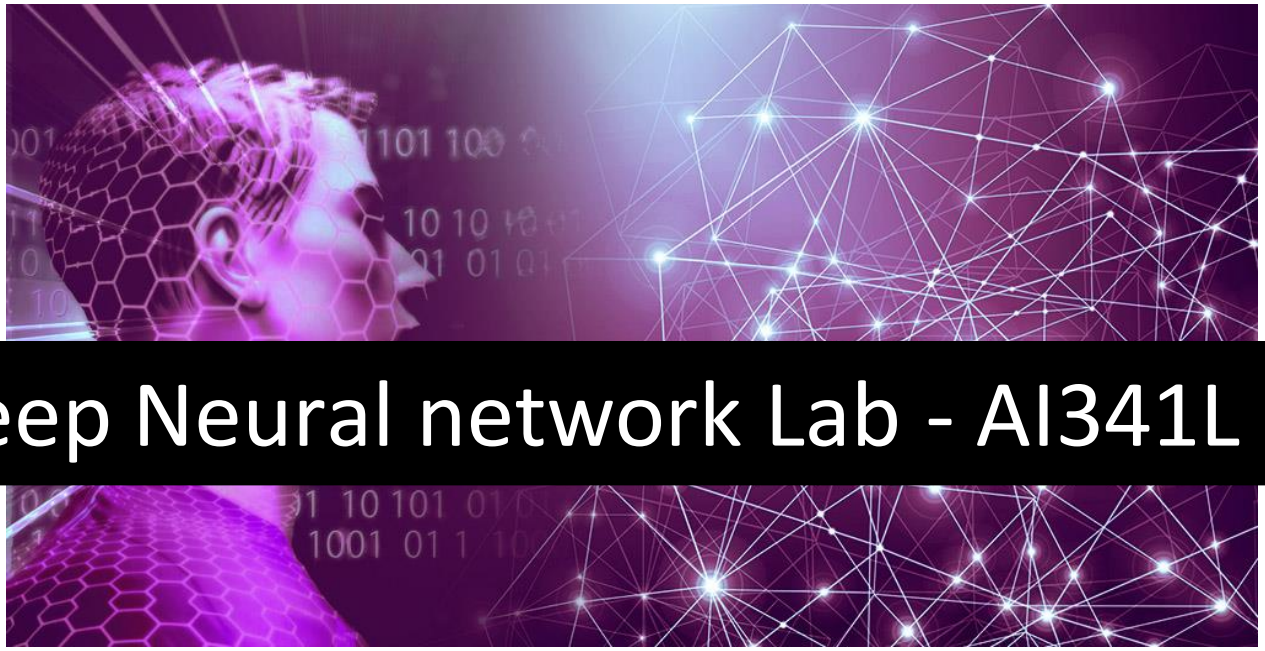


Deep Neural network Lab - AI341L

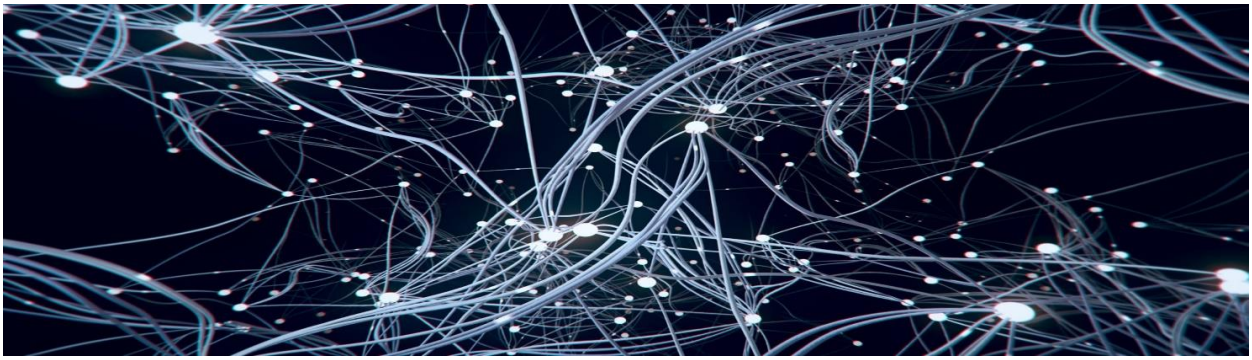


Table of Contents

By:
Hayat Ullah (Computer Engr., FCSE)

Lab. # 1: Linear Algebra, Calculus, Statistics & Probability Overview

If you are terrified at the mere mention of "math", you're probably not going to have much fun in Artificial Intelligence. But if you're willing to invest time to improve your familiarity with the principles underlying calculus, linear algebra, stats, and probability, nothing — not even math — should get in the way of you getting into AI. There are a few important mathematical concepts need to be reviewed from Linear Algebra, Calculus, Statistics & Probability, given below:

1.1 Linear Algebra:

A branch of mathematics that is concerned with mathematical structures closed under the operations of addition and scalar multiplication and that includes the theory of systems of linear equations, matrices, determinants, vector spaces, and linear transformations.

1.1.1 Vectors:

A vector in a simple term can be considered as a single-dimensional array. With respect to Python, a vector is a one-dimensional array of elements (the elements may be numbers). It occupies the elements in a similar manner as that of a Python list. A vector can be horizontal (one row and n columns) or vertical (n rows and one column).

Horizontal & vertical vectors in Python are given by the following Python code snippet:

In [1]:
```
# Horizontal Vector
import numpy as np
lst = [10,20,30,40,50]
vctr = np.array(lst)
vctr = np.array(lst)
print("Vector created from a list:")
print(vctr)
```

```
Vector created from a list:
[10 20 30 40 50]
```


By:
Hayat Ullah (Computer Engr., FCSE)

In [2]:
```
import numpy as np
lst = [[2], [4], [6], [10]]
vctr = np.array(lst)
vctr = np.array(lst)
print("Vector created from a list:")
print(vctr)
```

```
Vector created from a list:
[[ 2]
 [ 4]
 [ 6]
 [10]]
```

Addition, subtraction, multiplication (element-by-elemetn, dot product & cross product) & division (element-by-element) are given by:

In [3]:
```
# Vector Operations
v1 = np.array([10, 20, 30])
v2 = np.array([2, 10, 6])

Vsum = v1 + v2              # sum of two vects
Vdiff = v1 - v2             # Diff of two vects
Vmul = v1*v2                # Element-wise Product
Vcross = np.cross(v1, v2)   # Cross Product
Vdot = np.dot(v1, v2)       # Dot Product
Vdiv = v1/v2                # Element-wise Division

print('Sum: ', Vsum)
print('Difference: ', Vdiff)
print('Element-by-element Product: ', Vmul)
print('Cross Product:', Vcross)
print('Dot Product: ', Vdot)
print('Element-by-element Division: ', Vdiv)
```

```
Sum:  [12 30 36]
Difference:  [ 8 10 24]
Element-by-element Product:  [ 20 200 180]
Cross Product: [-180    0   60]
Dot Product:  400
Element-by-element Division:  [5. 2. 5.]
```

1.1.1.1 Norm of Vector:

The length of a vector is a nonnegative number that describes the extent of the vector in space, and is sometimes referred to as the vector's magnitude or the norm.

The norm of a vector is always a positive number, except for a vector of all zero values. It is calculated using some measure that summarizes the distance of the vector from the origin of the vector space. For example, the origin of a vector space for a vector with 3 elements is (0, 0, 0).

Usually, we use three kinds of Norms i.e. L_1, L_2 & L_∞.

By:
Hayat Ullah (Computer Engr., FCSE)

For conceptual understanding and relation between different norms, kindly watch the video on YouTube https://www.youtube.com/watch?v=NKuLYRui-NU&ab_channel=Dr.WillWood.

1.1.1.1.1 L_1 Norm or Absolute Norm:

Let $\boldsymbol{X}$ be a vector in N-dimensional space, given by $\vec{\boldsymbol{x}} = [x_1 \quad x_2 \quad x_3 \quad \dots \quad x_n]$.

L_1 Norm of $\vec{\boldsymbol{x}}$, denoted by $\left|\left|\vec{\boldsymbol{x}}\right|\right|_1$, is given by: $\left|\left|\vec{\boldsymbol{x}}\right|\right|_1 = \sum_{i=1}^{n} |x_i|$.

In Python, we may code it as:

```
In [4]:
1  X = np.array([1, 4, -10, 5])
2  X_l1 = np.abs(X).sum()
3  print('L1 Norm of X: ', X_l1)

L1 Norm of X:  20
```

1.1.1.1.2 L_2 Norm or Euclidean Norm:

Mathematically, L_2 norm is the vector magnitude, a direct length of vector from the origin. L_2 norm of $\vec{\boldsymbol{x}}$, denoted by $\left|\left|\vec{\boldsymbol{x}}\right|\right|_2$, given by: $\left|\left|\vec{\boldsymbol{x}}\right|\right|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$

In Python, we may code it as:

```
In [5]:
1  X = np.array([3, 4, 5, -10])
2  X_l2 = ((X**2).sum())**0.5
3  print('L1 Norm of X: ', X_l2)

L1 Norm of X:  12.24744871391589
```

1.1.1.1.3 L_∞ Norm or Max Norm:

Mathematically, L_∞ norm is the maximum of absolute value of . L_2 norm of $\vec{\boldsymbol{x}}$, denoted by $\left|\left|\vec{\boldsymbol{x}}\right|\right|_2$, given by: $\left|\left|\vec{\boldsymbol{x}}\right|\right|_\infty = \max_{i=1}^{n} |x_i|$.

In Python, we may code it this way:

```
In [6]:
1  X = np.array([0.8, 1, 8, -10])
2  X_inf = np.abs(X).max()
3  print('L infinity Norm of X: ', X_inf)

L infinity Norm of X:  10.0
```

1.1.1.2 Unit Vector

A vector in any direction, having one magnitude, is said to be unit vector. Unit vector in the direction of $\vec{\boldsymbol{x}}$ is denoted by $\hat{\boldsymbol{x}}$. Unit vector is evaluated by dividing it by its magnitude. Magnitude of a vector is defined by its L_2 norm, given by: $\hat{\boldsymbol{x}} = \frac{\vec{\boldsymbol{x}}}{\left|\left|\vec{\boldsymbol{x}}\right|\right|_2}$.

