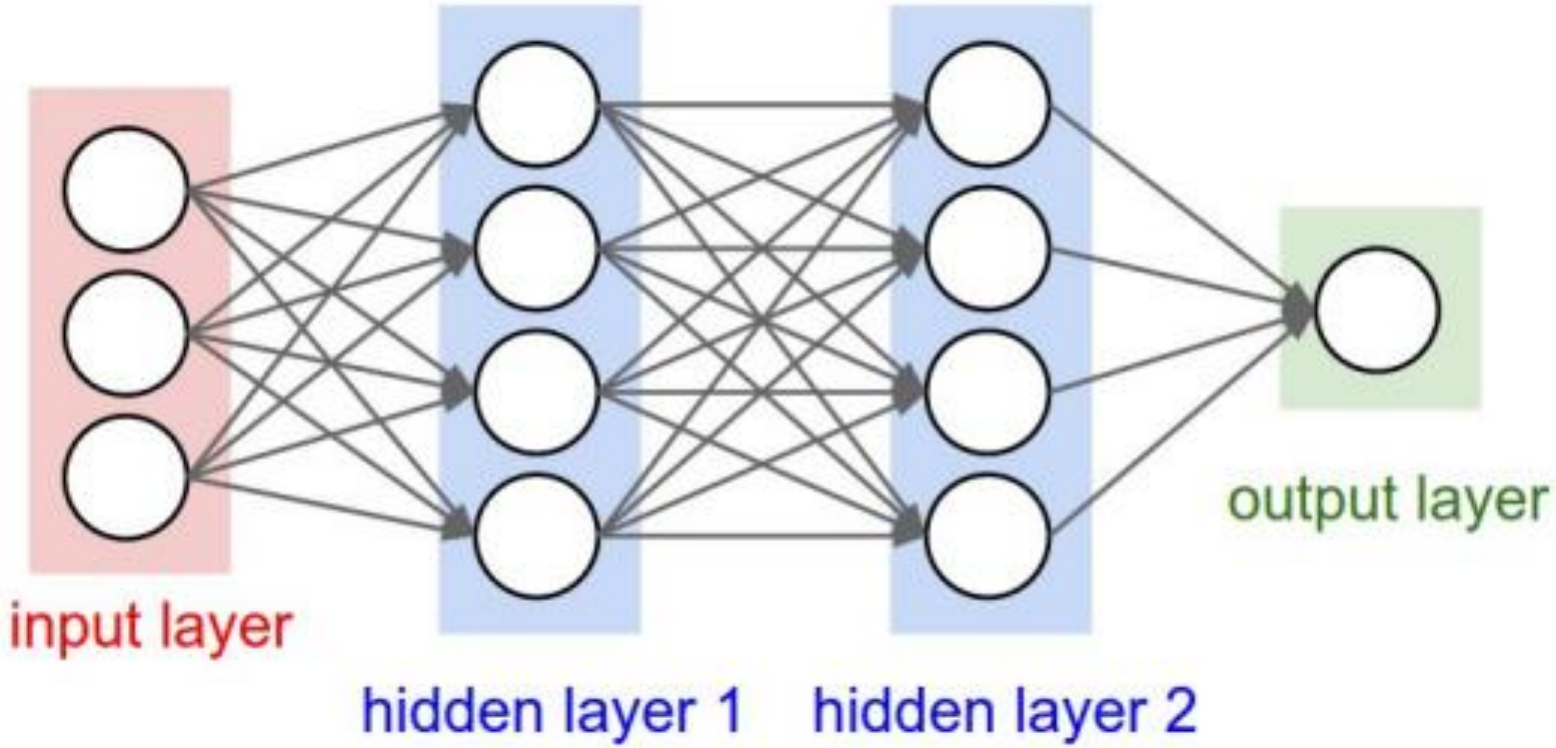
In other words, the Perceptron learning rule is guaranteed to converge to a weight vector that correctly classifies the examples provided the training examples are linearly separable.

Feed Forward Networks:

A **feedforward neural network** is an artificial neural network wherein connections between the units do *not* form a cycle. As such, it is different from recurrent neural networks.

The feedforward neural network was the first and simplest type of artificial neural network devised. In this network, the information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) and to the output nodes. There are no cycles or loops in the network.

A Feed-Forward Network:



Back Propagation Algorithm:

At the heart of back-propagation is an expression for the partial derivative $\partial C/\partial w$ of the cost function C with respect to any weight w (or bias b) in the network. The expression tells us how quickly the cost changes when we change the weights and biases.

And while the expression is somewhat complex, it also has a beauty to it, with each element having a natural, intuitive interpretation. And so back-propagation isn't just a fast algorithm for learning. It actually gives us detailed insights into how changing the weights and biases changes the overall behavior of the network.

Back-propagation has been used successfully for wide variety of applications, such as **speech or voice recognition, image pattern recognition, medical diagnosis, and automatic controls**.

