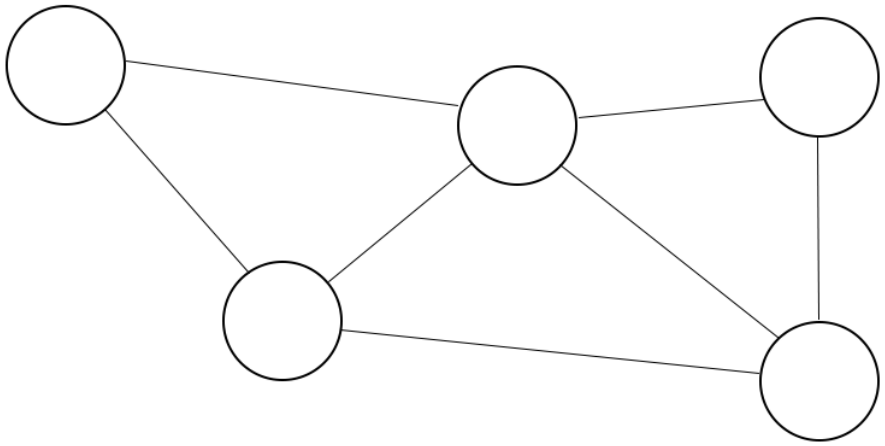
What is a graph?

A **graph**, in the context of graph theory, is a structured datatype that has **nodes** (entities that hold information) and **edges** (connections between nodes that can also hold information). A graph is a way of structuring data, but can be a datapoint itself. Graphs are a type of **Non-Euclidean data**, which means they exist in 3D, unlike other datatypes like images, text, and audio. Graphs can have certain properties, which limit the possible actions and analysis that can be performed on them. These properties can be defined.

Graph Definitions

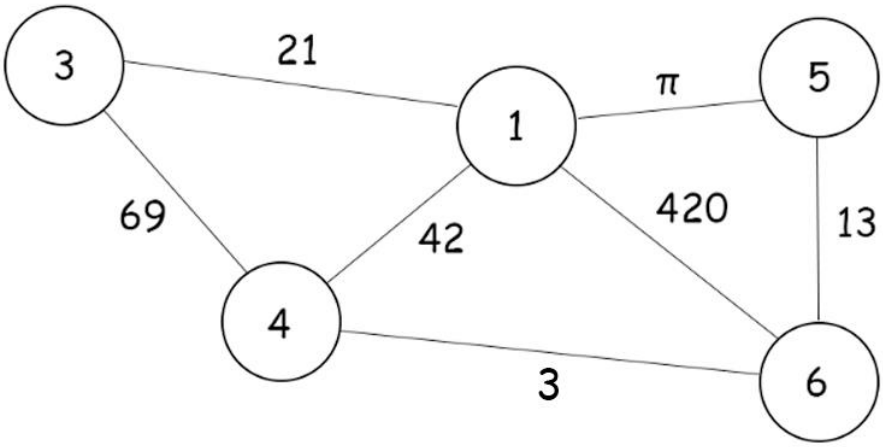
First let's cover some definitions, brought to you by the easiest Photoshop job of my life.

In computer science, we talk alot about a data structure known as **graphs:**



He's cute, let's call him Graham

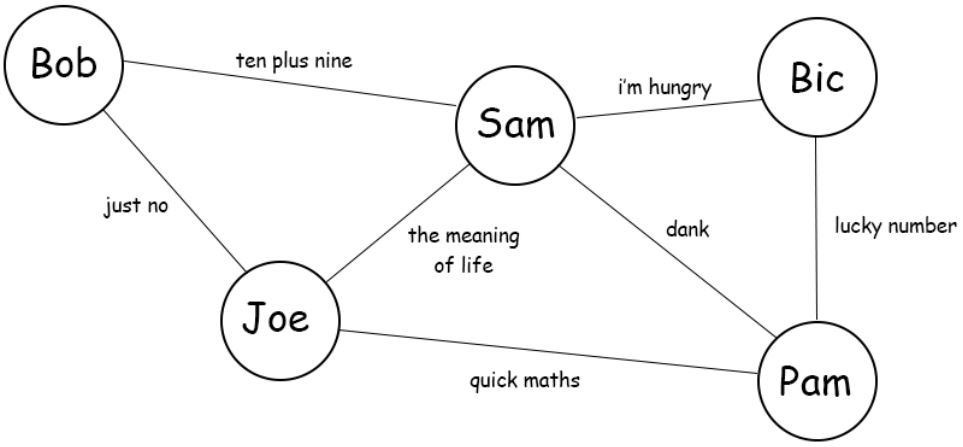
Graphs can have **labels on their edges and/or nodes**, let's give Graham some edge and node labels.



Graham looks rather dashing in his new attire

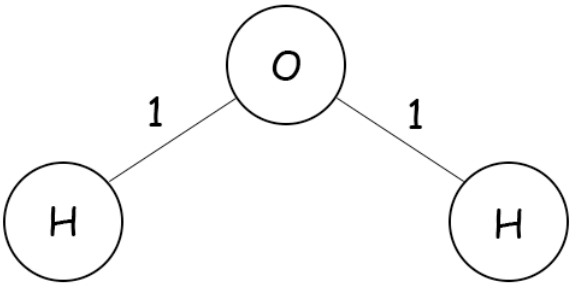
Labels can also be considered **weights**, but that's up to the graph's designer.

Labels don't have to be numerical, they can be textual.



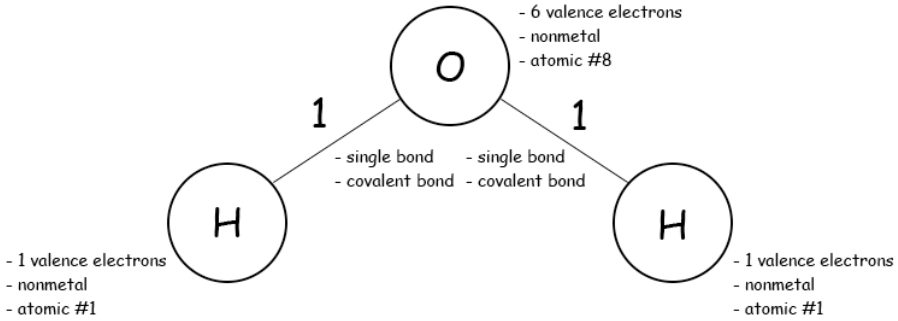
Graham also likes memes

Labels don't have to be unique; it's entirely possible and sometimes useful to give multiple nodes the same label. Take for example, a hydrogen molecule:



Notice the mix of numerical and textual datatypes

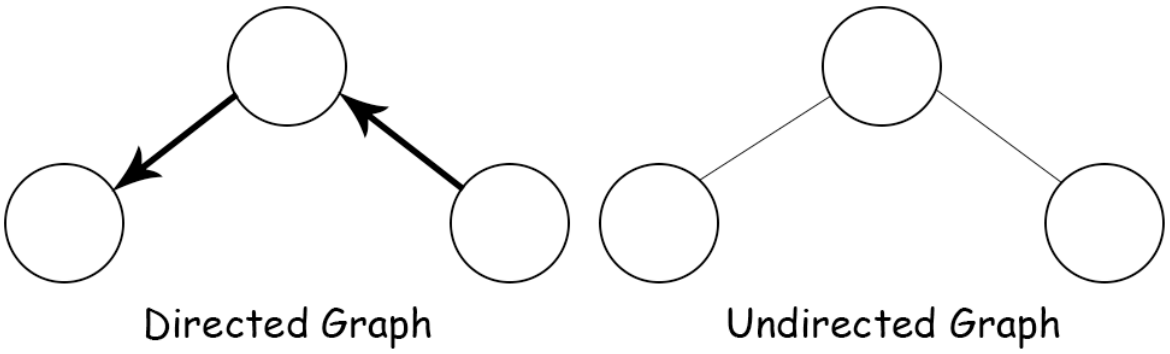
Graphs can have **features** (a.k.a attributes)**.**



Take care not to mix up features and labels. An easy way to think about it is using an analogy to names, characters, and people:

a node is a person, a node's label is a person's name, and the node's features are the person's characteristics.

Graphs can be **directed or undirected:**



A node in the graph can even have an edge that points/connects to itself. This is known as a **self-loop.**

Graphs can be either:

- **Heterogeneous** — composed of different types of nodes
- **Homogeneous** — composed of the same type of nodes

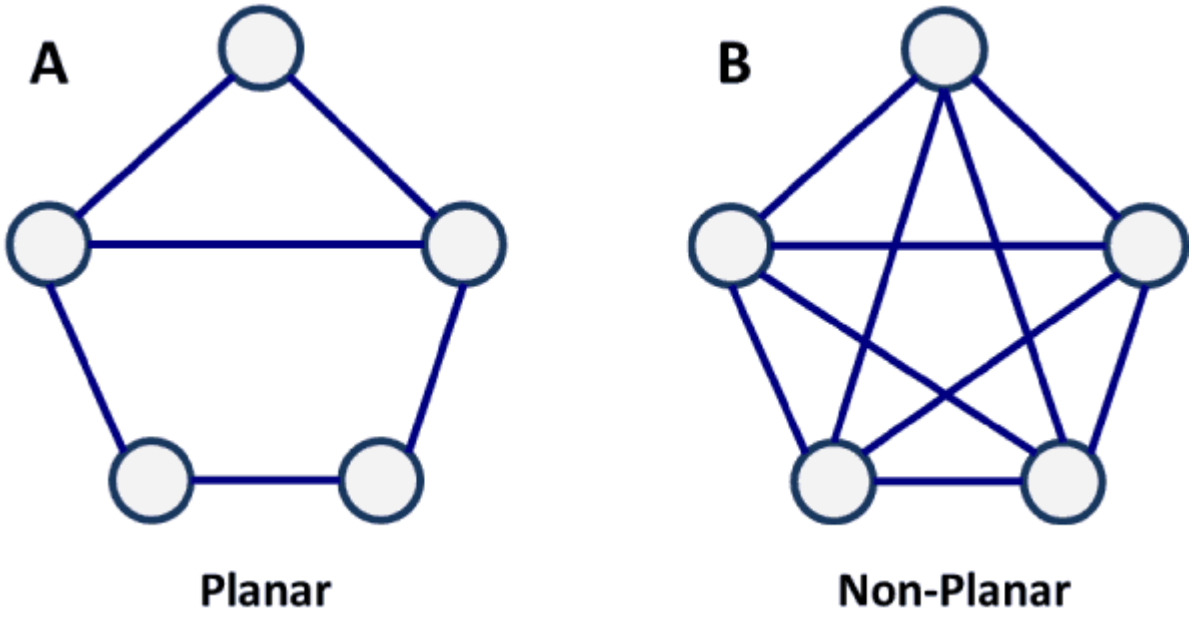
and are either:

- **Static** — nodes and edges do not change, nothing is added or taken away
- **Dynamic** — nodes and edges change, added, deleted, moved, etc.

Roughly speaking, graphs can be vaguely described as either

- **Dense** — composed of many nodes and edges
- **Sparse** — composed of fewer nodes and edges

Graphs can be made to look neater by turning them into their **planar** form, which basically means rearranging nodes such that edges don’t intersect



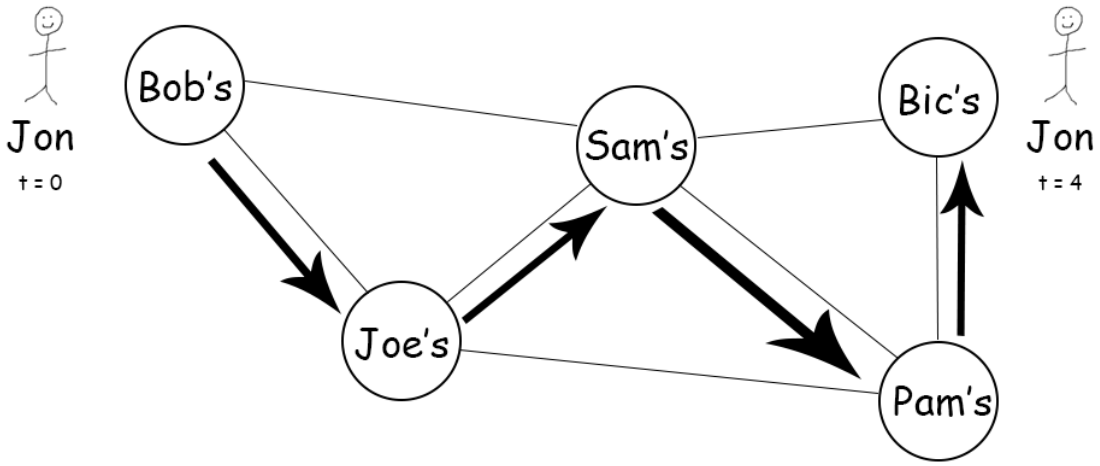
(Courtesy of The Geography of Transport Systems**)**

These concepts and terminology will come in handy as we explore the many different methods currently being employed in the various GNN architectures. Some of these basic methods are described in:

Graph Analysis

There are a host of different Graph structures available for a ML model to learn from (Wheel, Cycle, Star, Grid, Lollipop, Dense, Sparse, etc.)