By:
Hayat Ullah (Computer Engr., FCSE)

Let's consider some random vector and evaluate unit vector in that direction in Python:

```
In [7]:
# Evaluating Unit Vector in the direction X
X = np.array([4, 8, -10, 5])
X_mag = ((X**2).sum())**0.5   # Magnitude (L2 Norm)
X_unit = X/X_mag              # Evaluate Unit Vector
print('Unit Vector in the direction of X: ', X_unit)
```

```
Unit Vector in the direction of X:  [ 0.27937212  0.55874424 -0.6984303   0.34921515]
```

1.1.1.3 Vector Transposition:

When rows become columns and column become rows, the matrix/vector is said be transposed and the process is called transposition. In case of vector, a horizontal vector is converted into vertical, or vice versa, after transposition. In Python we use ".T" for transposition, as given the snippet below:

```
In [8]:
# Vector Transposition
X = np.array([[4], [5], [-20], [12]])
X_trans = X.T
print('Vector befor Transpose: ')
print(X)
print('Vector after Transpose: ')
print(X_trans)
```

```
Vector befor Transpose:
[[  4]
 [  5]
 [-20]
 [ 12]]
Vector after Transpose:
[[  4   5 -20  12]]
```

Note: In Python transposition structure is defined for matrices (2D array), so, while defining vector we need to use nested square brackets.

1.1.1.4 Linear Dependence & Independence:

The two vectors $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$ are said to be linearly dependent if one vector can be written as scalar multiple of other vector i.e. $\overrightarrow{v_1} = k\overrightarrow{v_2}$ where $\overrightarrow{v_1}, \overrightarrow{v_2} \epsilon \mathbb{R}^n$, otherwise $\overrightarrow{v_2}$ and $\overrightarrow{v_2}$ linear independent.

e.g. if $\overrightarrow{v_1} = [1 \quad 2 \quad 4]$ and $\overrightarrow{v_2} = [1.5 \quad 3 \quad 6]$ are linearly dependent because $\overrightarrow{v_1} = 1.5\overrightarrow{v_2}$.

if $\overrightarrow{v_1} = [1 \quad 2 \quad 4]$ and $\overrightarrow{v_2} = [10]$ are linearly independent because $\overrightarrow{v_1} \neq k\overrightarrow{v_2}$.

Cross product of linearly dependent vectors is always a null vector i.e. a vector containing all elements equal to zero.

In Python, if we divide $\overrightarrow{v_1}$ by $\overrightarrow{v_2}$ element-wise and we get a vector all same numbers, it shows linear dependence. Consider the code snippet below:

By:
Hayat Ullah (Computer Engr., FCSE)

In [9]:
```
# Checking Linear Dependence of two vectors
v1 = np.array([1, 3, 6, 4])
v2 = np.array([2.5, 7.5, 15, 10])
print(v1/v2)
print('In this case all elements are the same.')
print('So, v1 and v2 are not linear independent.')
```

```
[0.4 0.4 0.4 0.4]
In this case all elements are the same.
So, v1 and v2 are not linear independent.
```

1.1.1.5 Orthogonal Vectors:

Two vectors are said to be orthogonal if they are mutually perpendicular. The dot product of orthogonal vectors must always be zero. Orthogonal vectors are always linearly independent.

1.1.1.6 Orthonormal Vectors:

The two vectors are said to be orthonormal if they are orthogonal as well as both vectors are unit vectors.

Practice Question: Write a Python function that takes two vectors as input and check if they are orthogonal, orthonormal, linear dependent or independent.

1.1.2 Matrices:

Matrices are rectangular arrays consisting of numbers and can be seen as 2nd-order tensors. If m and n are positive integers, i.e. $m, n \in \mathbb{N}$ then the $m \times n$ matrix contains mn numbers of elements, with m number of rows and n number of columns. Matrix is denoted by:

$$A = [a_{ij}]_{m \times n} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

Let's create matrix in numpy and play with it.

In [12]:
```
# Define a matrix in numpy
import numpy as np
M = np.array([[45,34],[67,58]])       # Define a matrix
print("Original matrix:\n", M)        # Printing the matrix
print("No. of Dimension of Matrix", M.ndim) # Dimensions
```

```
Original matrix:
 [[45 34]
 [67 58]]
No. of Dimension of Matrix 2
```


By:
Hayat Ullah (Computer Engr., FCSE)

1.1.2.1 Shape & Size of a Matrix

In this section, we will be finding shape i.e., the number of rows and columns in the given matrix and size i.e., number of elements in the matrix of a given matrix.

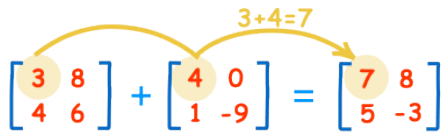
Consider the snippet below:

In [20]:
```
M = np.array([[45,34,75],[67,58,89]])
print('Original Matrix: \n', M)
print("The shape of Matrix is ", M.shape)
print("Matrix size is: " , M.size)
```

```
Original Matrix:
 [[45 34 75]
 [67 58 89]]
The shape of Matrix is  (2, 3)
Matrix size is:  6
```

1.1.2.2 Addition/Subtraction of two matrices:

In addition/subtraction, corresponding elements are added/subtracted.



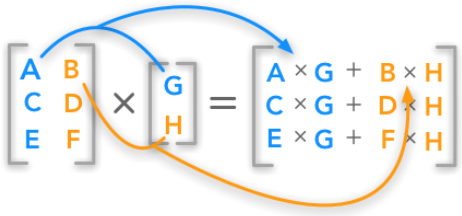
In [14]:
```
M1 = np.array([[1,2], [4,6]])
M2 = np.array([[1,1], [2,3]])
print('First Matrix')
print(M1)
print('Second Matrix')
print(M2)
print('Sum of Matrices:')
print(M1+M2)
print('Difference of Matrices:')
print(M1-M2)
```

```
First Matrix
[[1 2]
 [4 6]]
Second Matrix
[[1 1]
 [2 3]]
Sum of Matrices:
[[2 3]
 [6 9]]
Difference of Matrices:
[[0 1]
 [2 3]]
```


By:
Hayat Ullah (Computer Engr., FCSE)

1.1.2.3 Matrix Multiplication:

A matrix of shape (m x n) and B matrix of shape (n x p) multiplied gives C of shape (m x p). Remember while multiplying the matrices is that the number of columns in the first matrix is the same as the number of rows in the second matrix to perform multiplication without error.



numpy.dot() method is used for evaluating matrix multiplication, given by:

In [15]:
```
# Matrix Multiplication
M1 = np.array([[1,2], [4,6]])
M2 = np.array([[1,1], [2,3]])
M_prod = np.dot(M1, M2)
print('Matrix Multiplication:')
print(M_prod)
```

```
Matrix Multiplication:
[[ 5  7]
 [16 22]]
```

1.1.2.4 Determinant of a matrix:

The determinant is a special number that can be calculated from a matrix. The matrix has to be square (same number of rows and columns). Determinant of a matrix help to find inverse of matrix, and hence helps in solving system of linear equations.

The method numpy.linalg.det() is used to evaluate determinant of a matrix, as given in the snippet below:

By:
Hayat Ullah (Computer Engr., FCSE)

In [22]:
```
# Determinant of a matrix
M = np.array([[1,2], [4,6]])
det = np.linalg.det(M)
print('Determinant of \n\n', M, '\n\n is ', det)
```

```
Determinant of

 [[1 2]
 [4 6]]

 is  -2.0
```

1.1.2.5 Identity, Null & Ones Matrix:

Identity is a matrix having all the diagonal element equal to one and rest of the elements equal to zero. Multiplication of identity matrix by any matrix will result the same matrix. Null matrix has all element equal to zero and ones matrix has all elements equal to 1. Addition of Null matrix to any matrix will result the same matrix.

In numpy, they are created as:

In [24]:
```
# Identity, Null and ones matrix
print('3-by-3 Identity matrix:\n', np.eye(3))
print('\n4-by-2 Null Matrix:\n', np.zeros((4,2)))
print('\n3-by-5 Ones Matrix:\n', np.ones((3,5)))
```

```
3-by-3 Identity matrix:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

4-by-2 Null Matrix:
 [[0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]]

3-by-5 Ones Matrix:
 [[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

1.1.2.6 Matrix Transpose:

When row and column are interchanged, the matrix is said to be transposed. In Python, ".T" is used for matrix transposition, as shown:

By:
Hayat Ullah (Computer Engr., FCSE)

In [25]:
```
# Transpose of a matrix
M = np.array([[1,2], [4,6], [8,9]])
print('Original Matrix:\n', M)
print('Transposed:\n', M.T)
```

```
Original Matrix:
 [[1 2]
 [4 6]
 [8 9]]
Transposed:
 [[1 4 8]
 [2 6 9]]
```

1.1.2.7 Matrix Inverse:

For each matrix A having det(A) ≠ 0, there exists A^{-1} which results an identity matrix when multiplied by A. In this case A^{-1} is called inverse of A or vice versa. Consider the snippet below:

In [53]:
```
# Inverse of a matrix
M = np.array([[1,0,3], [4,6,8], [8,9,10]])
print('Original Matrix:\n', M)
M_inv = np.linalg.inv(M)
print('Inversed:\n', M_inv)
```

```
Original Matrix:
 [[ 1  0  3]
 [ 4  6  8]
 [ 8  9 10]]
Inversed:
 [[ 0.25       -0.5625      0.375     ]
 [-0.5         0.29166667 -0.08333333]
 [ 0.25        0.1875     -0.125     ]]
```

1.1.2.8 Eigenvalues & Eigenvectors:

In Linear Algebra, a scalar λ is called an eigenvalue of matrix A if there exists a column vector $\vec{v}$ such that

$$A\vec{v} = \lambda\vec{v}$$

and $\vec{v}$ is non-zero. Any vector satisfying the above relation is known as eigenvector of the matrix A corresponding to the eigenvalue λ. Consider the code snippet below.

By:
Hayat Ullah (Computer Engr., FCSE)

In [73]:
```
# Eigen Value & Eigen Vectors
A = np.array([[3, 1], [2, 2]])
Eig_val, Eig_vect = np.linalg.eig(A)

print('Eigen Values:\n', Eig_val)
print('Eigen Vectors:\n', Eig_vect)
```

```
Eigen Values:
 [4. 1.]
Eigen Vectors:
 [[ 0.70710678 -0.4472136 ]
 [ 0.70710678  0.89442719]]
```

The output of the above code show that we have two eigenvectors $\overrightarrow{v_1} = \begin{bmatrix} 0.7071 \\ 0.7071 \end{bmatrix}$ and $\overrightarrow{v_2} = \begin{bmatrix} -0.4472 \\ 0.8944 \end{bmatrix}$ and their corresponding eigenvalues are $\lambda_1 = 4$ and $\lambda_2 = 1$, respectively.

1.1.3 Tensors:

In deep learning it is common to see a lot of discussion around tensors as the cornerstone data structure. Tensor even appears in name of Google's flagship machine learning library: "TensorFlow". Tensors are a type of data structure used in linear algebra, and like vectors and matrices, you can calculate arithmetic operations with tensors. In this tutorial, you will discover what tensors are and how to manipulate them in Python with NumPy.

In the general case, an array of numbers arranged on a regular grid with a variable number of axes is known as a tensor.

Tensor having same shape can be added, subtracted, multiplied and divided element-by-element. Consider the snippet below:

In [55]:
```
# tensor addition
from numpy import array
A = array([
  [[1,2,3],    [4,5,6],    [7,8,9]],
  [[11,12,13], [14,15,16], [17,18,19]],
  [[21,22,23], [24,25,26], [27,28,29]],
  ])
B = array([
  [[1,2,3],    [4,5,6],    [7,8,9]],
  [[11,12,13], [14,15,16], [17,18,19]],
  [[21,22,23], [24,25,26], [27,28,29]],
  ])

print('Addition:\n', A+B)
print('Subtraction:\n', A-B)
print('Multiplication:\n',A*B)
print('Division:\n', A/B)
```

Output:

By:
Hayat Ullah (Computer Engr., FCSE)

