Neural Networks and Deep Learning with R & Python

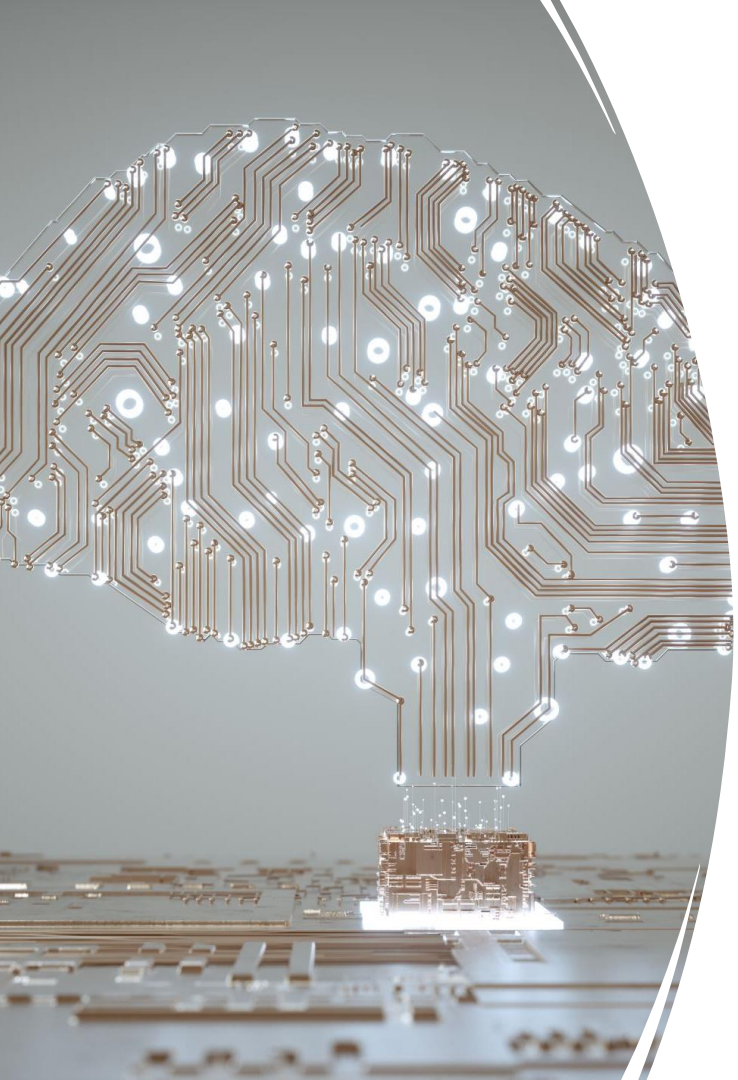
Part One: 04June24

Data Science Group

Mohammad Arashi

Professor and Director of Data Science Laboratory, Ferdowsi University of Mashhad, Iran
Extraordinary Professor, University of Pretoria, South Africa

Table of contents



What is Machine Learning?

- Machine learning (ML) is a sub-domain of artificial intelligence (AI), which makes a system automatically discover (learn) the statistical structure of the input data and convert those representations (patterns) to get closer to the expected output.
- A machine learning algorithm is an algorithm that is able to learn from the data.
- The process of learning is improved by a measure of the feedback, which compares the computed output to the expected output. The machine learning system is not explicitly programmed; it automatically searches for patterns within the hypothesis space and, uses the feedback signal (estimate) to correct those patterns.
- To enable a machine learning algorithm to work effectively on real-world data, we need to feed the machine a full range of features and possibilities to train on.

What is Learning?

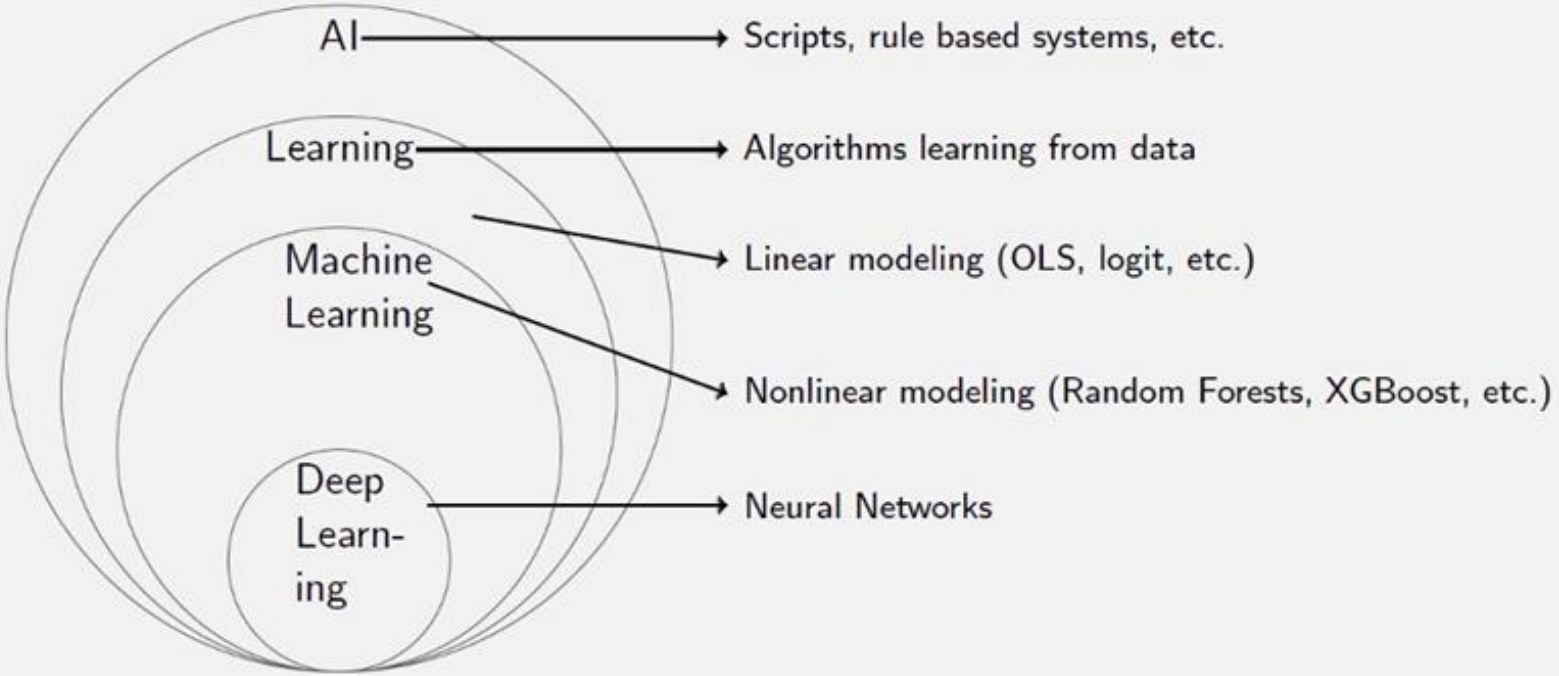


Figure 1: A Venn diagram of artificial intelligence. Inspired from Goodfellow, Bengio and Courville (2016)

Workflow

To enable a machine learning algorithm to work effectively on real-world data, we need to feed the machine a full range of features and possibilities to train on.

Training the model on a training data set, tuning the model on a validation (development) set and testing the model on an unseen test data set.

- Trying out the above on different yet appropriate algorithms using proper performance metrics.
- Selecting the most appropriate model.
- Testing the model on real-world data. If the results are not up to the speed, repeat the above by revaluating the data and/or model, possibly with different evaluation metrics.

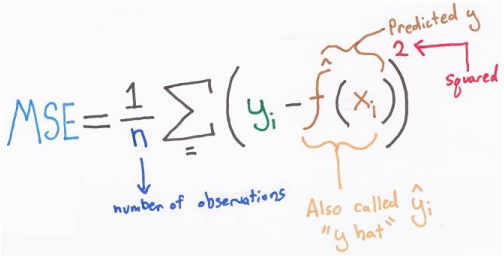
We define data sets which we use in machine learning as follows:

- **Training set**: is the data on which we learn the algorithm.
- **Validation set:** is the data on which we tune the hyperparameters of the model.
- **Test set**: is the data we use to evaluate the performance of our algorithm.
- Real-world set: is the data on which our selected model will be deployed.

Recipes of a Machine Learning Algorithm

- Input data points, e.g.
  - if the task is speech recognition, these data points could be sound files of people speaking
  - If the task is image tagging, they could be picture files

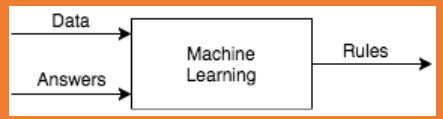
- Examples of the expected output
  - In a speech-recognition task, these could be human-generated transcripts of sound files
  - In an image task, expected outputs could tags such as "dog", "cat", and so on



- A way to measure whether the algorithm is doing a good job
  - This is necessary in order to determine the distance between the algorithm's current output and its expected output.
  - The measurement is used as a feedback signal to adjust the way the algorithm works. This adjustment step is what we call learning.

Differences

01



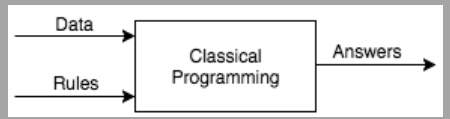
MACHINE LEARNING

focusses on predicting future data and evaluation of the model

algorithm that can learn from data without being explicitly programmed

Machine learning is a sub-domain of AI

02

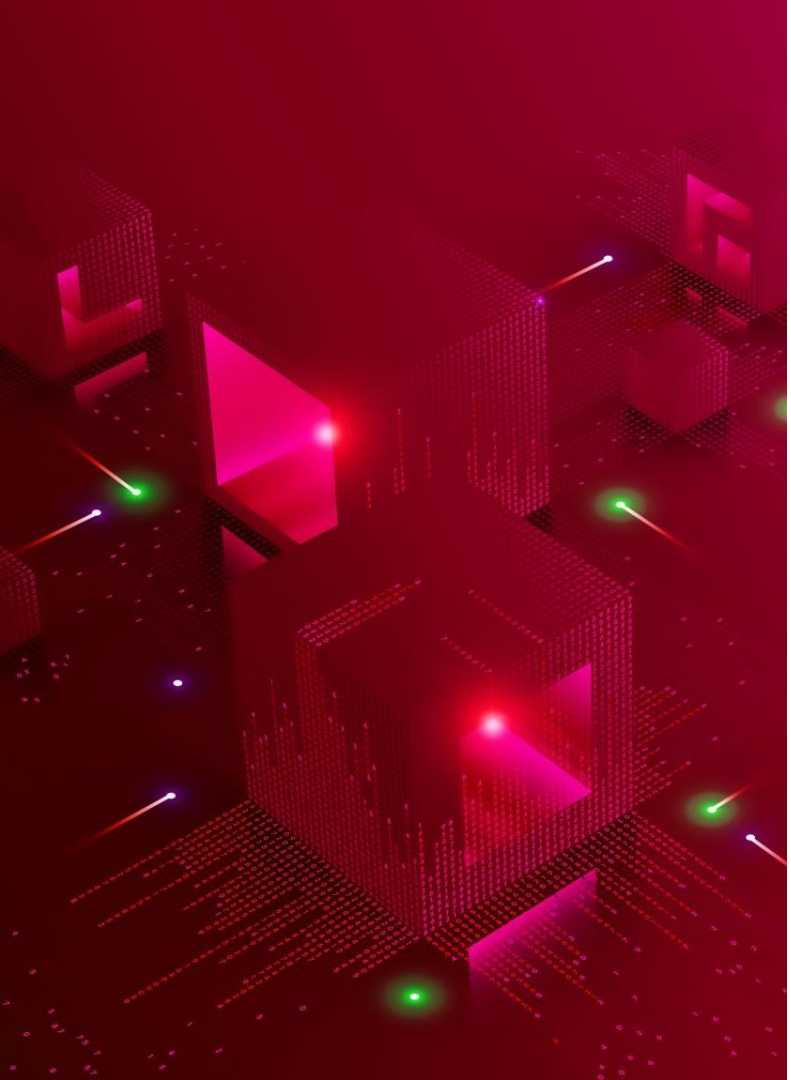


STATISTICS

focussed on the inference and explanation cum understanding of the phenomenon

statistical modeling is a mathematical representation of the relationship between different variables

statistics is a sub-domain of mathematics



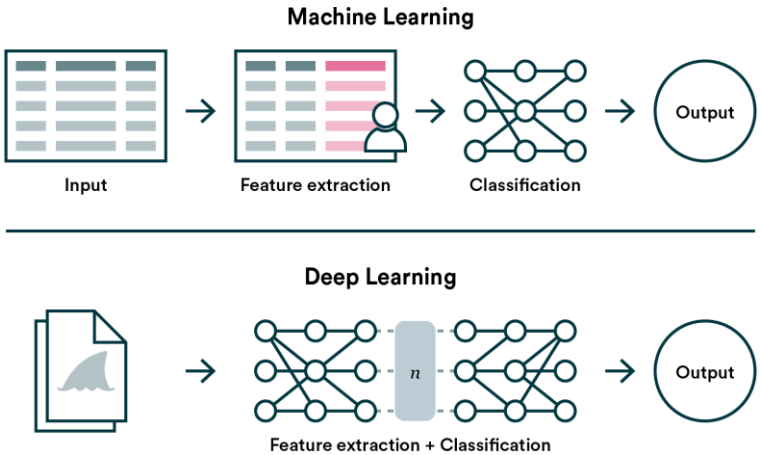
What is Deep Learning ?

Deep learning (DL) is a branch of ML which is based on artificial neural networks (ANNs). It is capable of learning complex patterns and relationships within data.

In DL, we don’t need to explicitly program everything. It has become increasingly popular in recent years due to the advances in processing power and the availability of large datasets. Because it is based on ANNs, it is also known as deep neural networks (DNNs).

These neural networks are inspired by the structure and function of the human brain’s biological neurons, and they are designed to learn from large amounts of data.

The difference between machine learning and deep learning



Machine learning vs. deep learning

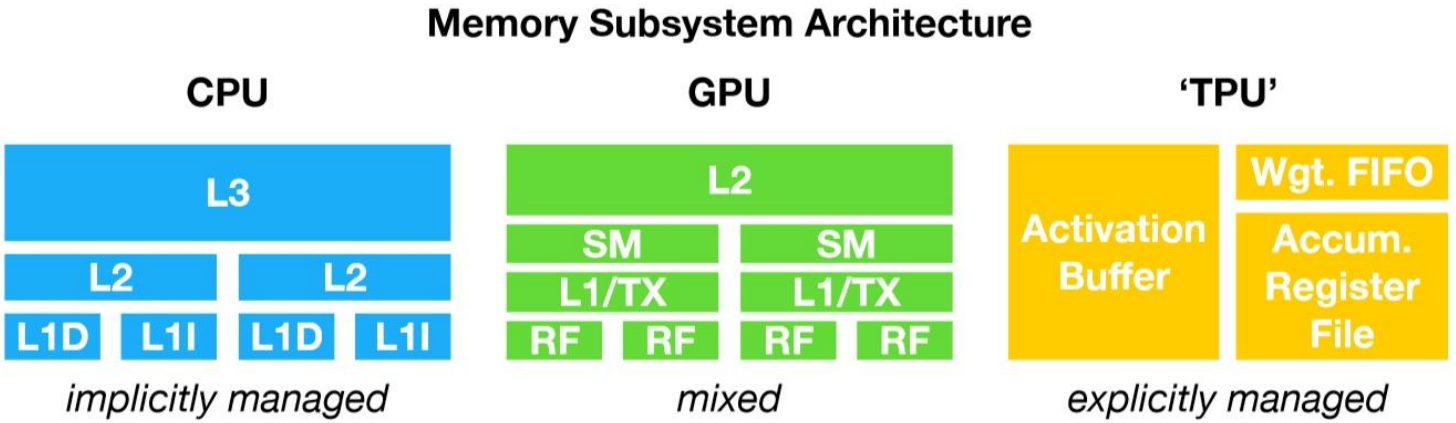
Machine learning

Uses algorithms and learns on its own but may need human intervention to correct errors

Deep learning

Uses advanced computing, its own neural network, to adapt with little to no human intervention

Applications of Deep Learning

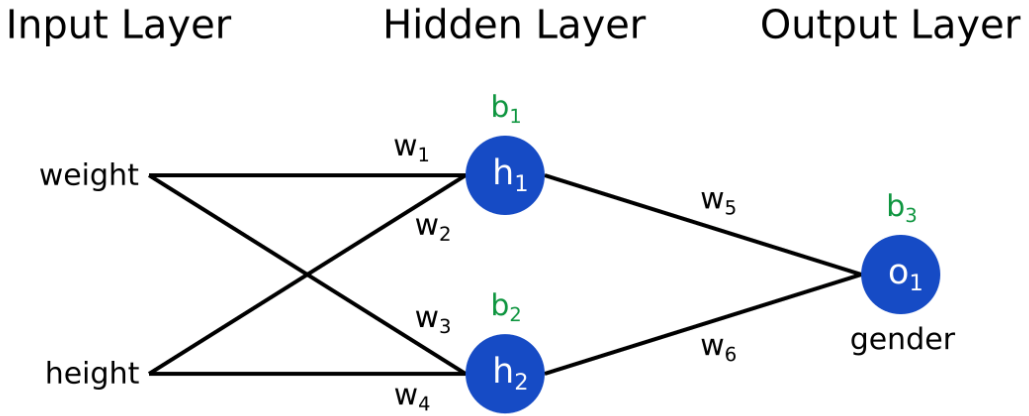


System requirements

- While deep learning models can run on CPUs, the truth is that any real work requires a workstation with a GPU.
- This does not mean that you need to go out and purchase one, as you can use cloud-computing to train your models. By running deep learning models in the cloud, we can use AWS, Azure, and Google Cloud to train deep learning models.

What Is a Neural Network?

- A neural network is a series of algorithms that endeavors to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates. In this sense, neural networks refer to systems of neurons, either organic or artificial in nature. Neural networks can adapt to changing input; so, the network generates the best possible result without needing to redesign the output criteria .



Bias, Variance, Under(Over)fitting

how well does our neural network model compare to human-level performance?

(a)If there exists a large gap between human-level error and the training error, then the model is too simple in relation to human-level performance, and it is a bias problem.

(b)If there is a small gap between the training error and human-level error and, a large difference between training error and validation error, it implies that our model has got high variance.

(c)If the training error is less than the human-level error, we cannot say for sure if our model suffers from bias or variance or a combination of both. There are a couple of domains where our model may surpass human-level performance and they include product recommendations, predicting transit time (Google maps), etc.