The algorithm repeats a **two phase cycle, propagation and weight update**. When an input vector is presented to the network, it is propagated forward through the network, layer by layer, until it reaches the output layer.

The output of the network is then compared to the desired output, using a loss function, and an error value is calculated for each of the neurons in the output layer.

The error values are then propagated backwards, starting from the output, until each neuron has an associated error value which roughly represents its contribution to the original output.

Backpropagation uses these error values to calculate the gradient of the loss function with respect to the weights in the network.

In the second phase, this gradient is fed to the optimization method, which in turn uses it to update the weights, in an attempt to minimize the loss function.

Back-propagation requires a known, desired output for each input value in order to calculate the loss function gradient – it is therefore usually considered to be a **supervised learning method**; nonetheless, it is also used in some unsupervised networks such as **auto-encoders**.

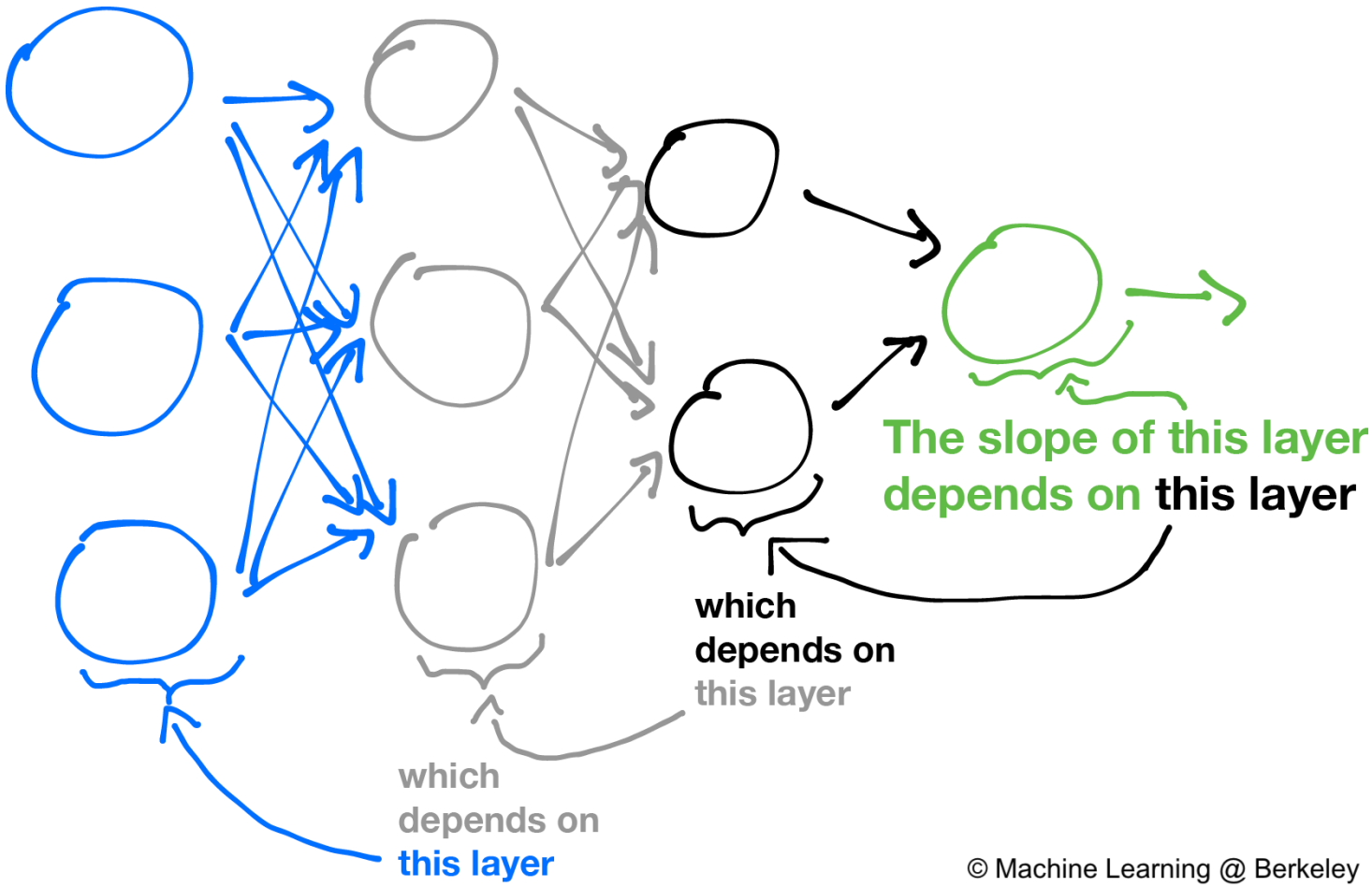
It is a generalization of the delta rule to multi-layered feed-forward networks, made possible by using the chain rule to iteratively compute gradients for each layer. Back-propagation requires that the activation function used by the artificial neurons (or "nodes") be differentiable.