```
Addition:
[[[ 2  4  6]
  [ 8 10 12]
  [14 16 18]]

 [[22 24 26]
  [28 30 32]
  [34 36 38]]

 [[42 44 46]
  [48 50 52]
  [54 56 58]]]

Subtraction:
[[[0 0 0]
  [0 0 0]
  [0 0 0]]

 [[0 0 0]
  [0 0 0]
  [0 0 0]]

 [[0 0 0]
  [0 0 0]
  [0 0 0]]]

Multiplication:
[[[  1   4   9]
  [ 16  25  36]
  [ 49  64  81]]

 [[121 144 169]
  [196 225 256]
  [289 324 361]]

 [[441 484 529]
  [576 625 676]
  [729 784 841]]]

Division:
[[[1. 1. 1.]
  [1. 1. 1.]
  [1. 1. 1.]]

 [[1. 1. 1.]
  [1. 1. 1.]
  [1. 1. 1.]]

 [[1. 1. 1.]
  [1. 1. 1.]
  [1. 1. 1.]]]
```

1.2 Calculus

Calculus, originally called infinitesimal calculus or "the calculus of infinitesimals", is the mathematical study of continuous change, in the same way that geometry is the study of shape, and algebra is the study of generalizations of arithmetic operations

1.2.1 Derivative of a Function:

Derivatives are the fundamental tools of Calculus. It is very useful for optimizing a loss function with gradient descent in Machine Learning is possible only because of derivatives.

Suppose we have a function y = f(x) which is dependent on x then the derivation of this function means the rate at which the value y of the function changes with the change in x.

By definition, derivative $f'(x)$ of a function $f(x)$ w.r.t. x is defined as:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

1.2.1.1 Rule of differentiation:

i) Power Rule:

Generally: $f'(x^n) = nx^{n-1}$. Using SymPy library in Python for verify this for n=5.

In [89]:
```
# Power Rule
import sympy as sp
#Power rule
x = sp.Symbol('x')
f = x**5
f.diff(x)
```

Out[89]: `5*x**4`

ii) Product Rule:

Let u(x) and v(x) be differentiable functions. Then the product of the functions u(x)v(x) is also differentiable, given by:–

Let $u = e^x$ and $v = \cos(x)$. Consider code snippet below:

By:
Hayat Ullah (Computer Engr., FCSE)

```
In [90]:
1 # Product Rule
2 x = sp.Symbol('x')
3 f = sp.exp(x)*sp.cos(x)
4 f.diff(x)

Out[90]: -exp(x)*sin(x) + exp(x)*cos(x)
```

iii) Chain Rule:

It is very important

If $h(x) = f(g(x))$, then $h'(x) = f'^{(g(x))}g'(x)$.

Let $g = x^2$ and $f(x) = \cos(x)$, then $f(g(x)) = \cos(x^2)$, evaluating using Python:

```
In [91]:
1 # Chain Rule:
2 x = sp.Symbol('x')
3 f = sp.cos(x**2)
4 f.diff(x)

Out[91]: -2*x*sin(x**2)
```

iv) Quotient Rule:

According to this rule, if $h(x) = \frac{u(x)}{v(x)}$ then $h'(x) = \frac{vu' - uv'}{v^2}$.

Let $u = 1$, and $v = cos(x)$.

Evaluating $\left(\frac{u}{v}\right)' = \left(\frac{1}{\cos(x)}\right)' = \frac{\cos(x)\times 0 - (1)(-\sin(x))}{\cos^2 x} = \frac{\sin(x)}{\cos^2(x)} = \tan(x)\sec(x)$

```
In [92]:
1 # Quotient Rule
2 x = sp.Symbol('x')
3 u = 1
4 v = sp.cos(x)
5 (u/v).diff(x)

Out[92]: sin(x)/cos(x)**2
```

1.2.1.2 Partial Derivatives:

A partial derivative is defined as a derivative in which some variables are kept constant and the derivative of a function with respect to the other variable can be determined.

Let $f(x,y) = 2x^2 + y^3 + 3xy$ where x and y are two independent variables, $\frac{\partial f(x,y)}{\partial x}$ (or simply $f_x(x,y)$) and $\frac{\partial f(x,y)}{\partial y}$ (or simply $f_y(x,y)$) denotes partial derivative of $f(x,y)$ w.r.t. x and y.

In Python, we can do it as:

By:
Hayat Ullah (Computer Engr., FCSE)

In [131]:
```
# Partial Derivatives:
x, y = sp.Symbol('x'), sp.Symbol('y')
f = 2*x**2+y**3+3*x*y
fx = f.diff(x)
fy = f.diff(y)
print(f'Original function: {f}')
print(f'Partial derivative w.r.t. x: {fx}')
print(f'Partial derivative w.r.t. y: {fy}')
```

```
Original function: 2*x**2 + 3*x*y + y**3
Partial derivative w.r.t. x: 4*x + 3*y
Partial derivative w.r.t. y: 3*x + 3*y**2
```

1.2.1.3 Numerical Derivative:

Sometimes, it's hard to evaluate analytical derivative, so we find approximate value of its which is called numerical derivative.

Since

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Which we put $h \ll x$, the returned value will be numerical derivation of f(x). Let $f(x) = x^2$, so $f'^{(x)} = 2x$ and analytically, $f'(10) = 20$.

Now evaluating this numerically in Python as: (let $h = 1 \times 10^{-5}$)

In [133]:
```
# Numerical Derivative:
f = lambda x: x**2
h = 1e-5  # step size = 0.00001
df = lambda x: (f(x+h)-f(x))/h
x = 10
print(f'Numerical Derivative at x={x} is {df(x)}')
```

```
Numerical Derivative at x=10 is 20.00000999942131
```

1.3 Statistics:

Statistics is a branch of mathematics that deals with collecting, analyzing, interpreting, and visualizing empirical data.

Descriptive statistics and inferential statistics are the two major areas of statistics. Descriptive statistics are for describing the properties of sample and population data (what has happened). Inferential statistics use those properties to test hypotheses, reach conclusions, and make predictions (what can you expect).

By:
Hayat Ullah (Computer Engr., FCSE)

1.3.1 Mean:

The arithmetic mean is the average of all the data points.

If there are n number of observations and xi is the ith observation, then mean is:

$$\bar{x} = \frac{1}{n}\sum_{i}^{n} x_i$$

1.3.2 Median:

Median is the middle value that divides the data into two equal parts once it sorts the data in ascending order. If the total number of data points (n) is odd, the median is the value at position (n+1)/2.

When the total number of observations (n) is even, the median is the average value of observations at n/2 and (n+2)/2 positions.

e.g. if we have $\vec{x} = [1\ 2\ 3\ 2\ 1\ 2\ 3\ 2\ 1]$, then for median, we have to sort it. After sorting we have $[1\ 1\ 1\ 2\ 2\ 2\ 2\ 3\ 3]$, so the middle value is 2 which median.

1.3.3 Mode:

Mode is the most repeating value in a vector.

The numpy.mean(), numpy.median() and scipy.stats.mode() functions in Python can help you find the median value of a column.

Consider the snippet below:

In [113]:
```
# Mean median & mode of data
# Import Library of statistics
from scipy import stats
x = np.array([155, 157, 160, 159, 162,
              160, 161, 165, 160, 158])
# Mean median & mode can be evaluated by
#      numpy / stats functions like:
print('mean={}, Median = {}, Mode = {}'.format(np.mean(x),
                                               np.median(x),
                                               stats.mode(x)[0][0]))
```

```
mean=159.7, Median = 160.0, Mode = 160
```

1.3.4 Variance & Standard Deviation:

Variance is used to measure the variability in the data from the mean. It is denoted by σ^2.

Mathematically,

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

By:
Hayat Ullah (Computer Engr., FCSE)

Where x_i is the i^{th} sample of data vector $\vec{x}$ and μ is mean of data.

Non-negative square-root of variance is called standard deviation and denoted by σ.

So,

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$$

In numpy, we can evaluate them as:

In [116]:
```
x = np.array([155, 157, 160, 159, 162,
              160, 161, 165, 160, 158])
xvar = np.var(x)
xstd = np.std(x)
print(f'Variance={xvar}, Standard Deviation={xstd}')
```

```
Variance=6.81, Standard Deviation=2.6095976701399777
```

1.3.5 Range & Interquartile Range (IQR):

The difference between max and minimum value of a vector is called range of that vector.

The difference between 3rd (75 percentile) and 1st (25th per) quartile of vector is called interquartile range(IQR).

Consider the Python snippet below:

In [119]:
```
x = np.array([155, 157, 160, 159, 162, 153, 150, 154,
              160, 161, 165, 160, 158, 145, 132, 160])
rng = x.max() - x.min()
print(f'Range = {rng}')