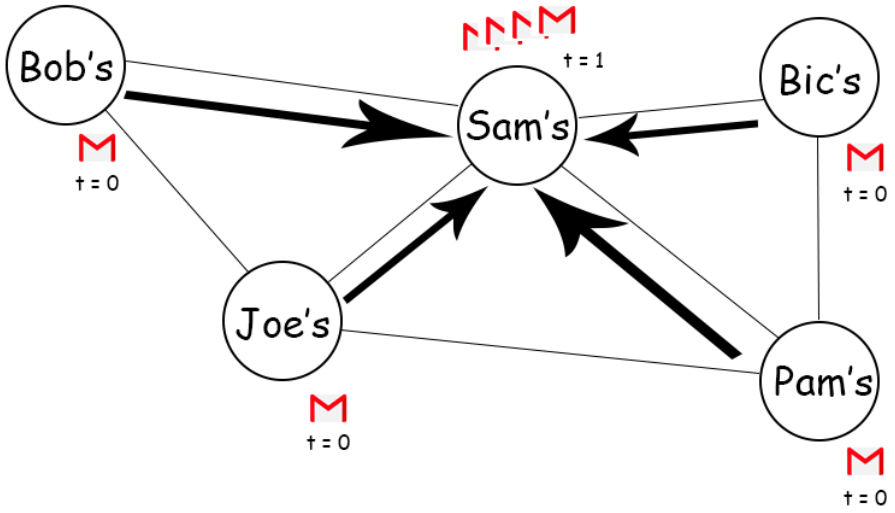
You can traverse a graph



In this case, we're traversing an **undirected** graph. Obviously, if the graph was **directed**, one would simply follow the direction of the edges. There are a couple different types of traversals, so be careful of the wording. These are a couple most common graph traversal terms and what they mean:

- **Walk:** A graph traversal — a **closed walk** is when the destination node is the same as the source node
- **Trail:** A walk with no repeated edges — a **circuit** is a closed trail
- **Path:** A walk with no repeated nodes — a c**ycle** is a closed path

Building on the concept of traversals, one can also **send messages across a graph.**



All of Sam's neighbors are sending him a message, where *t* stands for the timestep. Sam can choose to open his mailbox and update his own information. The concept of propagating information throughout a network is super important for models with **attention mechanisms**. In graphs, message passing is one way we **generalize convolutions.** More on that later.

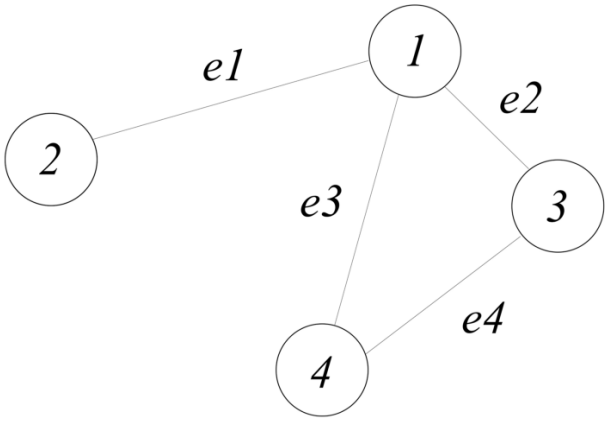
E-graphs — graphs on computers

Having learned all this, you now have a basic understanding of graph theory! Any other concepts important to GNNs will be explained as they come but in the meantime, there is still one last topic concerning graphs that we need to cover. **We must learn how to express graphs computationally.**

There are a couple ways one can turn a graph into a format that a computer can digest; all of them are different types of matrices.

Incidence Matrix (*I*):

The Incidence Matrix, commonly denoted with a capital ***I*** in research papers, Made up of 1s, 0s, and -1s, the incidence matrix can be made by following a simple pattern:



	e_1	e_2	e_3	e_4
1	1	1	1	0
2	1	0	0	0
3	0	1	0	1
4	0	0	1	1

$$= \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}.$$

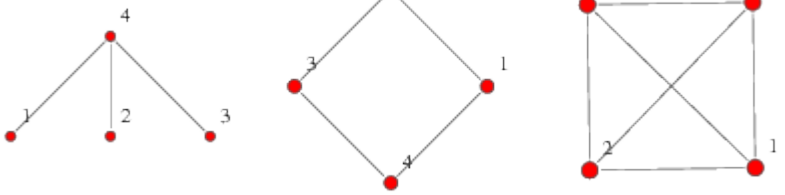
From Graph to Incidence Matrix

(Weighted) Adjacency Matrix (*A*):

The Adjacency Matrix of a graph is be made of 1s and 0s **unless** it is otherwise weighted or labelled. In any case ***A*** can be built by following this rule:

$$a_{ij} = \begin{cases} 1, \text{ if } (v_i, v_j) \in E \\ 0, \text{ otherwise} \end{cases}$$

The Adjacency Matrix of a undirected graph is therefore symmetrical along its diagonal, from the top left object to the bottom right:



$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

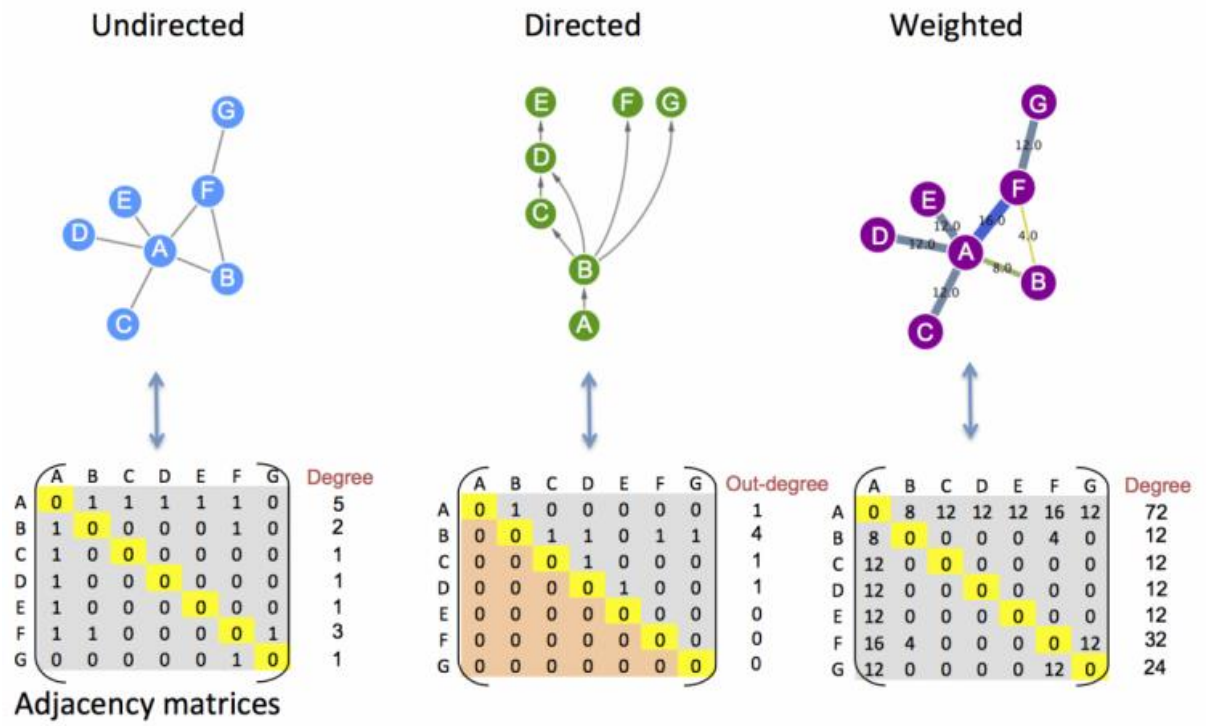
Adjacency matrices of directed graphs only cover one side of the diagonal line, since directed graphs have edges that go in only one direction.

An adjacency matrix can be "weighted", which basically means each edge has an associated value attached to it, so instead of 1s, the value is put in the respective matrix coordinate. These weights can represent anything you want. In the case of molecules for example, they may represent the type of bond between two nodes (atoms). In a social network like LinkedIn, they can represent 1st, 2nd, or 3rd, order connections between two nodes (people).

This concept of weights for edges is an attribute that makes GNNs so powerful; they allow us to take into account both **structural (dependent) and singular (independent) information.** For real world applications, this means we can consider external as well as internal information.

Degree Matrix (*D*):

The Degree Matrix of a graph can be found by using the concept of degrees covered earlier. ***D*** is essentially a diagonal matrix, where **each value of the diagonal is the degree of its corresponding node.**



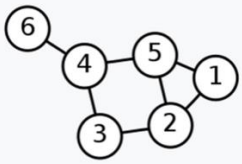
Notice that the degree is simply the sum of each row of the adjacency matrix. These degrees are then placed on the diagonal of the matrix (the line of symmetry for the adjacency matrix). This leads nicely to the final matrix:

Laplacian Matrix (L):

The Laplacian Matrix of a graph is the result of subtracting the Adjacency Matrix from the Degree Matrix:

$$L = D - A$$

Each value in the Degree Matrix is subtracted by its respective value in the Adjacency Matrix as such:

Labeled graph	Degree matrix	Adjacency matrix	Laplacian matrix
	$\begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 2 & -1 & 0 & 0 & -1 & 0 \\ -1 & 3 & -1 & 0 & -1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & -1 \\ -1 & -1 & 0 & -1 & 3 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}$

There exist other graph matrix representations like the **Incidence Matrix,** but the vast majority of GNN applications on graph type data utilize one, two, or all three of these matrices. This is because they, and the laplacian matrix in particular, provide substantial information about the **entities** (an element with attributes) and **relations** (a connection between entities).

The only thing missing is a **rule** (a function that maps entities to other entities via relations). This is where neural networks come in handy.

Deep Learning

Now let's do a quick run down of the other half of "Graph **Neural Networks**". Neural networks are the architecture we talk about when someone says "**Deep Learning**". The neural network architecture is built upon the concept of **perceptrons**, which are inspired by the neuron interactions in human brains.

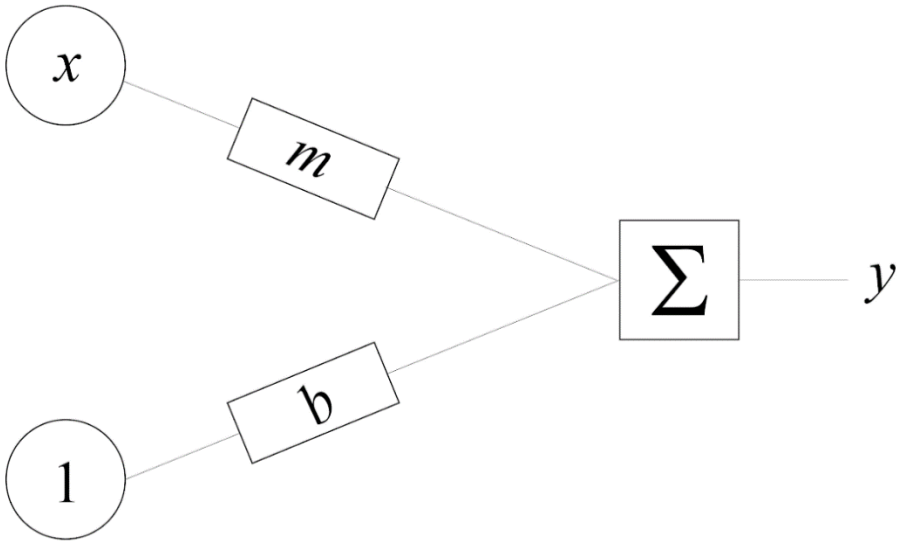
Artificial Neural Networks (or just NN for short) and its extended family, including Convolutional Neural Networks, Recurrent Neural Networks, and of course, Graph Neural Networks, are all types of Deep Learning algorithms.

Deep Learning is a type of machine learning algorithm, which in turn is a subset of artificial intelligence.

It all starts with the humble linear equation.

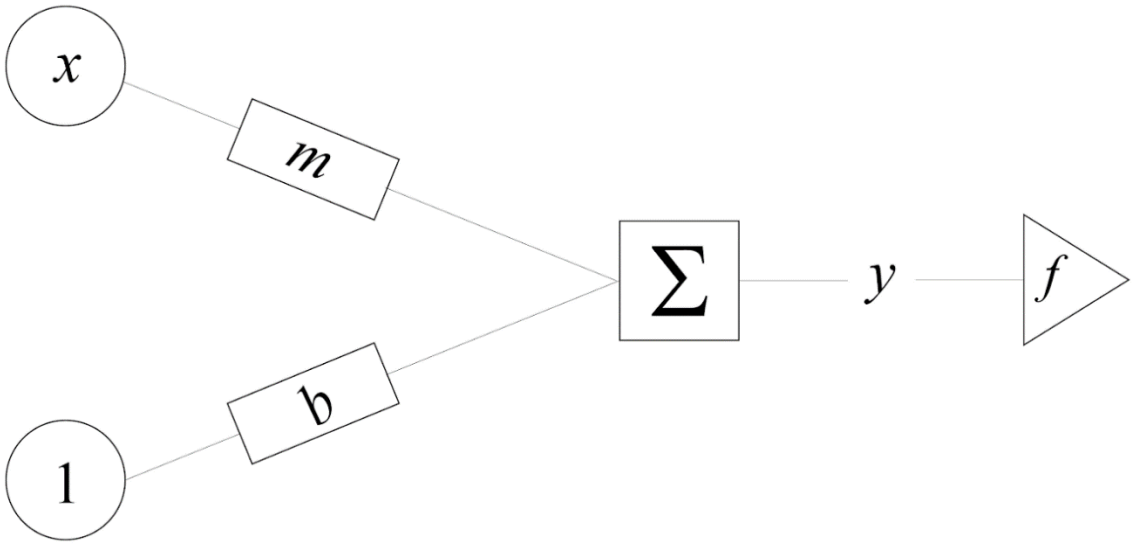
$y = mx + b$

If we structure this equation as a perceptron, we see:

