Neural Networks and Deep Learning with R & Python

Part Three: 19Nov24

Data Science Group

Mohammad Arashi

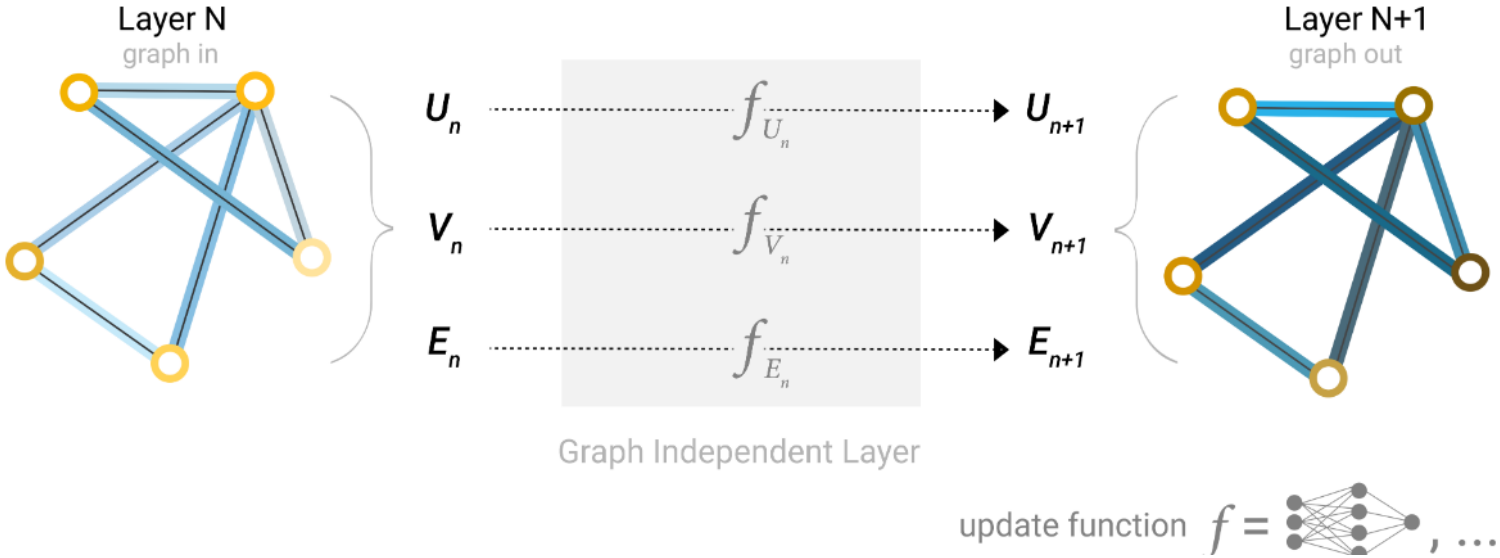
linkedin.com/in/mohammadarashi

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

GitHub

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

GNN and LLM



Graph Neural Network (GNN)

Graph neural networks (GNNs) are deep learning models that learn from graph data, a common data structure for representing relationships between entities. They propagate information across the graph using a **message-passing** mechanism, allowing them to learn complex patterns from interconnected data.

Applications of GNNs

Social Network Analysis: Identifying communities, predicting user behavior, and detecting fake accounts.

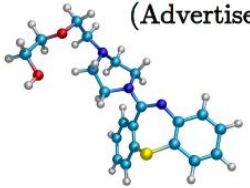
Recommender Systems: Recommending products, movies, or music based on user preferences and relationships.

Drug Discovery: Analyzing molecular structures to predict drug efficacy and toxicity.

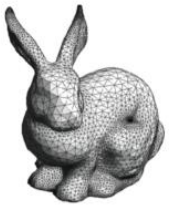
Traffic Prediction: Forecasting traffic patterns and optimizing traffic flow.

Natural Language Processing: Understanding relationships between words and sentences in a text.

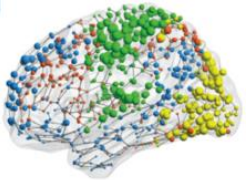
Social networks
(Advertisement)



Drug/Material
molecules
(Chemistry)

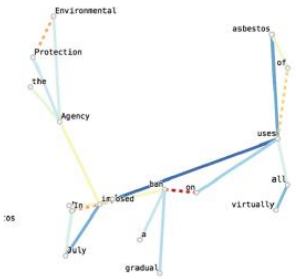


3D Meshes
(Computer Graphics)

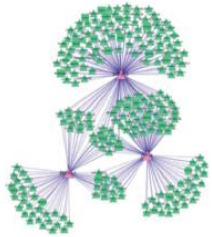


Brain
connectivity
(Neuroscience)

Transportation
networks

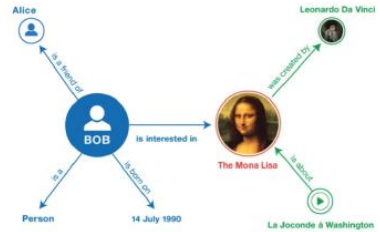


Words relationships
(NLP)

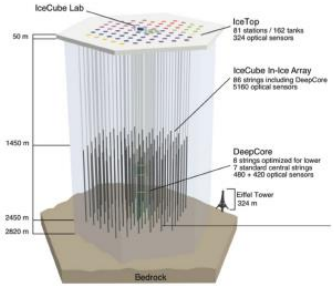


Gene Regulatory
Network

Recommender
systems (Amazon,
Netflix)

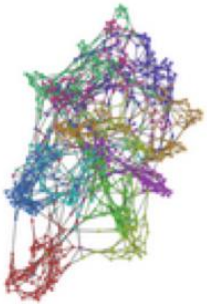


Knowledge graph
(Causality)



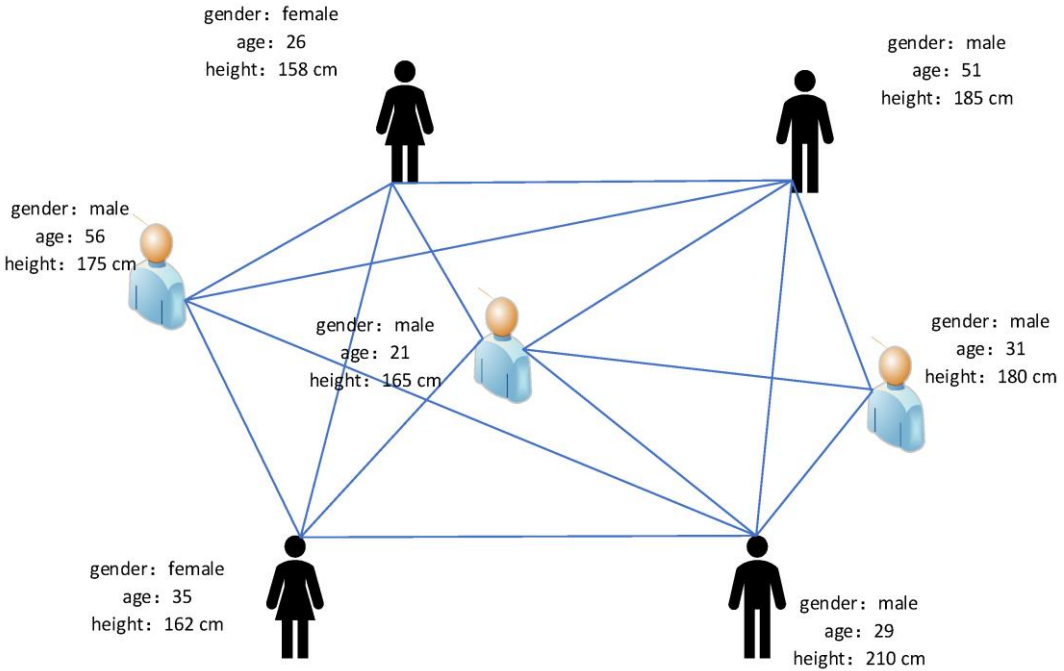
Neutrino
detection (High-
energy Physics)