q1 = np.percentile(x, 25)
q3 = np.percentile(x, 75)
iqr = q3 - q1

print(f'Inter Quartile Range (IQR) = {iqr}')
```

```
Range = 33
Inter Quartile Range (IQR) = 6.25
```

1.4 Probability:

At the most basic level, probability seeks to answer the question, “What is the chance of an event happening?” An event is some outcome of interest. To calculate the chance of an event happening, we also need to consider all the other events that can occur. The quintessential

By:
Hayat Ullah (Computer Engr., FCSE)

representation of probability is the humble coin toss. In a coin toss the only events that can happen are:

- tossing a coin and it landing on heads
- tossing a coin and it landing on tails
- rolling a '3' on a die
- rolling a number > 4 on a die
- it rains two days in a row
- drawing a card from the suit of clubs
- guessing a certain number between 000 and 999 (lottery)

Statistically, probability is number between 0 and 1 (inclusive). 0 corresponds to impossible event, while 1 corresponds to the event certainly happening.

Events that are certain:

- If it is Thursday, the probability that tomorrow is Friday is certain, therefore the probability is 1.
- If you are sixteen, the probability of you turning seventeen on your next birthday is 1. This is a certain event.

1.4.1 Sample Space:

Sample space is the set of all possible outcomes of an event. If we consider tossing coin experiment once, so there is a chance to get head (H) or tail (T). So the sample space of the event be $\{H, T\}$. Head and tail are equally probable, so head has 0.5 or 50% probability and tail has also the same probability. The sample space of Ludo die be $\{1,2,3,4,5,6\}$.

1.4.2 Random Variable:

In probability and statistics, random variables are used to quantify outcomes of a random occurrence, and therefore, can take on many values. Random variables are required to be measurable and are typically real numbers. For example, the letter X may be designated to represent the sum of the resulting numbers after three dice are rolled. In this case, X could be 3 (1 + 1+ 1), 18 (6 + 6 + 6), or somewhere between 3 and 18, since the highest number of a die is 6 and the lowest number is 1.

Random variable can be discrete or continuous.

1.4.2.1 Discrete Random Variables:

A discrete random variable is a type of random variable that has a countable number of distinct values that can be assigned to it, such as in a coin toss.

1.4.2.2 Continuous Random Variables:

A continuous random variable stands for any amount within a specific range or set of points and can reflect an infinite number of potential values, such as the average rainfall in a region.

By:
Hayat Ullah (Computer Engr., FCSE)

1.4.3 Probability Distribution Function (PDF):

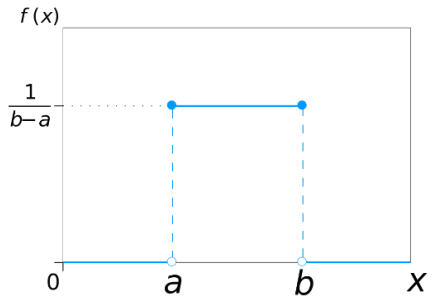
A probability distribution is a statistical function that describes all the possible values and likelihoods that a random variable can take within a given range. This range will be bounded between the minimum and maximum possible values, but precisely where the possible value is likely to be plotted on the probability distribution depends on a number of factors.

We will consider two basic types of distribution functions i.e. Uniform & Normal Distribution function.

1.4.3.1 Uniform Distribution:

In probability theory and statistics, the continuous uniform distribution or rectangular distribution is a family of symmetric probability distributions. The distribution describes an experiment where there is an arbitrary outcome that lies between certain bounds.

Uniform probability distribution function is given by:



$$f(x) = \begin{cases} \dfrac{1}{b-a} & for\ x \epsilon [a,b] \\ 0 & otherwise \end{cases}$$

Mean μ & standard deviation σ for uniform distribution are $\frac{a+b}{2}$ and $\sqrt{\frac{(b-a)^2}{12}}$ respectively.

In the uniform distribution, every sample is equally like to occur. For more details and mathematical depth, kindly follow the link.

https://en.wikipedia.org/wiki/Continuous_uniform_distribution

1.4.3.2 Normal Distribution:

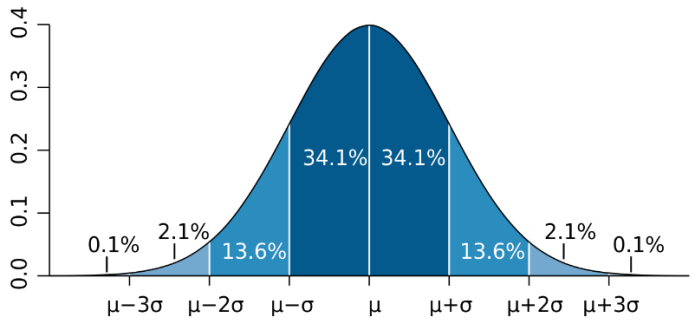
In statistics, a normal distribution (also known as Gaussian, Gauss, or Laplace–Gauss distribution) is a type of continuous probability distribution for a real-valued random variable. The general form of its probability density function is:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)$$

Where μ & σ are mean and standard deviation of the distribution.

By:
Hayat Ullah (Computer Engr., FCSE)

The curve plot is:



From the distribution function above, we conclude that samples closer to μ are more likely to occur. 68.3% area under curve is ranging $\mu - \sigma \leq x \leq \mu + \sigma$, it means when we draw sample x from normal distribution, it has probability of 0.683 to satisfy $\mu - \sigma \leq x \leq \mu + \sigma$.

Similarly, 95.5% of area under the curve is ranging $\mu - 2\sigma \leq x \leq \mu + 2\sigma$, it means when we draw sample x from normal distribution, it has probability of 0.955 to satisfy $\mu - 2\sigma \leq x \leq \mu + 2\sigma$.

Similarly, 99.7% of area under the curve is ranging $\mu - 3\sigma \leq x \leq \mu + 3\sigma$, it means when we draw sample x from normal distribution, it has probability of 0.955 to satisfy $\mu - 2\sigma \leq x \leq \mu + 2\sigma$.

The above three special probabilities are referred as 3 Sigma Rule. For more details about normal distribution and 3 Sigma Rule, please follow the link: https://en.wikipedia.org/wiki/68%E2%80%9395%E2%80%9399.7_rule .

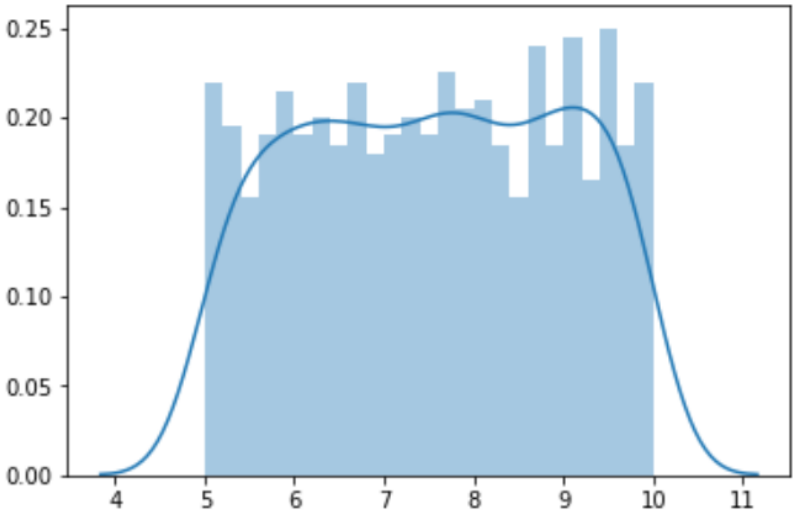
Now using Python, we take 1000 sample from Uniform Distribution and plot it histogram:

In [150]:
```
# Uniform Distribution:
import seaborn as sns # seaborn for plotting
a, b = 5, 10
n = 1000
x = np.random.uniform(low=a, high=b, size=n)
sns.distplot(x, bins=25)
print(f'Observed \tMean = {x.mean()}, STD = {x.std()}')
print(f'Theoritical \tMean = {(a+b)/2},\t\tSTD = {np.sqrt(((b-a)**2)/12)}')
```

```
Observed        Mean = 7.536967015264195, STD = 1.4527307593162864
Theoritical     Mean = 7.5,               STD = 1.4433756729740645
```

Generated plot:

By:
Hayat Ullah (Computer Engr., FCSE)

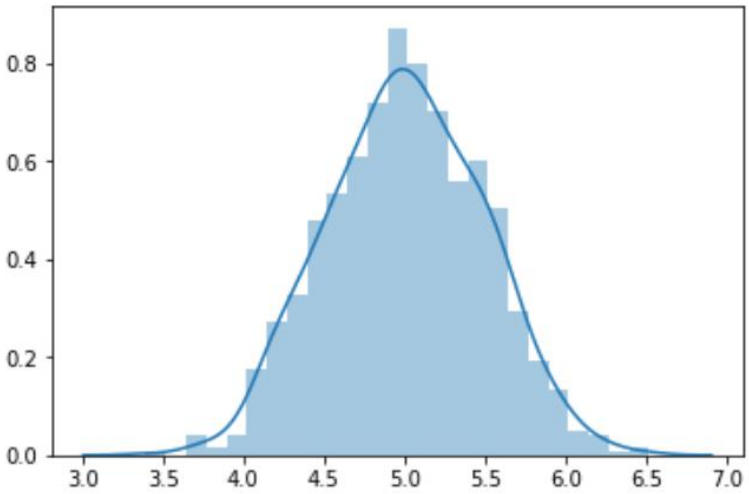


Now we sample from normal distribution with $\mu = 5$ and $\sigma = 0.5$, and evaluate the observed mean & STD.

In [153]:
```
# Normal Distribution:
import seaborn as sns # seaborn for plotting
mu, sig = 5, 0.5
n = 1000
x = np.random.normal(loc=mu, scale=sig, size=n)
sns.distplot(x, bins=25)
print(f'Observed \tMean = {x.mean()}, STD = {x.std()}')
print(f'Theoritical \tMean = {mu},\t\tSTD = {sig}')
```