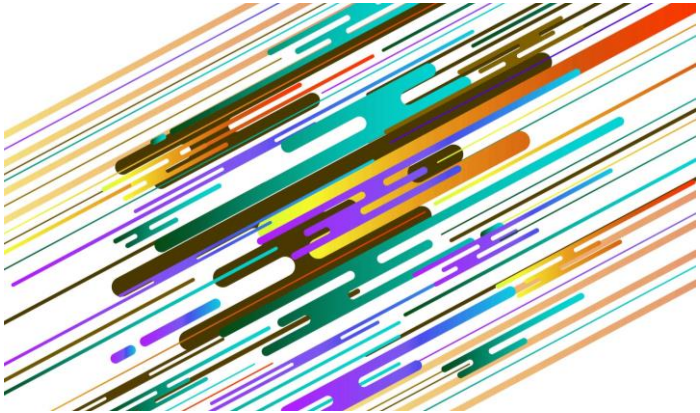
Bias

1. Train a more complex model.
2. Train for a longer duration.
3. Use better optimization algorithms—Momentum, RMSProp, Adam, etc.
4. Use different neural network architectures.
5. Carry out better hyperparameter search.

Variance

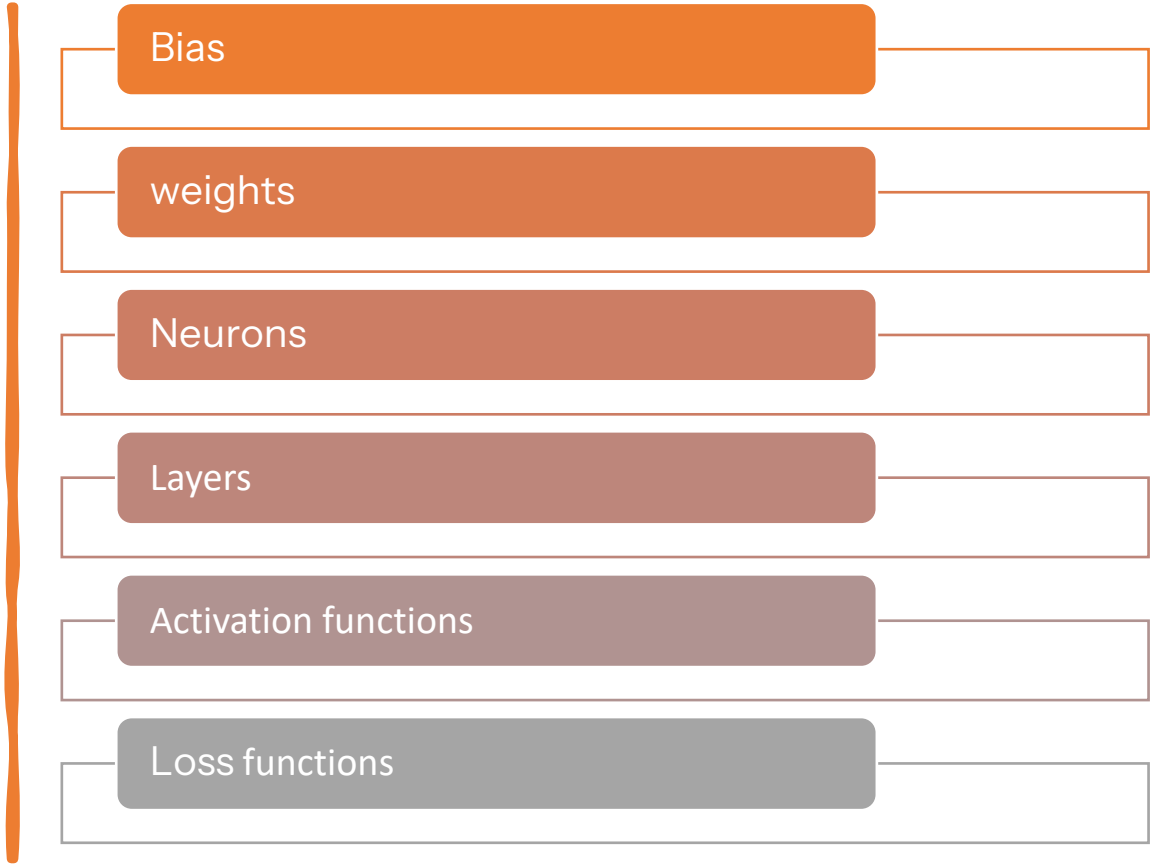
1. Use more data to train.
2. Use regularization methods, l_2, dropout, etc.
3. Use data augmentation.
4. Use different neural network architectures.
5. Carry out better hyperparameter search.

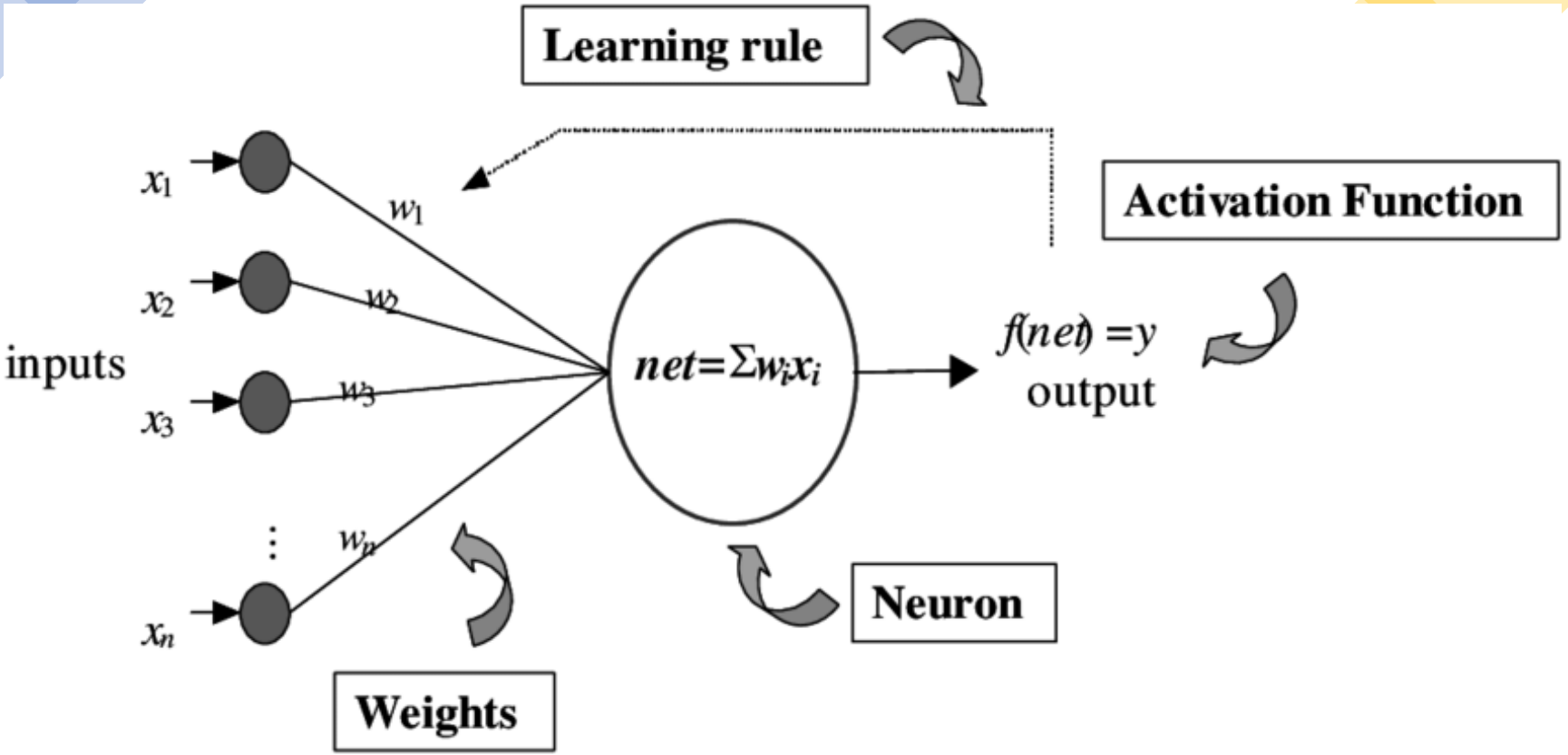
overfitting and underfitting

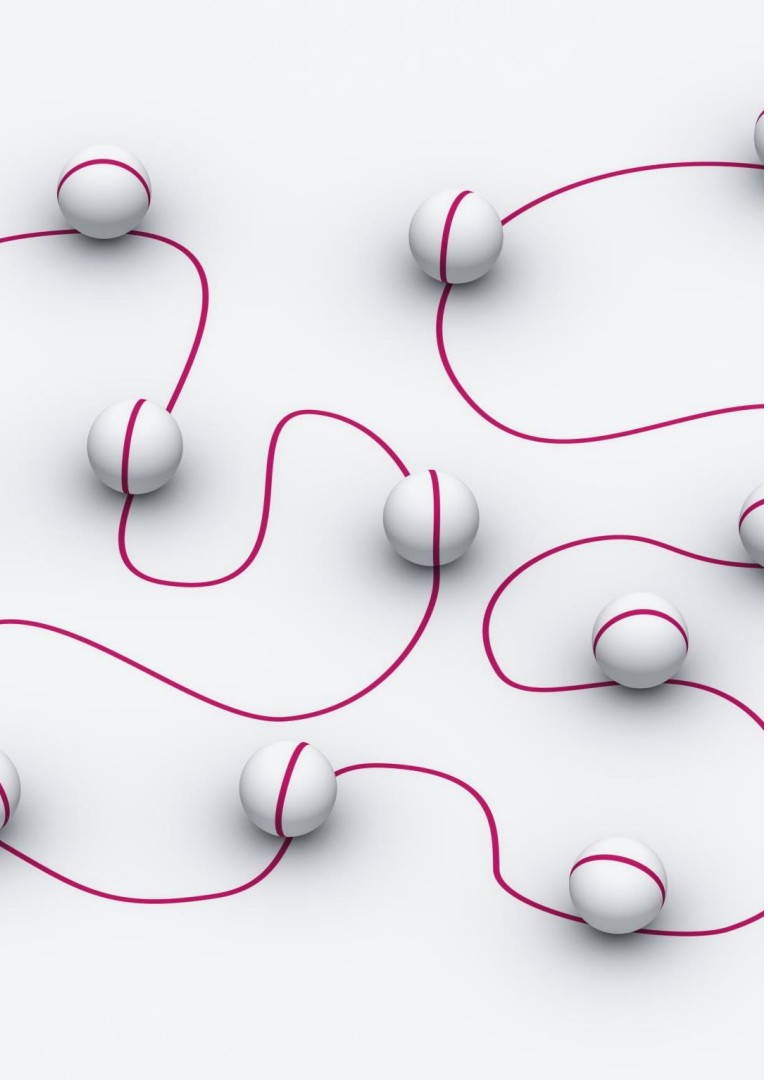


- When it comes to machine learning, overfitting and underfitting are terms that describe the performance of a predictive model. An overfit model is one that captures noise in the training data and consequently makes predictions that are too specific to that data, resulting in poor generalization to new data. On the other hand, an underfit model lacks the capacity to capture the underlying trend of the data, leading to poor performance even on the training data. In essence, overfitting and underfitting represent the delicate balance that a model must achieve to make accurate predictions.

Important concepts of neural network




Learning rule
Activation Function
x_1
x_2
x_3
x_n
w_1
w_2
w_3
w_n
inputs
$net=\Sigma w_i x_i$
$f(net) = y$
output
Neuron
Weights



Bias

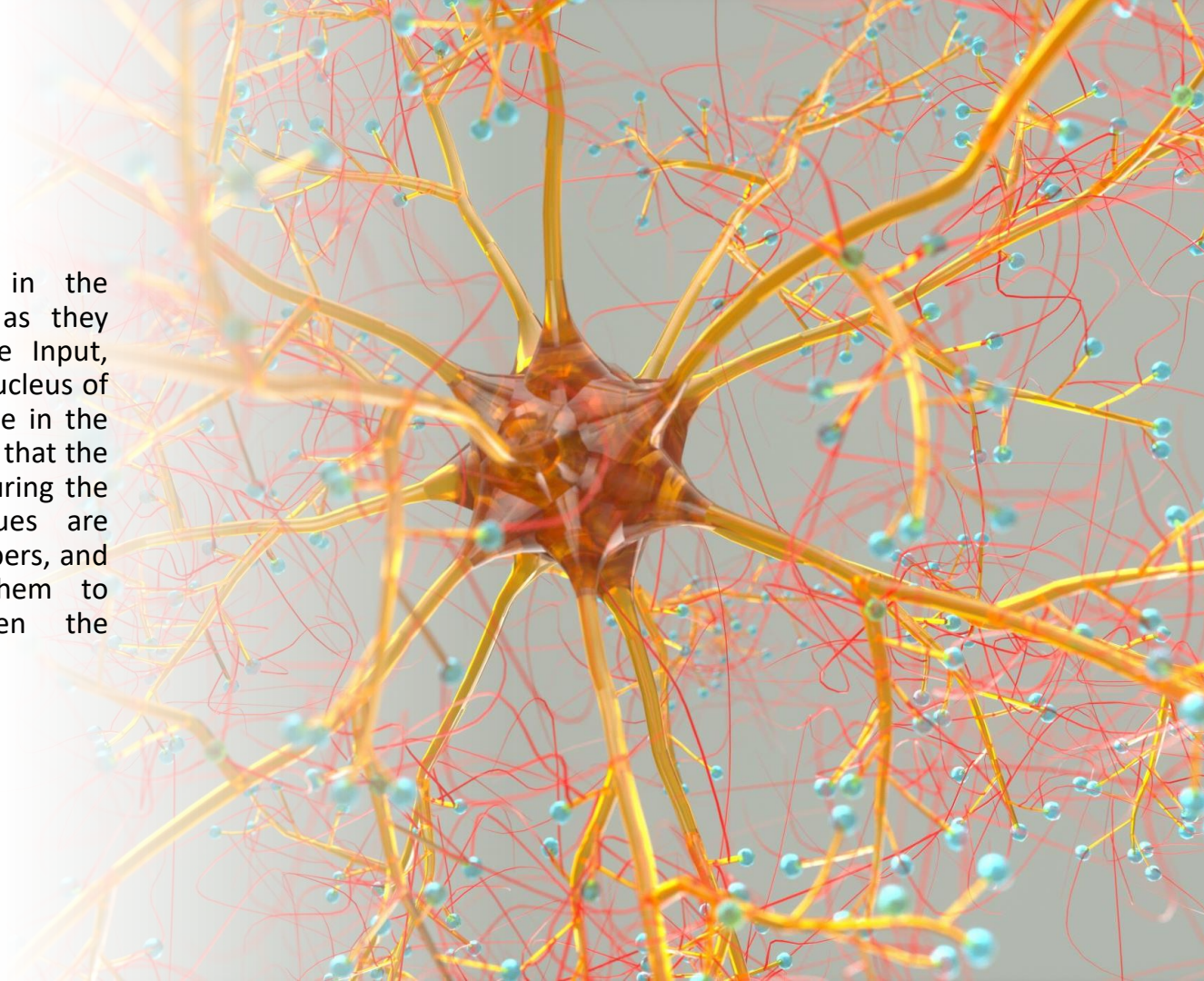
- In simple words, neural network bias can be defined as the constant which is added to the product of features and weights. It is used to offset the result. It helps the models to shift the activation function towards the positive or negative side. Bias is an additional input to each layer starting from the input layer. The bias is not dependent nor impacted by the preceding layer. In simple words, bias is the intercept term and is constant. It implies that even if there are no inputs or independent variables, the model will be activated with a default value of the bias. There, a bias works the same way it does in a linear equation:
- $y = wx + b$

Layers : Input Layer, Hidden Layer(s), and Output layer

- **Input layer** : The input layer is the first layer of any Neural Network and represents the input data to the network
- **Hidden layers** : The input and output pairs can have complex relationships, and to decode these relations, hidden layers exist between the input and output layers. Hidden layers also contain neurons; every Neuron connects to every other Neuron in adjacent layers.
- **Output layer** : The output layer of a NN represents the final predictions generated by the network. The number of neurons in this layer corresponds to the number of outputs desired for a given input. In a regression problem, where a single output value is expected, there will be one neuron in the output layer. However, in classification tasks, where multiple output classes are possible, there will be multiple neurons, one for each class. For example, in a handwritten digit recognition task, there will be 10 neurons corresponding to the 10 possible classes (0-9) .

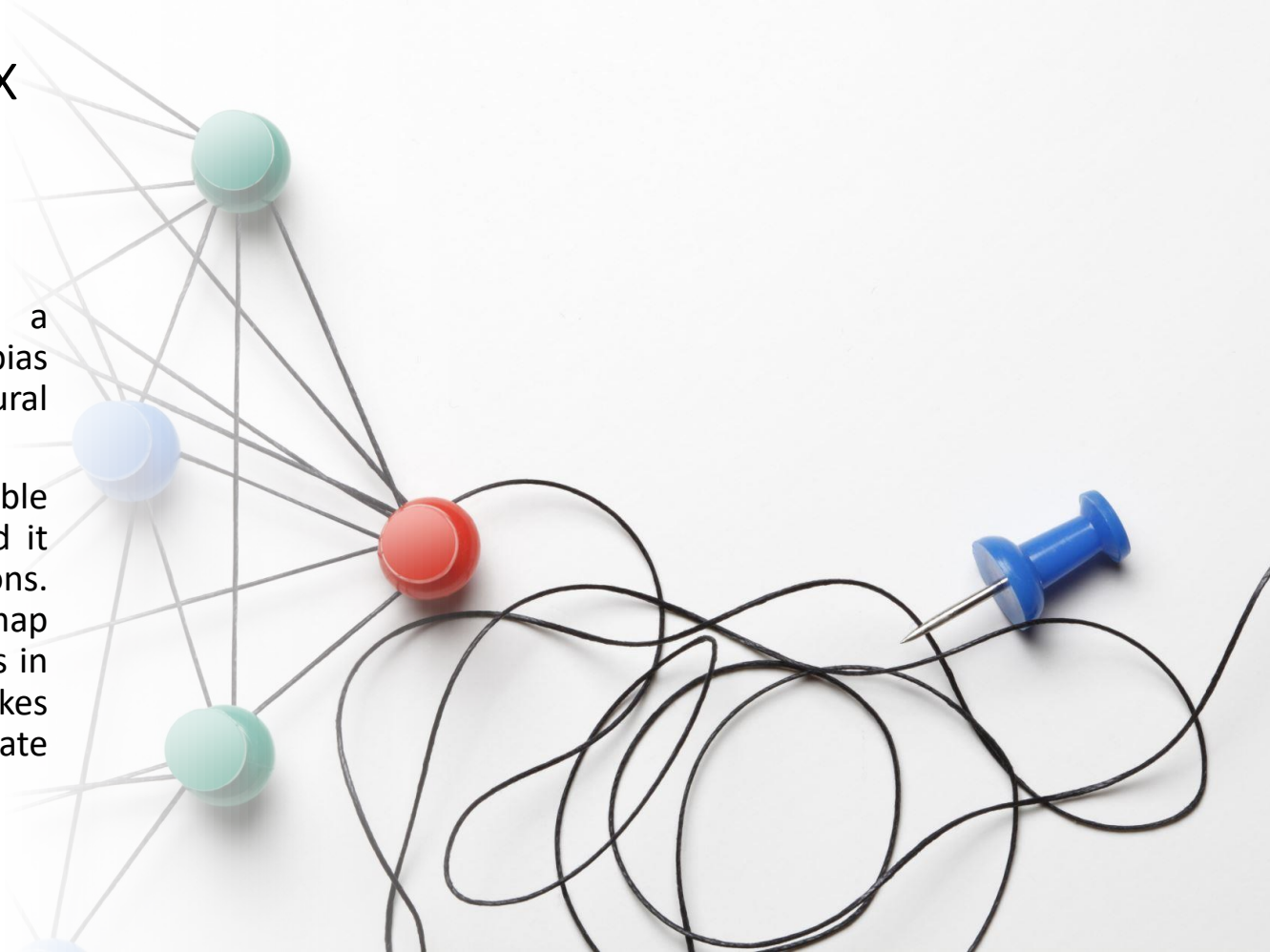
Neurons

- Neurons play a crucial role in the functioning of a Neural Network, as they constitute every layer, including the Input, Output, and Hidden layers. Like the nucleus of brain cells, each neuron, except those in the Input layer, contains a bias parameter that the neural network learns and adjusts during the training process. These bias values are typically initialized with random numbers, and the neural network fine-tunes them to minimize the difference between the computed and actual output.