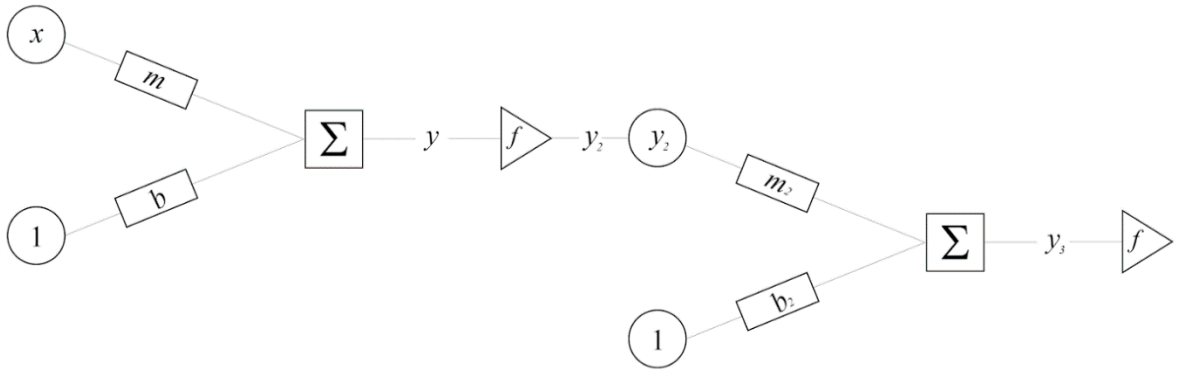


Where the **output (y) is the sum (E) of the bias (b) and the input (x) times the weight (m).**

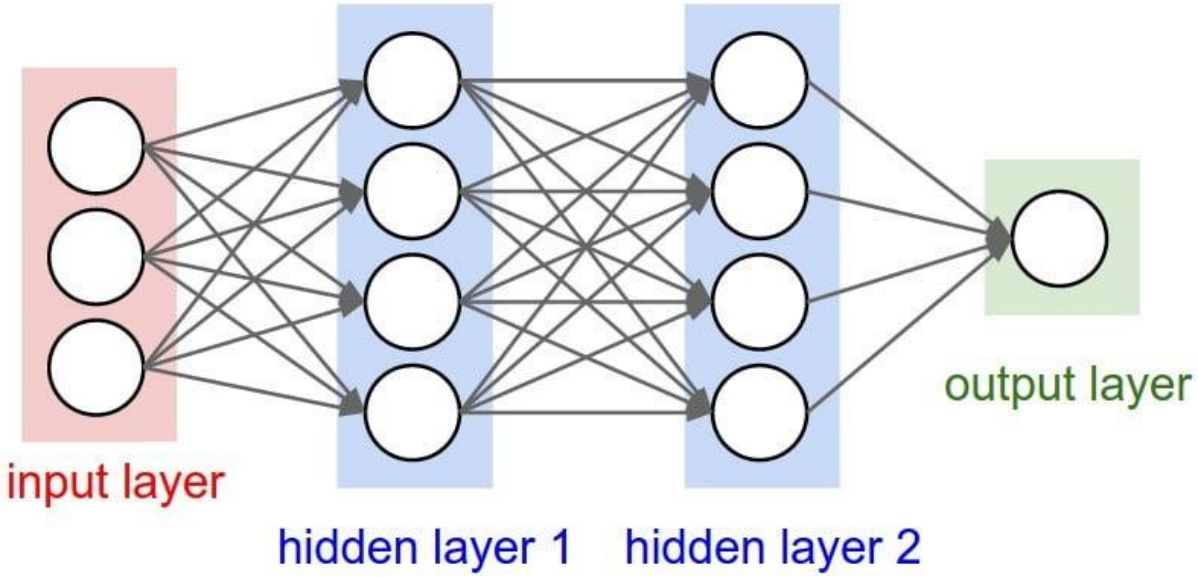
Neural networks usually have an **activation function**, which basically decides if a given neuron output (the *y*) should be considered as "activated", and keeps the output value of a perceptron within a reasonable and compute-able range. (sigmoid for 0–1, tanh for -1–1, ReLU for 0 OR 1, etc.). This is why we attach an activation function at the end of the perceptron.



When we put a bunch of perceptrons together, we get something that resembles the beginnings of a **neural network**! These perceptrons pass numerical values from one layer to another, with each pass bringing that numerical value closer to the target/label that the network is trained against.



When you put a bunch of perceptrons together, you get:



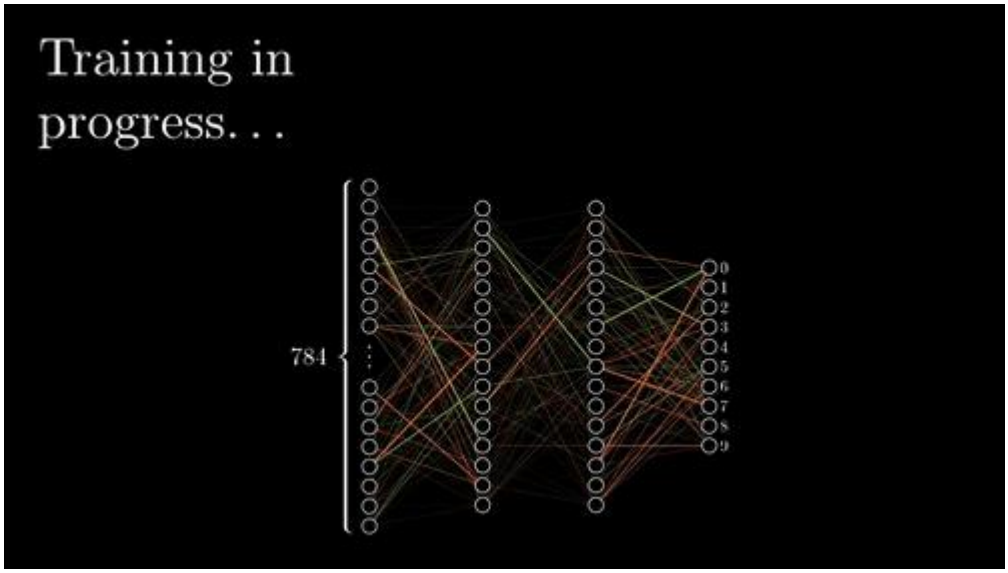
To train a neural network, we need to first calculate how much we need to adjust the model's weights. We do this with a **loss function**, which calculates the **error**.

$$e_i = Y_i - \hat{Y}_i$$

Where ***e*** is the error, ***Y*** is the expected output and $\hat{Y}$ is the actual output. At a high level, error is calculated as actual output (the NN's prediction) minus expected output (the target). The goal is to **minimize error**. Error is minimized by adjusting each of the layer's weights using a process known as **back propagation.**

Essentially back propagation distributes the adjustments throughout the network starting from the output layer to the input layer. The amount that is adjusted is determined by the **optimization function** which receives the error as input. The optimization function can be visualized as a ball rolling down a hill, with the ball's location being the error. Hence when the ball rolls to the bottom of the hill, the error is at its minimum.

Additionally there are some **hyperparameters** that must be defined, one of the most important of which is the **learning rate**. Learning rate adjusts the rate of which the optimization function is applied. Learning rate is like the gravity setting; the higher the gravity (higher the learning rate) the faster the ball rolls down the hill, and the same is true in reverse.



Neural networks have many different macro and micro customization that make each model unique, with varying levels of performance, but all of them are based off of this **vanilla** model.

Deep Learning is Graph Theory

Artificial Neural Networks are actually just graphs!

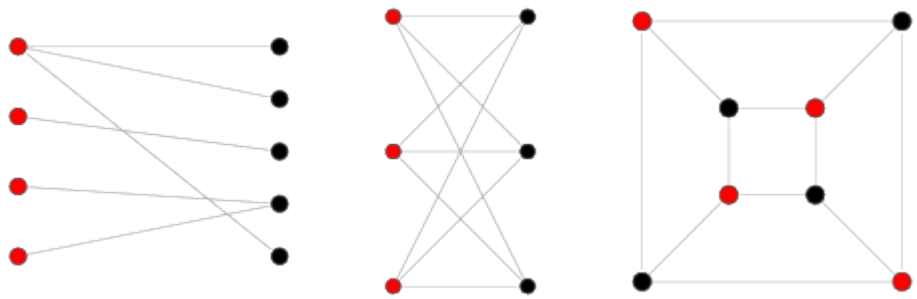
NNs are a special graph, but they have the same structure and therefore the same terminology, concepts, and rules.

Recall that the structure of a perceptron is essentially:

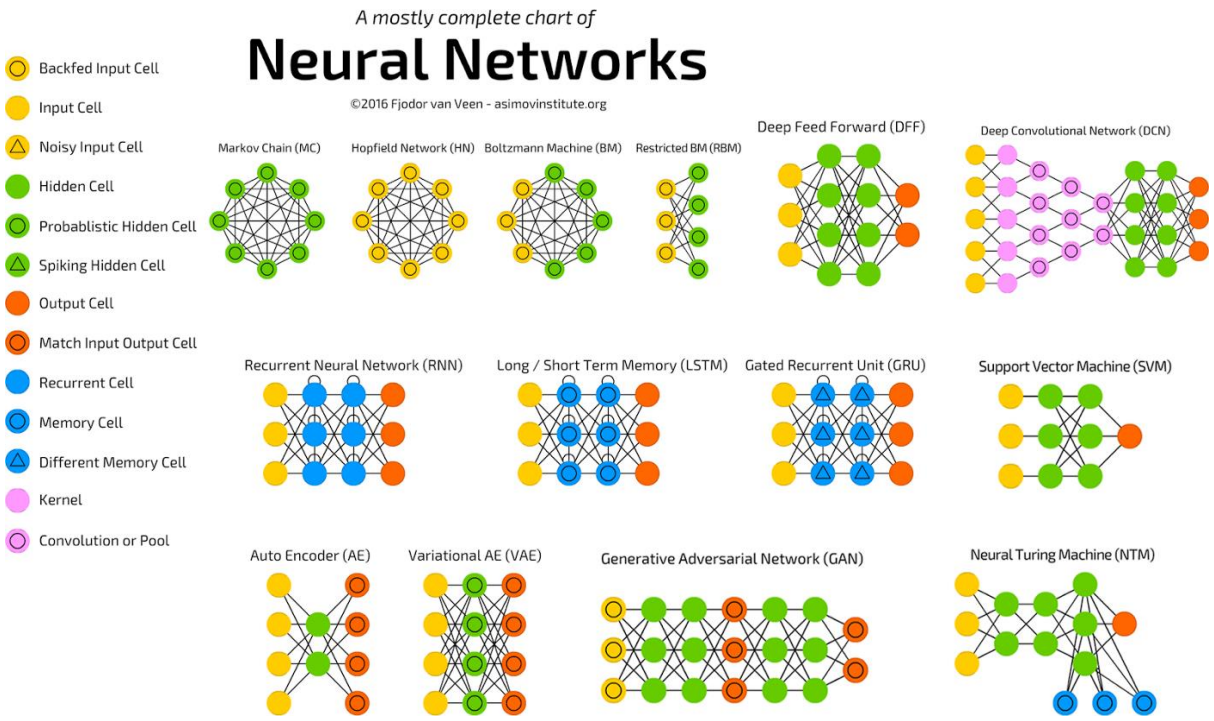
Picture of perceptron (modified)

We can think of the input value (x), the bias value (b), and the sum operation (E) as 3 nodes in a graph. We can think of the weight (m) as an edge connecting input value (x) to sum operation (E).

The specific type of graph that NNs most resemble are **multipartite graphs.** A multipartite graph is a graph that can be separated into different sets of nodes. **The nodes in each set can share edges between sets, but not within each set.**



Some neural networks even have fully connected nodes, conditional nodes, and other crazy architectures that give NNs their trademark versatility and power; here are some of the most popular architectures:



Each color corresponds to a different type of node, which can be arranged in a multitude of different ways. Propagating data forward or backwards through the network is analogous to **message passing in graphs**. The **edge or node features** in a graph are similar to the weights in a neural network. Notice some nodes even have the **self-loops** we mentioned earlier (native to RNNs — recurrent neural networks).

Neural networks aren't the only machine learning models to have a graph-like structure.

- K-means
- K-nearest neighbours
- Decision trees
- Random forests
- Markov chains

are all structured like graphs themselves, or output data in a graph structure.

Key Takeaways

There are many key takeaways, but the highlights are:

- All graphs have **properties that define the possible actions and limitations** for which it can be used or analyzed.
- **Graphs are represented computationally using various matrices**. Each matrix provides a different amount or type of information.

- **Deep learning is a subset of machine learning** that roughly mimics the way a human minds works using neurons.
- **Deep learning learns over iterations** by passing information forward through a network and propagating neuron adjustments backwards.
- Neural Networks (and other machine learning algorithms) have close ties with graph theory.

Biological Networks

Networks are widely used in many branches of biology as a convenient representation of patterns of interaction between appropriate biological elements. These biological networks include biochemical networks, neural networks, and ecological networks.

Biochemical networks

Biochemical networks represent the molecular-level patterns of interaction and mechanisms of control in the biological cell. The principal types of these networks are metabolic networks, protein-protein networks, and genetic regulatory networks.

Metabolic networks

Metabolism is the chemical process by which cells break down food and nutrients into usable building blocks and then reassemble those building block to form the biological molecules the cell needs to complete its other tasks. Typically this breakdown and reassembly involves chains or **pathways**, sets of successive chemical reactions that convert initial inputs into useful end products by a series of steps. The complete set of all reactions in all pathways forms the **metabolic network**.

The vertices in a metabolic network are chemicals produced and consumed by the reactions. These chemicals are known as **metabolites**. These are small molecules like carbohydrates, lipids, as well as amino acids and nucleotides. The metabolites consumed are called the **substrates** of the reaction, while those produced are called the **products**.

Most metabolic reactions do not occur spontaneously, or do so only at a very low rate. To make reactions occur at a usable rate, the cell employs an array of chemical catalysts, referred to as **enzymes**. Enzymes are not consumed in the reactions they catalyze. By increasing or decreasing the concentration of the enzyme that catalyzes a particular reaction, the cell can turn that reaction on or off, or moderate its speed. Enzymes tend to be highly specific to the reaction they catalyze.