```
Observed        Mean = 5.003480201193338, STD = 0.48415813923713763
Theoritical     Mean = 5,                 STD = 0.5
```

Generated Plot:



By:
Hayat Ullah (Computer Engr., FCSE)

For more distributions, kindly follow the links below:

- https://data-flair.training/blogs/python-probability-distributions/#:~:text=Python%20normal%20distribution%20is%20a,probability%20distribution%20for%20each%20value.
- https://www.datacamp.com/tutorial/probability-distributions-python

Lab. # 2: Single Layer Perceptron from Scratch

Prerequisites for this Lab.:

Before you start this lab, you must have clear understanding of the concepts below:

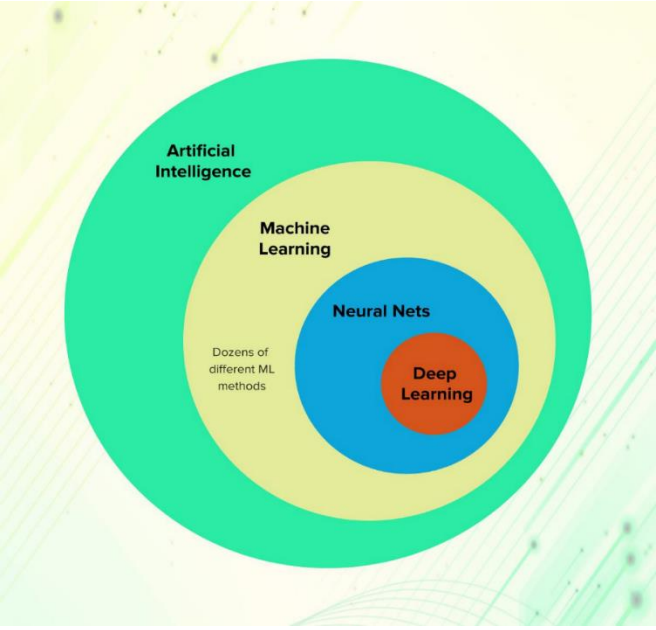
Machine learning, Supervised and unsupervised learning, classification & regression

2.1 Artificial Intelligence Vs Machine Learning Vs Deep Learning

In modern age, people benefit from artificial intelligence every day: music recommender systems, Google maps, Uber, and many more applications are powered with AI. However, the confusion between the terms artificial intelligence, machine learning, and deep learning remains. One of popular Google search requests goes as follows: "are artificial intelligence and machine learning the same thing?".

Let's clear things up: artificial intelligence (AI), machine learning (ML), and deep learning (DL) are three different things.

- Artificial intelligence is a science like mathematics or biology. It studies ways to build intelligent programs and machines that can creatively solve problems, which has always been considered a human prerogative.
- Machine learning is a subset of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. In ML, there are different algorithms (e.g. neural networks) that help to solve problems.
- Deep learning, or deep neural learning, is **a subset of machine learning, which uses the neural networks to analyze different factors with a structure that is similar to the human neural system.** This is how it looks on an Euler diagram in Figure below:



By:
Hayat Ullah (Computer Engr., FCSE)

2.2 Neural Networks:

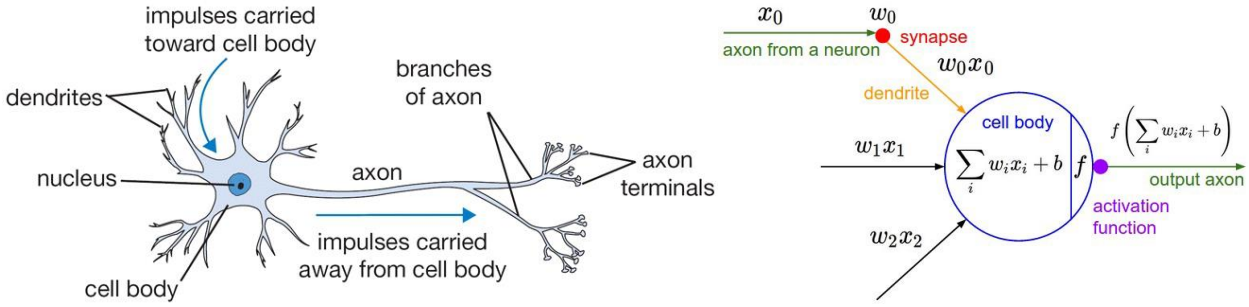
Artificial neural networks (ANNs), usually simply called neural networks (NNs) or neural nets, are computing systems inspired by the biological neural networks that constitute animal brains.

An ANN is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. An artificial neuron receives signals then processes them and can signal neurons connected to it. The "signal" at a connection is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs. The connections are called edges. Neurons and edges typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons may have a threshold such that a signal is sent only if the aggregate signal crosses that threshold. Typically, neurons are aggregated into layers. Different layers may perform different transformations on their inputs. Signals travel from the first layer (the input layer), to the last layer (the output layer), possibly after traversing the layers' multiple times.

2.3 Neural Network Architecture:

2.3.1 Biological Motivation:

The basic computational unit of the brain is a neuron. Approximately 86 billion neurons can be found in the human nervous system and they are connected with approximately 10^{14} — 10^{15} synapses. The diagram below shows a cartoon drawing of a biological neuron (left) and a common mathematical model (right).



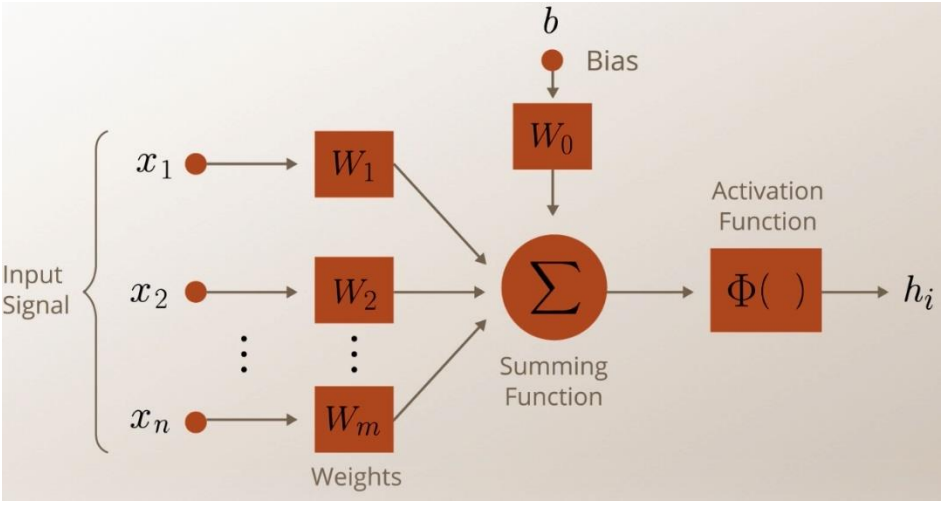
The basic unit of computation in a neural network is the neuron, often called a node or unit. It receives input from some other nodes, or from an external source and computes an output. Each input has an associated

weight (w), which is assigned on the basis of its relative importance to other inputs. The node applies a function to the weighted sum of its inputs.

The idea is that the synaptic strengths (the weights w) are learnable and control the strength of influence and its direction: excitory (positive weight) or inhibitory (negative weight) of one neuron on another. In the basic model, the dendrites carry the signal to the cell body where they all get summed. If the final sum is above a certain threshold, the neuron can fire, sending a

By:
Hayat Ullah (Computer Engr., FCSE)

spike along its axon. In the computational model, we assume that the precise timings of the spikes do not matter, and that only the frequency of the firing communicates information. we model the firing rate of the neuron with an activation function (e.x. sigmoid function), which represents the frequency of the spikes along the axon. The basic building block of an artificial neural network is known as perceptron.



2.3.2 Gradient Descent:

Before we go to the details of Perceptron, we need to explain gradient descent, an important concept of numerical analysis that is used by most of the neural networks.

Gradient descent is used for minimizing a function with respect to a given variable.

Let $f(x)$ a function of x, and we must minimize $f(x)$ w.r.t. x. According to gradient descent:

$$x \leftarrow x - \eta \frac{df(x)}{dx}$$

Where η is the called learning rate and $\frac{df(x)}{dx}$ is the first order derivative of $f(x)$ w.r.t. x at a given value of x.

When there is a function of multiple independent variables, say $f(x,y)$, and we have to minimize it, x and y are updated as:

$$x \leftarrow x - \eta \frac{\partial f(x,y)}{\partial x}$$
$$y \leftarrow y - \eta \frac{\partial f(x,y)}{\partial y}$$

The lesser you have learning rate and the more you iterate, the more $f(x)$ will minimize.

Implementing in Python:

By:
Hayat Ullah (Computer Engr., FCSE)

i. Define $f(x) = x^2$, and $\frac{df(x)}{dx} = 2x$, as given below:

In [134]:
```
# Gradient Descent:
# Declare function and its respective derivative:
f = lambda x: (x-4)**2
df = lambda x: 2*(x-4)
```

ii. Initialize the hyperparameters, learning rate and number of iterations:

In [135]:
```
# Hyperparmeters:
lr = .1  # Learning Rate
itr = 100        # Maximum Iterations
```

iii. Implement gradient descent according to the given formula:

In [144]:
```
import numpy as np
x = 30
xsol = np.zeros(itr)
for i in range(itr):
    xsol[i] = x
    x = x - lr*df(x)