Weight Matrix

- The weight matrix, a combination of weight and bias values, is a crucial aspect of neural networks.
- It represents the learnable parameters of the network, and it helps the model make predictions. The weight matrix is used to map the input to the output, and it is in the form of a matrix, which makes it easy to compute and update during the training process.

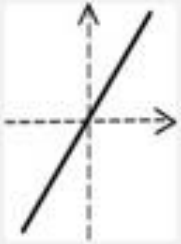
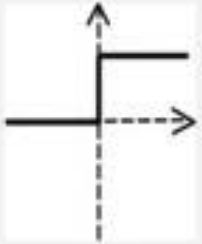
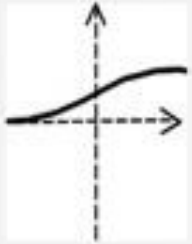
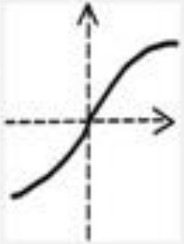
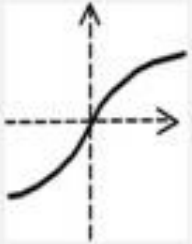
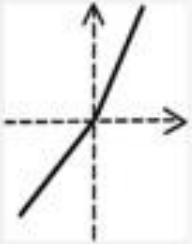
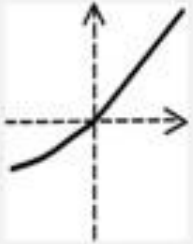
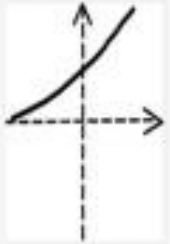


Activation Function

In neural networks, we have activation functions for neurons of every layer. We define the activation function for a layer, and all the neurons in that layer follow that same activation function. Neurons of every layer get the input from the previous layer multiplied by weight values. Based on the relationship we want to maintain between weighted input and the corresponding output from the Neuron, we can divide activation functions into two types :

- **Linear Activation Functions** : In a Neural Network, the linear activation function is used to directly pass the weighted input as the output without any additional transformation. This function ensures that the relationship between the incoming weighted input and the corresponding output from that neuron is linear. This type of activation function is commonly used in the output layer of a Neural Network.
- **Non-Linear Activation Functions** : Non-linear activation functions are mathematical functions that transform the weighted inputs received by a neuron to produce a non-linear relationship between the input and output. These functions help neurons in extracting complex patterns present in the data and are mostly used in the hidden layers of a Neural Network. Some popular non-linear activation functions include Sigmoid, ReLU, and Softmax.

Activation Function

Identity	Binary Step	Logistic	TanH	ArcTan	ReLU	PreLU	ELU	SoftPlus
								
$f(x)$	$f(x)$	$f(x)$	$f(x)$	$f(x)$	$f(x)$	$f(x)$	$f(x)$	$f(x)$
$=$	$=$	$=$	$=$ $tangh(x)$	$=$	$=$	$=$	$=$	$=$
x	$\begin{cases} 0 \text{ for } x<0 \\ 1 \text{ for } x\geq 0 \end{cases}$	$\frac{1}{1+e^{-x}}$	$=$ $\frac{2}{1+e^{-2x}}-1$	$tan^{-1}(x)$	$\begin{cases} 0 \text{ for } x<0 \\ x \text{ for } x\geq 0 \end{cases}$	$\begin{cases} \propto x \text{ for } x<0 \\ x \text{ for } x\geq 0 \end{cases}$	$\begin{cases} \propto(e^{-x}-1) \text{ for } x<0 \\ x \text{ for } x\geq 0 \end{cases}$	$log_e(1+e^x)$
$f'(x)$	$f'(x)$	$f'(x)$	$f'(x)$	$f'(x)$	$f'(x)$	$f'(x)$	$f'(x)$	$f'(x)$
$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$
1	$\begin{cases} 0 \text{ for } x\neq 0 \\ ? \text{ for } x=0 \end{cases}$	$f(x)(1-f(x))$	$1-f(x)^2$	$\frac{1}{x^2+1}$	$\begin{cases} 0 \text{ for } x<0 \\ 1 \text{ for } x\geq 0 \end{cases}$	$\begin{cases} \propto \text{ for } x<0 \\ 1 \text{ for } x\geq 0 \end{cases}$	$\begin{cases} f(x)+\propto \text{ for } x<0 \\ 1 \text{ for } x\geq 0 \end{cases}$	$\frac{1}{1+e^{-x}}$

Loss or Cost functions

- Loss and cost functions are one of the most vital components of any machine learning algorithm. Machines only understand numbers; hence we need to convey our objectives in numbers. If our framed objectives in the form of numbers do not represent what we want our machines to learn, the fault will be ours that machines will fail to learn. Machines try to optimize these costs by changing the parameter values and achieving the best suitable ones for which the cost becomes minimum.
- The cost function should represent the problem statement in the form of errors between the predicted output by the NN and the expected output by the NN.
- There can be thousands of parameters in NN. Hence slight changes in the parametric values should reflect the change in the cost function.
- The cost function should be differentiable. For example, the error between predicted and expected output can be one degree. Still, we can make the error as our cost function that will make it differentiable and understandable by machines in training.
- The cost function is defined only for the output layer and not the Input and Hidden layers.

Optimization Algorithm

• Optimizing the cost function in Machine Learning and Neural Networks is a crucial task, as trying all possible values for the weight matrix would take an excessive amount of time, even with the use of supercomputers. To aid in this process, various optimization algorithms such as Gradient Descent, Gradient Descent with momentum, Stochastic Gradient Descent, Coordinate Descent, and Adam are utilized. It is important to note that these optimization algorithms are only applied to the output layer, as the cost function is defined for this layer only.

Gradient

Minimizing a multivariate function involves finding a point where the gradient is zero:

$$\nabla_{\boldsymbol{\omega}} \mathcal{J} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{\omega}} = \mathbf{0}\ (the\ vector\ of\ zeros)$$

Points where the gradient is zero are local minima

- If the function is convex, also a global minimum

Let's solve the ordinary least squares problem!

Recall the OLS optimization problem

$$\text{minimize}_{\boldsymbol{\omega}} \frac{1}{2} \|\boldsymbol{X\omega} - \boldsymbol{y}\|_2^2$$

Gradient of the optimization objective

$$\nabla_{\omega} \frac{1}{2} \|\boldsymbol{X\omega} - \boldsymbol{y}\|_2^2 = \boldsymbol{X^T}(\boldsymbol{X\omega} - \boldsymbol{y})$$

points where the gradient equals zero are minima

$$\boldsymbol{X^T}(\boldsymbol{X\omega} - \boldsymbol{y}) = \mathbf{0} \Rightarrow \boldsymbol{X^T X\omega} = \boldsymbol{X^T y} \Rightarrow \left(\boldsymbol{X^T X}\right)^{-1} \boldsymbol{X^T X\omega} = \left(\boldsymbol{X^T X}\right)^{-1} \boldsymbol{X^T y} \Rightarrow \boldsymbol{\omega} = \left(\boldsymbol{X^T X}\right)^{-1} \boldsymbol{X^T y}$$

Gradient descent

Initialize the parameter vector $\boldsymbol{\omega}$:

$$\boldsymbol{\omega} \leftarrow \mathbf{0}$$

Repeat until satisfied (e.g., exact or approximate convergence)---choose a new value for $\boldsymbol{\omega}$ to reduce $\mathcal{J}(\boldsymbol{\omega})$:

- Compute gradient:

$$\nabla_{\boldsymbol{\omega}} \mathcal{J}(\boldsymbol{\omega})$$

- Update parameters: (α is the learning rate)

$$\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} - \alpha \nabla_{\boldsymbol{\omega}} \mathcal{J}(\boldsymbol{\omega})$$

Let's solve the ordinary least squares problem using gradient descent!

Recall the OLS optimization problem $\text{minimize}_{\boldsymbol{\omega}} \frac{1}{2} \|\boldsymbol{X}\boldsymbol{\omega} - \boldsymbol{y}\|_2^2$

$$\frac{1}{2} \|\boldsymbol{X}\boldsymbol{\omega} - \boldsymbol{y}\|_2^2 = \frac{1}{2} \sum_{i=1}^{n} (\boldsymbol{\omega}^T \boldsymbol{x}_i - y_i)^2 , \qquad \boldsymbol{\omega} = \begin{pmatrix} \omega_0 \\ \omega_1 \\ \vdots \\ \omega_m \end{pmatrix}, \boldsymbol{X} = \begin{pmatrix} \boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_n^T \end{pmatrix}, \boldsymbol{x}_i = \begin{pmatrix} x_{i0} \\ x_{i1} \\ \vdots \\ x_{im} \end{pmatrix}, \boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

Gradient descent

Recall the OLS optimization problem $\text{minimize}_{\boldsymbol{\omega}} \frac{1}{2}\|\boldsymbol{X\omega} - \boldsymbol{y}\|_2^2$ and thus

$$\nabla_{\omega_j} \mathcal{J}(\boldsymbol{\omega}) = \nabla_{\omega_j} \frac{1}{2} \sum_{i=1}^{n} (\boldsymbol{\omega}^T \boldsymbol{x}_i - y_i)^2 = \frac{1}{2} \sum_{i=1}^{n} (\boldsymbol{\omega}^T \boldsymbol{x}_i - y_i) x_{ij}\,, j = 0,1, \dots, m$$

$\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} - \boldsymbol{\alpha} \nabla_{\boldsymbol{\omega}} \mathcal{J}(\boldsymbol{\omega})$ ↓	
$\omega_0 \coloneqq \omega_0 - \alpha \nabla_{\omega_0} \mathcal{J}(\boldsymbol{\omega})$ $\omega_1 \coloneqq \omega_1 - \alpha \nabla_{\omega_1} \mathcal{J}(\boldsymbol{\omega})$ $\vdots$ $\omega_m \coloneqq \omega_m - \alpha \nabla_{\omega_m} \mathcal{J}(\boldsymbol{\omega})$	$\omega_0 \coloneqq \omega_0 - \alpha \frac{1}{2} \sum_{i=1}^{n} (\boldsymbol{\omega}^T \boldsymbol{x}_i - y_i) x_{i0}$ $\omega_1 \coloneqq \omega_1 - \alpha \frac{1}{2} \sum_{i=1}^{n} (\boldsymbol{\omega}^T \boldsymbol{x}_i - y_i) x_{i1}$ $\vdots$ $\omega_m \coloneqq \omega_m - \alpha \frac{1}{2} \sum_{i=1}^{n} (\boldsymbol{\omega}^T \boldsymbol{x}_i - y_i) x_{im}$

Note: $x_{i0} = 1$

Gradient descent vs closed-form solution

Gradient descent

- Requires multiple iterations
- Need to choose α (it is small)
- Works well when n is large
- Can support incremental learning

Closed-from solution

- Non-iterative
- No need for α
- **Slow if n is large**
- $(X^T X)$ might not be invertible

Parameters

Trainable parameters, are continuously updated during an experiment to minimize the cost function.

Hyperparameters, such as the number of hidden layers or the learning rate, remain mostly fixed or are modified in a systematic manner across several experiments to ensure the overall minimum cost function.

Trainable Parameters

- Can be modified during the learning process. These parameters include the biases of neurons in all layers except the input layer and the weights of connections between neurons. The total number of trainable parameters can be calculated by adding the total number of neurons in the hidden and output layers to the total number of connections.

Hyperparameters (Untrainable Parameters)

- It is important to set certain fixed values, known as hyperparameters, which are then fine-tune through experimentation to achieve the lowest possible cost value. Understanding and optimizing these hyperparameters plays a crucial role in effectively training and utilizing ANNs for a wide range of machine learning tasks and helps to ensure the best possible performance and accuracy in predictions.

Example

- Input neurons = 7
- Hidden layers = 2 (5 and 4 neurons)
- Output layer = 3 neurons
- Total number of weights = $(7 \times 5) + (5 \times 4) + (4 \times 3) = 67$
- Total number of biases = $(5 + 4 + 3) = 12$
- Total number of parameters = $67 + 12 = 79$

Type of Hyperparameters

Design-related hyperparameters

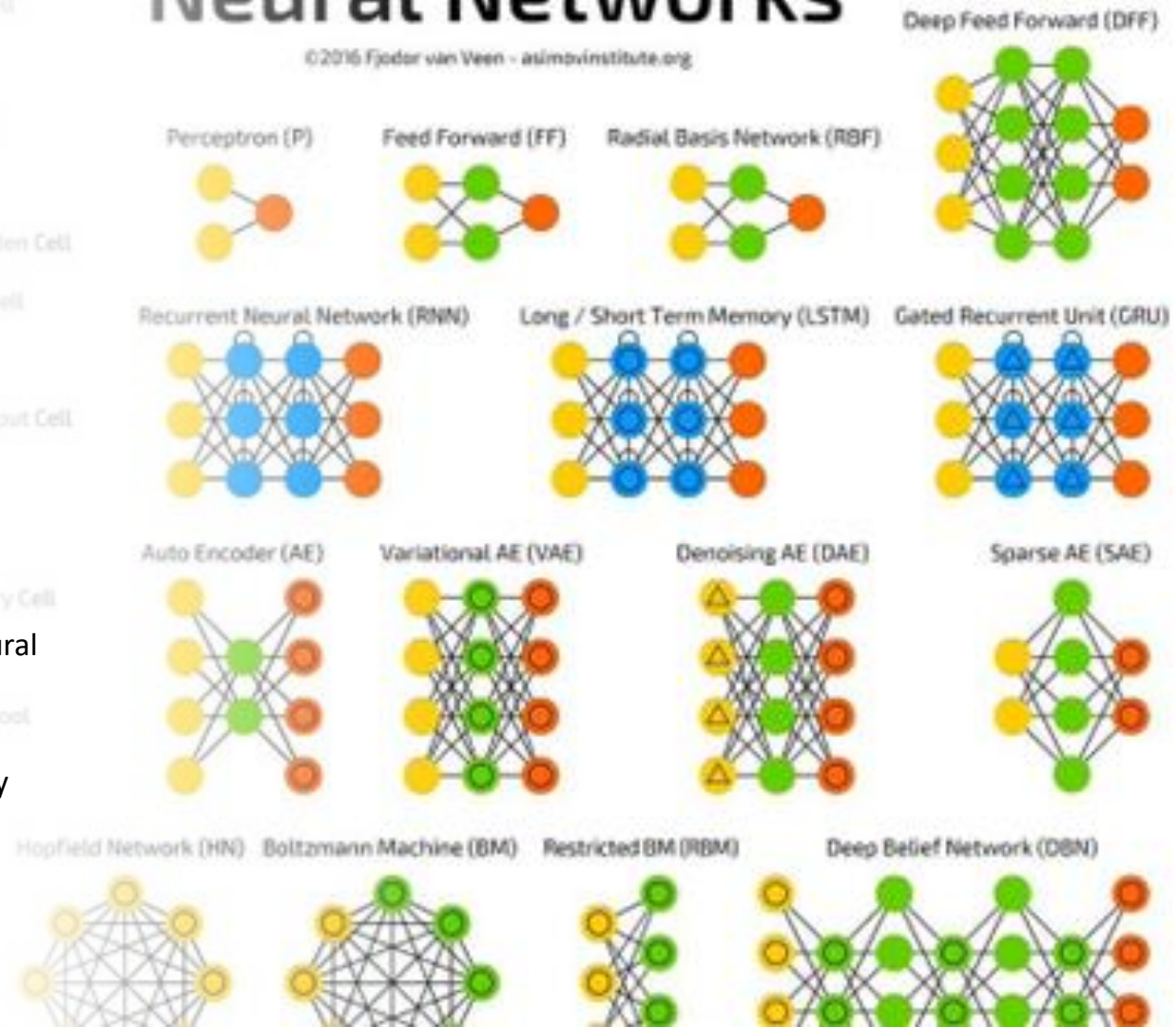
- Number of hidden layers - The number of neurons per hidden layer - The choice of activation function - The optimization algorithm used - The cost function selected - and cost function selected .