The most correct representation of a metabolic network is as a **bipartite graph**. The two types of vertices represent metabolites and metabolic reactions, with edges joining each metabolite to the reaction in which it participates. The edges are directed, since some metabolites (the substrates) go into the reaction and some (the products) come out of it. Enzymes can be incorporated by adding a third class of vertex to represent them, with

undirected edges connecting them to the reactions they catalyze. The resulting graph is a **mixed (directed and undirected) tripartite network**.

Nevertheless, the most common representations of metabolic network **project** the bipartite network onto metabolite vertices. In one approach, the vertices in the network represent metabolites and there is an undirected edge between any two metabolites that participate in the same reaction, either as substrates or as products. Clearly this projection loses much of the information contained in the bipartite network. Another more informative approach is to represent the network as a directed graph with a single type of vertex representing metabolites and a directed edge from one metabolite to another if there is a reaction in which the first metabolite appears as a substrate and the second as a product. This representation is more reach in terms of information with respect to the undirected one but still loses the association of metabolites with reactions.

Protein-protein interaction networks

Proteins do interact with one another and with other biomolecules, both large and small, but the interactions are not purely chemical. **Proteins** are long-chain molecules formed by the concatenation of a series of basic units called **amino acids**. Once created, a protein does not stay in a loose chain-like form, but folds on itself in a folded form whose shape depends on the amino acid sequence. The folded form dictates the physical interaction it can have with other molecules. Hence, the primary mode of **protein-protein interaction** is physical rather than chemical, their complicated folded shapes interlocking to create so-called **protein complexes** but without the exchange of particles that defines chemical reactions.

In a **protein-protein interaction network** the vertices are proteins and two vertices are connected by an undirected edge if the corresponding protein interact. However, this representation omits useful information. Interactions that involve three or more proteins are represented by multiple edges, and there is no way to tell from the network itself that such edges represent aspects of the same interaction. This problem could be addressed by adopting a bipartite representation, with proteins and interactions as different types of vertex, and undirected edges connection proteins to the interactions in which they participate.

Genetic regulatory networks

The small molecules needed by biological organisms, such as sugars and fats, are manufactured in the cell by the chemical reactions of metabolism. Proteins, however, which are much larger molecules, are manufactured in a different manner, following recipes recorded in the cell's genetic material, DNA.

Proteins are long-chain molecules formed by the concatenation of amino acids. The protein amino acid sequence is determined by a corresponding sequence stored in the **DNA** of the cell in which the protein is synthesized. This is the primary function of DNA, to act as an information storage medium containing the sequences of proteins needed by the cell. DNA is itself made up of units called **nucleotides**, of which there are four distinct species, denoted A, C, G, and T. The amino acids in proteins are encoded in DNA as trios of consecutive nucleotides called **codons**, such as ACG and TTT, and a succession of codons spells out the complete sequence of amino acids in a protein. A single strand of DNA can code for many proteins, and two special codons, called the start and end codons, signal the beginning and

end of the sequence coding for a protein. The DNA code for a single protein, from start to stop codon, is called a **gene**.

Proteins are created in the cell by a mechanism that operates in two stages. In the first stage, known as **transcription**, an enzyme makes a copy of the coding sequence of a single gene. The copy is made of RNA, another information-bearing similar to DNA. In the second stage, called **translation**, the protein is assembled step by step from the RNA sequence. In the jargon of molecular biology, one says that the gene has been **expressed**.

The cell does not, in general, need to produce at all times every possible protein for which it contains a gene. Individual proteins serve specific purposes, such as catalyzing metabolic reactions, and it is important for the cell to respond to its environment by turning on or off the production of individual proteins. It does this by the use of **transcription factors**, which are themselves proteins. The presence in the cell of the transcription factor for the gene turns on or enhances the expression of that gene, or inhibits it, depending on the type of transcription factor (promoting and inhibiting).

Here comes the network. Being proteins, transcription factors are themselves produced by transcription from genes, and the transcription process is regulated by other transcription factors, which again are proteins, and so forth. The complete set of such interactions forms a **genetic regulatory network**. The vertices in this network are proteins, or equivalently the genes that code for them and a directed edge from gene A to gene B indicates that A regulates the expression of B. One can distinguish between promoting and inhibiting transcription factors, giving the network two distinct types of edges.

Neural networks

One of the main functions of the brain is to process information and the primary information processing element is the **neuron**, a specialized brain cell that combines several inputs to generate a single output.

A typical neuron consists of a cell body or **soma**, along with a number of protruding tentacles, called **dendrites**, which are input wires for carrying signals in the cell. Most neurons have only one output, called the **axon**, which is typically longer than the dendrites. It usually branches near its end into **axon terminals** to allow the output of the cell to feed the input of several others. There is a small gap, called **synapse**, between terminal and dendrite across which the output signal of the first (presynaptic) neuron must be conveyed to reach the second (postsynaptic) neuron.

The actual signals that travel within neurons are electrochemical in nature. They consists in travelling waves of electrical voltage created by the motion of positively charged ions in and out of the cell. These waves are called **action potentials**. When an action potential reaches a synapse, it cannot cross the gap between the axon terminal and the opposing dendrite and the signal is instead transmitted chemically. The arrival of the action potential stimulates the production of a chemical neuro-transmitter by the terminal, which diffuses across the gap and is detected by receptor molecules on the dendrite at the other side. This in turn causes ions to move in and out the dendrite, changing its voltage.

These voltage changes, however, do not yet give rise to another travelling wave. The soma of the postsynamptic neuron sums the inputs from its dendrites and as a result may or may not send an output signal down its own axon. Thus the neuron acts as a gate that aggregates the signals at its inputs and only fires when enough inputs are excited. Furthermore, inputs can also be inhibiting; signals received at inhibiting inputs make the receiving neuron less likely to fire.

Current science cannot tell us exactly how the brain performs the more sophisticated cognitive tasks that allow animals to survive, but it is known that the brain constantly changes the pattern of wiring between neurons in response to inputs and experience, and it is presumed that this pattern - the **neural network** - holds much of the secret.

At the simplest level, a neural network can be represented as a set of vertices, the neurons, connected by two types of directed edges, one for excitatory inputs and one for inhibiting inputs. In practice, neurons are not all the same. This variation can be encoded in our network representation by different types of vertices.

Ecological networks

Ecological networks are networks of ecological interactions between species. Species in an ecosystem can interact in different ways: they can eat one another, they can parasitize one another, or they can have any of a variety of mutually advantageous interactions, such as pollination or seed dispersal.

Food webs are the most studied ecological networks. The biological organisms on our planet can be divided into ecosystems, groups of organisms that interact with one another and with elements of their environment. A **food web** is a directed network that represents which species prey on which others in a given ecosystem. The vertices of the network correspond to species and the directed edges to predator-prey interactions. In fact, ecologists conventionally draw edges in the opposite direction, from prey to predator, that is, in the direction of energy flow.

Food webs are approximately **directed acyclic graphs** (DAGs). The acyclic nature of food webs indicate that there is an intrinsic hierarchy among the species in ecosystems. Those higher up the hierarchy prey on those lower down, but not vice versa, although there are some counterexamples. A species's position in this hierarchy is called by ecologists its **trophic level**. This is the **rank** of the species' vertex on the acyclic food graph, that is the length of the longest path leading to the vertex representing the species. Those species in lower trophic levels tend to be smaller and more abundant, while those in higher trophic positions are usually larger-bodied and less numerous predators.

Community food webs are complete food webs for an entire ecosystem. Source food webs and sink food webs are sub-networks of community food webs that focus on species connected to a specific prey or predator. In a **source food web**, one records all species that derive energy from a particular source species. A **sink food web**, on the other hand, specifies through which species the energy is, directly or indirectly, consumed by a given sink species. In some cases attempts have been made to measure not merely the presence of interactions between species but also their **strength**, for instance as the fraction of energy that a species

derives from each of its preys. The result is a weighted directed network that sheds more light on the flow of energy through an ecosystem.

Other ecological networks include host-parasite networks and mutualistic networks. **Host-parasite networks** are networks of parasitic relationships between organisms. Parasites tend to be smaller-bodied then their hosts and parasites can live off their hosts without killing them. Host-parasite networks are directed acyclic networks. **Mutualistic networks** are networks of mutually beneficial interactions between species. These include networks of plants and the animals (insects) that pollinate them, networks of plants and the animals (birds) that disperse their seeds, and networks of ant species and the plants that thy protect and eat. Mutualistic networks can be represented as undirected bipartite graphs.

Video

https://www.youtube.com/watch?v=G4oekeQMFmE

Implementing PageRank (Using Random Walk Method) part - 1

https://www.youtube.com/watch?v=PYECYKxBXNU

Implementing PageRank (Using Random Walk Method) part – 2

https://www.youtube.com/watch?v=E9aoTVmQvok

Lecture 11 — How we Really Compute PageRank | Stanford University

https://www.baeldung.com/cs/random-walk

Introduction to Random Walk

A random walk can be defined as a series of discrete steps an object takes in some direction. Moreover, we determine the direction and movement of the object in each step probabilistically. In mathematics and probability theory, a random work is a random process.

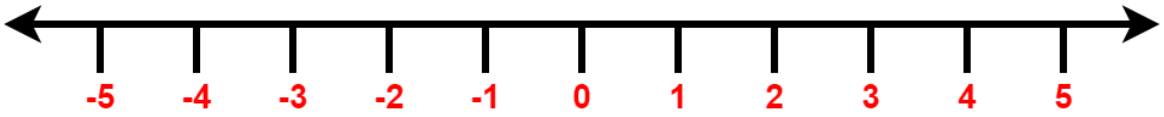
In a random walk, the future position is entirely independent of the current position of an object. Additionally, it's an example of the Markov process. Starting from a position, the object can go in any direction. Each step taken by the object in any direction has a probability associated with it. Hence, **the final position is completely independent of the point of origin.**

A simple example of a random walk is a drunkard's walk. A drunk man has no preferential direction. Therefore, he's equally likely to move in all directions.

In the random work concept, **the utmost significant problem is finding a probability distribution function** that can estimate the probability of the current position of an object after taking a random walk for a fixed amount of time.

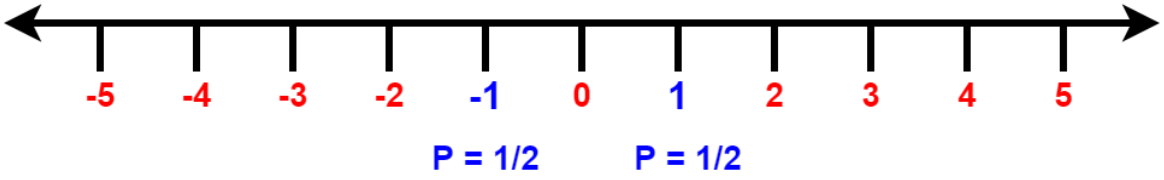
3. One Dimensional Random Walk

The simplest and basic random walk is a one-dimensional walk. **Let's look at a random walk on integers:**

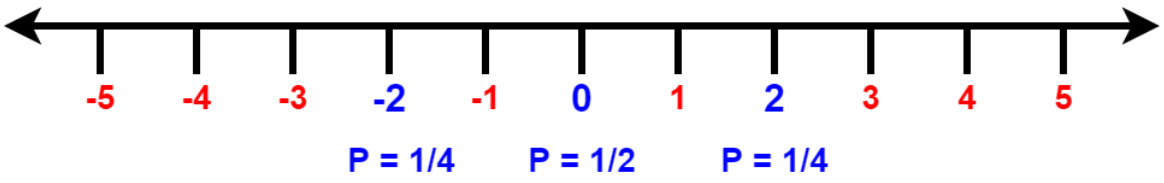


So here, an object is standing at point . It can move in two directions: forwards and backward. Now we'll decide the direction of each step of the object by flipping a coin. In the case of a head, the object will move forward. If it's a tail, the object will move backward. Here we'll flip a coin, move the object one step according to the rule and flip the coin again.

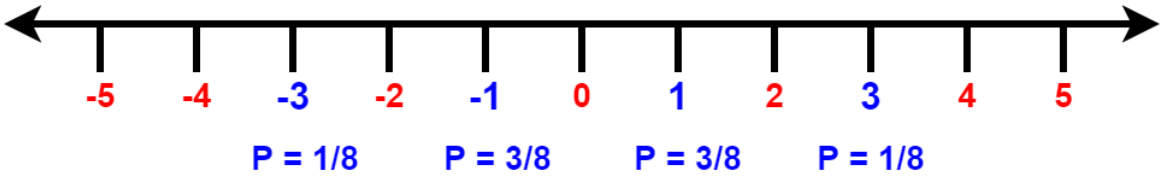
Now let's start the process. Let's take the first turn:



After the first turn, the object can either go to or position with equal probability . Now let's take the second turn:



After taking the second step, we can find the object in three positions: , or . Here, the probability associated with positions and is the same. Although with the probability of , the object might be standing on the position . Likewise, let's look at the probabilities at the third turn:



Here the object can be found in positions. The probability of finding the object in either position or is . However, the probability of finding the object in either position or is .