print(x)
```

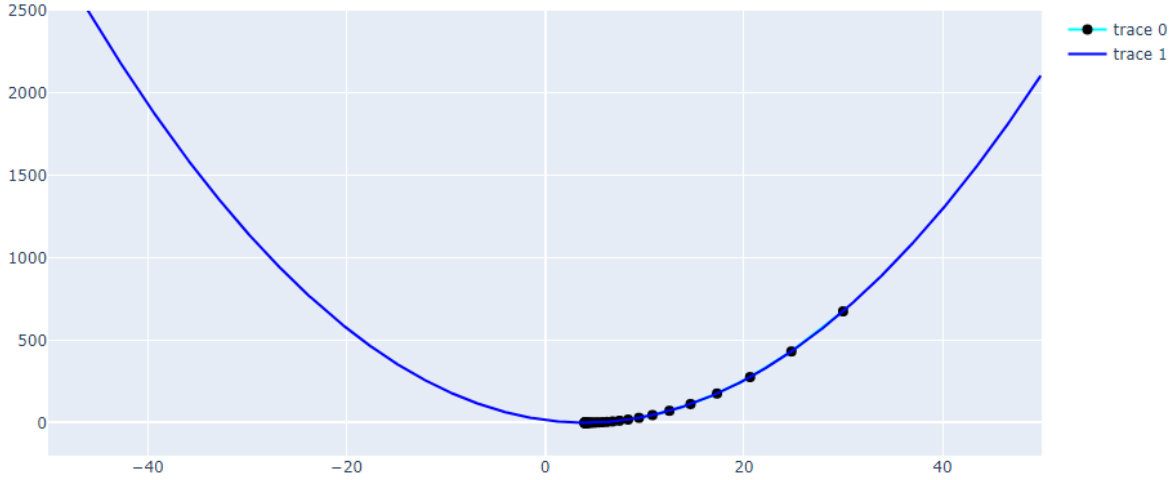
```
4.000000005296293
```

Gradient descent may not always converge, it diverges when learn rate is kept high.

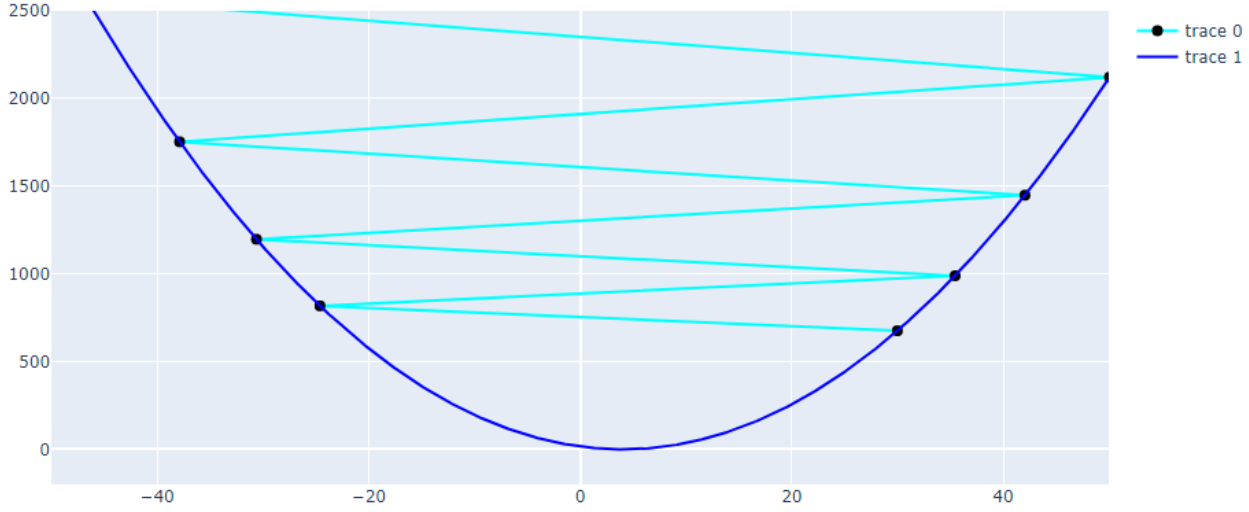
Here we consider convergence and divergence:

By:
Hayat Ullah (Computer Engr., FCSE)

When $\eta = 0.1$, it converges, as given below:



When $\eta = 1.05$, instead of convergence, it diverges, as show below. We started with x=30.



2.3.3 Perceptron:

The original Perceptron was designed to take several inputs and produce one binary output (0 or 1). The idea was to use different weights to represent the importance of each input, and that the sum of the values should be greater than a threshold value before deciding like true or false (0 or 1).

2.3.3.1 Perceptron Terminology

- **Perceptron Inputs:** It is input data to the neuron.
- **Node values:** The output is called node value.
- **Node Weights:** It is feature weights. Weights are multiplied by input x.

By:
Hayat Ullah (Computer Engr., FCSE)

- **Activation Function:** When weights are multiplied by inputs and bias is added, we have to threshold the data to convert it into some meaningful form. For doing this job, we use activation functions. The activation function associated to perceptron is called sigmoid function, explained in more details in the next section.

2.3.3.2 Steps in Perceptron:

Let suppose data has n number of features, so for perceptron, we need n number of weights and a bias to produce node value of $z^{(i)}$ associated with i-th input, as given by:

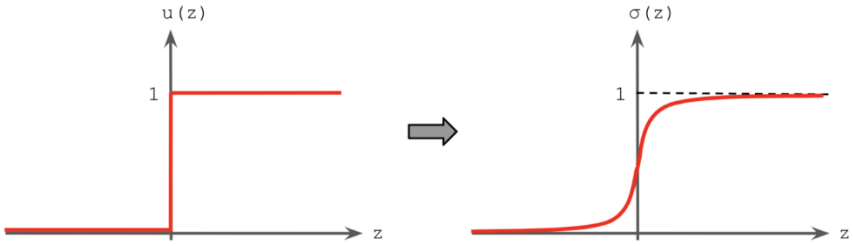
$$z^{(i)} = b + \sum_{j=1}^{n} w_j x_j^{(i)}$$

$z^{(i)}$ corresponds to line boundary between two classes associated to i-th input example. Generally, we represent b by w_0 and $x_0 = 1$. So, the above equation becomes:

$$z^{(i)} = \sum_{j=0}^{n} w_j x_j^{(i)}$$

Then z is threshold to decide. Ideally step function is used, denoted by $u(z)$, which means if $z \geq 0$, then the model should predict 1 otherwise 0, but with step function, $u(z)$, there is a problem that it is indifferentiable at $z = 0$, for updating weights, we need to evaluate derivative of the function.

To avoid this problem we use sigmoid function, $\sigma(z)$, instead of step function. Please look at the picture showing that sigmoid is approximation of step function.



Mathematically, for i-th input, we have output, given by:

$$\sigma\left(z^{(i)}\right) = \frac{1}{1 + e^{-z^{(i)}}}$$

Derivative of sigmoid function is:

$$\frac{\partial \sigma\left(z^{(i)}\right)}{\partial z^{(i)}} = \sigma\left(z^{(i)}\right)\left(1 - \sigma\left(z^{(i)}\right)\right)$$

Proof of the above is left as practice for students.

By:
Hayat Ullah (Computer Engr., FCSE)

Steps of Perceptron:

1) **Feed Forward:** In this step the data flows from input to output of the network. Now weight or bias is updated in this step.
   These steps include the following sub-steps:
   i) Multiply input by weights and add bias.
   ii) Pass the accumulated value to sigmoid which is the predicted value. The predicted output is output from sigmoid:

$$y_{pred}^{(i)} = \sigma\left(z^{(i)}\right)$$

   For i^{th} input example.
2) **Back-Propagation:** In this step the weights are updated. Machines learns from its mistakes. Mistake is technically error which is nothing but mean of squared differences of actual output and the predicted output by Artificial Neuron (Perceptron in this case).
   Let X (having m samples and n features) be the input and $\vec{y}$ (a vector having m element) be the actual output.
   i) Error is calculated sum of squared differences, as:

$$e = \frac{1}{m}\sum_{i=1}^{m}\left(y_{pred}^{(i)} - y^{(i)}\right)^2$$

   In the above case, m and $y^{(i)}$ are constant while $y_{pred}^{(i)}$ is variable and is dependent upon the value of weights. For Gradient Descent, we need to minimize error, e, with respect to weights.
   ii) Differentiating with respect to $y_{pred}^{(i)}$, we have:

$$\frac{\partial e}{\partial y_{pred}^{(i)}} = \frac{1}{m}\sum_{i}^{m} 2\left(y_{pred}^{(i)} - y^{(i)}\right)$$

   iii) Differentiating $y_{pred}^{(i)}$ w.r.t. $z^{(i)}$, we have:

$$\frac{\partial y_{pred}^{(i)}}{\partial z^{(i)}} = \sigma\left(z^{(i)}\right)\left(1 - \sigma\left(z^{(i)}\right)\right)$$

   OR

$$\frac{\partial y_{pred}^{(i)}}{\partial z^{(i)}} = y_{pred}^{(i)}\left(1 - y_{pred}^{(i)}\right)$$

   iv) Differentiating $z^{(i)}$ w.r.t. to w_j, given by:

By:
Hayat Ullah (Computer Engr., FCSE)

$$\frac{\partial z^{(i)}}{\partial w_j} = \frac{\partial}{\partial w_j}\left(\sum_{k=0}^{n} w_k x_k^{(i)}\right) = x_j^{(i)}$$

v) Using chain rule of derivative to evaluate $\frac{\partial e}{\partial w_j}$, as given by:

$$\frac{\partial e}{\partial w_j} = \frac{\partial e}{\partial y_{pred}^{(i)}} \times \frac{\partial y_{pred}^{(i)}}{\partial z^{(i)}} \times \frac{\partial z^{(i)}}{\partial w_j}$$

$$\Rightarrow \frac{\partial e}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m} 2\left(y_{pred}^{(i)} - y^{(i)}\right)\left(y_{pred}^{(i)}\right)\left(1 - y_{pred}^{(i)}\right)\left(x_j^{(i)}\right)$$

vi) Updating weights using Gradient Descent, given by:

$$w_j \leftarrow w_j - \eta \frac{\partial e}{\partial w_j}$$

Python Code:

Data Generation from Uniform Distribution:

```
# Creating a dummy dataset
import numpy as np
n = 100
mu_hor1 = 1
mu_ver1 = 2
hor1 = np.random.uniform(size=n) + mu_hor1
ver1 = np.random.uniform(size=n) + mu_ver1
```

```
# Creating a dummy dataset
import numpy as np
n = 100
mu_hor2 = 2
mu_ver2 = 3
hor2 = np.random.uniform(size=n) + mu_hor2
ver2 = np.random.uniform(size=n) + mu_ver2

import matplotlib.pyplot as plt
plt.scatter(hor1, ver1)
plt.scatter(hor2, ver2)
plt.show()
```

Data Concatenation:

```
# concatenating
x1 = np.concatenate((hor1,hor2))
x2 = np.concatenate((ver1,ver2))

# Labels
y = np.concatenate((np.zeros(shape=n), np.ones(shape=n)))
```


By:
Hayat Ullah (Computer Engr., FCSE)

Randomly Initialize the weights:

```
w0, w1, w2 = tuple(np.random.uniform(size=3))
w0, w1, w2
```

```
(0.13400746499696836, 0.9771388122157311, 0.5319842005169761)
```

Declare Sigmoid and its Derivative:

```
def sigmoid(z):
    return 1/(1+np.exp(-z))
```

```

def dsig(z):
    return sigmoid(z)*(1-sigmoid(z))
```

Decision Boundary:

```
z = lambda X: w0 + w1*X[:,0] + w2*X[:,1]
```

Gradient Descent:

```
lr = 0.1
for i in range(2000):
    if i%300 == 0:
        xx = np.linspace(0.5, +3., 5)
        fig, ax = plt.subplots(1, 1, figsize=(8, 6))
        ax.plot(xx, -(w0/w2) - (w1/w2)*xx)
        ax.scatter(hor1, ver1)
        ax.scatter(hor2, ver2)
        ax.grid(True)

    ypred = sigmoid(z(X))
    dw0 = (2*(ypred - y)*ypred*(1-ypred)).mean()
    dw1 = (2*(ypred - y)*ypred*(1-ypred)*X[:,0]).mean()
    dw2 = (2*(ypred - y)*ypred*(1-ypred)*X[:,1]).mean()