General Hyperparameters tuned after design finalization

- Learning rate – Regularization - Batch size

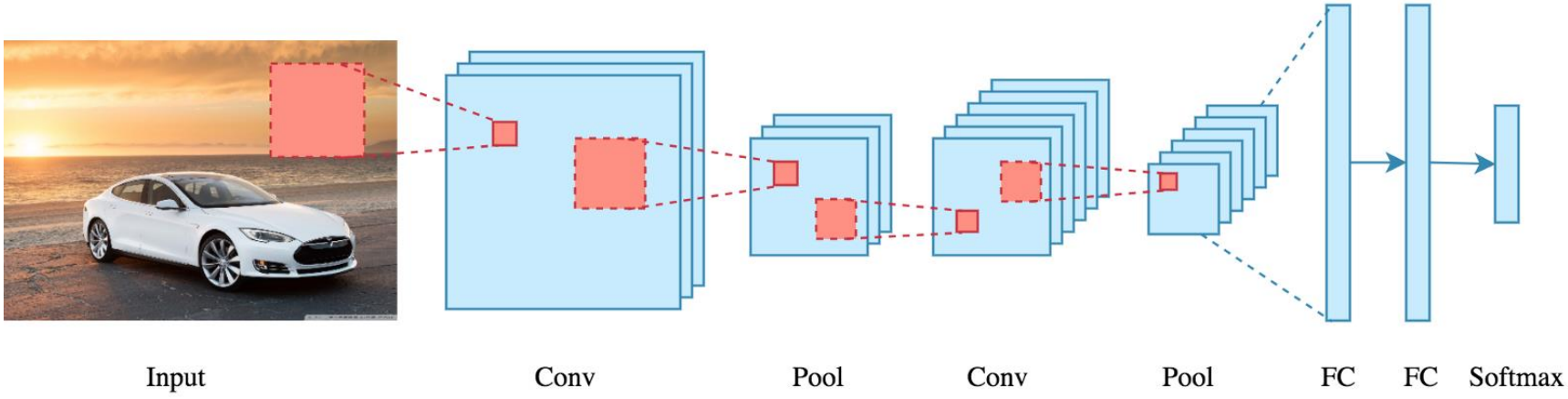
Types of neural networks

- Perceptron
- Feed Forward Neural Network
- Multilayer Perceptron
- Convolutional Neural Network
- Radial Basis Functional Neural Network
- Recurrent Neural Network
- LSTM – Long Short-Term Memory
- Sequence to Sequence Models
- Modular Neural Network



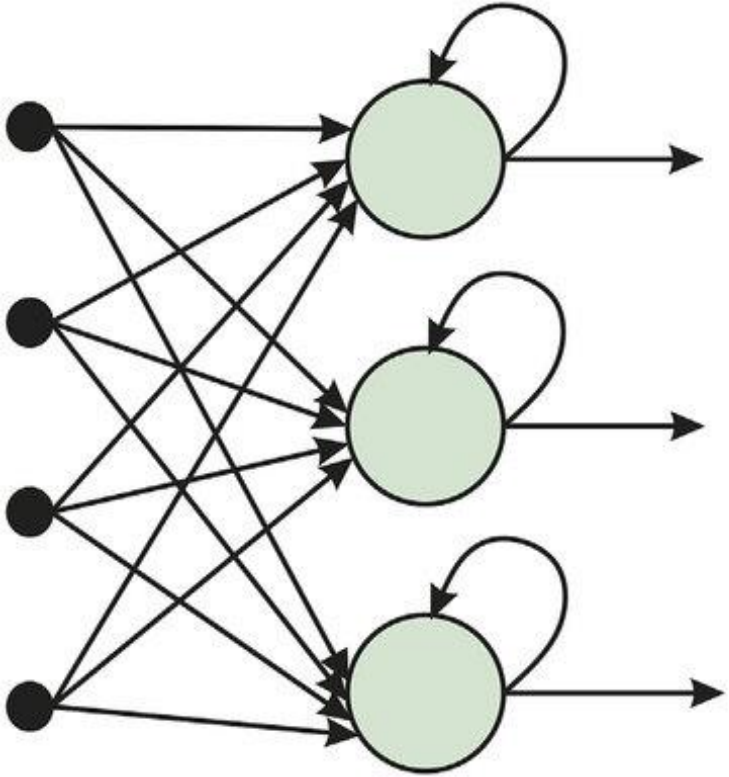
Convolutional Neural Network

A convolutional neural network (CNN/ConvNet) is a class of deep neural networks, most applied to analyze visual imagery. Now when we think of a neural network, we think about matrix multiplications but that is not the case with ConvNet. It uses a special technique called Convolution. Now in mathematics convolution is a mathematical operation on two functions that produces a third function that expresses how the shape of one is modified by the other.

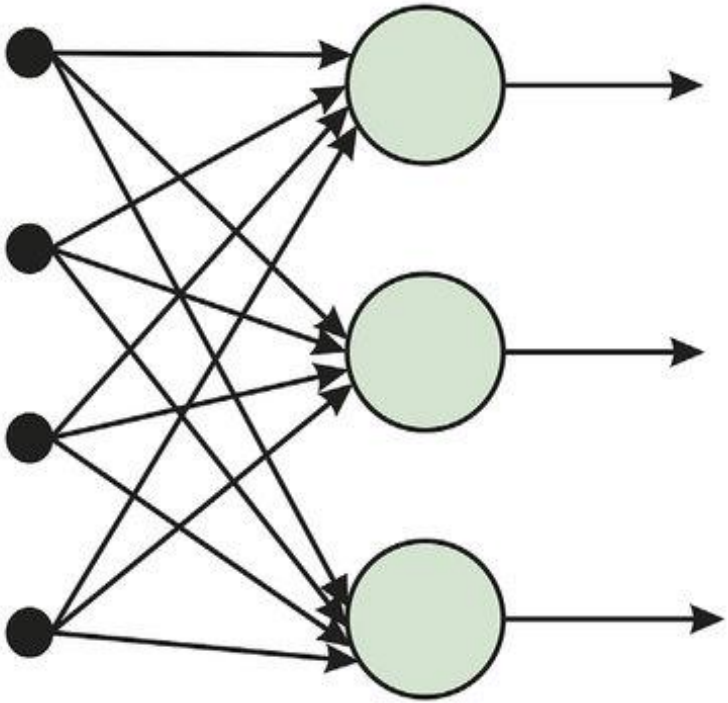

Input
Conv
Pool
Conv
Pool
FC
FC
Softmax

Recurrent Neural Network

A recurrent neural network (RNN) is a type of artificial neural network which uses sequential data or time series data. These deep learning algorithms are commonly used for ordinal or temporal problems, such as language translation, natural language processing (NLP), speech recognition, and image captioning; they are incorporated into popular applications such as Siri, voice search, and Google Translate. Like feedforward and convolutional neural networks (CNNs), recurrent neural networks utilize training data to learn. They are distinguished by their “memory” as they take information from prior inputs to influence the current input and output. While traditional deep neural networks assume that inputs and outputs are independent of each other, the output of recurrent neural networks depend on the prior elements within the sequence. While future events would also be helpful in determining the output of a given sequence, unidirectional recurrent neural networks cannot account for these events in their predictions.

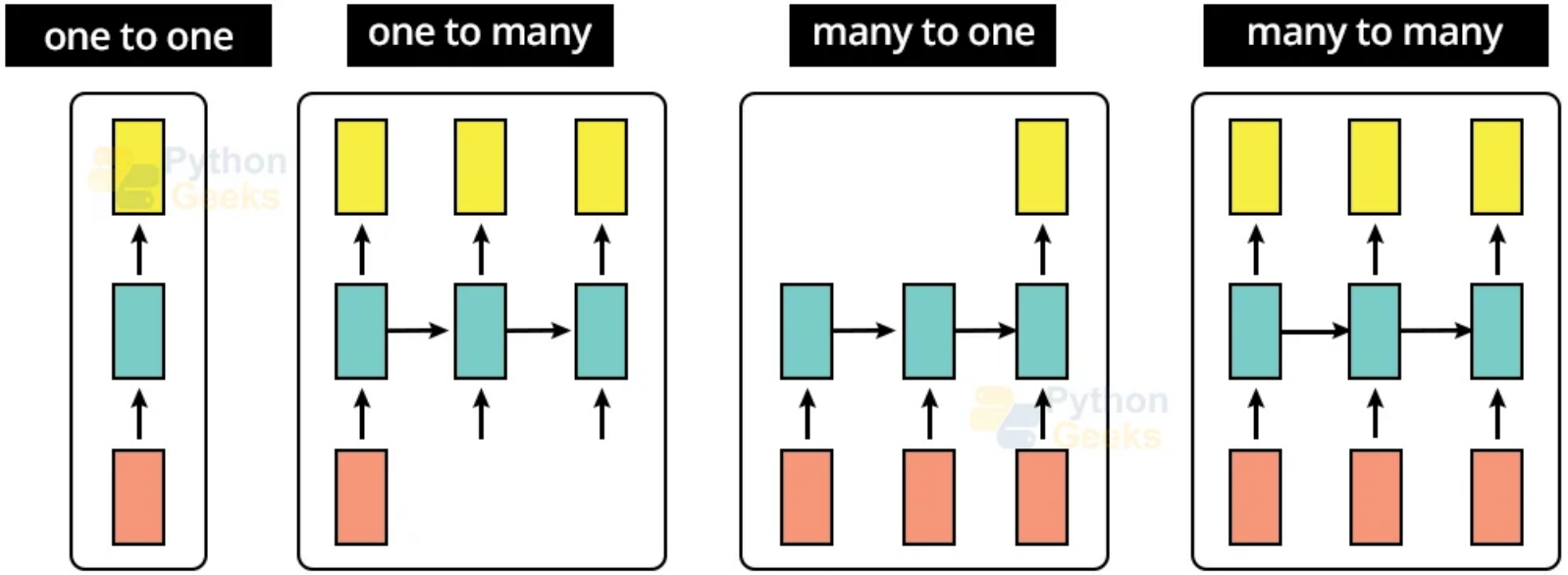


(a) Recurrent Neural Network



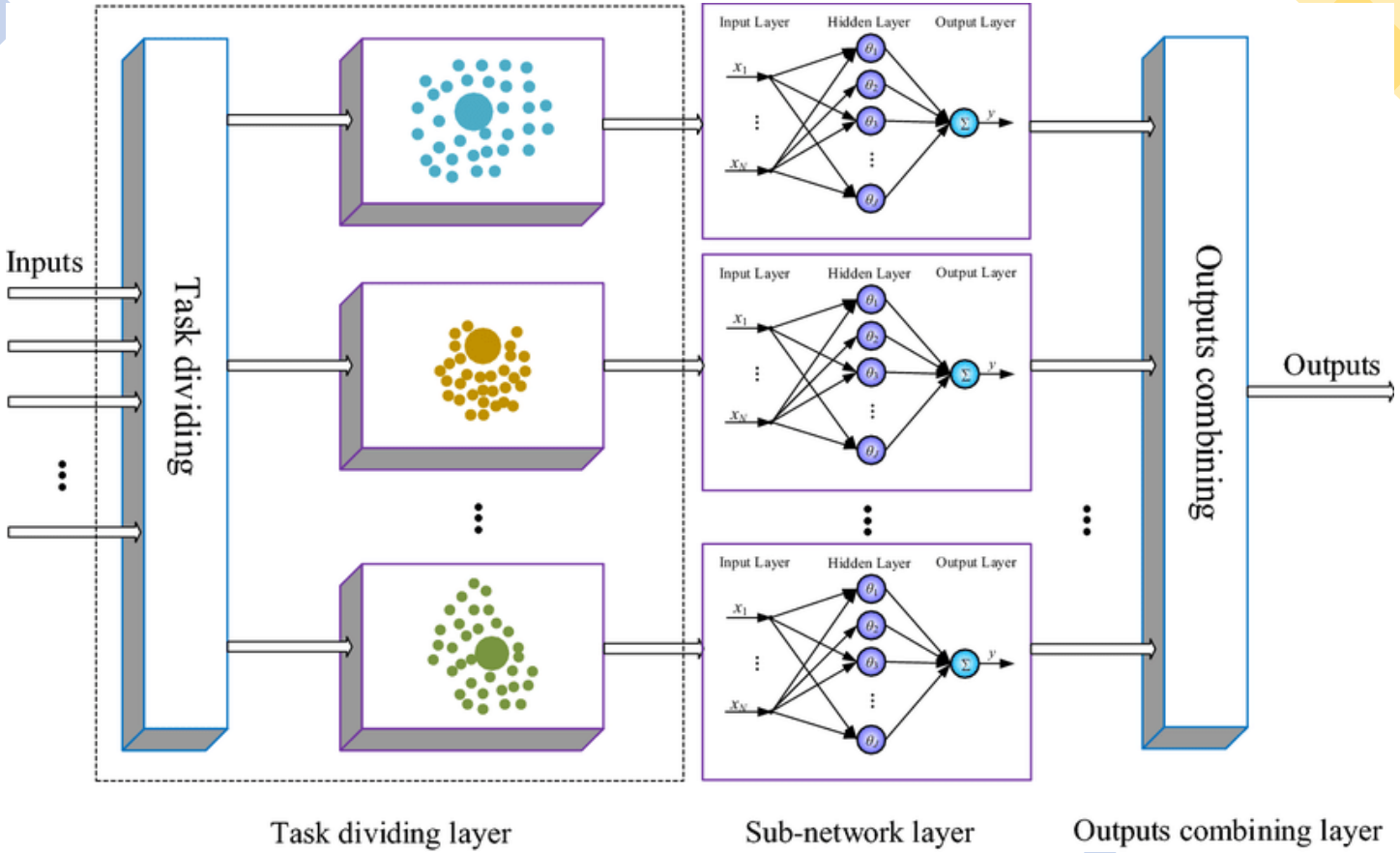
(b) Feed-Forward Neural Network

Types of recurrent neural networks



Modular Neural Network

- A modular neural network is made up of several neural network models that are linked together via an intermediate. Modular neural networks allow for more complex management and handling of more basic neural network systems. In this case, the multiple neural networks act as modules, each solving a portion of the issue. An integrator is responsible for dividing the problem into multiple modules as well as integrating the answers of the modules to create the system's final output. Modular neural networks, in general, allow engineers to expand the possibilities of employing these technologies to push the limits of what neural networks can do.
- Each network is converted into a module that may be freely combined with modules of different sorts. We get to the notion of modular neural networks in this way.


Inputs
Task dividing
Input Layer
Hidden Layer
Output Layer
Outputs combining
Outputs
Task dividing layer
Sub-network layer
Outputs combining layer

Why do we use MNN?

Reducing model complexity: Controlling the degrees of freedom of the system is one method to minimize training time.

Data fusion and prediction averaging:

Network committees may be thought of as composite systems consisting of comparable parts.

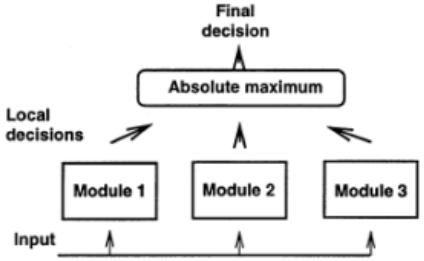
Combination of techniques: As a building block, more than one method or network class can be utilized.

Learning several tasks at the same time:

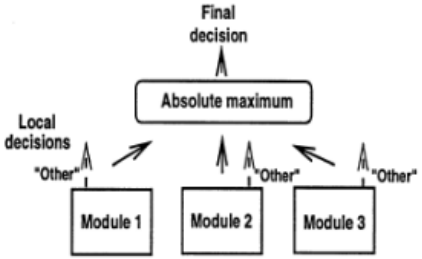
Trained modules can be transferred between systems that are built for various tasks.

Robustness and incrementality:

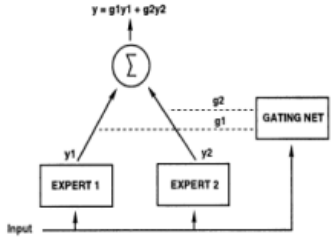
The integrated network may be fault-tolerant and develop progressively.



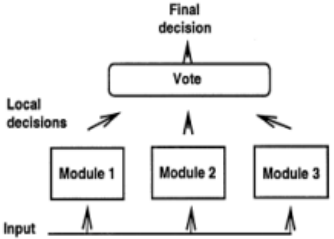
Decoupled modules



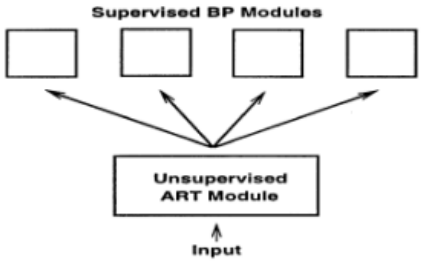
Other-o/p model



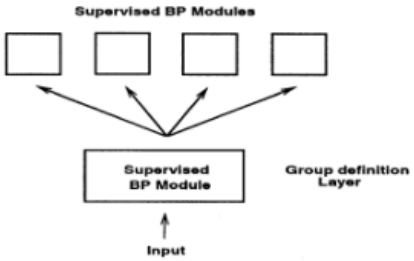
Multiple experts



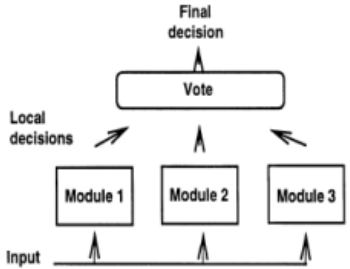
Ensemble (majority vote)



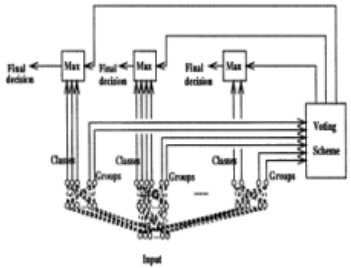
ART-BP model



Hierarchical modules



Ensemble (average vote)



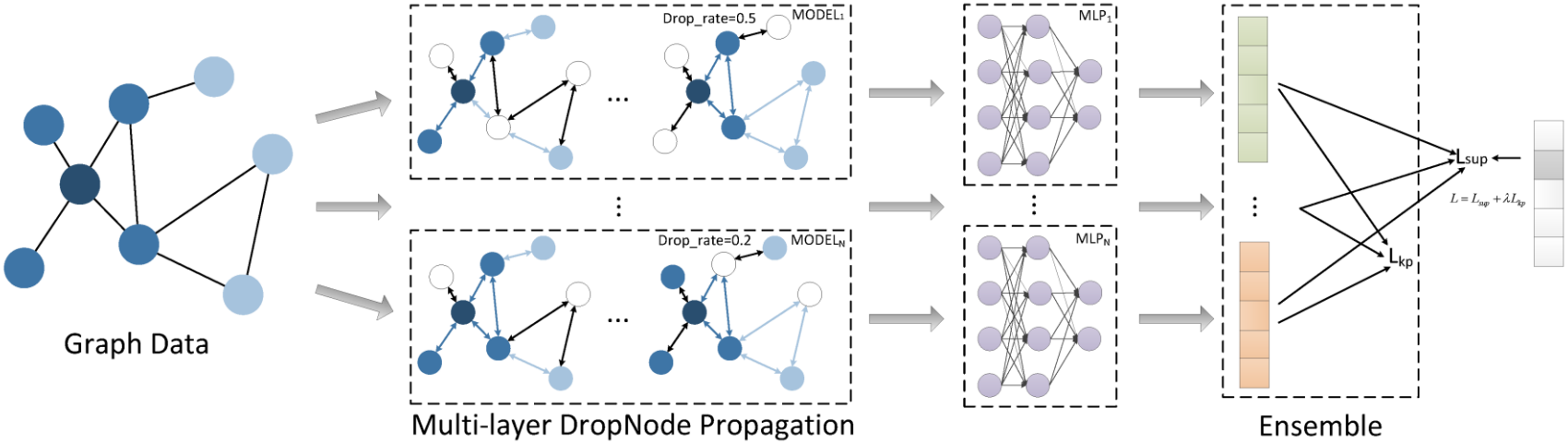
Cooperative MNN

Graph Neural Network (GNN)

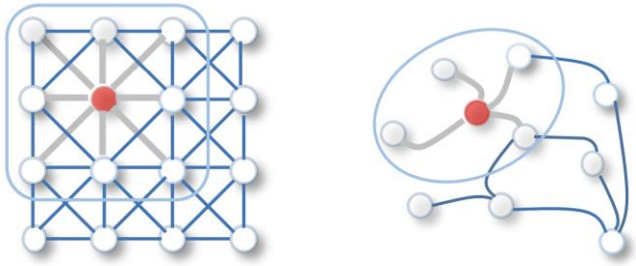
Due to their extraordinarily powerful expressive capabilities, graphs are getting significant interest in the field of machine learning. Each node is paired with an embedding. This embedding establishes the node's location in the data space. Graph Neural Networks are topologies of neural networks that operate on graphs.