=

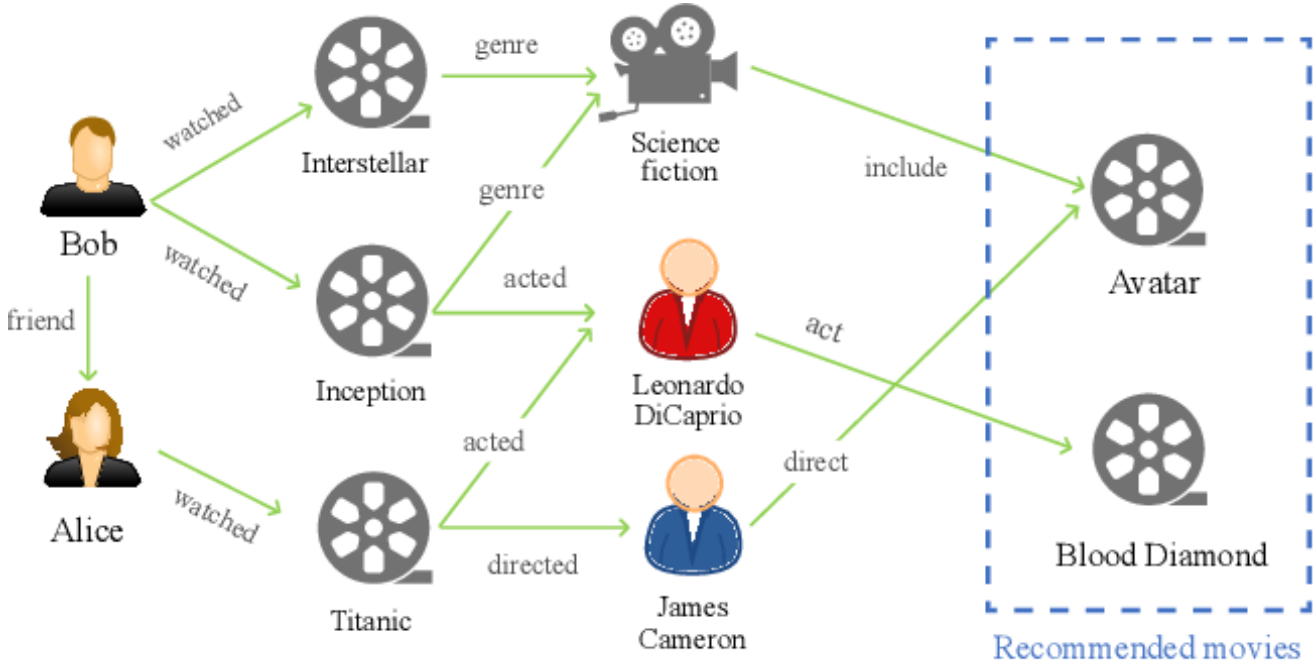


Graphs/
Networks

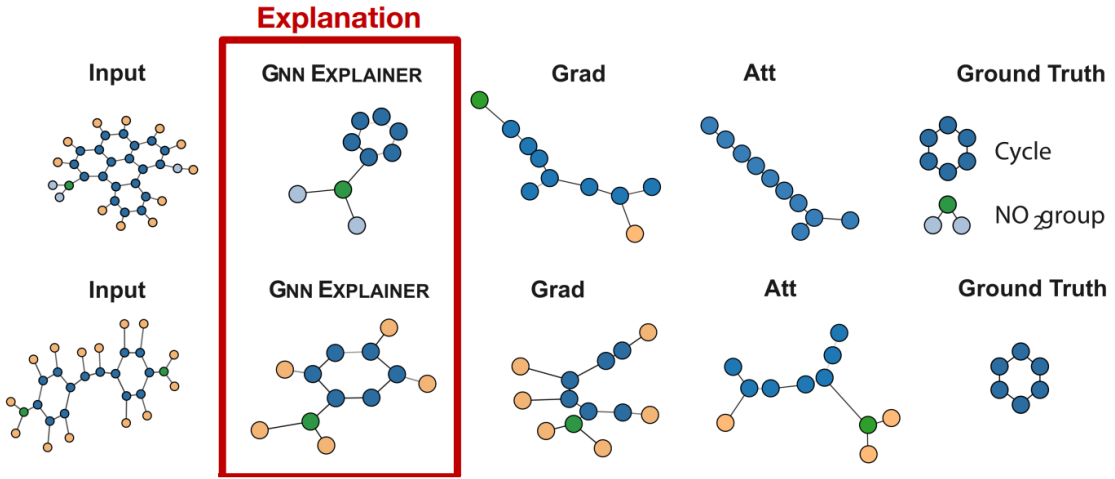
Social Network Analysis: Identifying communities, predicting user behavior, and detecting fake accounts.



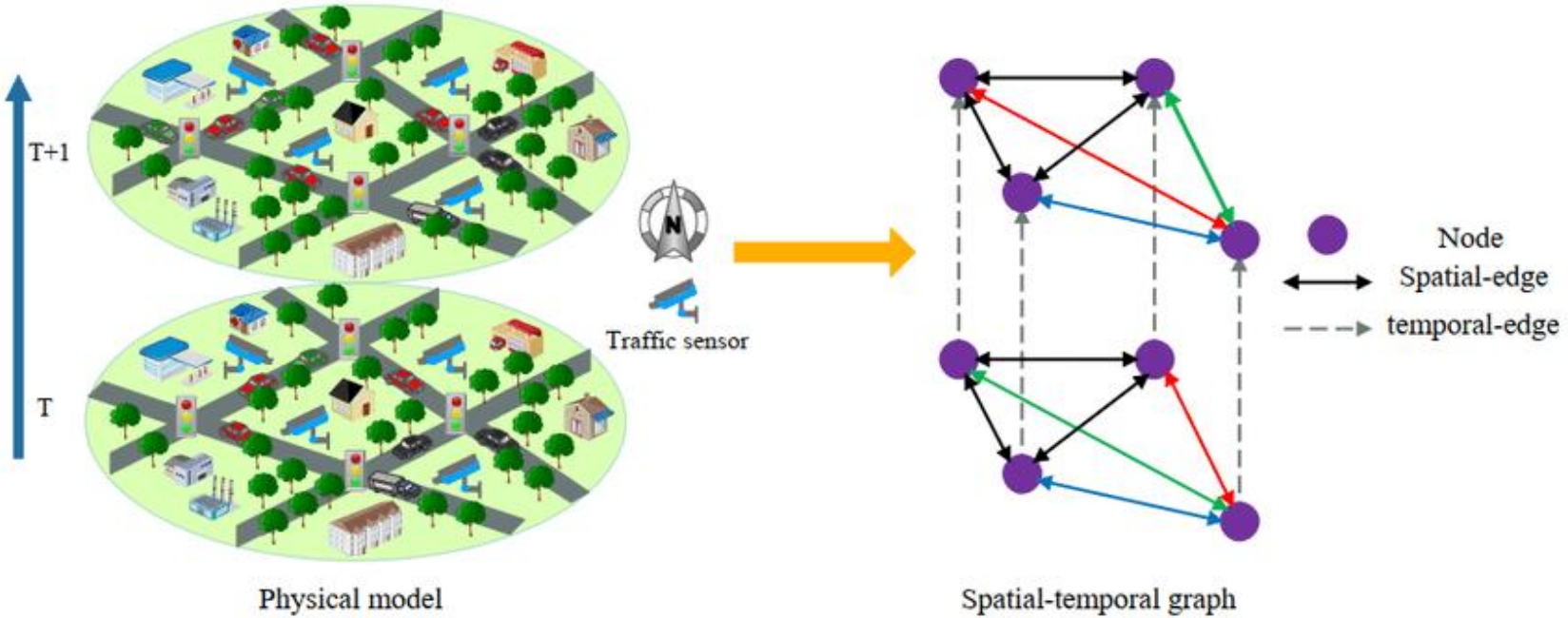
Recommender Systems: Recommending products, movies, or music based on user preferences and relationships.