Algorithm:

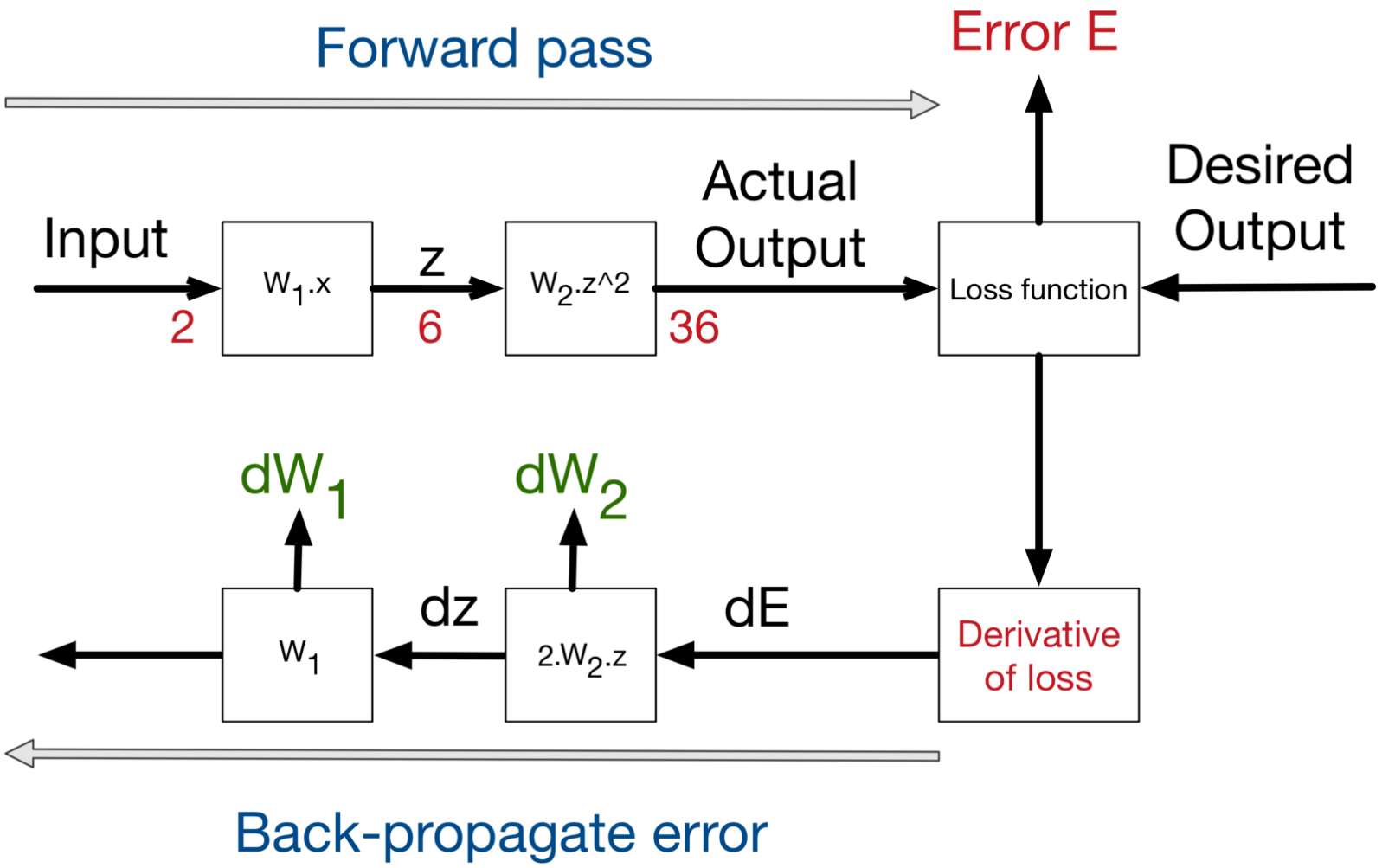
1. Compute the error values for the output units using the observed error.

2. Starting with output layer, repeat the following for each layer in the network, until the earliest hidden layer is reached:
    I. Propagate the error values back to the previous layer.
    II. Update the weights between the two layers.

Back-Propagation Algorithm:



Back-Propagation Algorithm:



Applications of neural networks:

- ✓ Computer Vision
- ✓ Speech Recognition
- ✓ Sentiment Analysis
- ✓ Natural Language Processing
- ✓ Medical Diagnosis
- ✓ Conversational Agents
- ✓ Autonomous Cars
- ✓ Machine Translation
- ✓ Stock Market Prediction
- ✓ Optical Character Recognition, etc.

- For course material and 100 free resources on AI and Machine Learning:
  - Fill the short feedback form.
  - Link: bit.ly/mlifeedback

Priyanka Kasture
pkasture2010@gmail.com
GitHub: @priyanka-kasture
Instagram: @priyanka.kasture
Community page: @ml.india
Telegram Channel: @machinelearning24x7
Community Membership: bit.ly/mlindia
Contact: 8329941605