A GNN architecture's primary goal is to learn an embedding that contains information about its neighborhood. We may use this embedding to tackle a variety of issues, including node labeling, node and edge prediction, and etc.

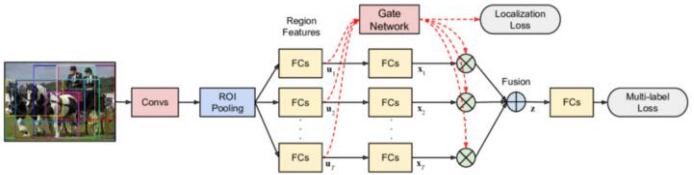
In other words, GNNs are a subclass of deep learning techniques that are specifically built to do inference on graph-based data. They are applied to graphs and can perform prediction tasks at the node, edge, and graph levels.


Graph Data
Drop_rate=0.5
MODEL1
Drop_rate=0.2
MODELN
Multi-layer DropNode Propagation
MLP1
MLPN
Lsup
Lkp
$L = L_{sup} + \lambda L_{kp}$
Ensemble

Types of Graph Neural Networks



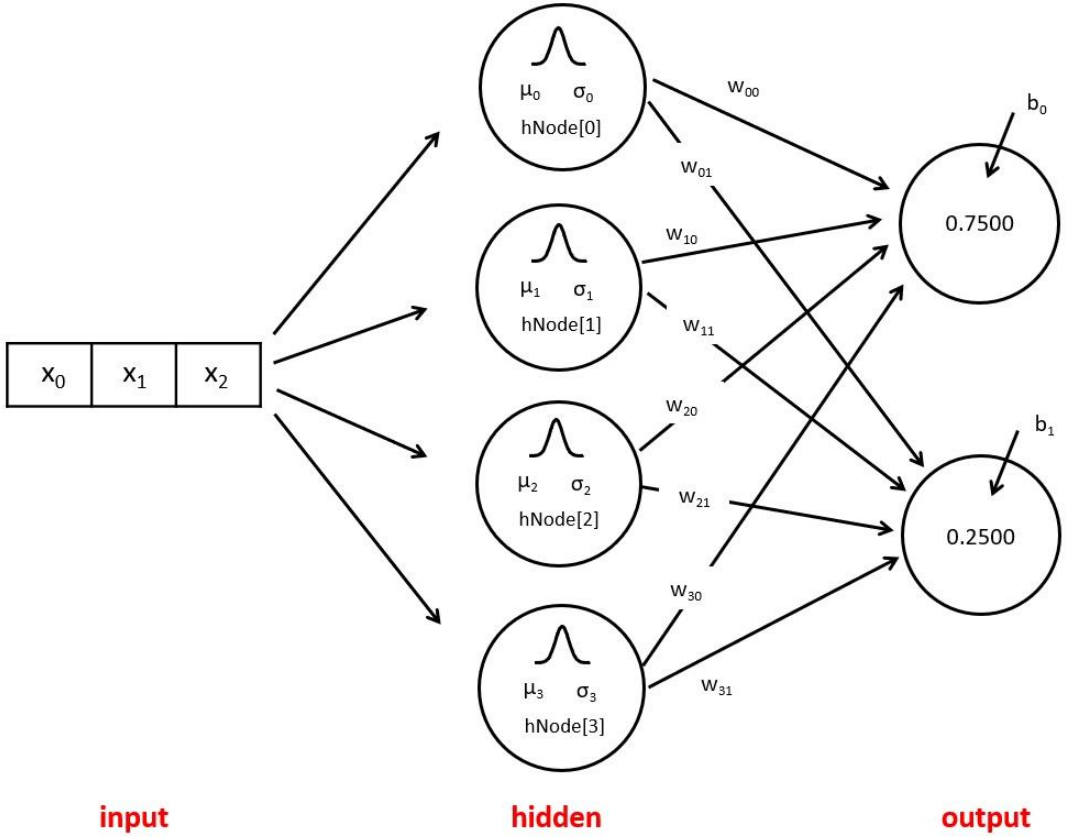
- **Recurrent Graph Neural Network**
- **Spatial Convolutional Network**
- **Spectral Convolutional Network**



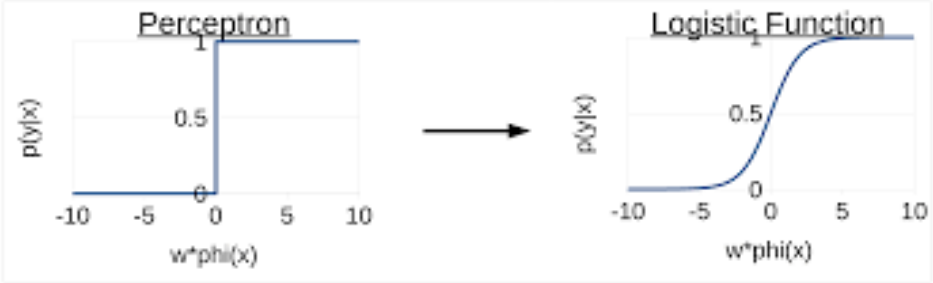
Architecture of RGNN. Layers such as pooling, ReLU and soft-max are omitted in the illustration for clarity. Example region proposals are depicted on the input image.

Radial Basis Function Neural Network

A radial basis function (RBF) neural network is a type of artificial neural network that uses radial basis functions as activation functions. It typically consists of three layers: an input layer, a hidden layer, and an output layer. The hidden layer applies a radial basis function, usually a **Gaussian function**, to the input. The output layer then linearly combines these outputs to generate the final output. RBF neural networks are highly versatile and are extensively used in pattern classification tasks, function approximation, and a variety of machine learning applications. They are especially known for their ability to handle non-linear problems effectively.


x0
x1
x2
μ0 σ0
hNode[0]
μ1 σ1
hNode[1]
μ2 σ2
hNode[2]
μ3 σ3
hNode[3]
w00
w01
w10
w11
w20
w21
w30
w31
b0
b1
0.7500
0.2500
input
hidden
output

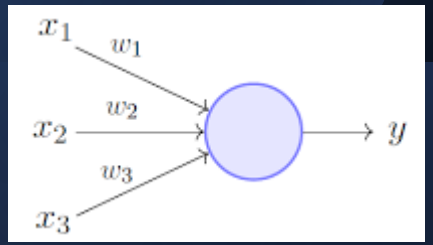
$$P(y=1|x)=\frac{e^{w\cdot\varphi(x)}}{1+e^{w\cdot\varphi(x)}}$$



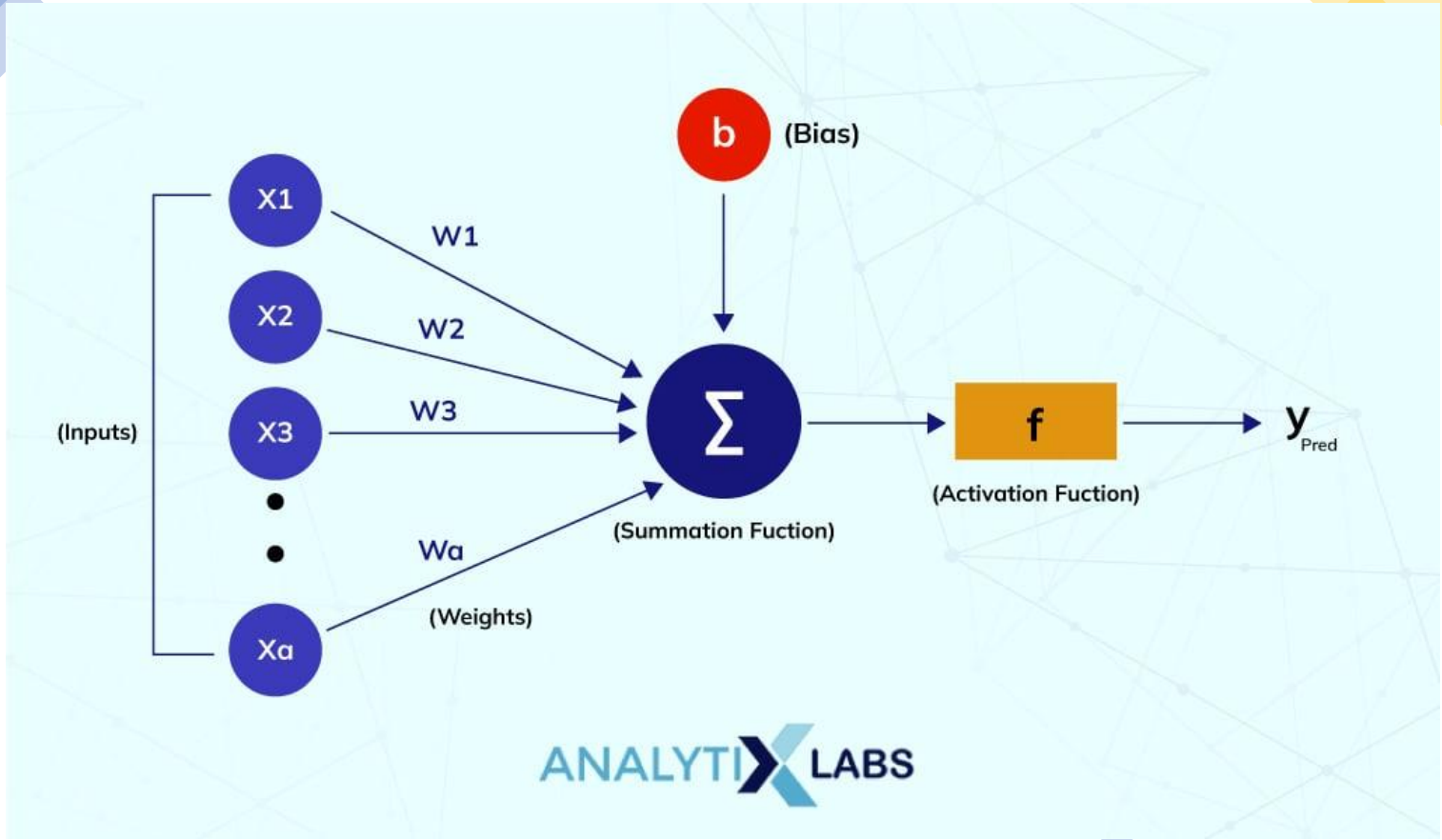
Neural networks and logistic regression

Neural networks are somewhat related to logistic regression. Basically, we can think of logistic regression as a one layer neural network . In fact, it is very common to use logistic sigmoid functions as activation functions in the hidden layer of a neural network – like the schematic above.

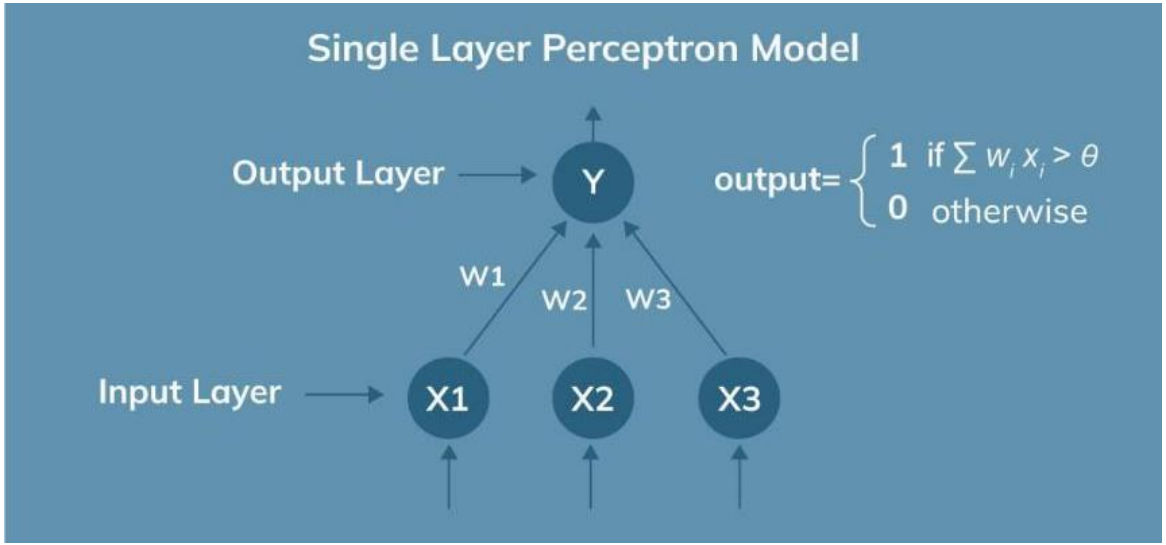
What is Perceptron?



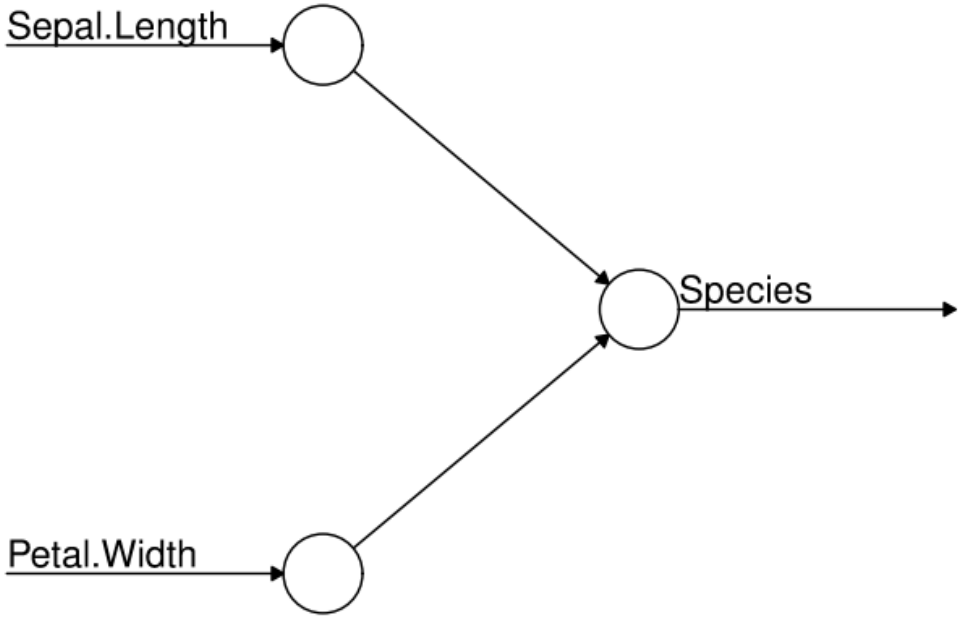
A perceptron is the smallest element of a neural network. Perceptron is a single-layer neural network used for supervised learning of various binary classifiers. A perceptron network is a group of simple logical statements that come together to create an array of complex logical statements, known as the neural network.


b
(Bias)
X1
X2
X3
Xa
(Inputs)
W1
W2
W3
Wa
(Weights)
Σ
(Summation Fuction)
f
(Activation Fuction)
y
Pred
ANALYTIX LABS

Single Layer Perceptron Model

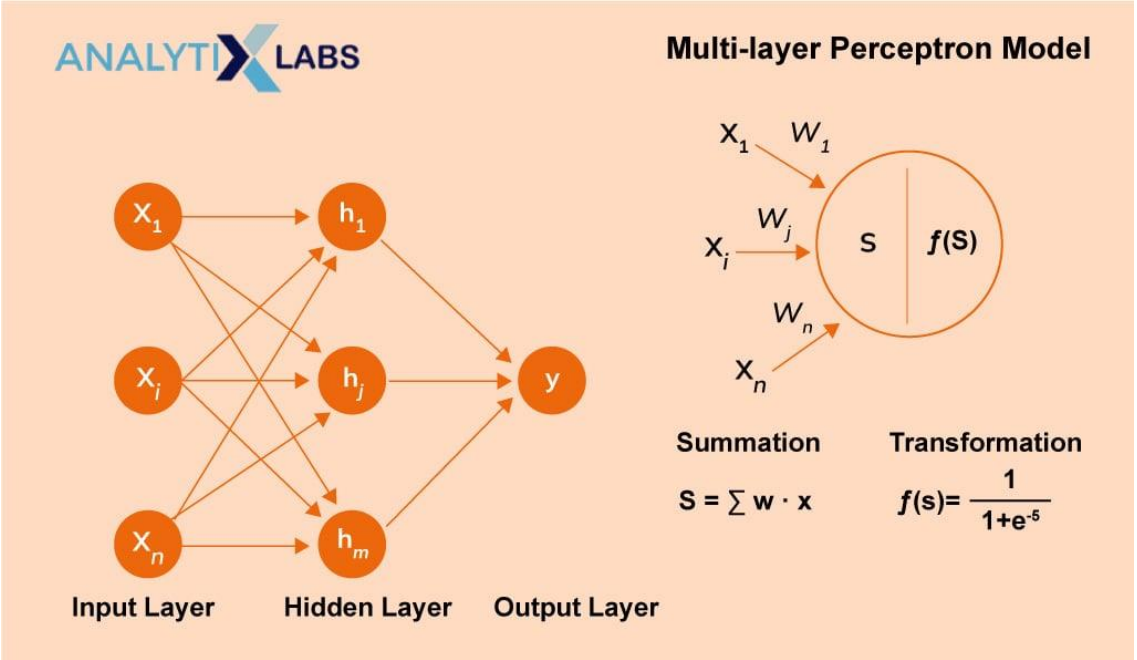


Iris (We call it Zanbagh in Persian)



$$\text{Species} = g(w_0 + w_1 \text{Sepal.Length} + w_2 \text{Petal.Width})$$

Multilayer Perceptron Model



Perceptron loss equation

In order to have an effective classification system, we must also have a loss function. A loss function is defined based on samples that were misclassified. In the following formula M is the set of misclassified samples, w is the vector of weights, φ is the feature vector, and t is the value of the target or label. The goal is to minimize the amount of perceptron loss. To minimize this loss function, a learning algorithm is needed to change the weights enough that we achieve our goal of minimizing waste.