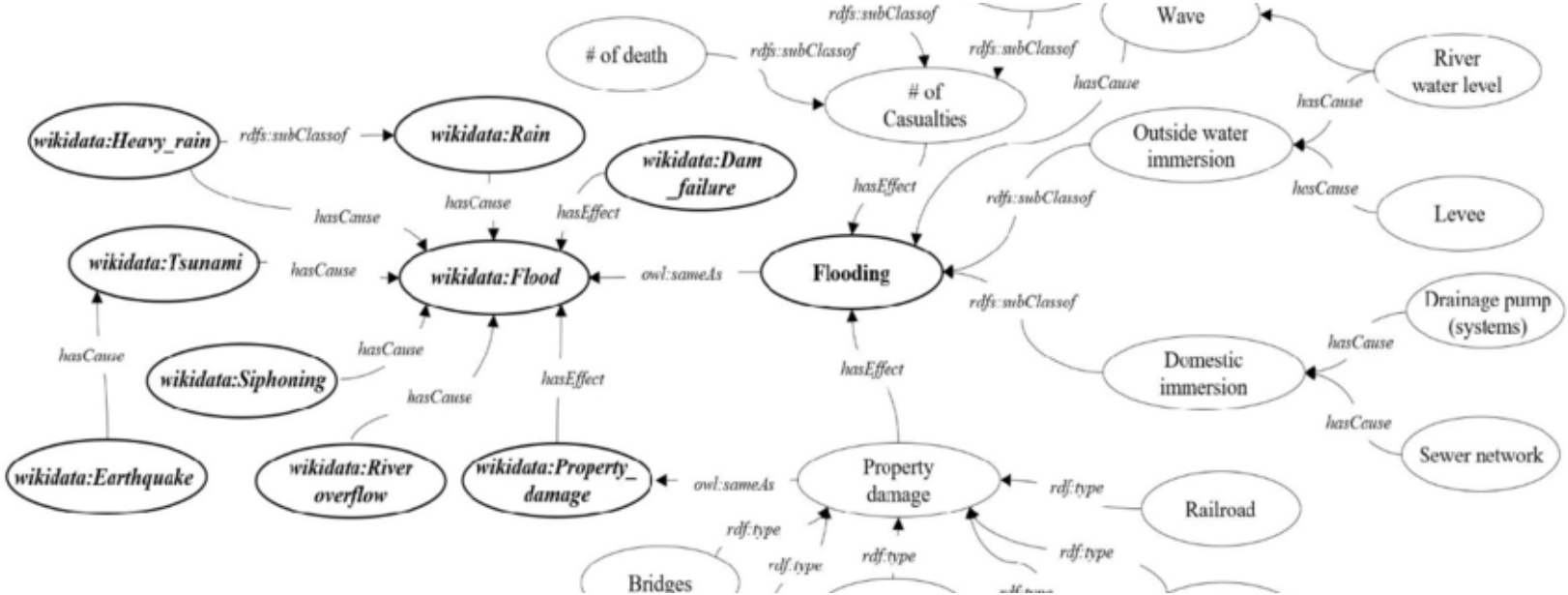


Drug Discovery: Analyzing molecular structures to predict drug efficacy and toxicity.



Traffic Prediction: Forecasting traffic patterns and optimizing traffic flow.



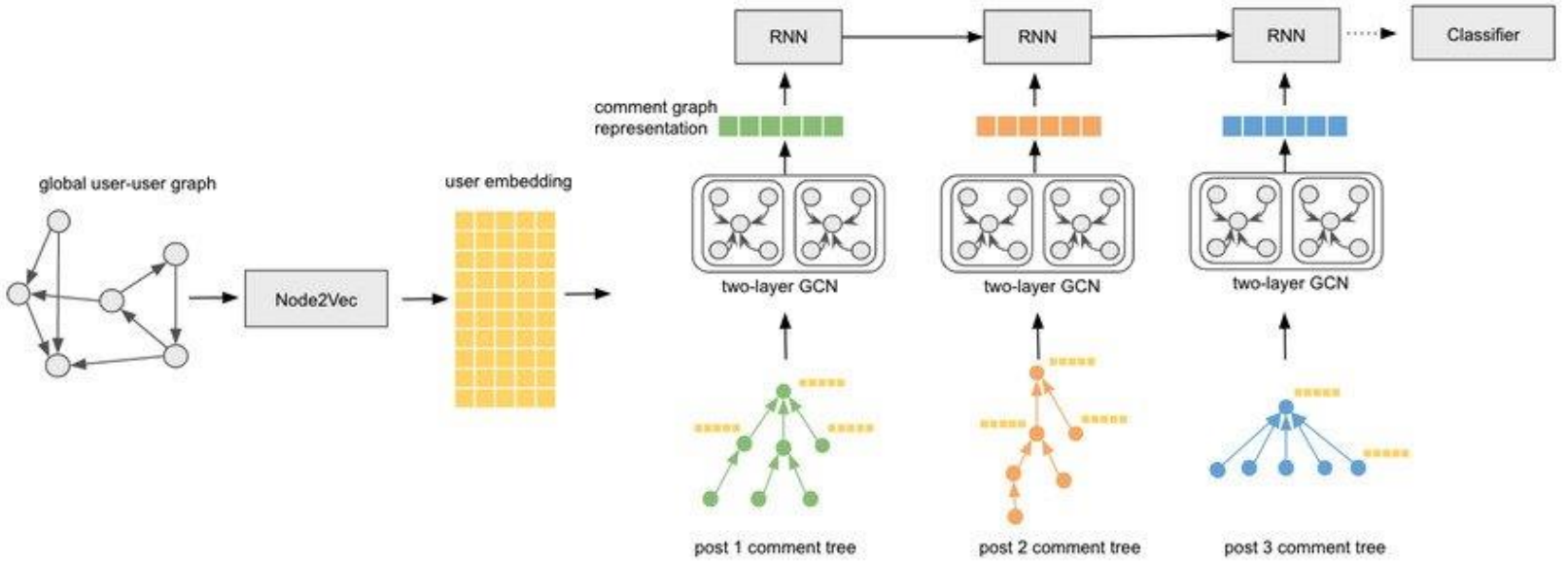


Natural Language Processing:

Understanding relationships between words and sentences in a text.

Why GNN?

The primary benefit of GNN is its capability to perform tasks that Convolutional Neural Networks (CNN) cannot. In contrast, CNN excels in object identification, image categorization, and recognition tasks, achieved through hidden convolutional and pooling layers. CNN is computationally challenging to perform on graph data because the topology is very arbitrary and complicated, implying no spatial locality. Additionally, there is an unfixed node ordering, which complicates using CNN.