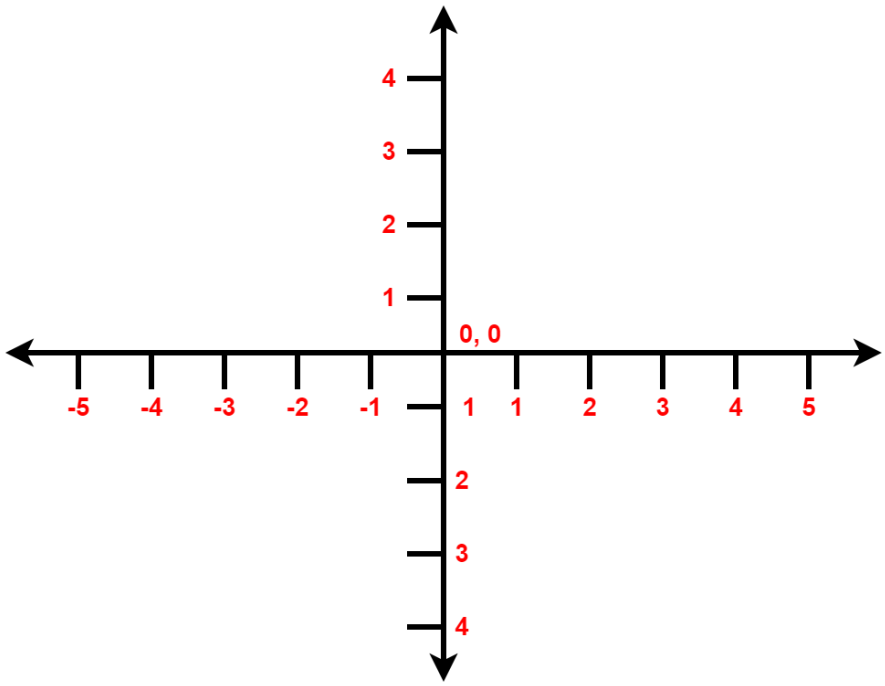
Now it's easy to observe that when the number of turns is odd, all possible positions where the object can go is odd. Similarly, in the case of even turns, all possible positions are even numbers.

Now in order to get more insights from the one-dimensional random walk, let's assume indicates the outcome of coin in the random process. Hence, is the outcome of the first coin flipped at the first turn. Additionally, the variable is known as the random variable.

Now in the case of a one-dimensional random walk, **the expected or average value of the random variable is always** .

4. Two Dimensional Random Walk

A random walk can happen in any dimensional. We'll discuss the random walk in a two-dimensional integer lattice here. **Let's look at a two-dimensional integer lattice:**



In a two-dimensional random walk, an object can move in four different directions: forward, backward, left, right. Therefore, in this environment, in order to move the object, we need to flip a coin twice at each step. We can decide whether to move the object forward or backward in the first flip. The second flip will determine whether to go in the right or left direction.

Interestingly, in the random walk, **the probability of reaching any point in the 2D grid is** when we set the number of steps to infinite.

5. Types

Let's now talk about the different types of random walks. **There're two types of random walk based on the position of an object: recurrent and transient.**

Recurrent random walks guarantee to return to the starting position starting from a random position. **One and two-dimensional random walk falls under this category.** We can verify this from the examples presented. Given a large number of steps, it's guaranteed that the object will return to its starting position.

In contrast to a recurrent random walk, a transient random walk doesn't guarantee to return to the starting position. In fact, in most cases, there's a positive probability that the walk will never return to its starting position.

A random walk in 3 or higher dimensions comes under the transient category. The huge number of choices for an object to move at each step makes it impossible to return to the starting position.

6. Application of Random Walk

There're many applications of random walk in mathematics, computer science, biology, chemistry, physics. In biological genetic drift, random walks can give us a general idea of the statistical processes involved. In physics, we can use them to describe an ideal chain in polymer physics.

The random work concept is also crucial and used in several fields such as psychology, finance, ecology. Moreover, **we can describe fluctuations in the share market with the random walk concept.** Additionally, Google search engine algorithms also use them.

7. Conclusion

In this tutorial, we discuss the concept of random walk in detail. We described random walks in one and two-dimensional with examples.

Finally, we presented various applications of random walk.

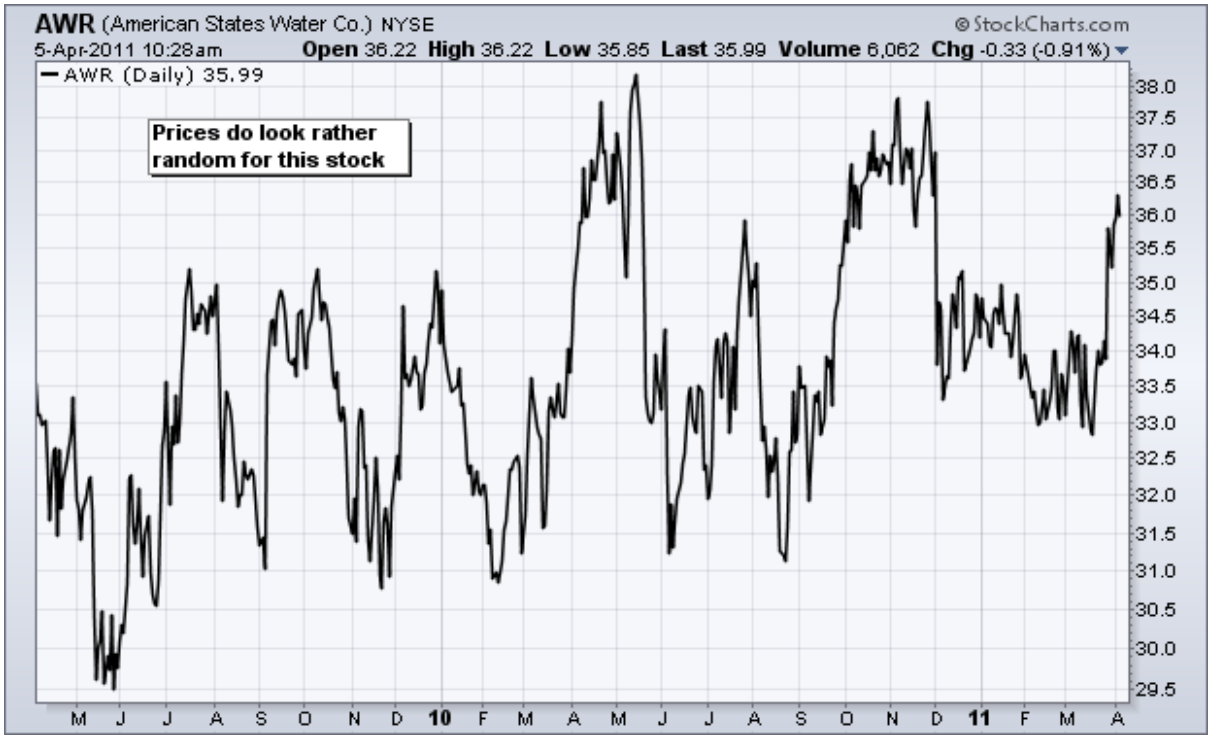
Random Walk Theory

A mathematical model of the stock market

What is the Random Walk Theory?

The Random Walk Theory, or the Random Walk Hypothesis, is a mathematical model of the stock market. Proponents of the theory believe that the prices of securities in the stock market evolve according to a random walk.

A “random walk” is a statistical phenomenon where a variable follows no discernible trend and moves seemingly at random. The random walk theory, as applied to trading, most clearly laid out by Burton Malkiel, an economics professor at Princeton University, posits that the price of securities moves randomly (hence the name of the theory) and that, therefore, any attempt to predict future price movement, either through fundamental or technical analysis, is futile.



The implication for traders is that it is impossible to outperform the overall market average other than by sheer chance. Those who subscribe to the random walk theory recommend using a “buy and hold” strategy, investing in a selection of stocks that represent the overall market – for example, an index mutual fund or ETF based on one of the broad stock market indexes, such as the S&P 500 Index.

Basic Assumptions of the Random Walk Theory

1. The Random Walk Theory assumes that the price of each security in the stock market follows a random walk.
2. The Random Walk Theory also assumes that the movement in the price of one security is independent of the movement in the price of another security.

Brief History of the Random Walk Theory

In 1863, a French mathematician turned stock broker named Jules Regnault published a book titled “Calcul des Chances et Philosophie de la Bourse” or “The Study of Chance and the Philosophy of Exchange.” Regnault’s work is considered one of the first attempts at the use of advanced mathematics in the analysis of the stock market.

Influenced by Regnault’s work, Louis Bachelier, another French mathematician, published a paper titled “Théorie de a Spéculation” or the “Theory of Speculation.” This paper is credited

with establishing the ground rules that would be key to the use of mathematics and statistics in the stock market.

In 1964, American financial economist Paul Cootner published a book entitled "The Random Character of Stock Market Prices." Considered a classic text in the field of financial economics, it inspired other works such as "A Random Walk Down Wall Street" by Burton Malkiel (another classic) and "Random Walks in Stock Market Prices" by Eugene Farma.

Implications of the Random Walk Theory

Since the Random Walk Theory posits that it is impossible to predict the movement of stock prices, it is also impossible for a stock market investor to outperform or "beat" the market in the long run. It implies that it is impossible for an investor to outperform the market without taking on large amounts of additional risk.

As such, the best strategy available to an investor is to invest in the market portfolio, i.e., a portfolio that bears a resemblance to the total stock market and whose price reflects perfectly the movement of the prices of every security in the market.

A flurry of recent performance studies reiterating the failure of most money managers to consistently outperform the overall market has indeed led to the creation of an ever-increasing number of passive index funds.

Also, it appears that an increasing number of investors are firm believers in the wisdom of index investing. According to data from Vanguard and Morningstar, 2016 saw an unprecedented inflow of more than $235 billion into index funds.

Random Walk Theory in Practice

In 1988, the Random Walk Theory was put to the test in the famous Dart Throwing Investment Contest. Devised by the Wall Street Journal, this contest pitted professional investors working out of the New York Stock Exchange against dummy investors. The dummy investors consisted of the Wall Street Journal staff who chose stocks by throwing darts at a board.

The experiment, titled "The Wall Street Journal Dartboard Contest," gained much fanfare and media attention. Out of 100 contests, the professional investors won 61, whereas the dart-throwing dummies won 39. However, the professional investors only beat the market (as represented by the performance of the Dow Jones Industrial Average) 51 times out of 100.

Criticism of the Random Walk Theory

One of the main criticisms of the Random Walk Theory is that the stock market consists of a large number of investors, and the amount of time each investor spends in the market is different. Thus, it is possible for trends to emerge in the prices of securities in the short run, and a savvy investor can outperform the market by strategically buying stocks when the price is low and selling stocks when the price is high within a short time span.

Other critics argue that the entire basis of the Random Walk Theory is flawed and that stock prices do follow patterns or trends, even over the long run. They argue that because the price

of a security is affected by an extremely large number of factors, it may be impossible to discern the pattern or trend followed by the price of that security. However, just because a pattern cannot be clearly identified, that doesn't mean that a pattern does not exist.

A Non-Random Walk

In contrast to the Random Walk Theory is the contention of believers in technical analysis – those who think that future price movements *can* be predicted based on trends, patterns, and historical price action. The implication arising from this point of view is that traders with superior market analysis and trading skills can significantly outperform the overall market average.

Both sides can present evidence to support their position, so it's up to each individual to choose what they believe. However, there is one fact – perhaps a decisive one – which goes against the random walk theory. This is the fact that there are some individual traders who consistently outperform the market average for long periods of time.

According to the Random Walk Theory, a trader should only be able to outperform the overall market average by chance or luck. It would allow for there to be *some* traders who, at any given point in time, would – purely by chance – be outperforming the market average.

However, what are the odds that the *same* traders would be "lucky" year in and year out for decades? Yet there are indeed such traders, people like Paul Tudor Jones, who have managed to generate significantly above-average trading returns on a consistent basis over a long span of time.

It's important to note that even the most devout believers in technical analysis – those who think that future price movements in the market can be predicted – don't believe that there's any way to *infallibly* predict future price action. It is more accurate to say that *probable* future price movement can be predicted by using technical analysis and that by trading based on such probabilities, it is possible to generate higher returns on investment.

Conclusion

So, who do you believe? If you believe in the Random Walk Theory, then you should just invest in a good ETF or mutual fund designed to mirror the performance of the S&P 500 Index and hope for an overall bull market.

If, on the other hand, you believe that price movements are not random, then you should be polishing your fundamental and/or technical analysis skills, confident that doing such work will pay off with superior profits through actively trading the market.

Random Walk (Implementation in Python

Introduction A random walk is a mathematical object, known as a stochastic or random process, that describes a path that consists of a succession of random steps on some mathematical space such as the integers. An elementary example of a random walk is the

random walk on the integer number line, which starts at 0 and at each step moves +1 or -1 with equal probability. Other examples include the path traced by a molecule as it travels in a liquid or a gas, the search path of a foraging animal, the price of a fluctuating stock and the financial status of a gambler can all be approximated by random walk models, even though they may not be truly random in reality. As illustrated by those examples, random walks have applications to many scientific fields including ecology, psychology, computer science, physics, chemistry, biology as well as economics. Random walks explain the observed behaviors of many processes in these fields and thus serve as a fundamental model for the recorded stochastic activity. As a more mathematical application, the value of pi can be approximated by the usage of random walk in the agent-based modeling environment.

Enough with the boring theory. Let's take a break while getting some knowledge of the code. So, to code out the random walk we will basically require some libraries in python some to do maths, and some others to plot the curve.

Libraries Required

- **matplotlib** It's an external library that helps you to plot the curve. To install this library type the following code in your cmd.

```
pip install matplotlib
```

It would be enough to get you through the installation.

- **numpy** It's also an external library in python it helps you to work with arrays and matrices. To install the library type the following code in cmd.

```
pip install numpy
```

- **random** It's a built-in library of python we will use to generate random points.