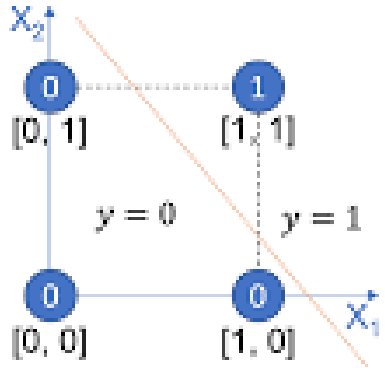
$$E^{perc}(w) = -\sum_{\varphi^n \in M} w^T(\varphi^n t^n)$$

- M is the set of misclassified samples
- w is the vector of weights
- φ is the feature vector
- t is the value of the target or label

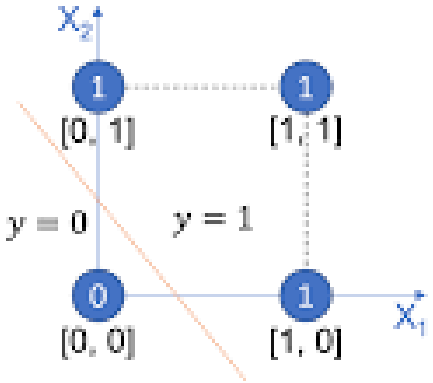
Limitations of the Perceptron Model

- The input vectors must be presented to the network one at a time or in batches so that the corrections can be made to the network based on the results of each presentation.
- The perceptron generates only a binary number (0 or 1) as an output due to the hard limit transfer function.
- It can classify linearly separable sets of inputs easily whereas non-linear input vectors cannot be classified properly.

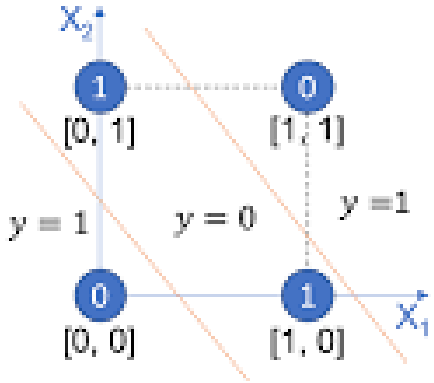
AND



OR



XOR



x_1	x_2	AND
0	0	0
0	1	0
1	0	0
1	1	1

x_1	x_2	OR
0	0	0
0	1	1
1	0	1
1	1	1

x_1	x_2	XOR
0	0	0
0	1	1
1	0	0
1	1	1

OR

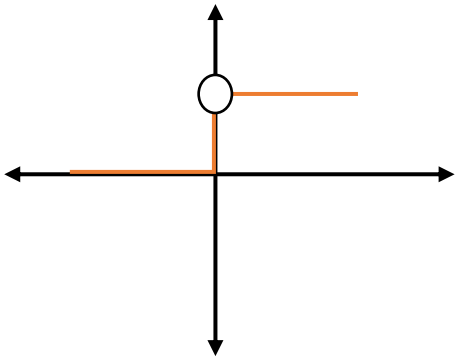
$f(x_1,x_2) = 20x_1+20x_2-10$

$f(0,0) = -10 \rightarrow$  = 0

$f(0,1) = +10 \rightarrow$  = 1

$f(1,0) = +10 \rightarrow$  = 1

$f(1,1) = +10 \rightarrow$  = 1



AND

$f(x_1,x_2) = 20x_1+20x_2-30$

$f(0,0) = -30 \rightarrow$  = 0

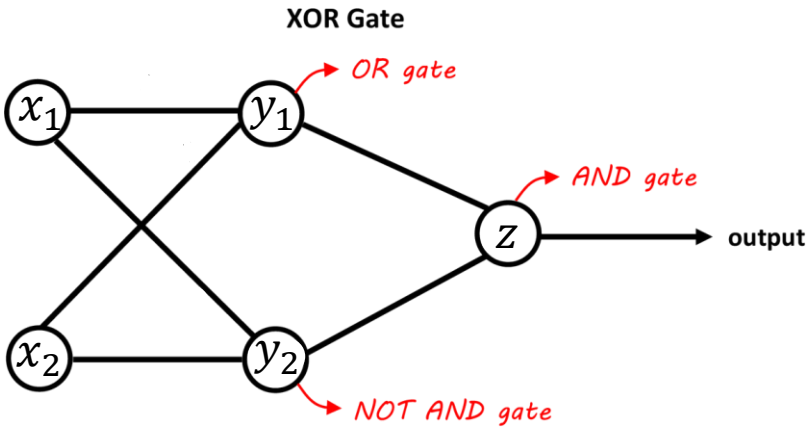
$f(0,1) = -10 \rightarrow$  = 0

$f(1,0) = -10 \rightarrow$  = 0

$f(1,1) = +10 \rightarrow$  = 1

NOT

x	not
0	1
1	0



x_1	x_2	XOR
0	0	0
0	1	1
1	0	0
1	1	1

$f(x_1,x_2) = 20x_1+20x_2-10$

y_1
0
1
1
1

$NOT(f(x_1,x_2) = 20x_1+20x_2-30)$

y_2
0
0
0
1

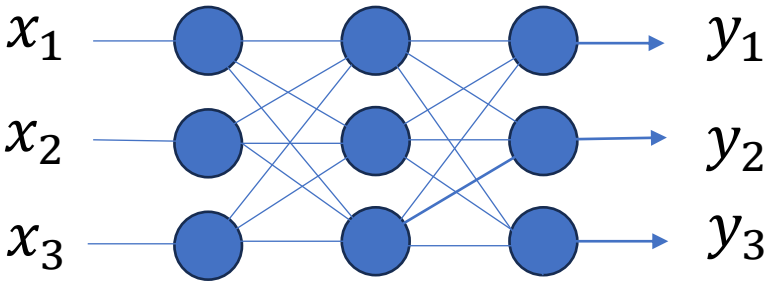
NOT →

y_2
1
1
1
0

$f(x_1,x_2) = 20x_1+20x_2-30$

y_1	y_2	AND
0	1	0
1	1	1
1	1	1
1	0	1

Example

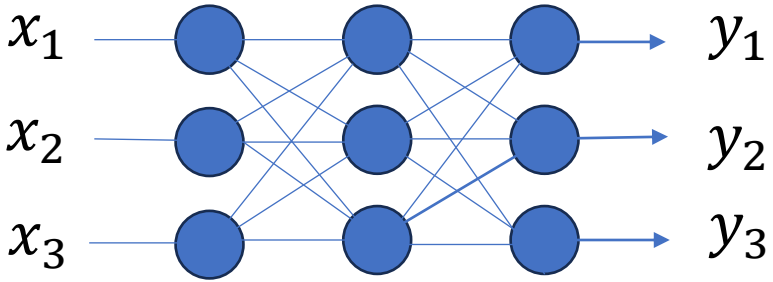


$$y = \begin{pmatrix} ? \\ ? \\ ? \end{pmatrix}$$

$$Activation\ Function = Sigmoid$$

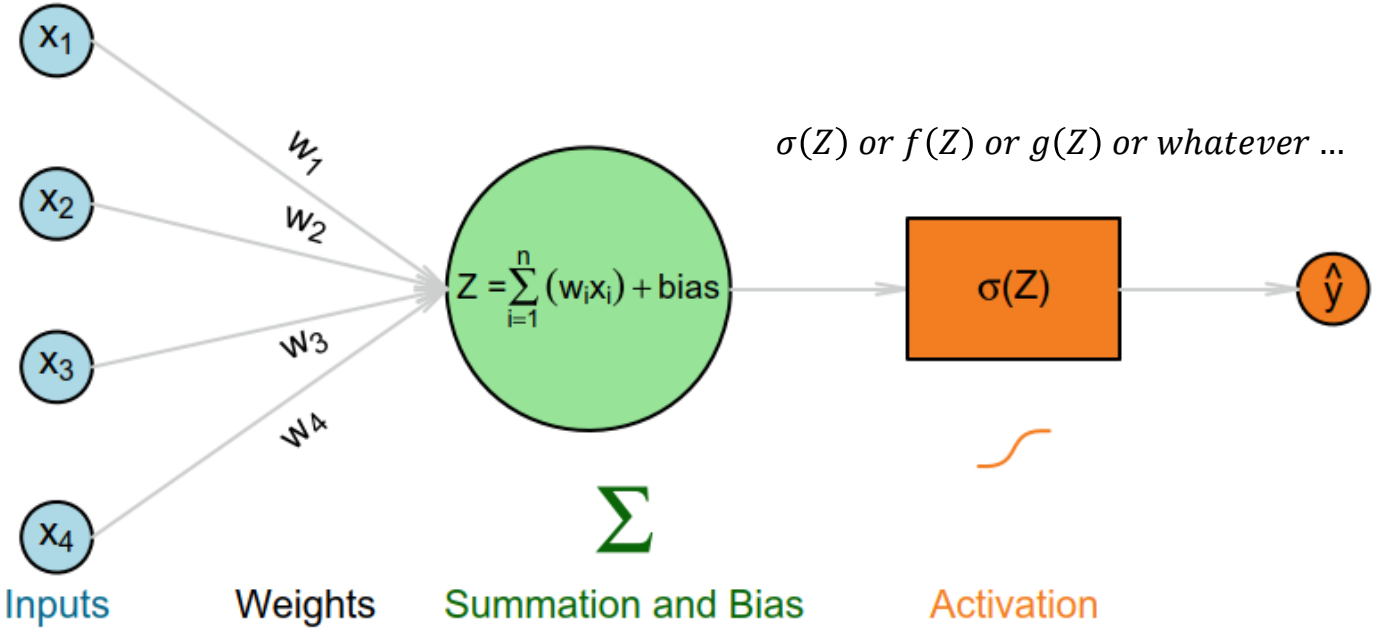
$$w_1 = \begin{pmatrix} 0.9 & 0.3 & 0.4 \\ 0.2 & 0.8 & 0.2 \\ 0.1 & 0.5 & 0.6 \end{pmatrix} \quad w_2 = \begin{pmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{pmatrix} \quad B = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \quad X = \begin{pmatrix} 0.9 \\ 0.1 \\ 0.8 \end{pmatrix}$$

$$w_1.X + b_1 = \begin{pmatrix} 0.9 & 0.3 & 0.4 \\ 0.2 & 0.8 & 0.2 \\ 0.1 & 0.5 & 0.6 \end{pmatrix} \times \begin{pmatrix} 0.9 \\ 0.1 \\ 0.8 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1.16 \\ 0.42 \\ 0.62 \end{pmatrix} \xrightarrow[y = \frac{1}{1+e^{-x}}]{sigmoid} \begin{pmatrix} \frac{1}{1+e^{-1.16}} \\ \frac{1}{1+e^{-0.42}} \\ \frac{a1}{1+e^{-0.62}} \end{pmatrix} = \begin{pmatrix} 0.761 \\ 0.603 \\ 0.650 \end{pmatrix}$$



$$w_2.z + b_2 = \begin{pmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{pmatrix} \times \begin{pmatrix} 0.761 \\ 0.603 \\ 0.650 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.975 \\ 0.888 \\ 1.254 \end{pmatrix} \xrightarrow[y = \frac{1}{1+e^{-x}}]{sigmoid} \begin{pmatrix} \frac{1}{1+e^{-0.975}} \\ \frac{1}{1+e^{-0.888}} \\ \frac{1}{1+e^{-1.254}} \end{pmatrix} = \begin{pmatrix} 0.726 \\ 0.708 \\ 0.778 \end{pmatrix}$$

Fig. is a representation of a hidden layer having as input (in this case) the input layer.



Forward propagation computes the predicted values by successively transforming the data through the layers up to the output layer, where the output of each layer is the input for the next successive layer. A loss function compares the predicted values from the output layer to the actual values and computes a distance score between these values, thereby capturing how well the network has done on a batch of input data.

linear function

$$z_m = b + \sum_m x_n w_{mn}$$

matrix multiplication
input to hidden layer

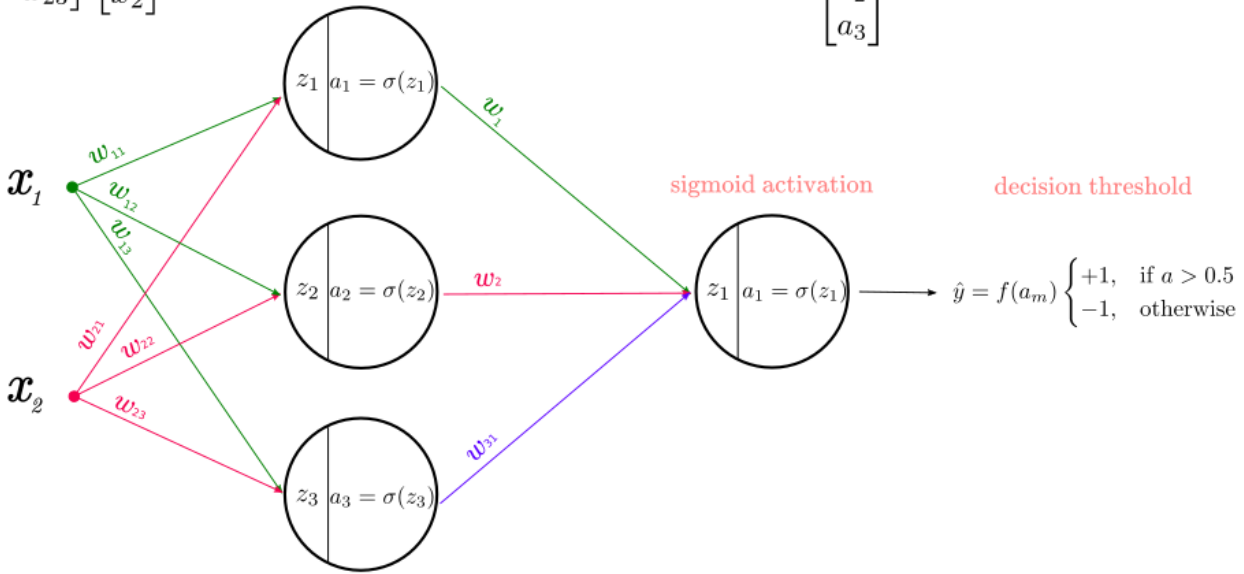
$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

sigmoid activation

$$a_m = \sigma(z_m) = \frac{1}{1 + e^{-z_m}}$$

matrix multiplication
hidden to output layer

$$\begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$



Information flow forward pass neural network with one hidden layer

Summary of Forward Propagation

The forward propagation equations for a three-layer network for a single observation can be represented as

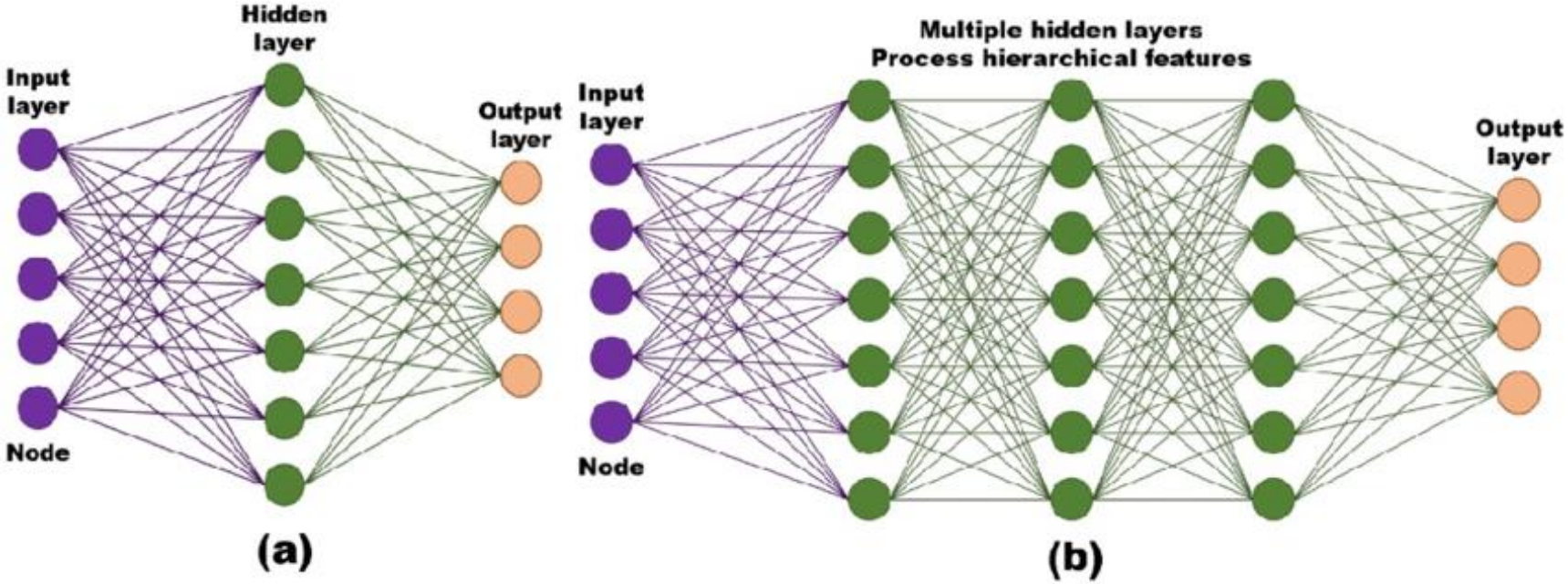
$$
\begin{aligned}
z^{[1](i)} &= w^{[1]}x^{(i)} + b^{[1](i)} \\
a^{[1](i)} &= g^{[1]}(z^{[1](i)}) \\
z^{[2](i)} &= w^{[2]}a^{[1](i)} + b^{[2](i)} \\
a^{[2](i)} &= g^{[2]}(z^{[2](i)}) \\
z^{[3](i)} &= w^{[3]}a^{[2](i)} + b^{[3](i)} \\
a^{[3](i)} &= g^{[3]}(z^{[3](i)})
\end{aligned}
$$

And the vectorized

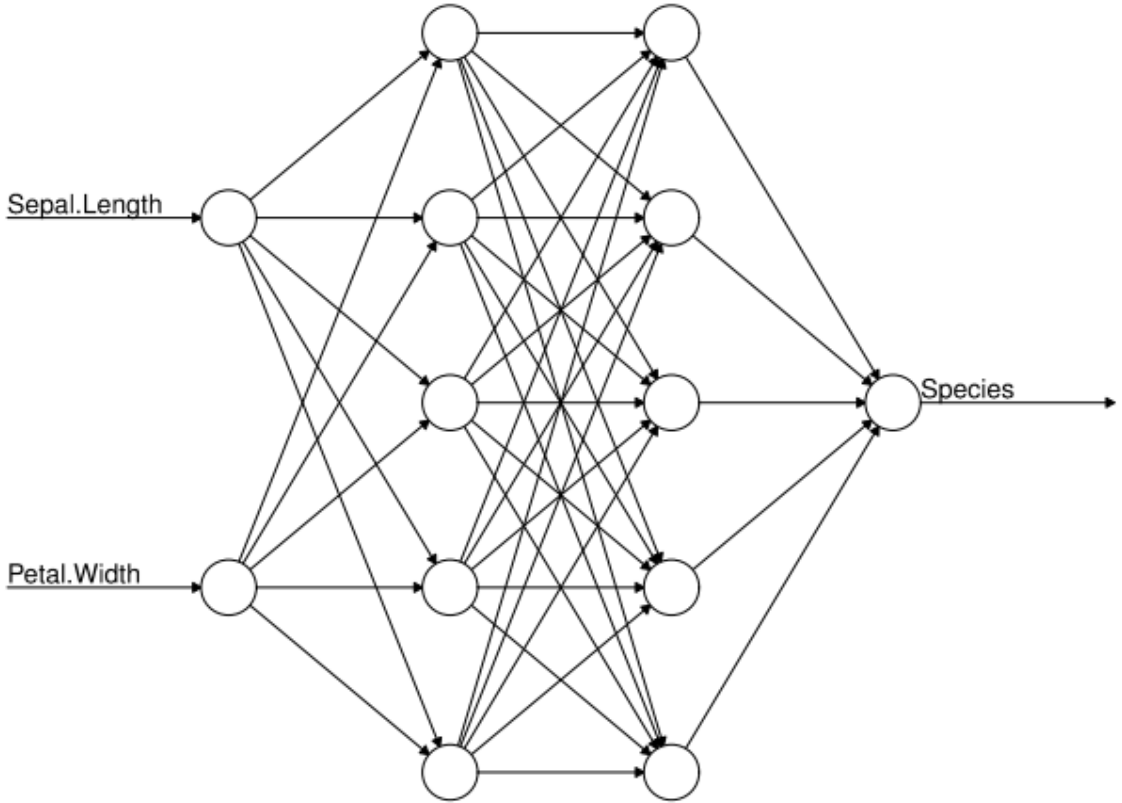
$$
\begin{aligned}
&\text{for } \ell \text{in} 1, 2, 3, \ldots, \mathrm{L} \\
Z^{[\ell]} &= W^{[\ell]}A^{[\ell]} + b^{[\ell]} \\
A^{[\ell]} &= g^{[\ell]}(Z^{[\ell]})
\end{aligned}
$$