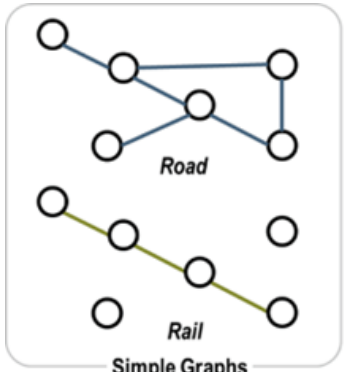


Types of GNNs

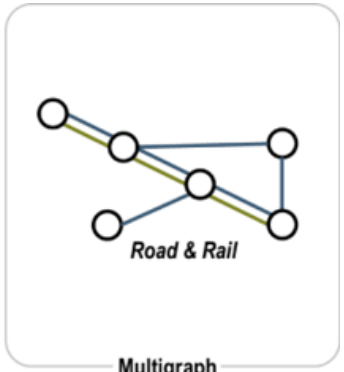
- Recurrent Graph Neural Network
- Spatial Convolutional Network
- Spectral Convolutional Network

Basic Definitions and Concepts in Graph Theory

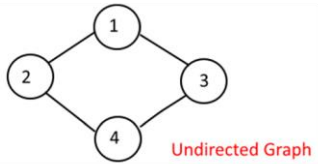
A graph G(V, E) is a set V of vertices and an E of edges. In an undirected graph, an edge is an unordered pair of vertices. An ordered pair of vertices is called a directed edge. We obtain a multigraph if we allow multi-sets of edges, i.e., multiple edges between two vertices. A self-loop or loop is an edge between a vertex and itself. An undirected graph without loops or multiple edges is known as a simple graph.



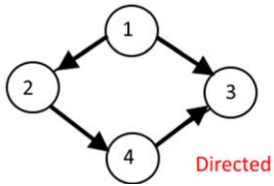
Simple Graphs



Multigraph



Undirected Graph



Directed Graph

Adjacency Matrix

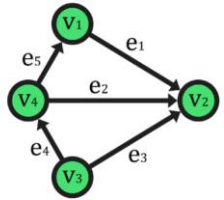
An adjacency matrix is one of the most popular methods for storing a graph on a computer. However, its major drawback is the consumption of unused memory.
For **directed** graphs like the one above, there is a matrix size $|V| \times |V|$ (being $|V|$ the cardinality of the vertices set, thus the number of vertices on the graph) where each element can be a 0 if there is no connection between vertices or a one if the row element links the column one by an outgoing edge. Also, if the graph is **weighted**, the one value is substituted with the **weight** parameter associated with each edge when necessary. if the graph is **undirected**, the same criteria apply with the difference that no distinction is made between outgoing and incoming edges this time. So, there will be a 1 value if an edge exists between the row and column elements.

Undirected graph

$$a_{ij} = \begin{cases} 0 & \text{if exists an edge between } v_i \text{ and } v_j \\ 1 & \text{if not} \end{cases}$$

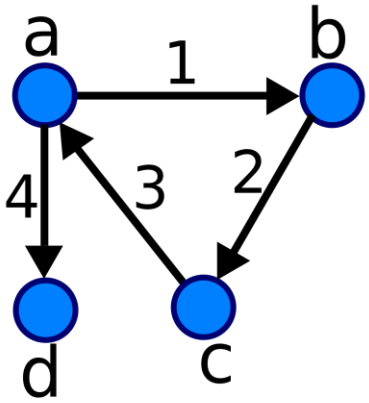
directed graph

$$a_{ij} = \begin{cases} 0 & \text{if exists an outgoing edge from } v_i \text{ to } v_j \\ 1 & \text{if not} \end{cases}$$



Incidence matrix **Adjacency matrix**

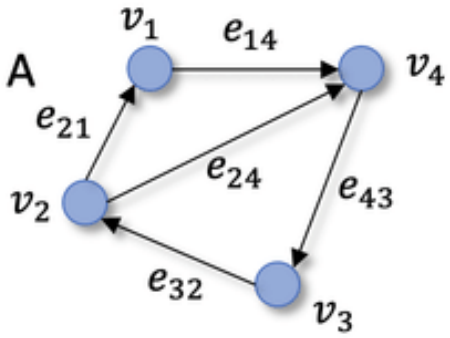
Incidence Matrix



	1	2	3	4
a	1	0	-1	1
b	-1	1	0	0
c	0	-1	1	0
d	0	0	0	-1

Similar to the previous method, there is a matrix size |V|x|E| in which the same rules are fulfilled. The difference is that if an edge e is incoming to a vertex v, the corresponding element will be a -1 instead of 0.

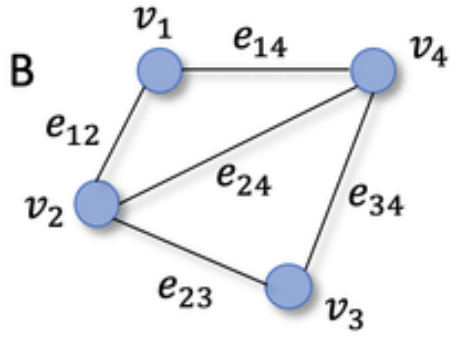
Directed graph G(V,E)



E

	v_1	v_2	v_3	v_4
v_1	0	0	0	1
v_2	1	0	0	1
v_3	0	1	0	0
v_4	0	0	1	0

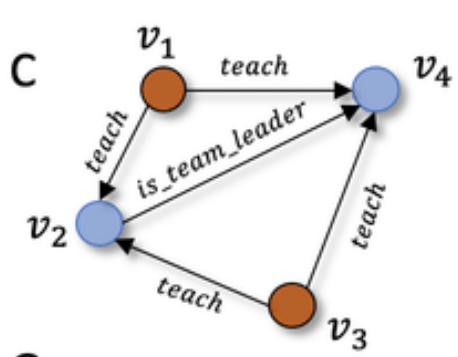
Undirected graph G(V,E)



F

	v_1	v_2	v_3	v_4
v_1	0	1	0	1
v_2	1	0	1	1
v_3	0	1	0	1
v_4	1	1	1	0

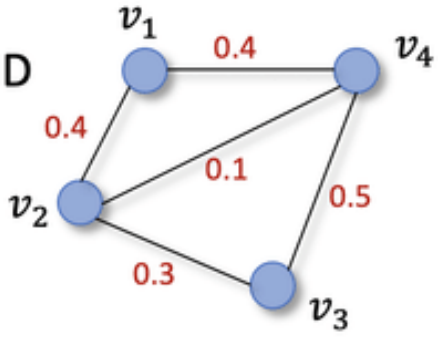
Knowledge graph G(V,E)



G

	v_1	v_2	v_3	v_4
v_1	0	1	0	1
v_2	0	0	0	1
v_3	0	1	0	1
v_4	0	0	0	0

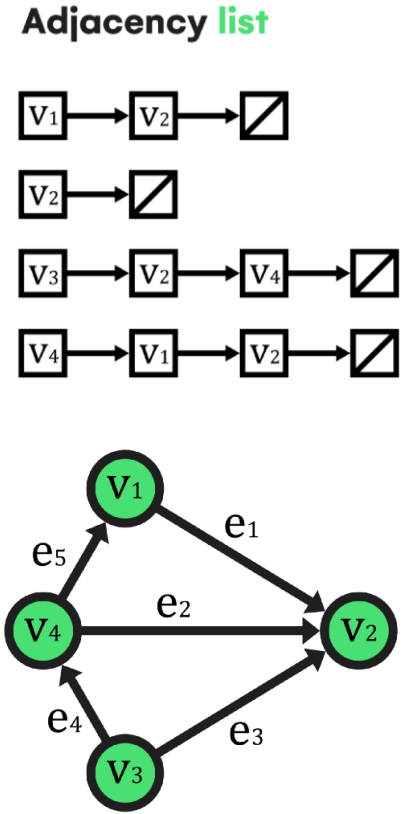
Weighted graph G(V,E)



H

	v_1	v_2	v_3	v_4
v_1	0	0.4	0	0.4
v_2	0.4	0	0.3	0.1
v_3	0	0.3	0	0.5
v_4	0.4	0.1	0.5	0

Adjacency Lists



When using matrices, if the graph has many vertices but few edges (a sparse graph), the matrix will contain a high number of zeroes. This wastes much memory and makes the representation inefficient in terms of space. To solve this issue, adjacency lists appeared as an alternative, replacing matrices with a combination of different data structures – arrays and linked lists. The kernel of this method is an array containing all the graph's nodes. Each array element will have a linked list holding each leading node's neighbor vertices (adjacent vertices). In the case of directed graphs, only the neighbor elements connected by an outgoing edge from the lead node will be inside the linked list. So, if we have a dense graph with many edges, we should store it in matrix form. This has the advantage of O(1) time complexity when checking vertex connection and matrix symmetry along the main diagonal in undirected graphs. But, if our graph is sparse, the low density of edges makes an adjacency list the best choice to depict it computationally.