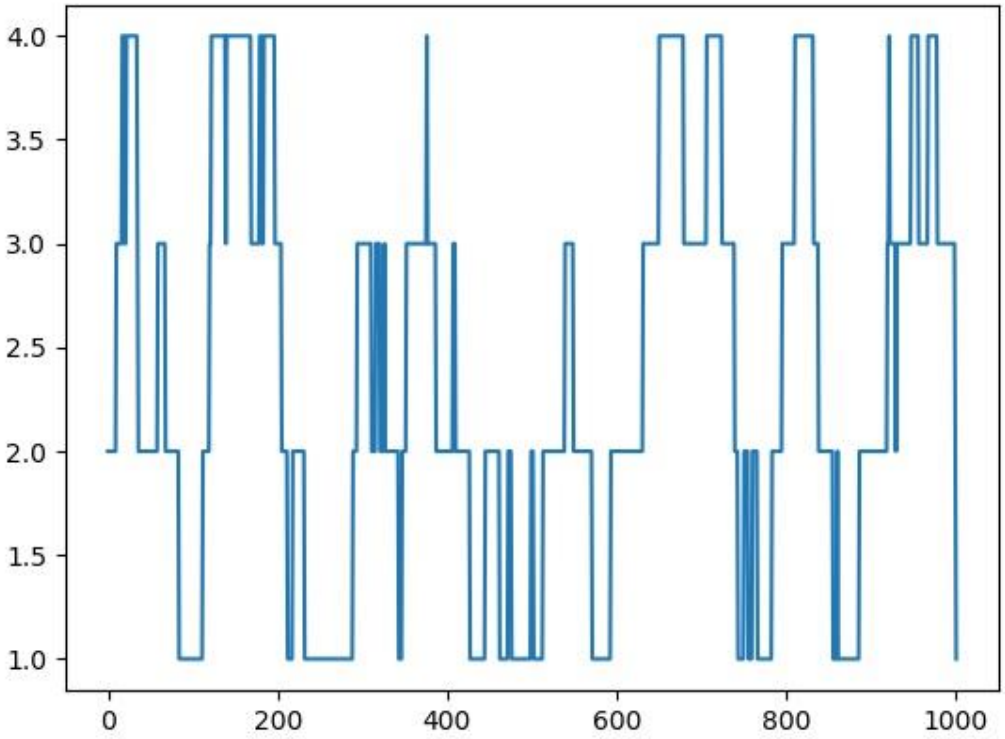
One-dimensional random walk An elementary example of a random walk is the random walk on the integer number line, which starts at 0 and at each step moves +1 or ? 1 with equal probability.

So let's try to implement the 1-D random walk in python.

```
# Python code for 1-D random walk.

import random

import numpy as np

import matplotlib.pyplot as plt


# Probability to move up or down

prob = [0.05, 0.95]


# statically defining the starting position
```

```
start = 2
positions = [start]

# creating the random points
rr = np.random.random(1000)
downp = rr < prob[0]
upp = rr > prob[1]



for idownp, iupp in zip(downp, upp):
    down = idownp and positions[-1] > 1
    up = iupp and positions[-1] < 4
    positions.append(positions[-1] - down + up)

# plotting down the graph of the random walk in 1D
plt.plot(positions)
plt.show()
```

Output:



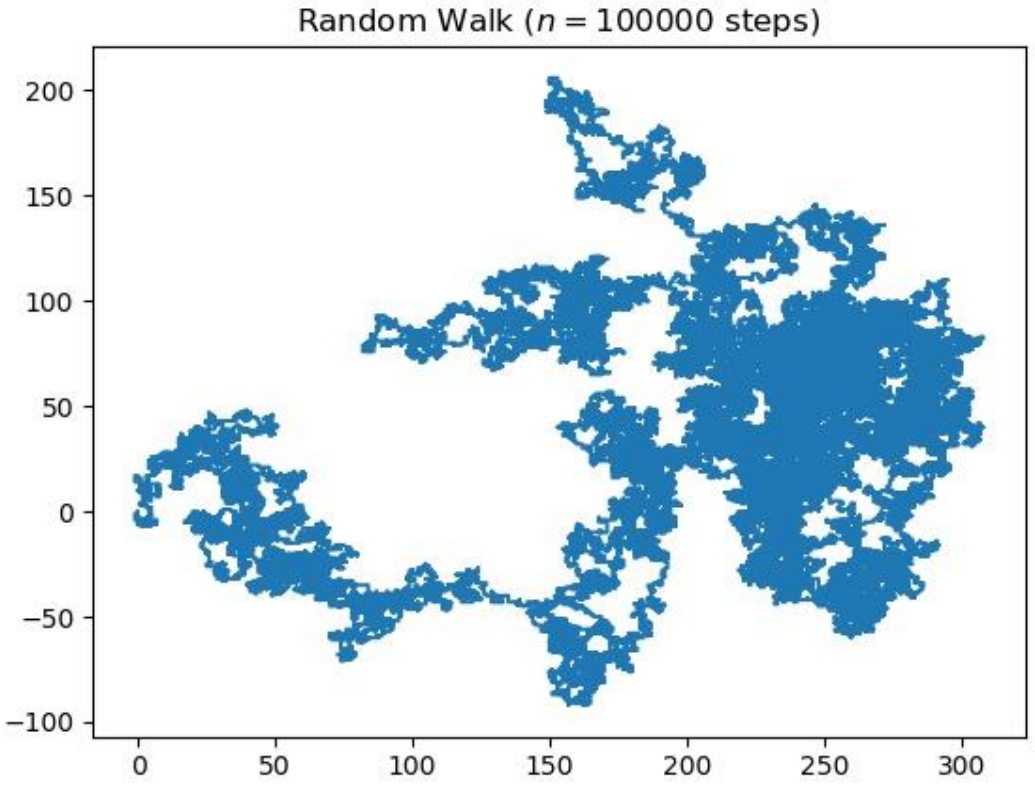
Higher dimensions In higher dimensions, the set of randomly walked points has interesting geometric properties. In fact, one gets a discrete fractal, that is, a set that exhibits stochastic self-similarity on large scales. On small scales, one can observe "jaggedness" resulting from the grid on which the walk is performed. Two books of Lawler referenced below are good sources on this topic. The trajectory of a random walk is the collection of points visited, considered as a set with disregard to when the walk arrived at the point. In one dimension, the trajectory is simply all points between the minimum height and the maximum height the walk achieved (both are, on average, on the order of ? n).

Let's try to create a random walk in 2D.

```
# Python code for 2D random walk.

import numpy

import pylab

import random


# defining the number of steps

n = 100000


#creating two array for containing x and y coordinate
```

```
#of size equals to the number of size and filled up with 0's
x = numpy.zeros(n)
y = numpy.zeros(n)

# filling the coordinates with random variables
for i in range(1, n):
    val = random.randint(1, 4)
    if val == 1:
        x[i] = x[i - 1] + 1
        y[i] = y[i - 1]
    elif val == 2:
        x[i] = x[i - 1] - 1
        y[i] = y[i - 1]
    elif val == 3:
        x[i] = x[i - 1]
        y[i] = y[i - 1] + 1
    else:
        x[i] = x[i - 1]
        y[i] = y[i - 1] - 1



# plotting stuff:
pylab.title("Random Walk ($n = " + str(n) + "$ steps)")
pylab.plot(x, y)
pylab.savefig("rand_walk"+str(n)+".png",bbox_inches="tight",dpi=600)
pylab.show()
```

Output:



Applications

1. In computer networks, random walks can model the number of transmission packets buffered at a server.
2. In population genetics, a random walk describes the statistical properties of genetic drift.
3. In image segmentation, random walks are used to determine the labels (i.e., “object” or “background”) to associate with each pixel.
4. In brain research, random walks and reinforced random walks are used to model cascades of neuron firing in the brain.
5. Random walks have also been used to sample massive online graphs such as online social networks.

Page Rank Algorithm and Implementation

PageRank (PR) is an algorithm used by Google Search to rank websites in their search engine results. PageRank was named after Larry Page, one of the founders of Google. PageRank is a way of measuring the importance of website pages. According to Google:

PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.

It is not the only algorithm used by Google to order search engine results, but it is the first algorithm that was used by the company, and it is the best-known.
The above centrality measure is not implemented for multi-graphs.

Algorithm
The PageRank algorithm outputs a probability distribution used to represent the likelihood that a person randomly clicking on links will arrive at any particular page. PageRank can be calculated for collections of documents of any size. It is assumed in several research papers that the distribution is evenly divided among all documents in the collection at the beginning of the computational process. The PageRank computations require several passes, called "iterations", through the collection to adjust approximate PageRank values to more closely reflect the theoretical true value.

Simplified algorithm
Assume a small universe of four web pages: A, B, C, and D. Links from a page to itself, or multiple outbound links from one single page to another single page, are ignored. PageRank is initialized to the same value for all pages. In the original form of PageRank, the sum of PageRank over all pages was the total number of pages on the web at that time, so each page in this example would have an initial value of 1. However, later versions of PageRank, and the remainder of this section, assume a probability distribution between 0 and 1. Hence the initial value for each page in this example is 0.25.
The PageRank transferred from a given page to the targets of its outbound links upon the next iteration is divided equally among all outbound links.
If the only links in the system were from pages B, C, and D to A, each link would transfer 0.25 PageRank to A upon the next iteration, for a total of 0.75.

Suppose instead that page B had a link to pages C and A, page C had a link to page A, and page D had links to all three pages. Thus, upon the first iteration, page B would transfer half of its existing value, or 0.125, to page A and the other half, or 0.125, to page C. Page C would transfer all of its existing value, 0.25, to the only page it links to, A. Since D had three outbound links, it would transfer one-third of its existing value, or approximately 0.083, to A. At the completion of this iteration, page A will have a PageRank of approximately 0.458.

In other words, the PageRank conferred by an outbound link is equal to the document's own PageRank score divided by the number of outbound links L( ).

In the general case, the PageRank value for any page u can be expressed as:

,
i.e. the PageRank value for a page u is dependent on the PageRank values for each page v contained in the set Bu (the set containing all pages linking to page u), divided by the number L(v) of links from page v. The algorithm involves a damping factor for the calculation of the PageRank. It is like the income tax which the govt extracts from one despite paying him itself.

Following is the code for the calculation of the Page rank.

```
def pagerank(G, alpha=0.85, personalization=None,
             max_iter=100, tol=1.0e-6, nstart=None, weight='weight',
             dangling=None):
    """Return the PageRank of the nodes in the graph.

    PageRank computes a ranking of the nodes in the graph G based on
    the structure of the incoming links. It was originally designed as
    an algorithm to rank web pages.

    Parameters
    ----------
    G : graph
      A NetworkX graph.  Undirected graphs will be converted to a directed
      graph with two directed edges for each undirected edge.

    alpha : float, optional
      Damping parameter for PageRank, default=0.85.

    personalization: dict, optional
      The "personalization vector" consisting of a dictionary with a
      key for every graph node and nonzero personalization value for each
node.
      By default, a uniform distribution is used.

    max_iter : integer, optional
      Maximum number of iterations in power method eigenvalue solver.
```

```
    tol : float, optional
      Error tolerance used to check convergence in power method solver.

    nstart : dictionary, optional
      Starting value of PageRank iteration for each node.

    weight : key, optional
      Edge data key to use as weight.  If None weights are set to 1.

    dangling: dict, optional
      The outedges to be assigned to any "dangling" nodes, i.e., nodes without
      any outedges. The dict key is the node the outedge points to and the dict
      value is the weight of that outedge. By default, dangling nodes are given
      outedges according to the personalization vector (uniform if not
      specified). This must be selected to result in an irreducible transition
      matrix (see notes under google_matrix). It may be common to have the
      dangling dict to be the same as the personalization dict.

    Returns
    -------
    pagerank : dictionary
       Dictionary of nodes with PageRank as value

    Notes
    -----
    The eigenvector calculation is done by the power iteration method
    and has no guarantee of convergence.  The iteration will stop
    after max_iter iterations or an error tolerance of
    number_of_nodes(G)*tol has been reached.

    The PageRank algorithm was designed for directed graphs but this
```

```
algorithm does not check if the input graph is directed and will
execute on undirected graphs by converting each edge in the
directed graph to two edges.



"""
if len(G) == 0:
    return {}

if not G.is_directed():
    D = G.to_directed()
else:
    D = G

# Create a copy in (right) stochastic form
W = nx.stochastic_graph(D, weight=weight)
N = W.number_of_nodes()

# Choose fixed starting vector if not given
if nstart is None:
    x = dict.fromkeys(W, 1.0 / N)
else:
    # Normalized nstart vector
    s = float(sum(nstart.values()))
    x = dict((k, v / s) for k, v in nstart.items())

if personalization is None:

    # Assign uniform personalization vector if not given
    p = dict.fromkeys(W, 1.0 / N)
else:
    missing = set(G) - set(personalization)
    if missing:
```

```
            raise NetworkXError('Personalization dictionary '
                                'must have a value for every node. '
                                'Missing nodes %s' % missing)
        s = float(sum(personalization.values()))
        p = dict((k, v / s) for k, v in personalization.items())

    if dangling is None:

        # Use personalization vector if dangling vector not specified
        dangling_weights = p
    else:
        missing = set(G) - set(dangling)
        if missing:
            raise NetworkXError('Dangling node dictionary '
                                'must have a value for every node. '
                                'Missing nodes %s' % missing)
        s = float(sum(dangling.values()))
        dangling_weights = dict((k, v/s) for k, v in dangling.items())
    dangling_nodes = [n for n in W if W.out_degree(n, weight=weight) == 0.0]

    # power iteration: make up to max_iter iterations
    for _ in range(max_iter):
        xlast = x
        x = dict.fromkeys(xlast.keys(), 0)
        danglesum = alpha * sum(xlast[n] for n in dangling_nodes)
        for n in x:

            # this matrix multiply looks odd because it is
            # doing a left multiply x^T=xlast^T*W
            for nbr in W[n]:
                x[nbr] += alpha * xlast[n] * W[n][nbr][weight]
            x[n] += danglesum * dangling_weights[n] + (1.0 - alpha) * p[n]
```

```
        # check convergence, l1 norm
        err = sum([abs(x[n] - xlast[n]) for n in x])
        if err < N*tol:
            return x
    raise NetworkXError('pagerank: power iteration failed to converge '
                        'in %d iterations.' % max_iter)
```

The above code is the function that has been implemented in the networkx library.