Summary o

	Dim W	Dim b	Activation	Dim Activation
Layer 1	$(num\ features, n^{[1]})$	$(n^{[1]}, 1)$	$Z^{[1]} = W^{[1]}X + b^{[1]}$	$(n^{[1]}, num\ obs)$
Layer ℓ	$(n^{[\ell-1]}, n^{[\ell]})$	$(n^{[\ell]}, 1)$	$Z^{[\ell]} = W^{[\ell]}A^{[\ell-1]} + b^{[\ell]}$	$(n^{[\ell]}, num\ obs)$
Layer L	$(n^{[L-1]}, n^{[L]})$	$(n^{[L]}, 1)$	$Z^{[L]} = W^{[L]}A^{[L-1]} + b^{[L]}$	$(n^{[L]}, num\ obs)$


Hidden
layer
Input
layer
Output
layer
Node
(a)
Multiple hidden layers
Process hierarchical features
Input
layer
Output
layer
Node
(b)

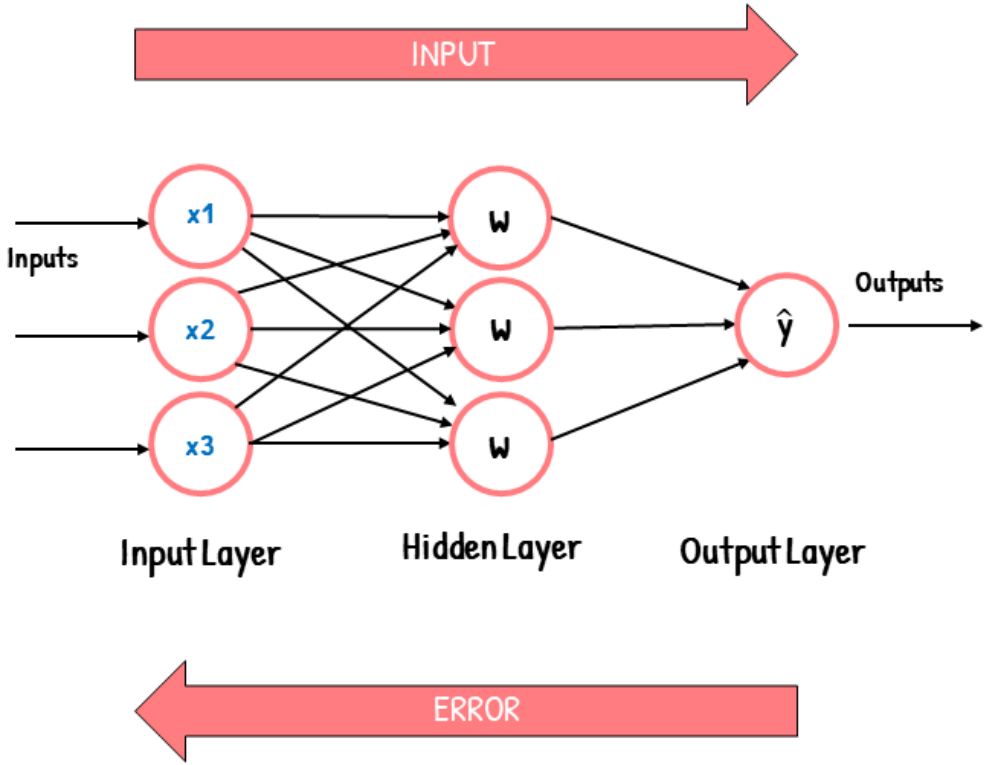
Iris



Neural Network: Depth=2 Width=5

Backpropagation

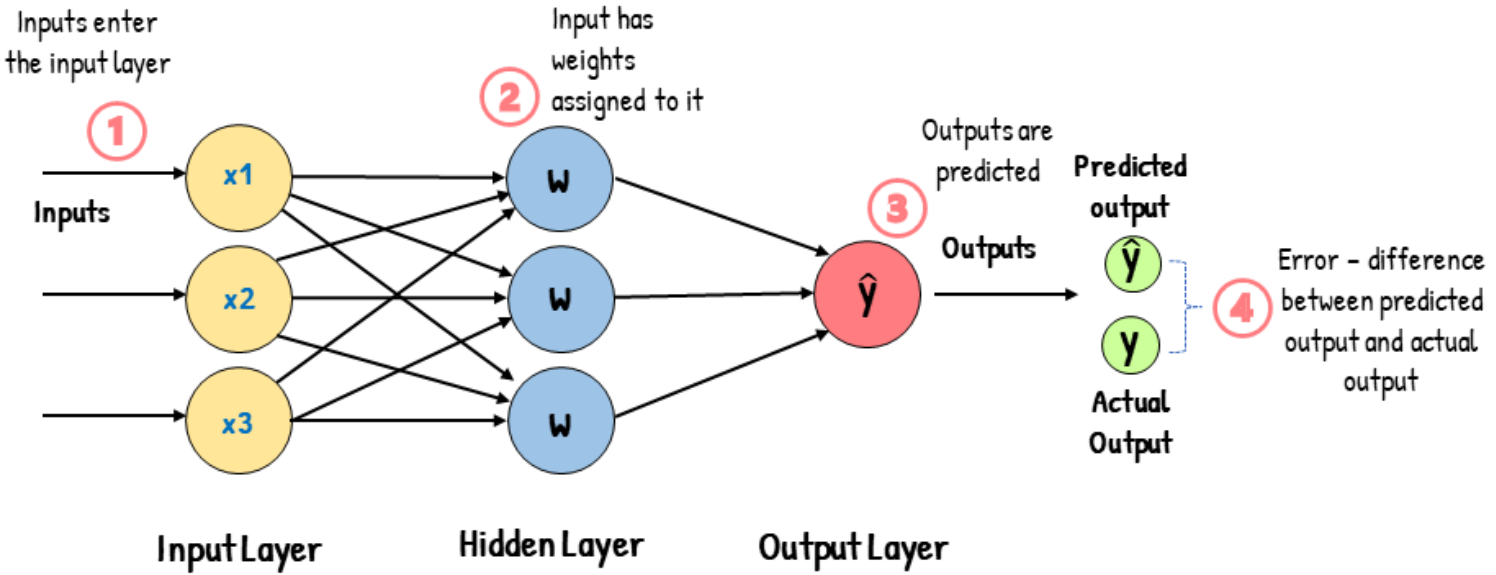
In a feedforward neural network, the input moves forward from the input layer to the output layer. Backpropagation helps improve the neural network's output. It does this by propagating the error backward from the output layer to the input layer. To understand how backpropagation works, let's first understand how a feedforward network works.



Feed Forward Networks

A feedforward network consists of an input layer, one or more hidden layers, and an output layer. The input layer receives the input into the neural network, and each input has a weight attached to it. The weights associated with each input are numerical values. These weights are an indicator of the importance of the input in predicting the final output. For example, an input associated with a large weight will have a greater influence on the output than an input associated with a small weight. When a neural network is first trained, it is first fed with input. Since the neural network isn't trained yet, we don't know which weights to use for each input. And so, each input is randomly assigned a weight. Since the weights are randomly assigned, the neural network will likely make the wrong predictions. It will give out the incorrect output.

Feed-Forward Neural Network



When the neural network gives out the incorrect output, this leads to an output error.

This error is the difference between the actual and predicted outputs.

A cost function measures this error. The cost function (J) indicates how accurately the model performs. It tells us how far-off our predicted output values are from our actual values. It is also known as the error.

Because the cost function quantifies the error, we aim to minimize the cost function.

To reduce the output error, since the weights affect the error, we will need to readjust the weights that minimize the cost function. Backpropagation allows us to readjust our weights to reduce output error. The error is propagated backward during backpropagation from the output to the input layer.

This error is then used to calculate the gradient of the cost function with respect to each weight.

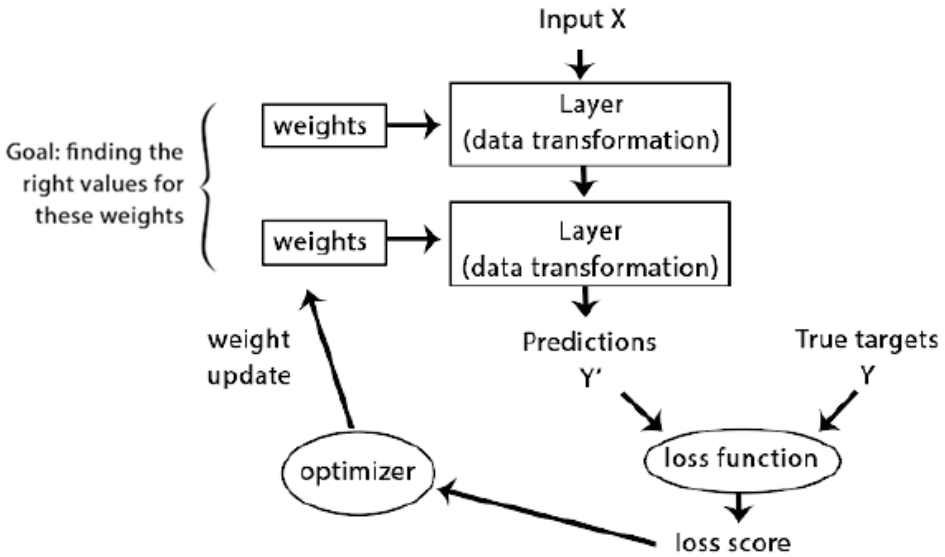
Backpropagation aims to calculate the negative gradient of the cost function. This negative gradient is what helps in adjusting of the weights. It gives us an idea of how we need to change the weights so that we can reduce the cost function.

Backpropagation uses the chain rule to calculate the gradient of the cost function.

The chain rule involves taking the derivative. This involves calculating the partial derivative of each parameter. These derivatives are calculated by differentiating one weight and treating the other(s) as a constant. As a result of doing this, we will have a gradient.

- The *input data* and corresponding *targets*
- *Layers*, which are combined into a *network* (or *model*)
- The *loss function*, which defines the feedback signal used for learning
- The *optimizer*, which determines how learning proceeds

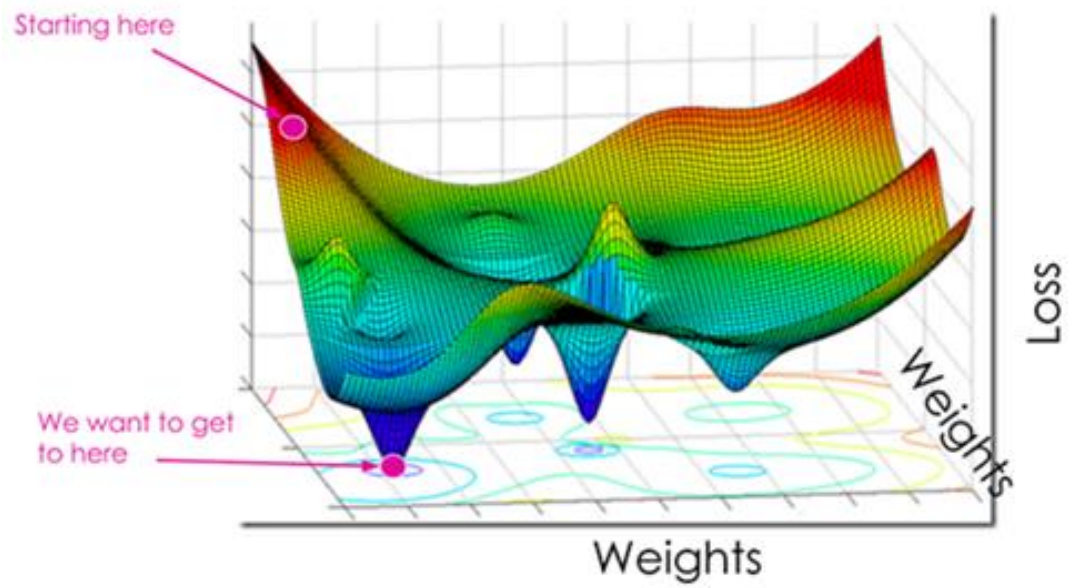
Anatomy

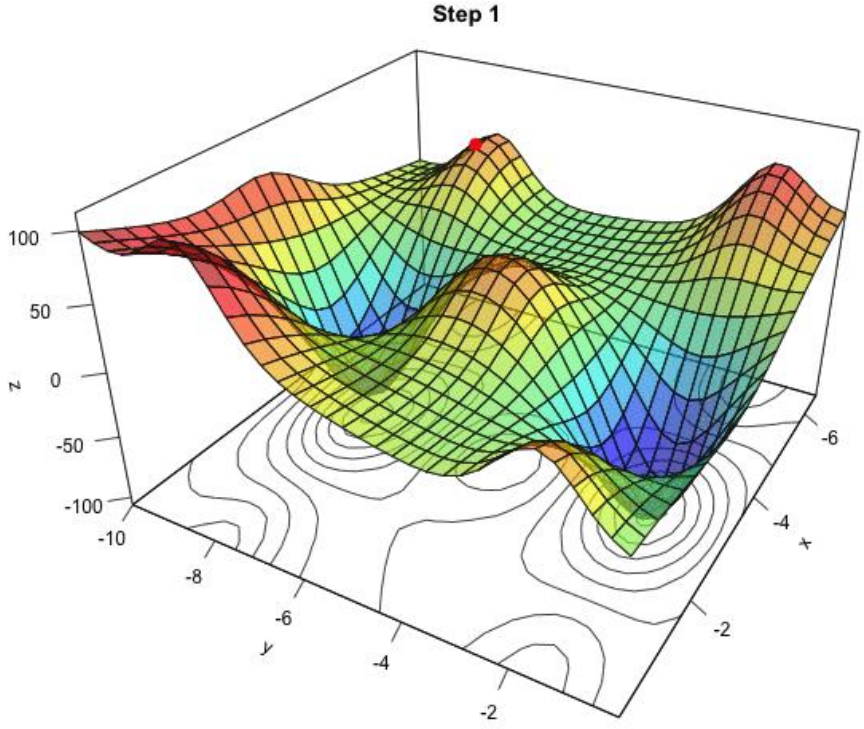


Gradient Descent

The weights are adjusted using a process called gradient descent.

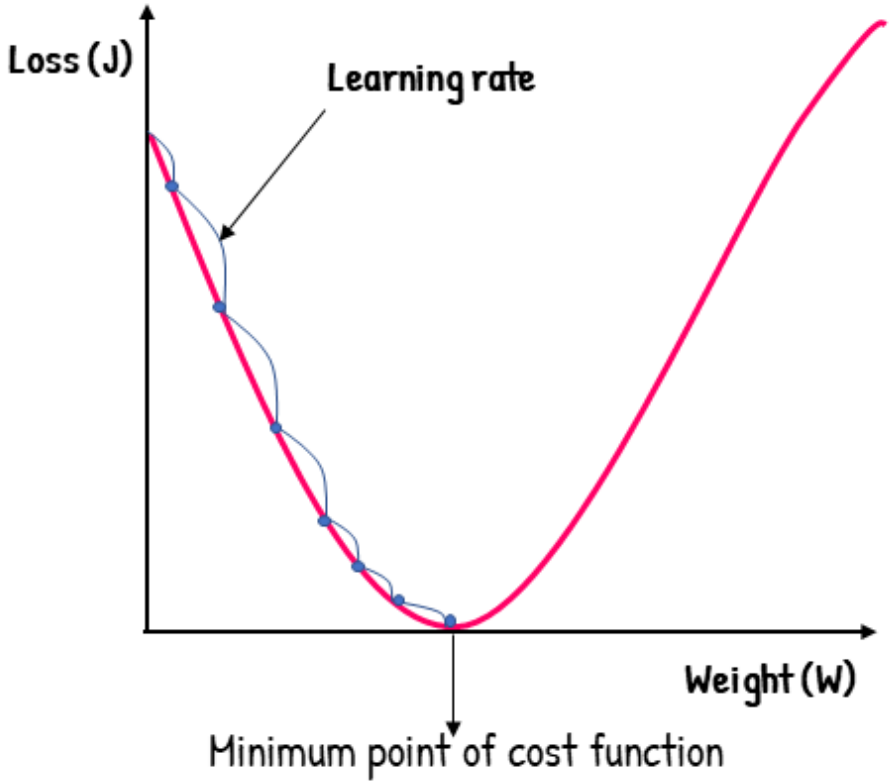
Recall gradient descent is an optimization algorithm that is used to find the weights that minimize the cost function. Minimizing the cost function means getting to the minimum point of the cost function. So, gradient descent aims to find a weight corresponding to the cost function's minimum point. To find this weight, we must navigate down the cost function until we find its minimum point.