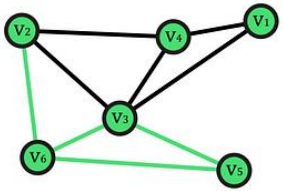
First, we need a starting node v1 and an ending node v2 to traverse a graph. Then, we can define a walk from v1 to v2 as an alternate sequence of vertices and edges. We can go through these elements as much as we need there, and there is always an edge after a vertex (except the last one).

In the case of v1 being equal to v2, the walk would be closed.

Still, we can add repetition restrictions. So, if we want a walk in which no edge is repeated, it's renamed as "trail". Consequently, if the trail is closed, it would be denoted as a "circuit."

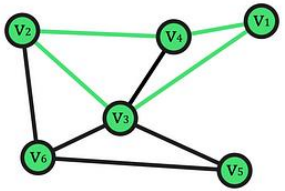
The same happens if we restrict vertex repetition—the walk is renamed a "path," and a closed path is known as a "cycle".

Walk



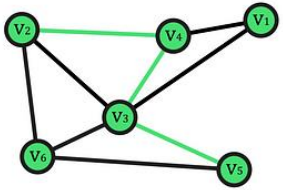
$$w = v_3\{v_3, v_5\}v_5\{v_5, v_6\}v_6\{v_6, v_2\}v_2\{v_2, v_6\}v_6\{v_3, v_6\}v_3$$

Cycle



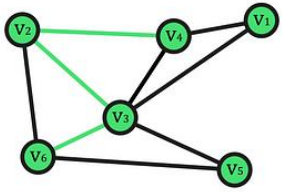
$$c = v_2\{v_2, v_4\}v_4\{v_4, v_1\}v_1\{v_1, v_3\}v_3\{v_2, v_3\}v_2$$

Trail



$$t = v_2\{v_2, v_4\}v_4\{v_4, v_3\}v_3\{v_3, v_5\}v_5$$

Path

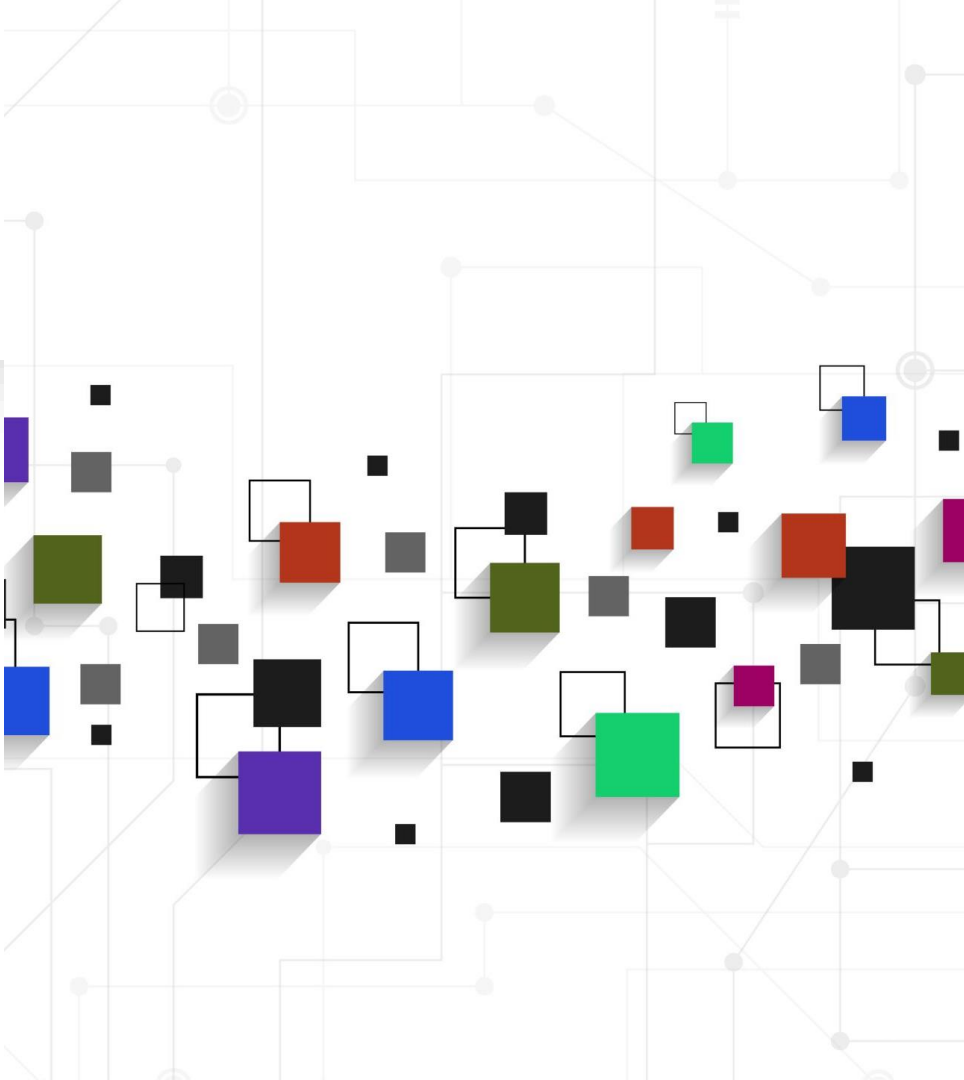


$$p = v_6\{v_6, v_3\}v_3\{v_3, v_2\}v_2\{v_2, v_4\}v_4$$

Images as graphs

We typically think of images as rectangular grids with image channels, representing them as arrays (e.g., 244x244x3 floats). Another way to think of images is as graphs with regular structure, where each pixel represents a node and is connected via an edge to adjacent pixels. Each non-border pixel has precisely 8 neighbors, and the information stored at each node is a 3-dimensional vector representing the RGB value of the pixel. A way of visualizing the connectivity of a graph is through its adjacency matrix. We order the nodes, in this case each of 25 pixels in a simple 5x5 image of a smiley face, and fill a matrix of nnodes×nnodes

with an entry if two nodes share an edge. Note that the three representations below have different views of the same piece of data.