    w0 = w0 - lr*dw0
    w1 = w1 - lr*dw1
    w2 = w2 - lr*dw2
```


By:
Hayat Ullah (Computer Engr., FCSE)

Lab. # 3: Multilayer Perceptron

We need to keep in mind the big picture here:

1. We feed **input** data into the neural network.
2. The data flows **from layer to layer** until we have the **output**.
3. Once we have the output, we can calculate the **error** which is a **scalar**.
4. Finally, we can adjust a given parameter (weight or bias) by subtracting the **derivative** of the error with respect to the parameter itself.
5. We iterate through that process.

The most important step is the **4th**. We want to be able to have as many layers as we want, and of any type. But if we modify/add/remove one layer from the network, the output of the network is going to change, which is going to change the error, which is going to change the derivative of the error with respect to the parameters. We need to be able to compute the derivatives regardless of the network architecture, regardless of the activation functions, regardless of the loss we use.

In order to achieve that, we must implement **each layer separately**.

What every layer should implement

Every layer that we might create (fully connected, convolutional, maxpooling, dropout, etc.) have at least 2 things in common: **input** and **output** data.

$$X \rightarrow \boxed{\text{layer}} \rightarrow Y$$

3.1 Forward propagation

We can already emphasize one important point which is: the output of one layer is the input of the next one.

This is called forward propagation. Essentially, we give the input data to the first layer, then the output of every layer becomes the input of the next layer until we reach the end of the network. By comparing the result of the network (Y) with the desired output (let's say Y*), we can calculate en error E. The goal is to minimize that error by changing the parameters in the network. That is backward propagation (backpropagation).

3.2 Gradient Descent

This is a quick reminder, if you need to learn more about gradient descent there are tons of resources on the internet.

By:
Hayat Ullah (Computer Engr., FCSE)

Basically, we want to change some parameter in the network (call it w) so that the total error E decreases. There is a clever way to do it (not randomly) which is the following:

$$w \leftarrow w - \alpha \frac{\partial E}{\partial w}$$

Where α is a parameter in the range [0,1] that we set and that is called the learning rate. Anyway, the important thing here is ∂E/∂w (the derivative of E with respect to w). We need to be able to find the value of that expression for any parameter of the network regardless of its architecture.

3.3 Backward propagation

Suppose that we give a layer the derivative of the error with respect to its output (∂E/∂Y), then it must be able to provide the derivative of the error with respect to its input (∂E/∂X).

$$\frac{\partial E}{\partial X} \leftarrow \boxed{\text{layer}} \leftarrow \frac{\partial E}{\partial Y}$$

Remember that E is a scalar (a number) and X and Y are matrices.

$$\frac{\partial E}{\partial X} = \begin{bmatrix} \frac{\partial E}{\partial x_1} & \frac{\partial E}{\partial x_2} & \cdots & \frac{\partial E}{\partial x_i} \end{bmatrix}$$

$$\frac{\partial E}{\partial Y} = \begin{bmatrix} \frac{\partial E}{\partial y_1} & \frac{\partial E}{\partial y_2} & \cdots & \frac{\partial E}{\partial y_j} \end{bmatrix}$$

Let's forget about ∂E/∂X for now. The trick here, is that if we have access to ∂E/∂Y we can very easily calculate ∂E/∂W (if the layer has any trainable parameters) without knowing anything about the network architecture! We simply use the chain rule:

$$\frac{\partial E}{\partial w} = \sum_j \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial w}$$

The unknown is ∂y_j/∂w which totally depends on how the layer is computing its output. So, if every layer have access to ∂E/∂Y, where Y is its own output, then we can update our parameters!

By:
Hayat Ullah (Computer Engr., FCSE)

3.4 But why $\partial E/\partial X$?

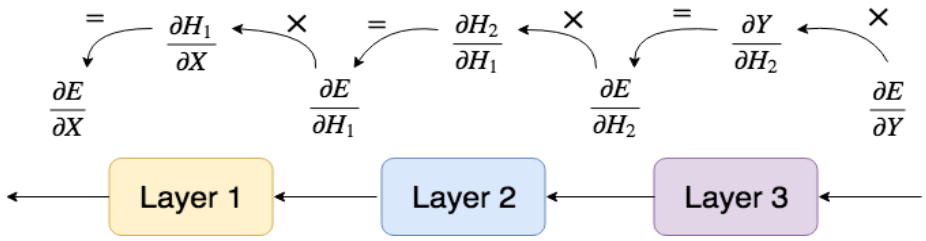
Don't forget, the output of one layer is the input of the next layer. Which means ∂E/∂X for one layer is $\partial E/\partial Y$ for the previous layer! That's it! It's just a clever way to propagate the error! Again, we can use the chain rule:

$$\frac{\partial E}{\partial x_i} = \sum_j \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

This is very important, it's the key to understand backpropagation! After that, we'll be able to code a Deep Convolutional Neural Network from scratch in no time!

3.5 Diagram to understand backpropagation

This is what I described earlier. Layer 3 is going to update its parameters using ∂E/∂Y and is then going to pass ∂E/∂H2 to the previous layer, which is its own "∂E/∂Y". Layer 2 is then going to do the same, and so on and so forth.



This may seem abstract here, but it will get very clear when we will apply this to a specific type of layer. Speaking of abstract, now is a good time to write our first python class.

3.6 Abstract Base Class: Layer

The abstract class Layer, which all other layers will inherit from, handles simple properties which are an input, an output, and both a forward and backward methods.

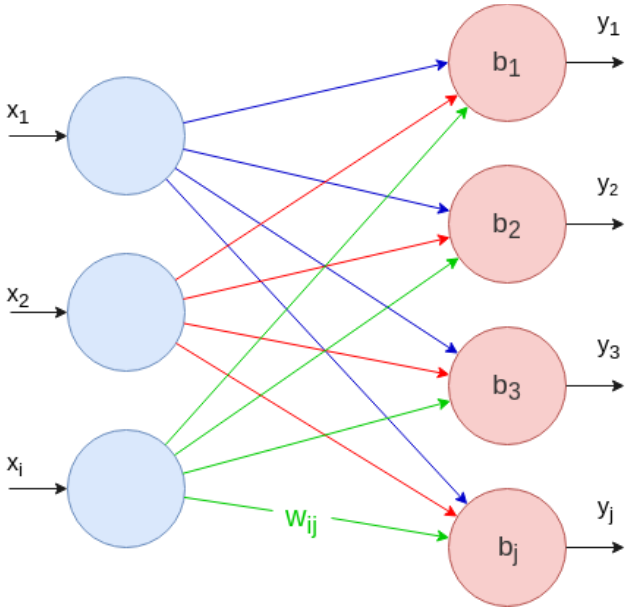
```
# Base class
class Layer:
    def __init__(self):
        self.input = None
        self.output = None

    # computes the output Y of a layer for a given input X
    def forward_propagation(self, input):
        raise NotImplementedError

    # computes dE/dX for a given dE/dY (and update parameters if any)
    def backward_propagation(self, output_error, learning_rate):
        raise NotImplementedError
```


By:
Hayat Ullah (Computer Engr., FCSE)

As you can see there is an extra parameter in backward_propagation that I didn't mention, it is the learning_rate. This parameter should be something like an update policy, or an optimizer as they call it in Keras, but for the sake of simplicity we're simply going to pass a learning rate and update our parameters using gradient descent.

3.7 Fully Connected Layer

Now let's define and implement the first type of layer: fully connected layer or FC layer. FC layers are the most basic layers as every input neurons are connected to every output neurons.



3.8 Forward Propagation

The value of each output neuron can be calculated as the following:

$$y_j = b_j + \sum_i x_i w_{ij}$$

With matrices, we can compute this formula for every output neuron in one shot using a dot product:

$$X = \begin{bmatrix} x_1 & \dots & x_i \end{bmatrix} \quad W = \begin{bmatrix} w_{11} \dots w_{1j} \\ \vdots \quad \ddots \quad \vdots \\ w_{i1} \dots w_{ij} \end{bmatrix} \quad B = \begin{bmatrix} b_1 & \dots & b_j \end{bmatrix}$$

$$Y = XW + B$$

We're done with the forward pass. Now let's do the backward pass of the FC layer.

By:
Hayat Ullah (Computer Engr., FCSE)

3.9 Backward Propagation

As we said, suppose we have a matrix containing the derivative of the error with respect to that layer's output (∂E/∂Y). We need :

1. The derivative of the error with respect to the parameters (∂E/∂W, ∂E/∂B)
2. The derivative of the error with respect to the input (∂E/∂X)

Let's calculate ∂E/∂W. This matrix should be the same size as W itself: ixj where i is the number of input neurons and j the number of output neurons. We need one gradient for every weight:

$$\frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial w_{11}} & \cdots & \frac{\partial E}{\partial w_{1j}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial w_{i1}} & \cdots & \frac{\partial E}{\partial w_{ij}} \end{bmatrix}$$

Using the chain rule stated earlier, we can write:

$$\begin{aligned} \frac{\partial E}{\partial w_{ij}} &= \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial w_{ij}} + \ldots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial w_{ij}} \\ &= \frac{\partial E}{\partial y_j} x_i \end{aligned}$$

Therefore,

$$\begin{aligned} \frac{\partial E}{\partial W} &= \begin{bmatrix} \frac{\partial E}{\partial y_1} x_1 & \cdots & \frac{\partial E}{\partial y_j} x_1 \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial y_1} x_i & \cdots & \frac{\partial E}{\partial y_j} x_i \end{bmatrix} \\ &= \begin{bmatrix} x_1 \\ \vdots \\ x_i \end{bmatrix} \begin{bmatrix} \frac{\partial E}{\partial y_1} & \cdots & \frac{\partial E}{\partial y_j} \end{bmatrix} \\ &= X^t \frac{\partial E}{\partial Y} \end{aligned}$$

By:
Hayat Ullah (Computer Engr., FCSE)

3.10 Coding the Fully Connected Layer

We can now write some python code to bring this math to life!

```python
from layer import Layer
import numpy as np

# inherit from base class Layer
class FCLayer(Layer):
    # input_size = number of input neurons
    # output_size = number of output neurons
    def __init__(self, input_size, output_size):
        self.weights = np.random.rand(input_size, output_size) - 0.5
        self.bias = np.random.rand(1, output_size) - 0.5

    # returns output for a given input
    def forward_propagation(self, input_data):
        self.input = input_data
        self.output = np.dot(self.input, self.weights) + self.bias
        return self.output

    # computes dE/dW, dE/dB for a given output_error=dE/dY. Returns input_error=dE/dX.
    def backward_propagation(self, output_error, learning_rate):
        input_error = np.dot(output_error, self.weights.T)
        weights_error = np.dot(self.input.T, output_error)
        # dBias = output_error

        # update parameters
        self.weights -= learning_rate * weights_error
        self.bias -= learning_rate * output_error
        return input_error
```


By:
Hayat Ullah (Computer Engr., FCSE)

3.11 Coding the Activation Layer

The code for the activation layer is as straightforward.

```
from layer import Layer

# inherit from base class Layer
class ActivationLayer(Layer):
    def __init__(self, activation, activation_prime):
        self.activation = activation
        self.activation_prime = activation_prime

    # returns the activated input
    def forward_propagation(self, input_data):
        self.input = input_data
        self.output = self.activation(self.input)
        return self.output

    # Returns input_error=dE/dX for a given output_error=dE/dY.
    # learning_rate is not used because there is no "learnable" parameters.
    def backward_propagation(self, output_error, learning_rate):
        return self.activation_prime(self.input) * output_error
```

3.12Loss Function

Until now, for a given layer, we supposed that ∂E/∂Y was given (by the next layer). But what happens to the last layer? How does it get ∂E/∂Y? We simply give it manually, and it depends on how we define the error.

The error of the network, which measures how good or bad the network did for a given input data, is defined by you. There are many ways to define the error, and one of the most known is called MSE — Mean Squared Error.

$$E = \frac{1}{n}\sum_{i}^{n}(y_i^* - y_i)^2$$

Where `y*` and `y` denote **desired output** and **actual output** respectively. You can think of the loss as a last layer which takes all the output neurons and squashes them into one single

By:
Hayat Ullah (Computer Engr., FCSE)

neuron. What we need now, as for every other layer, is to define **∂E/∂Y**. Except now, we finally reached `E`!

$$\begin{aligned}\frac{\partial E}{\partial Y} &= \begin{bmatrix}\frac{\partial E}{\partial y_1} & \cdots & \frac{\partial E}{\partial y_i}\end{bmatrix} \\ &= \frac{2}{n}\begin{bmatrix}y_1 - y_1^* & \cdots & y_i - y_i^*\end{bmatrix} \\ &= \frac{2}{n}(Y - Y^*)\end{aligned}$$

These are simply two python functions that you can put in a separate file. They will be used when creating the network.