To implement the above in networkx, you will have to do the following:

```
>>> import networkx as nx
>>> G=nx.barabasi_albert_graph(60,41)
>>> pr=nx.pagerank(G,0.4)
>>> pr
```

Below is the output, you would obtain on the IDLE after required installations.

```
{0: 0.012774147598875784, 1: 0.013359655345577266, 2: 0.013157355731377924,
3: 0.012142198569313045, 4: 0.013160014506830858, 5: 0.012973342862730735,
 6: 0.012166706783753325, 7: 0.011985935451513014, 8: 0.012973502696061718,
9: 0.013374146193499381, 10: 0.01296354505412387, 11: 0.013163220326063332,
 12: 0.013368514624403237, 13: 0.013169335617283102, 14:
0.012752071800520563,
15: 0.012951601882210992, 16: 0.013776032065400283, 17:
0.012356820581336275,
18: 0.013151652554311779, 19: 0.012551059531065245, 20:
0.012583415756427995,
 21: 0.013574117265891684, 22: 0.013167552803671937, 23:
0.013165528583400423,
 24: 0.012584981049854336, 25: 0.013372989228254582, 26:
0.012569416076848989,
 27: 0.013165322299539031, 28: 0.012954300960607157, 29:
0.012776091973397076,
 30: 0.012771016515779594, 31: 0.012953404860268598, 32:
0.013364947854005844,
33: 0.012370004022947507, 34: 0.012977539153099526, 35:
0.013170376268827118,
 36: 0.012959579020039328, 37: 0.013155319659777197, 38:
0.013567147133137161,
```

```
 39: 0.012171548109779459, 40: 0.01296692767996657, 41:
0.028089802328702826,

 42: 0.027646981396639115, 43: 0.027300188191869485, 44:
0.02689771667021551,

 45: 0.02650459107960327, 46: 0.025971186884778535, 47:
0.02585262571331937,

48: 0.02565482923824489, 49: 0.024939722913691394, 50: 0.02458271197701402,

 51: 0.024263128557312528, 52: 0.023505217517258568, 53:
0.023724311872578157,

 54: 0.02312908947188023, 55: 0.02298716954828392, 56: 0.02270220663300396,

 57: 0.022060403216132875, 58: 0.021932442105075004, 59:
0.021643288632623502}
```

The above code has been run on IDLE(Python IDE of windows). You would need to download the networkx library before you run this code. The part inside the curly braces represents the output. It is almost similar to Ipython(for Ubuntu users).

Thus, this way the centrality measure of Page Rank is calculated for the given graph. This way we have covered 2 centrality measures. I would like to write further on the various centrality measures used for the network analysis.

Implementation of Page Rank using Random Walk method in Python

In Social Networks page rank is a very important topic. Basically page rank is nothing but how webpages are ranked according to its importance and relevance of search. All search engines use page ranking. Google is the best example that uses page rank using the web graph.

Random Walk Method – In the random walk method we will choose 1 node from the graph uniformly at random. After choosing the node we will look at its neighbors and choose a neighbor uniformly at random and continue these iterations until convergence is reached. After N iterations a point will come after which there will be no change In points of every node. This situation is called convergence.

Algorithm: Below are the steps for implementing the Random Walk method.

1. Create a directed graph with N nodes.
2. Now perform a random walk.
3. Now get sorted nodes as per points during random walk.
4. At last, compare it with the inbuilt PageRank method.

Below is the python code for the implementation of the points distribution algorithm.

```
import networkx as nx

import random
```

```
import numpy as np

# Add directed edges in graph
def add_edges(g, pr):
    for each in g.nodes():
        for each1 in g.nodes():
            if (each != each1):
                ra = random.random()
                if (ra < pr):
                    g.add_edge(each, each1)
                else:
                    continue
    return g

# Sort the nodes
def nodes_sorted(g, points):
    t = np.array(points)
    t = np.argsort(-t)
    return t

# Distribute points randomly in a graph
def random_Walk(g):
    rwp = [0 for i in range(g.number_of_nodes())]
    nodes = list(g.nodes())
    r = random.choice(nodes)
    rwp[r] += 1
    neigh = list(g.out_edges(r))
    z = 0

    while (z != 10000):
        if (len(neigh) == 0):
            focus = random.choice(nodes)
        else:
            r1 = random.choice(neigh)
```

```
            focus = r1[1]
        rwp[focus] += 1
        neigh = list(g.out_edges(focus))
        z += 1
    return rwp


# Main
# 1. Create a directed graph with N nodes
g = nx.DiGraph()
N = 15
g.add_nodes_from(range(N))

# 2. Add directed edges in graph
g = add_edges(g, 0.4)

# 3. perform a random walk
points = random_Walk(g)

# 4. Get nodes rank according to their random walk points
sorted_by_points = nodes_sorted(g, points)
print("PageRank using Random Walk Method")
print(sorted_by_points)

# p_dict is dictionary of tuples
p_dict = nx.pagerank(g)
p_sort = sorted(p_dict.items(), key=lambda x: x[1], reverse=True)

print("PageRank using inbuilt pagerank method")
for i in p_sort:
    print(i[0], end=", ")
```

Output:

```
PageRank using Random Walk Method
```

```
[ 9 10  4  6  3  8 13 14  0  7  1  2  5 12 11]
PageRank using inbuilt pagerank method
9, 10, 6, 3, 4, 8, 13, 0, 14, 7, 1, 2, 5, 12, 11,
```

Video

https://www.youtube.com/watch?v=k0uxnVEuuz0

Lecture 24 — Community Detection in Graphs - Motivation | Stanford University

https://www.youtube.com/watch?v=c0_vNfNZ4JM

Lecture 28 — Detecting Communities as Clusters (Advanced) | Stanford University

Graph similarity involves **determining the degree of similarity between these two graphs** (a number between 0 and 1). Intuitively, since we know the node correspondences, the same node in both graphs would be similar if its neighbors are similar (and its connectivity, in terms of edge weights, to its neighbors)

https://neo4j.com/docs/graph-data-science/current/management-ops/graph-catalog-ops/

https://www.youtube.com/watch?v=Twxbku8PZ3U

Graph Similarity and its Applications to Hardware Security

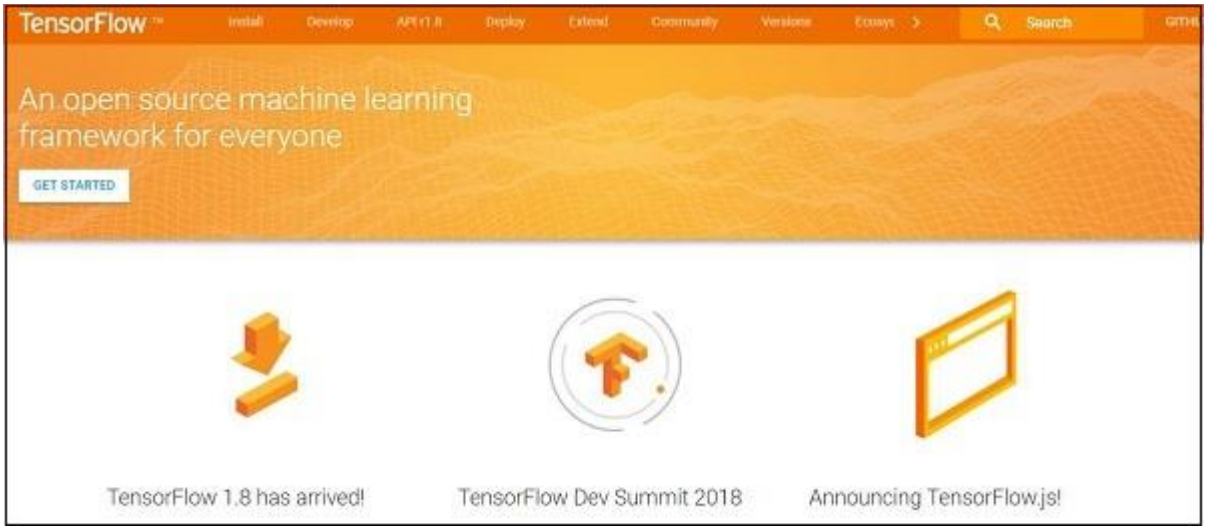
https://www.youtube.com/watch?v=VE-UrwNOaEE&list=PLNxJ2KOMQPGNtbOCvydglXwZMQTsqRGmo

Play List

TensorFlow is a software library or framework, designed by the Google team to implement machine learning and deep learning concepts in the easiest manner. It combines the computational algebra of optimization techniques for easy calculation of many mathematical expressions.

The official website of TensorFlow is mentioned below −

www.tensorflow.org



Let us now consider the following important features of TensorFlow −

- It includes a feature of that defines, optimizes and calculates mathematical expressions easily with the help of multi-dimensional arrays called tensors.
- It includes a programming support of deep neural networks and machine learning techniques.
- It includes a high scalable feature of computation with various data sets.
- TensorFlow uses GPU computing, automating management. It also includes a unique feature of optimization of same memory and the data used.

Why is TensorFlow So Popular?

TensorFlow is well-documented and includes plenty of machine learning libraries. It offers a few important functionalities and methods for the same.

TensorFlow is also called a “Google” product. It includes a variety of machine learning and deep learning algorithms. TensorFlow can train and run deep neural networks for handwritten digit classification, image recognition, word embedding and creation of various sequence models.

Tensor Data Structure

Tensors are used as the basic data structures in TensorFlow language. Tensors represent the connecting edges in any flow diagram called the Data Flow Graph. Tensors are defined as multidimensional array or list.

Tensors are identified by the following three parameters −

Rank

Unit of dimensionality described within tensor is called rank. It identifies the number of dimensions of the tensor. A rank of a tensor can be described as the order or n-dimensions of a tensor defined.

Shape

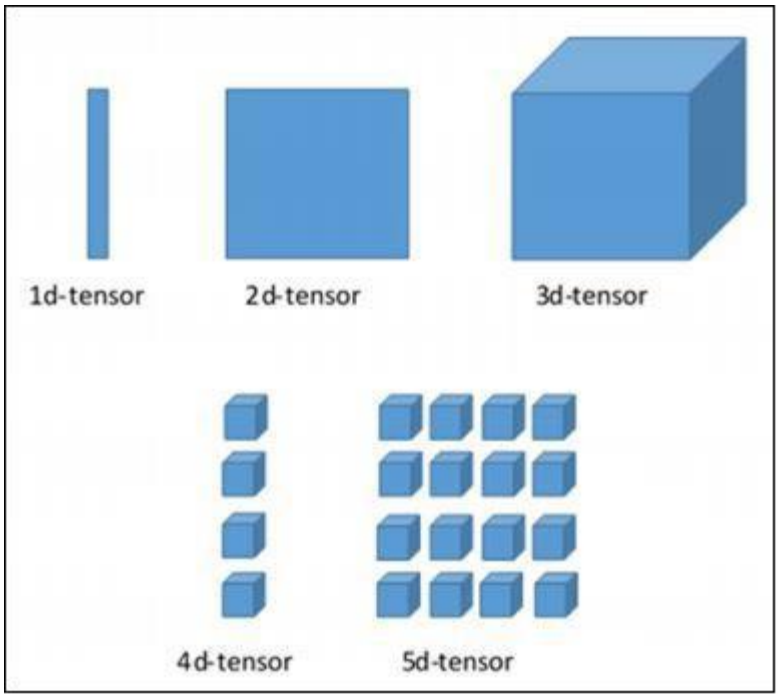
The number of rows and columns together define the shape of Tensor.

Type

Type describes the data type assigned to Tensor's elements.

A user needs to consider the following activities for building a Tensor −

- Build an n-dimensional array
- Convert the n-dimensional array.



Various Dimensions of TensorFlow

TensorFlow includes various dimensions. The dimensions are described in brief below −

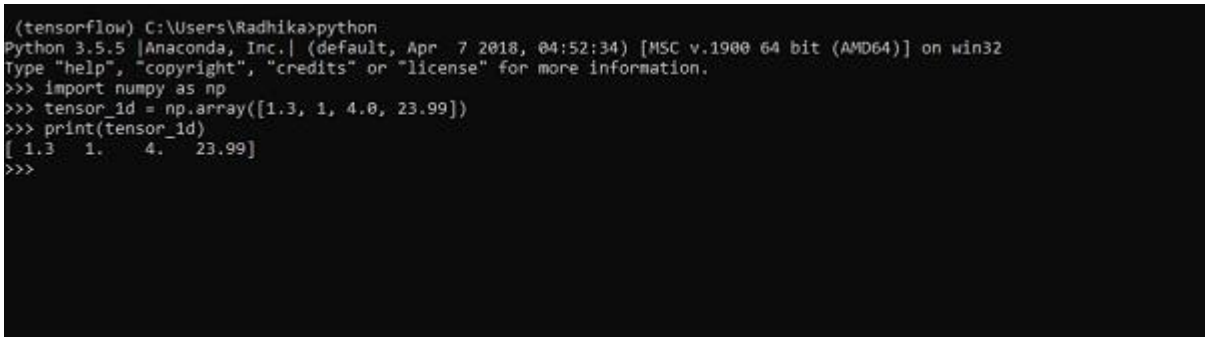
One dimensional Tensor

One dimensional tensor is a normal array structure which includes one set of values of the same data type.