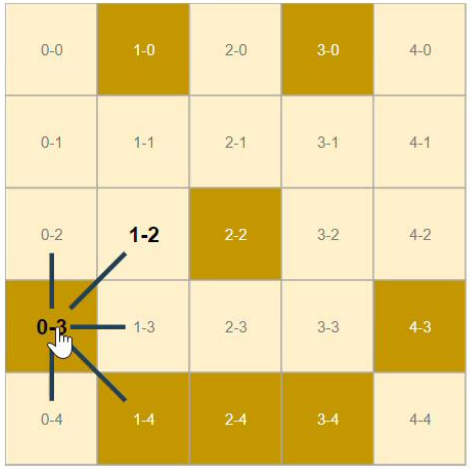
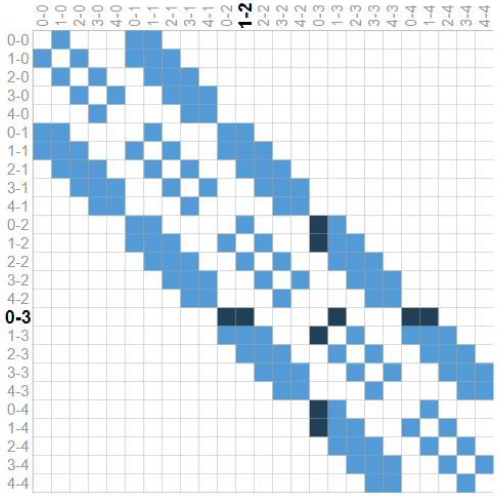
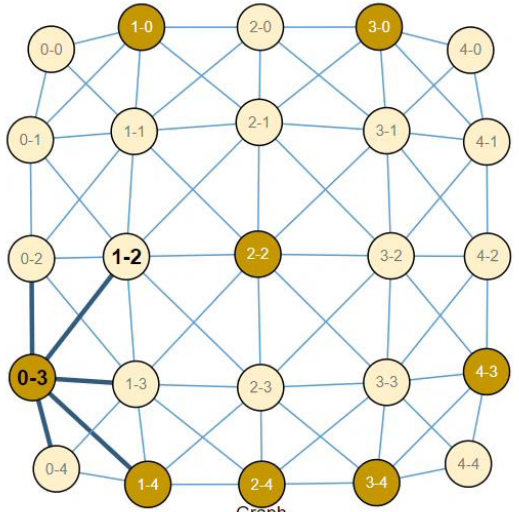


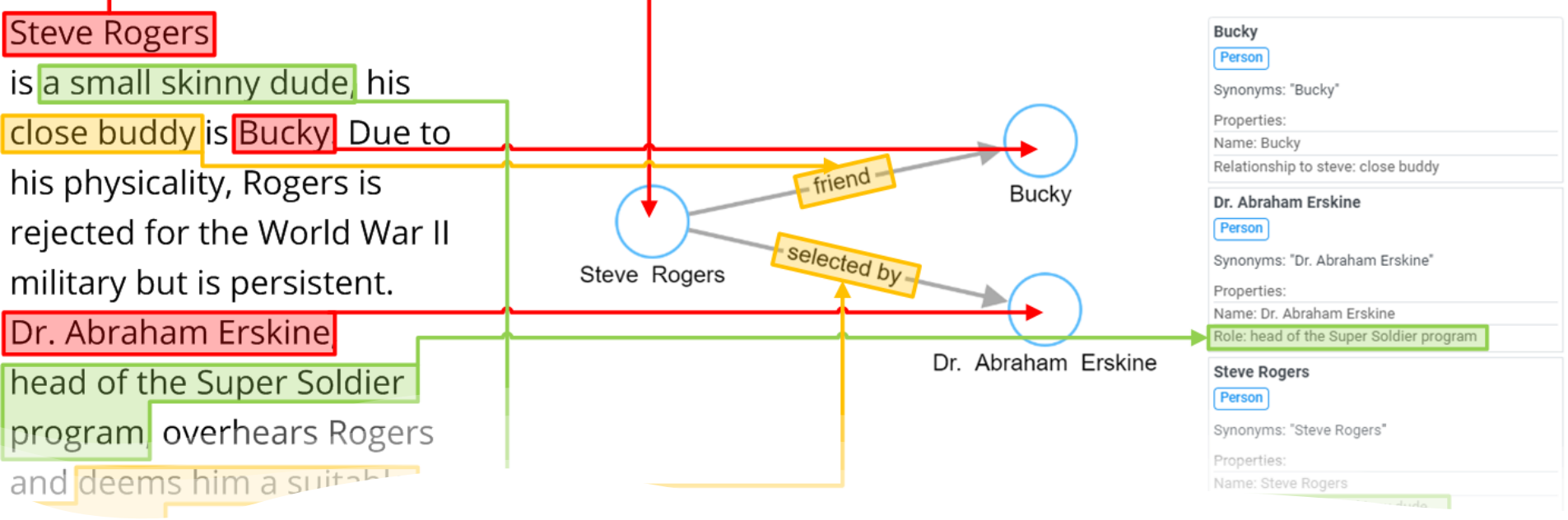
Image Pixels



Adjacency Matrix

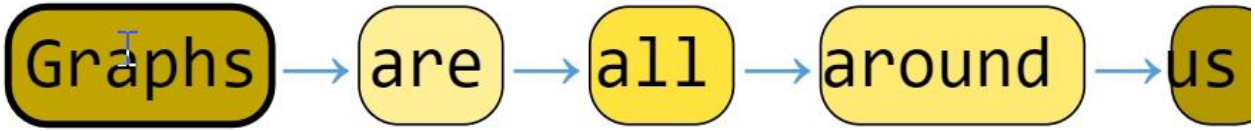


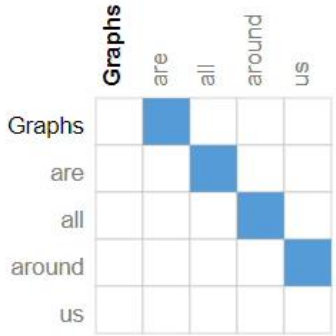
Graph



Text as graphs

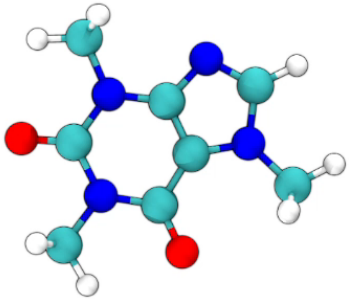
We can digitize text by associating indices to each character, word, or token and representing text as a sequence of these indices. This creates a simple directed graph, where each character or index is a node connected via an edge to the node that follows it. Of course, in practice, this is different from how text and images are encoded: these graph representations are redundant since all images and all text will have very regular structures. For instance, images have a banded structure in their adjacency matrix because all nodes (pixels) are connected in a grid. The adjacency matrix for text is just a diagonal line because each word only connects to the prior word and to the next one.