Step 1
100
50
0
-50
-100
z
-10
-8
-6
-4
-2
y
-6
-4
-2
x

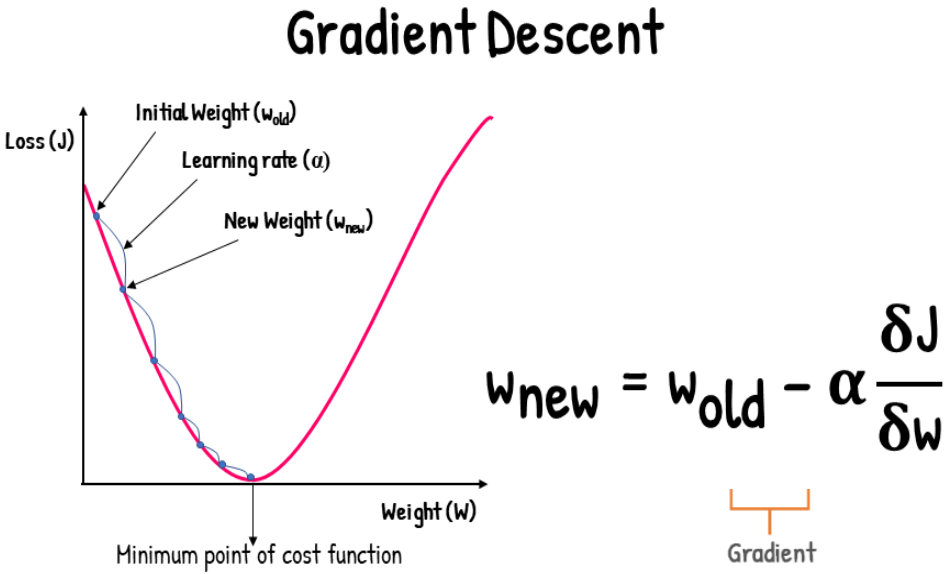
Learning Rate

Recall, the learning rate is a tuning parameter that determines the step size at each iteration of gradient descent. It determines the speed at which we move down the slope. The step size plays an important part in ensuring a balance between optimization time and accuracy. The step size is measured by a parameter alpha (α). A small α means a small step size, and a large α means a large step size. If the step sizes are too large, we could miss the minimum point completely. This can yield inaccurate results. If the step size is too small, the optimization process could take too much time. This will lead to a waste of computational power.



Descending the Cost Function

Navigating the cost function consists of adjusting the weights. As we can see, to obtain the new weight, we use the gradient, the learning rate, and an initial weight. Adjusting the weights consists of multiple iterations. We take a new step down for each iteration and calculate a new weight. Using the initial weight and the gradient and learning rate, we can determine the subsequent weights.



Backpropagation

we have $L = 3$ layers

$$
\begin{aligned}
z^{[1](i)} &= w^{[1]}x^{(i)} + b^{[1](i)} \\
a^{[1](i)} &= g^{[1]}(z^{[1](i)}) \\
z^{[2](i)} &= w^{[2]}a^{[1](i)} + b^{[2](i)} \\
a^{[2](i)} &= g^{[2]}(z^{[2](i)}) \\
z^{[3](i)} &= w^{[3]}a^{[2](i)} + b^{[3](i)} \\
a^{[3](i)} &= g^{[3]}(z^{[3](i)})
\end{aligned}
$$

The neural network model has unknown parameters, often called *weights*, and we seek values for them that make the model fit the training data well. Taking a loss function, we minimize $R(\boldsymbol{w})$, summation over the loss function. The generic approach to minimizing $R(\boldsymbol{w})$ is by gradient descent, called back-propagation in this setting. Here is back-propagation in detail for squared error loss:

Let for one layer, i.e., $l = 1$, (drop the superscript l)

$$R(\boldsymbol{w}) = \sum_{i=1}^{n} R_i = \sum_{i=1}^{n} \left(y_i - g\left(z^{(i)}\right)\right)^2, \qquad \boldsymbol{w} = (w_1, w_2, \ldots, w_L)^T$$

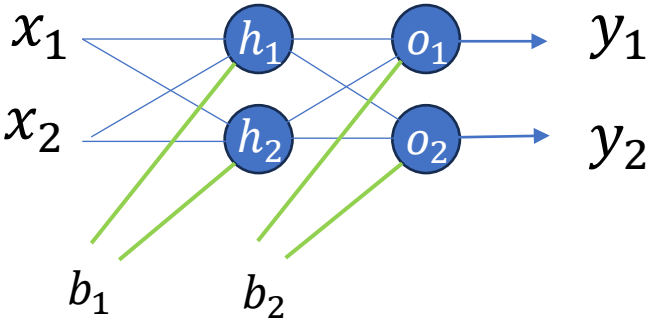
with derivatives $\frac{\partial R_i}{\partial w_j} = s_i x_j^{(i)}$, where $s_i = -2\left(y_i - g\left(z^{(i)}\right)\right) g'\left(z^{(i)}\right)$ is the error. Given these derivatives, a gradient descent update at the $(r + 1)$st iteration has the form (recall gradient descent in Chapter 1)

$$w_j^{(r+1)} = w_j^{(r)} - \alpha_r \sum_{i=1}^{n} \frac{\partial R_i}{\partial w_j^{(r)}}$$

where α_r is the learning rate.

Using this, the updates can be implemented with a two-pass algorithm. In the forward pass, the current weights are fixed and the predicted values are computed (forward propagation). In the backward pass, the errors are computed, and then back-propagated to be used to compute the gradients for the updates.

Example



$$Y = \begin{pmatrix} ? \\ ? \end{pmatrix} \qquad real\ Y = \begin{pmatrix} 0.01 \\ 0.99 \end{pmatrix}$$

$$Activation\ Function = Sigmoid$$

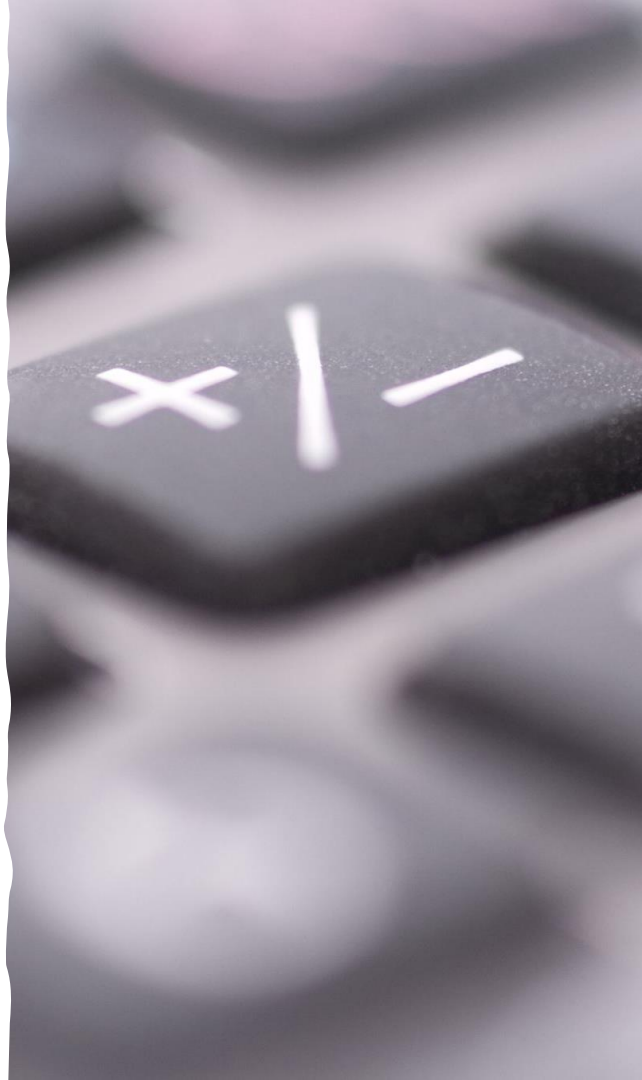
$$X = \begin{pmatrix} 0.05 \\ 0.1 \end{pmatrix} \quad w_1 = \begin{pmatrix} 0.15 & 0.2 \\ 0.25 & 0.3 \end{pmatrix} \quad w_2 = \begin{pmatrix} 0.4 & 0.45 \\ 0.55 & 0.6 \end{pmatrix} \quad b_1 = \begin{pmatrix} 0.35 \\ 0.35 \end{pmatrix} \quad b_2 = \begin{pmatrix} 0.6 \\ 0.6 \end{pmatrix}$$

$$W_1 \,.\, \mathrm{X} + b_1 = \begin{pmatrix} 0.15 & 0.2 \\ 0.25 & 0.3 \end{pmatrix} \times \begin{pmatrix} 0.05 \\ 0.1 \end{pmatrix} + \begin{pmatrix} 0.35 \\ 0.35 \end{pmatrix} = \begin{matrix} neth_1 \\ \begin{pmatrix} 0.3775 \\ 0.3925 \end{pmatrix} \\ neth_2 \end{matrix} \xrightarrow{sigmoid} y = \frac{1}{1+e^{-x}} = \begin{matrix} outh_1 \\ \begin{pmatrix} 0.5932 \\ 0.5968 \end{pmatrix} \\ outh_2 \end{matrix}$$

$$W_2 \,.\, \mathrm{Z} + b_2 = \begin{pmatrix} 0.4 & 0.45 \\ 0.55 & 0.6 \end{pmatrix} \times \begin{pmatrix} 0.5932 \\ 0.5968 \end{pmatrix} + \begin{pmatrix} 0.6 \\ 0.6 \end{pmatrix} \xrightarrow{sigmoid} y = \frac{1}{1+e^{-x}} = \begin{matrix} outo_1 \\ \begin{pmatrix} 0.7513 \\ 0.7832 \end{pmatrix} \\ outo_2 \end{matrix}$$

Example : Calculate the sum of squared error in the previous example

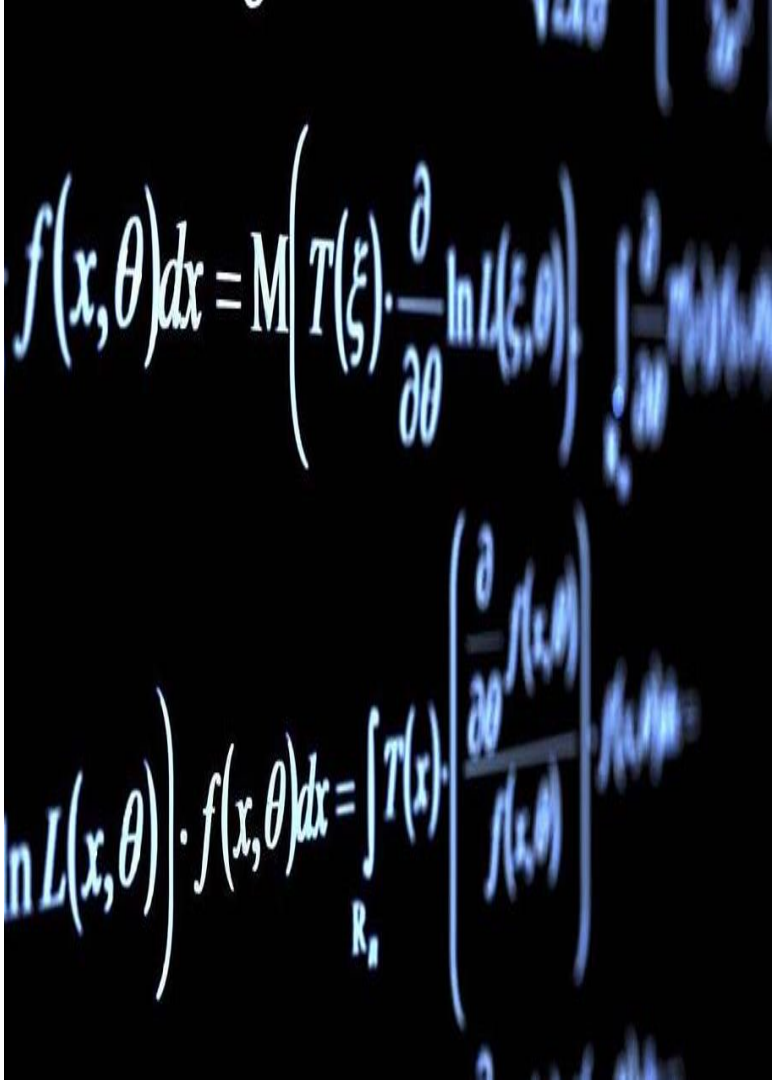
$$E = \sum_{i=1}^{2} \frac{1}{2}\left(Y_i - \hat{Y}_i\right)^2 = [\frac{1}{2}(0.01 - 0.7513)^2] + [\frac{1}{2}(0.99 - 0.7832)^2] = 0.30$$



chain rule differentiation

$$\frac{\partial h}{\partial x_i} = \frac{\partial h}{\partial u_1} \cdot \frac{\partial u_1}{\partial x_i} + \frac{\partial h}{\partial u_2} \cdot \frac{\partial u_2}{\partial x_i} + \ldots + \frac{\partial h}{\partial u_n} \cdot \frac{\partial u_n}{\partial x_i}$$

$$\frac{\partial h}{\partial x_i} = \sum_j \frac{\partial h}{\partial u_j} \cdot \frac{\partial u_j}{\partial x_i}$$



Example : Update w_5

w_5

$w_2 = \begin{pmatrix} 0.4 & 0.45 \\ 0.55 & 0.6 \end{pmatrix}$

$$New\ weight = w_5 - \eta \times \frac{\partial E_{total}}{\partial w_5}$$

chain rule

$$Updat\ W_5 = \frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} \times \frac{\partial out_{o1}}{\partial net_{o1}} \times \frac{\partial net_{o1}}{\partial w_5}$$

$$E_{total} = \frac{1}{2}(y_{o1} - out_{o1})^2 + \frac{1}{2}(y_{02} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 \times \frac{1}{2}(y_{o1} - out_{o1})^{2-1} \times -1 + 0$$

$$\frac{\partial E_{total}}{\partial out_{01}} = -(y_{o1} - out_{o1})$$

$$\frac{\partial E_{total}}{\partial out_{01}} = -(0.01 - 0.7513) = 0.7413$$

$$out_{o1} = \frac{1}{1 + e^{-net_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1} \times (1 - out_{01}) \qquad \frac{\partial out_{o1}}{\partial net_{o1}} = 0.7512 \times (1 - 0.7513) = 0.1868$$

$$net_{o1} = w_5 \times out_{h1} + w_6 \times out_{h2} + b_2 \times 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 \times out_{h1} \times w^{(1-1)}{}_5 + 0 + 0 = 0.5932$$

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} \times \frac{\partial out_{o1}}{\partial net_{o1}} \times \frac{\partial net_{o1}}{\partial w_5} = 0.7413 \times 0.1868 \times 0.5932 = 0.0821$$

$$New\ weight = w_5 - \eta \times \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 \times 0.0821 = 0.3589$$

in every epoch we use this algorithm for updating weights

epochs are fundamental parts of the training process for neural networks and other machine learning algorithms. They represent the number of times the entire dataset is passed through the algorithm. The right number of epochs is crucial for the model to learn effectively without overfitting.

Some Deep Learning Packages in R

R Package	Description
nnet	Software for feed-forward neural networks with a single hidden layer, and for multinomial log-linear models.
neuralnet	Training of neural networks using backpropagation
h2o	R scripting functionality for H2O
RSNNS	Interface to the Stuttgart Neural Network Simulator (SNNS)
tensorflow	Interface to TensorFlow
deepnet	Deep learning toolkit in R
darch	Package for Deep Architectures and Restricted Boltzmann Machines
rnn	Package to implement Recurrent Neural Networks (RRNs)
FCNN4R	Interface to the FCNN library that allows user-extensible ANNs
rcppDL	Implementation of basic machine learning methods with many layers (deep learning), including dA (Denoising Autoencoder), SdA (Stacked Denoising Autoencoder), RBM (Restricted Boltzmann machine) and DBN (Deep Belief Nets)
deepr	Package to streamline the training, fine-tuning and predicting processes for deep learning based on darch and deepnet
MXNetR	Package that brings flexible and efficient GPU computing and state-of-art deep learning to R

See: https://www.datacamp.com/community/tutorials/keras-r-deep-learning

https://github.com/M-Arashi/SASA-DS

Example: University admission

We aim to predict whether a candidate will get admitted to a job (e.g. university!) with variables such as gre (candidates GRE score), gpa (his/her grade point average in the previous college), and rank (previous college rank). This is a kaggle dataset.

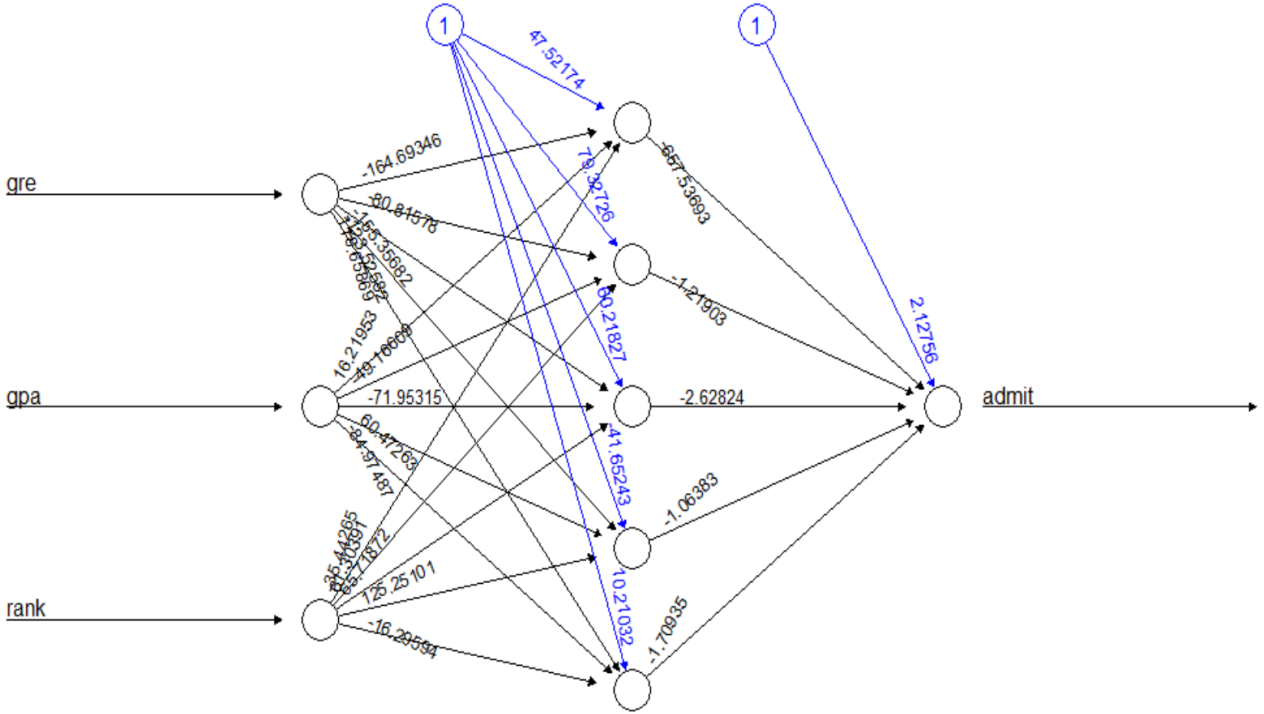
Steps:

1. Scale the data to calibrate (The common techniques to scale data are min-max normalization, Z-score normalization, median and MAD, and tan-h estimators).
2. Train-Test (sample splitting)
3. Fit the network
4. Predict (be careful since you have scaled the raw data)
5. Confusion Matrix and Misclassification error

Google

Example cont.

The model has 5 neurons in its hidden layer. The black lines show the connections with weights. The weights are calculated using the backpropagation algorithm. The blue line is displays the bias term (constant in a regression equation). We use the sigmoid activation function



Check the python code