Declaration

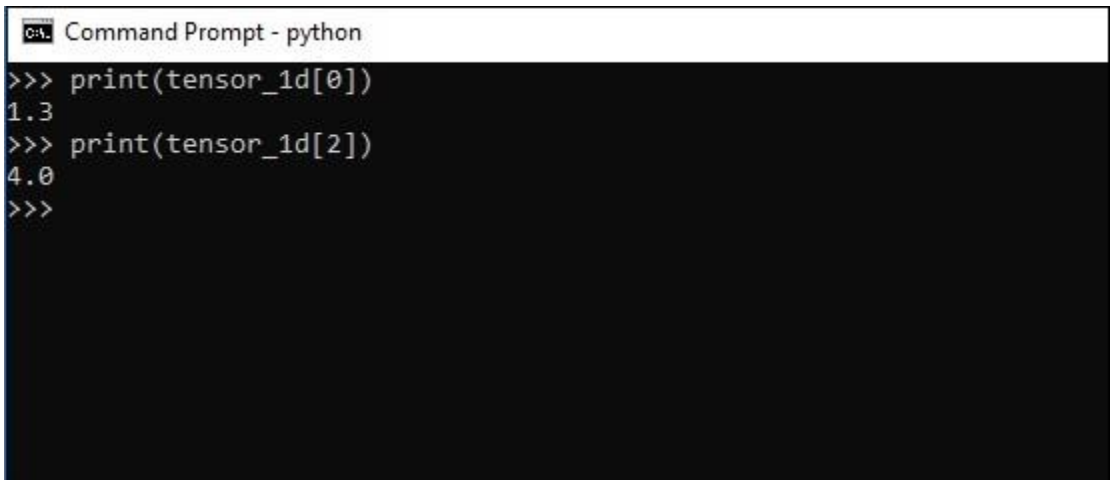
```
>>> import numpy as np
>>> tensor_1d = np.array([1.3, 1, 4.0, 23.99])
>>> print tensor_1d
```

The implementation with the output is shown in the screenshot below −



The indexing of elements is same as Python lists. The first element starts with index of 0; to print the values through index, all you need to do is mention the index number.

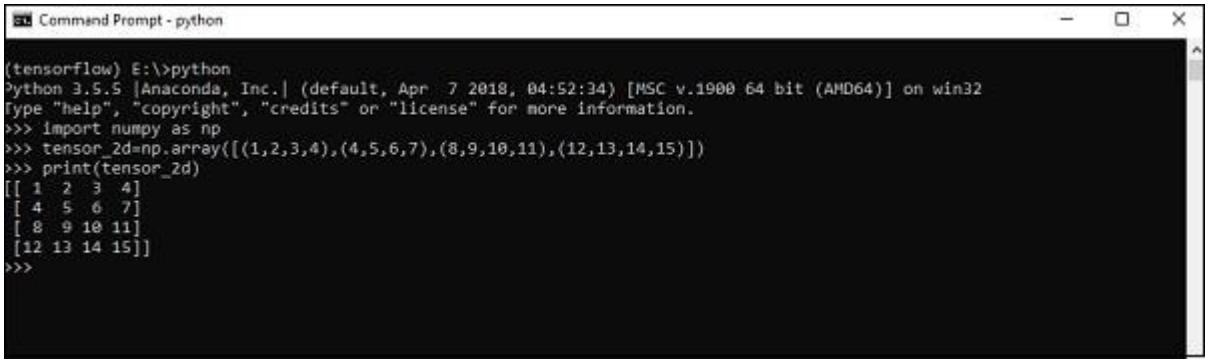
```
>>> print tensor_1d[0]
1.3
>>> print tensor_1d[2]
4.0
```



Two dimensional Tensors

Sequence of arrays are used for creating “two dimensional tensors”.

The creation of two-dimensional tensors is described below −

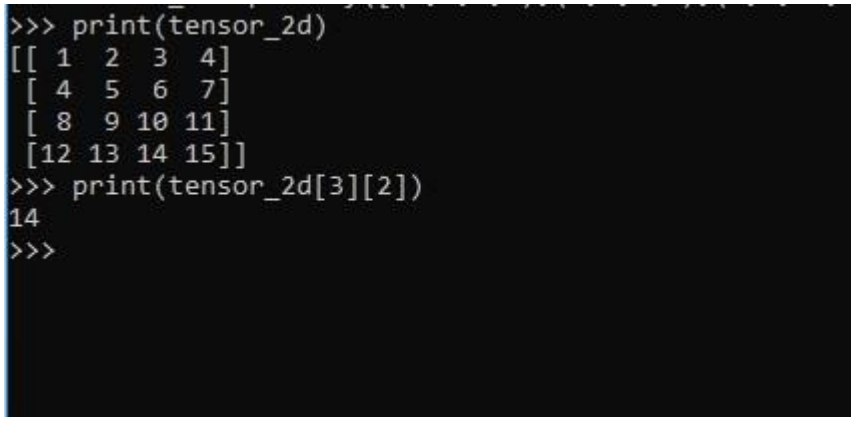


Following is the complete syntax for creating two dimensional arrays −

```
>>> import numpy as np
>>> tensor_2d = np.array([(1,2,3,4),(4,5,6,7),(8,9,10,11),(12,13,14,15)])
>>> print(tensor_2d)
[[ 1 2 3 4]
[ 4 5 6 7]
[ 8 9 10 11]
[12 13 14 15]]
>>>
```

The specific elements of two dimensional tensors can be tracked with the help of row number and column number specified as index numbers.

```
>>> tensor_2d[3][2]
14
```



Tensor Handling and Manipulations

In this section, we will learn about Tensor Handling and Manipulations.

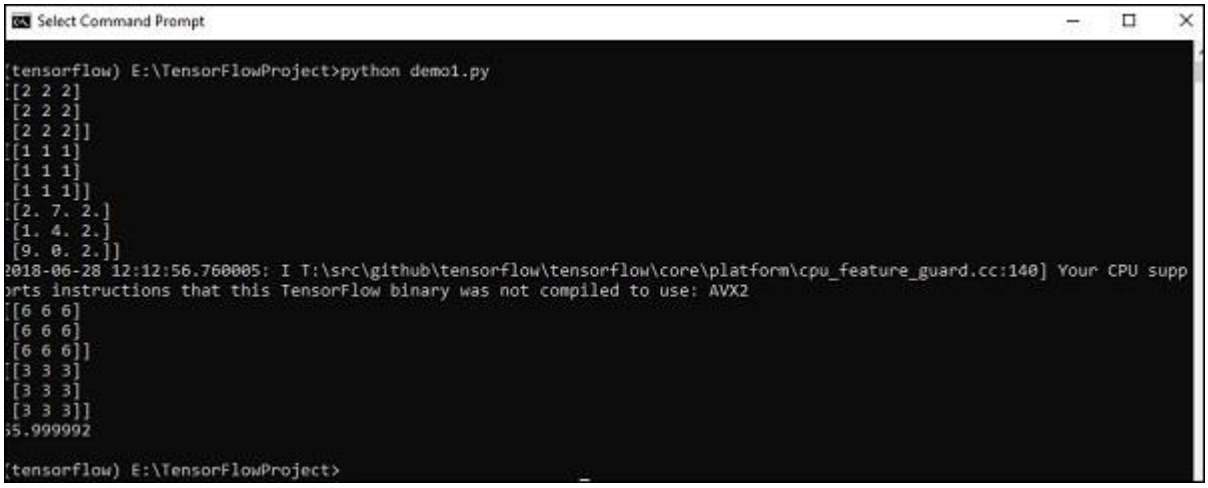
To begin with, let us consider the following code −

```
import tensorflow as tf
import numpy as np

matrix1 = np.array([(2,2,2),(2,2,2),(2,2,2)],dtype = 'int32')
matrix2 = np.array([(1,1,1),(1,1,1),(1,1,1)],dtype = 'int32')

print (matrix1)
print (matrix2)
```

```
matrix1 = tf.constant(matrix1)
matrix2 = tf.constant(matrix2)
matrix_product = tf.matmul(matrix1, matrix2)
matrix_sum = tf.add(matrix1,matrix2)
matrix_3 = np.array([(2,7,2),(1,4,2),(9,0,2)],dtype = 'float32')
print (matrix_3)

matrix_det = tf.matrix_determinant(matrix_3)
with tf.Session() as sess:
   result1 = sess.run(matrix_product)
   result2 = sess.run(matrix_sum)
   result3 = sess.run(matrix_det)

print (result1)
print (result2)
print (result3)
```

Output

The above code will generate the following output −



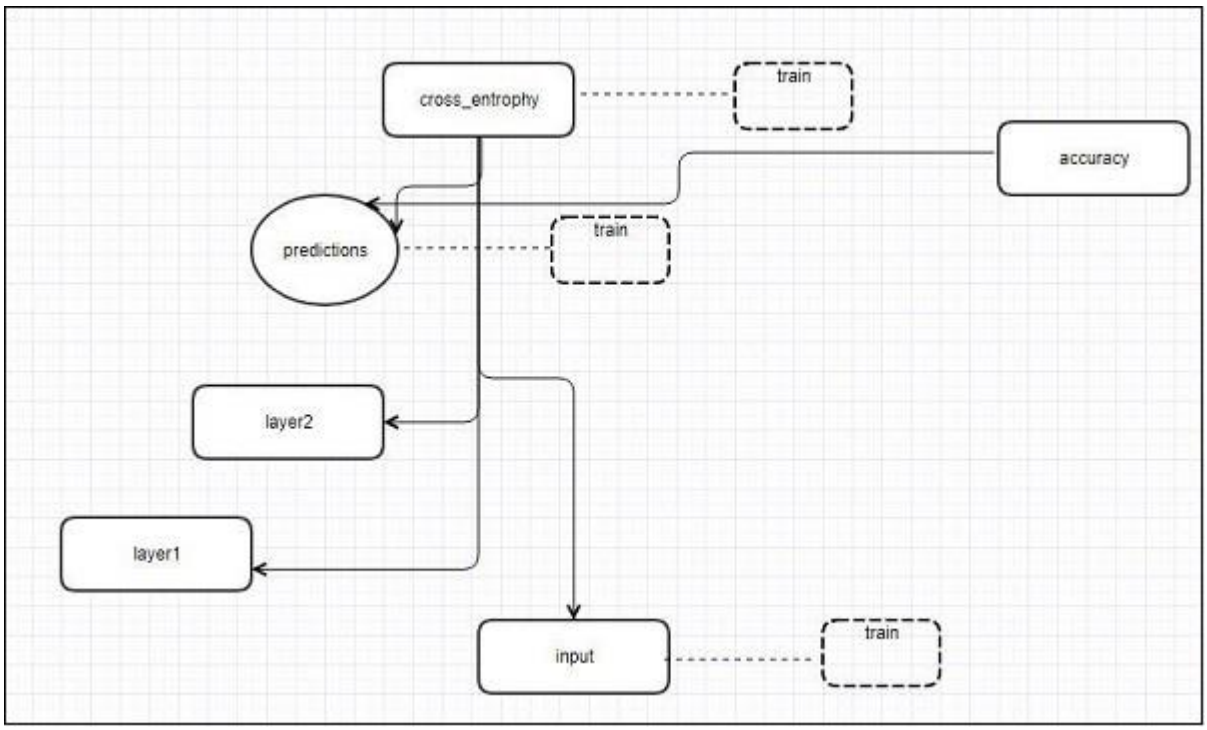
Explanation

We have created multidimensional arrays in the above source code. Now, it is important to understand that we created graph and sessions, which manage the Tensors and generate the appropriate output. With the help of graph, we have the output specifying the mathematical calculations between Tensors.

TensorFlow includes a visualization tool, which is called the TensorBoard. It is used for analyzing Data Flow Graph and also used to understand machine-learning models. The important feature of TensorBoard includes a view of different types of statistics about the parameters and details of any graph in vertical alignment.

Deep neural network includes up to 36,000 nodes. TensorBoard helps in collapsing these nodes in high-level blocks and highlighting the identical structures. This allows better analysis of graph focusing on the primary sections of the computation graph. The TensorBoard visualization is said to be very interactive where a user can pan, zoom and expand the nodes to display the details.

The following schematic diagram representation shows the complete working of TensorBoard visualization −



The algorithms collapse nodes into high-level blocks and highlight the specific groups with identical structures, which separate high-degree nodes. The TensorBoard thus created is useful and is treated equally important for tuning a machine learning model. This visualization tool is designed for the configuration log file with summary information and details that need to be displayed.

Let us focus on the demo example of TensorBoard visualization with the help of the following code −

```
import tensorflow as tf

# Constants creation for TensorBoard visualization
a = tf.constant(10,name = "a")
b = tf.constant(90,name = "b")
y = tf.Variable(a+b*2,name = 'y')
model = tf.initialize_all_variables() #Creation of model

with tf.Session() as session:
   merged = tf.merge_all_summaries()
   writer = tf.train.SummaryWriter("/tmp/tensorflowlogs",session.graph)
   session.run(model)
   print(session.run(y))
```

The following table shows the various symbols of TensorBoard visualization used for the node representation −

Symbol	Meaning
	*High-level* node representing a name scope. Double-click to expand a high-level node.
	Sequence of numbered nodes that are not connected to each other.
	Sequence of numbered nodes that are connected to each other.
	An individual operation node.
	A constant.
	A summary node.
	Edge showing the data flow between operations.
	Edge showing the control dependency between operations.
	A reference edge showing that the outgoing operation node can mutate the incoming tensor.

https://www.toptal.com/machine-learning/tensorflow-machine-learning-tutorial

https://www.tensorflow.org/guide/basics

https://www.tensorflow.org/guide/tensor

https://www.tensorflow.org/guide/variable

https://www.tensorflow.org/guide/intro_to_graphs

Node classification using graph convolutions

https://stellargraph.readthedocs.io/en/stable/demos/node-classification/gcn-node-classification.html