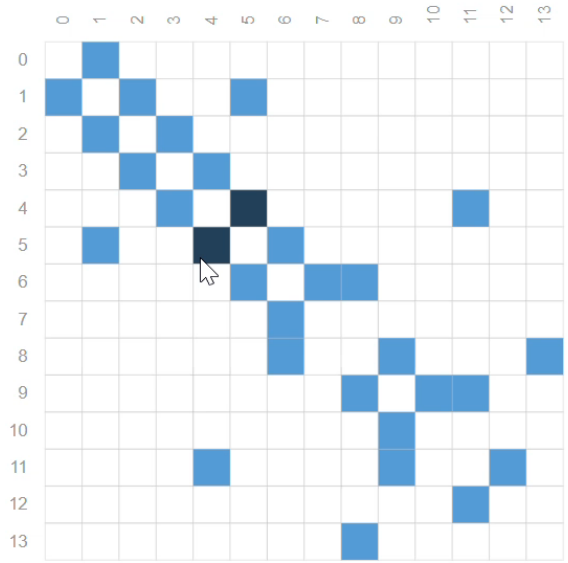

Graphs
are
all
around
us

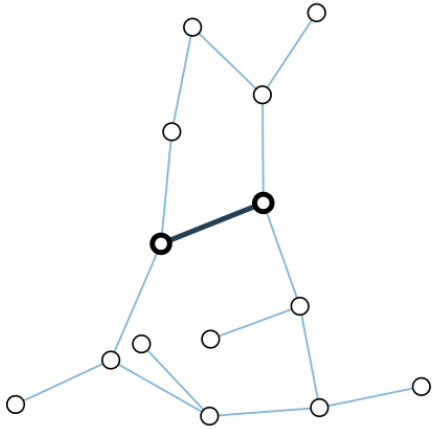

Graphs
are
all
around
us
Graphs
are
all
around
us

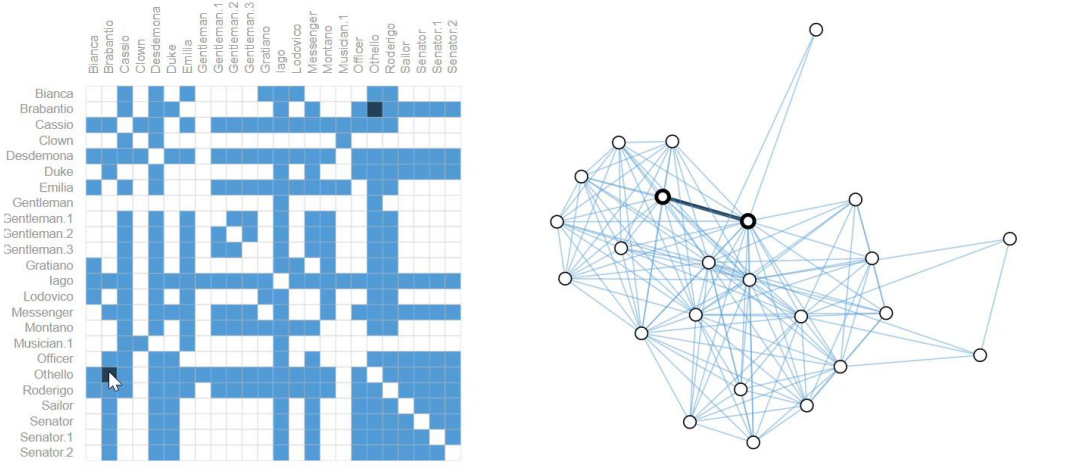
Graph-valued data in the wild

Molecules as graphs









Social networks as graphs

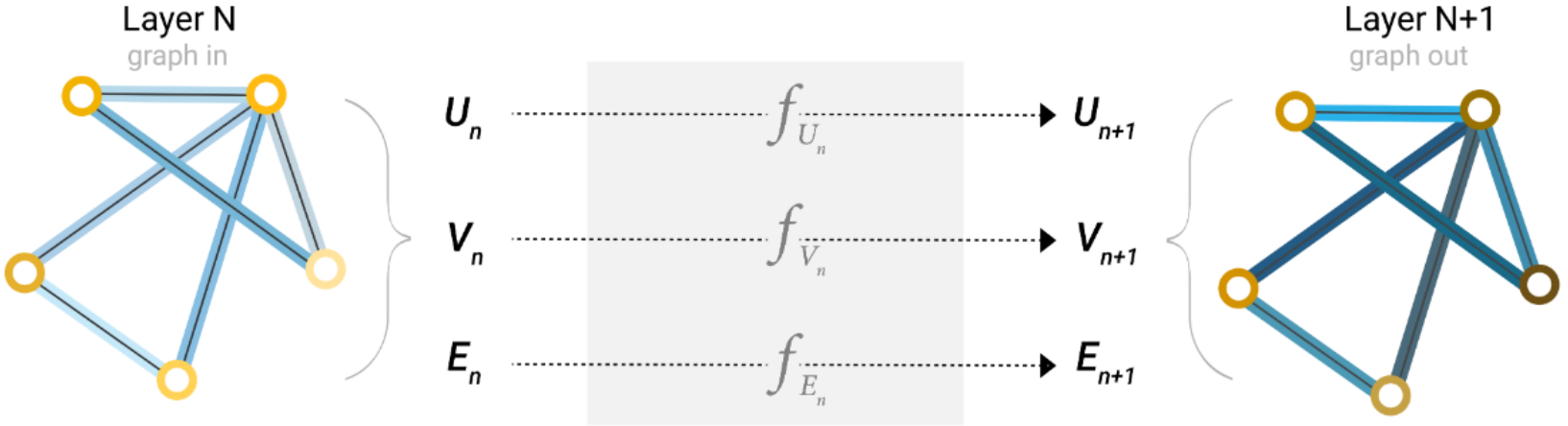
Social networks are tools to study patterns in the collective behavior of people, institutions, and organizations.

We can build a graph representing groups of people by modeling individuals as nodes and their relationships as edges.

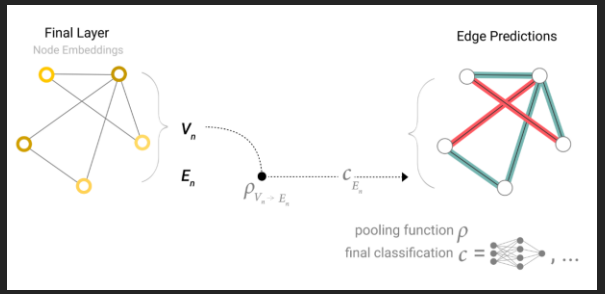
The challenges of using graphs in machine learning

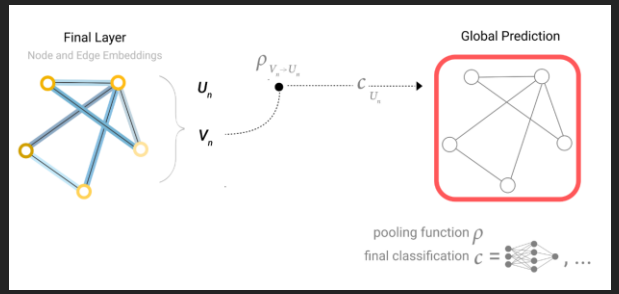
Machine learning models typically take rectangular or grid-like arrays as input. So, it's not immediately intuitive to represent them in a format compatible with deep learning. Graphs have up to four types of information we will potentially want to use to make predictions: nodes, edges, global context, and connectivity. The first three are relatively straightforward: for example, with nodes, we can form a node feature matrix N by assigning an index i to each node and storing the feature for node i in N. While these matrices have various examples, they can be processed without special techniques. However, representing a graph's connectivity is more complicated. Perhaps the most obvious choice would be to use an adjacency matrix since this is easily tensorisable. However, this representation has a few drawbacks. The example dataset table shows that the number of nodes in a graph can be on the order of millions, and the number of edges per node can be highly variable. Often, this leads to very sparse adjacency matrices, which are space-inefficient. Another problem is that many adjacency matrices can encode the same connectivity, and there is no guarantee that these different matrices would produce the same result in a deep neural network (that is to say, they are not permutation invariant).

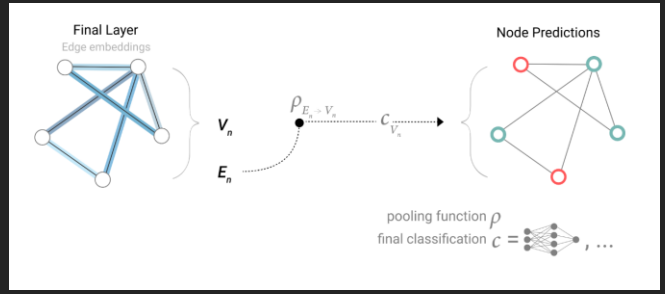
The simplest GNN



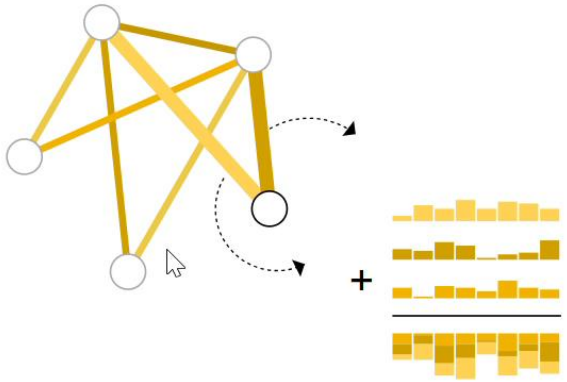
update function $f =$, ...