```
1  import numpy as np
2
3  # loss function and its derivative
4  def mse(y_true, y_pred):
5      return np.mean(np.power(y_true-y_pred, 2));
6
7  def mse_prime(y_true, y_pred):
8      return 2*(y_pred-y_true)/y_true.size;
```

3.13Network Class

Almost done! We are going to make a `Network` class to create neural networks very easily akin the first picture!

I commented almost every part of the code, it shouldn't be too complicated to understand if you grasped the previous steps. Nevertheless, leave a comment if you have any question, I will gladly answer!

By:
Hayat Ullah (Computer Engr., FCSE)

```python
class Network:
    def __init__(self):
        self.layers = []
        self.loss = None
        self.loss_prime = None

    # add layer to network
    def add(self, layer):
        self.layers.append(layer)

    # set loss to use
    def use(self, loss, loss_prime):
        self.loss = loss
        self.loss_prime = loss_prime

    # predict output for given input
    def predict(self, input_data):
        # sample dimension first
        samples = len(input_data)
        result = []

        # run network over all samples
        for i in range(samples):
            # forward propagation
            output = input_data[i]
            for layer in self.layers:
                output = layer.forward_propagation(output)
            result.append(output)

        return result

    # train the network
    def fit(self, x_train, y_train, epochs, learning_rate):
        # sample dimension first
        samples = len(x_train)

```


By:
Hayat Ullah (Computer Engr., FCSE)

```
        # training loop
        for i in range(epochs):
            err = 0
            for j in range(samples):
                # forward propagation
                output = x_train[j]
                for layer in self.layers:
                    output = layer.forward_propagation(output)

                # compute loss (for display purpose only)
                err += self.loss(y_train[j], output)

                # backward propagation
                error = self.loss_prime(y_train[j], output)
                for layer in reversed(self.layers):
                    error = layer.backward_propagation(error, learning_rate)

            # calculate average error on all samples
            err /= samples
            print('epoch %d/%d   error=%f' % (i+1, epochs, err))
```

3.14 Building Neural Networks

Finally! We can use our class to create a neural network with as many layers as we want! We are going to build two neural networks: a simple **XOR** and a **MNIST** solver.

3.15 Solve XOR

Starting with XOR is always important as it's a simple way to tell if the network is learning anything at all.

By:
Hayat Ullah (Computer Engr., FCSE)

```python
import numpy as np

from network import Network
from fc_layer import FCLayer
from activation_layer import ActivationLayer
from activations import tanh, tanh_prime
from losses import mse, mse_prime

# training data
x_train = np.array([[[0,0]], [[0,1]], [[1,0]], [[1,1]]])
y_train = np.array([[[0]], [[1]], [[1]], [[0]]])

# network
net = Network()
net.add(FCLayer(2, 3))
net.add(ActivationLayer(tanh, tanh_prime))
net.add(FCLayer(3, 1))
net.add(ActivationLayer(tanh, tanh_prime))

# train
net.use(mse, mse_prime)
net.fit(x_train, y_train, epochs=1000, learning_rate=0.1)

# test
out = net.predict(x_train)
print(out)
```


By:
Hayat Ullah (Computer Engr., FCSE)

Lab. # 4: Introduction to Tensorflow from DNN Perspective

4.1 What is TensorFlow?

TensorFlow is an open-source end-to-end machine learning library for preprocessing data, modelling data and serving models (getting them into the hands of others).

Why use TensorFlow?

Rather than building machine learning and deep learning models from scratch, it's more likely you'll use a library such as TensorFlow. This is because it contains many of the most common machine learning functions you'll want to use.

4.2 Introduction to Tensors

If you've ever used NumPy, tensors are kind of like NumPy arrays (we'll see more on this later). For the sake of this lab and going forward, you can think of a tensor as a multi-dimensional numerical representation (also referred to as n-dimensional, where n can be any number) of something. Where something can be almost anything you can imagine:

- It could be numbers themselves (using tensors to represent the price of houses).
- It could be an image (using tensors to represent the pixels of an image).
- It could be text (using tensors to represent words).

Or it could be some other form of information (or data) you want to represent with numbers.

The main difference between tensors and NumPy arrays (also an n-dimensional array of numbers) is that tensors can be used on GPUs (graphical processing units) and TPUs (tensor processing units).

> *"The benefit of being able to run on GPUs and TPUs is faster computation, this means, if we wanted to find patterns in the numerical representations of our data, we can generally find them faster using GPUs and TPUs."*

Okay, we've been talking enough about tensors as we already learnt about tensor lab 1, let's see them.

The first thing we'll do is import TensorFlow under the common alias tf, as:

```
# Import TensorFlow
import tensorflow as tf
print(tf.__version__) # find the version number (should be 2.x+)
```

```
2.4.1
```

4.2.1 Creating Tensors with `tf.constant()`

As mentioned before, in general, you usually won't create tensors yourself. This is because TensorFlow has modules built-in (such as tf.io and tf.data) which can read your data sources

By:
Hayat Ullah (Computer Engr., FCSE)

and automatically convert them to tensors and then later on, neural network models will process these for us. But for now, because we're getting familiar with tensors themselves and how to manipulate them, we'll see how we can create them ourselves. We'll begin by using `tf.constant()`.

```
# Create a scalar (rank 0 tensor)
scalar = tf.constant(7)
scalar
```

```
<tf.Tensor: shape=(), dtype=int32, numpy=7>
```

A scalar is known as a rank 0 tensor. Because it has no dimensions (it's just a number).

Note: For now, you don't need to know too much about the different ranks of tensors (but we will see more on this later). The important point is knowing tensors can have an unlimited range of dimensions (the exact amount will depend on what data you're representing).

```
# Check the number of dimensions of a tensor (ndim stands for number of dimensions)
scalar.ndim
```

```
0
```

A constant number is zero-dimensional tensor. Similary vectors and matrix are 1D and 2D tensor, respectively.

```
# Create a vector (more than 0 dimensions)
vector = tf.constant([10, 10])
vector
```

```
<tf.Tensor: shape=(2,), dtype=int32, numpy=array([10, 10])>
```

```
# Check the number of dimensions of our vector tensor
vector.ndim
```

```
1
```

```
# Create a matrix (more than 1 dimension)
matrix = tf.constant([[10, 7],
                      [7, 10]])
matrix
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[10,  7],
       [ 7, 10]])>
```

```
matrix.ndim
```

```
2
```


By:
Hayat Ullah (Computer Engr., FCSE)

Similarly floating constant is declared as:

```
# Create another matrix and define the datatype
another_matrix = tf.constant([[10., 7.],
                              [3., 2.],
                              [8., 9.]], dtype=tf.float16) # specify the datatype with 'dtype'
another_matrix
```

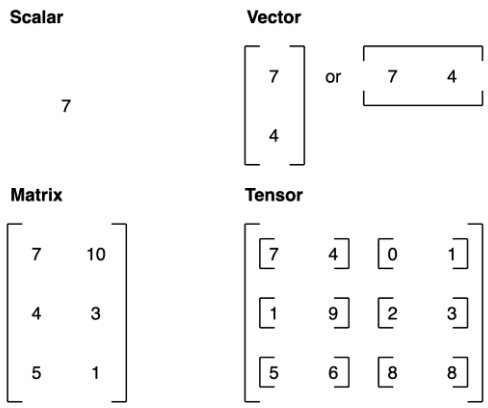
```
<tf.Tensor: shape=(3, 2), dtype=float16, numpy=
array([[10.,  7.],
       [ 3.,  2.],
       [ 8.,  9.]], dtype=float16)>
```

Multi-dimensional tensors:

```
# How about a tensor?
#   (more than 2 dimensions, although,
#     all of the above items are also technically tensors)
tensor = tf.constant([[[1, 2, 3],
                       [4, 5, 6]],
                      [[7, 8, 9],
                       [10, 11, 12]],
                      [[13, 14, 15],
                       [16, 17, 18]]])
tensor
```

```
<tf.Tensor: shape=(3, 2, 3), dtype=int32, numpy=
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]],

       [[13, 14, 15],
        [16, 17, 18]]], dtype=int32)>
```

In the above case, tensor.ndims will return 3 as it has rows columns and height coordinates.



By:
Hayat Ullah (Computer Engr., FCSE)

4.2.2 Creating Tensors with `tf.Variable()`

You can also (although you likely rarely will, because often, when working with data, tensors are created for you automatically) create tensors using `tf.Variable()`.

The difference between `tf.Variable()` and `tf.constant()` is tensors created with tf.constant() are immutable (can't be changed, can only be used to create a new tensor), whereas, tensors created with `tf.Variable()` are mutable (can be changed).

```
# Create the same tensor with tf.Variable() and tf.constant()
changeable_tensor = tf.Variable([10, 7])
unchangeable_tensor = tf.constant([10, 7])
changeable_tensor, unchangeable_tensor
```

```
(<tf.Variable 'Variable:0' shape=(2,) dtype=int32, numpy=array([10,  7], dtype=int32)>,
 <tf.Tensor: shape=(2,), dtype=int32, numpy=array([10,  7], dtype=int32)>)
```

Converting one tensor to another form, as:

```
# Create the same tensor with tf.Variable() and tf.constant()
changeable_tensor = tf.Variable([10, 7])
unchangeable_tensor = tf.constant([10, 7])
changeable_tensor, unchangeable_tensor
```

```
(<tf.Variable 'Variable:0' shape=(2,) dtype=int32, numpy=array([10,  7])>,
 <tf.Tensor: shape=(2,), dtype=int32, numpy=array([10,  7])>)
```

Now let's try to change one of the elements of the changeable tensor.

```
# Will error (requires the .assign() method)
changeable_tensor[0] = 7
changeable_tensor
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-daecfbad2415> in <module>()
      1 # Will error (requires the .assign() method)
```

ReLU:

https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/#:~:text=The%20ReLU%20function%20is%20another,neurons%20at%20the%20same%20time.

One-hot Encoder,

https://hackernoon.com/what-is-one-hot-encoding-why-and-when-do-you-have-to-use-it-e3c6186d008f

By:
Hayat Ullah (Computer Engr., FCSE)

Cross entropy loss function,

By:
Hayat Ullah (Computer Engr., FCSE)