GNN Predictions by Pooling Information



Aggregate information
from adjacent edges

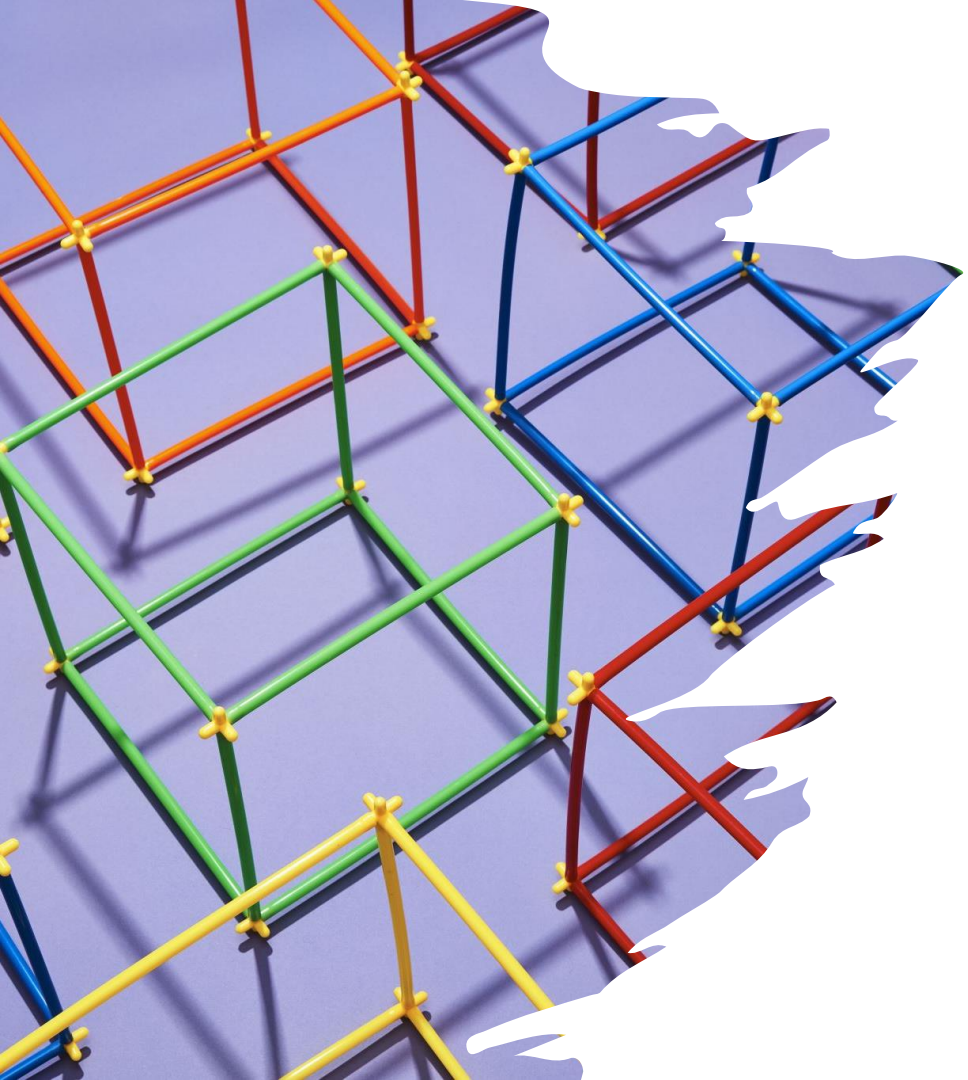
Inductive vs Transductive learning

The notion of inductive vs. transductive, which is often used in the GNN literature, can be confusing. So, let's clarify it before we proceed.

In transductive learning, the model has already encountered both the training and the test input data. In our case these are the nodes of a large graph where we want to predict the node labels. If a new node is added to the graph, we need to retrain the model.

In inductive learning, the model sees only the training data. Thus, the generated model will be used to predict graph labels for unseen data.

To understand that from the GNNs perspective, imagine the following example. Suppose that we have a graph with 10 nodes. Also consider that the structure of the graph, how nodes are connected, is not important for the following example. We use 6 of them for the training set (with the labels) and 4 for the test set. How do we train this model.



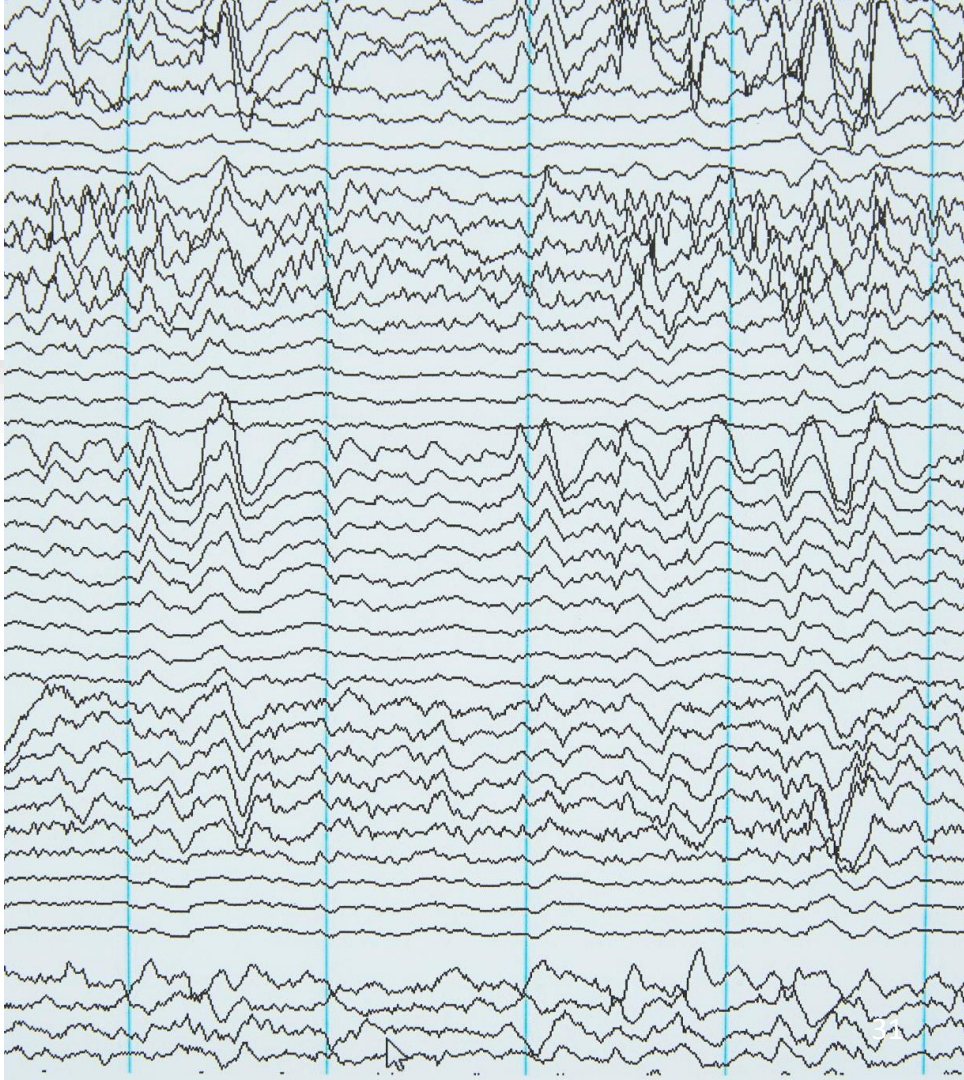
Use a semi-supervised learning approach and train the whole graph using only the 6 labeled data points. This is called inductive learning. Models trained correctly with inductive learning can generalize well but it can be quite hard to capture the complete structure of the data.

Use a self-supervised approach which will label the unlabeled data points using additional information and train the model on all 10 nodes. This is called transductive learning and is quite common in GNNs since we use the whole graph to train the model.

With that out of the way, let’s now proceed with the most popular GNN architectures.

Spectral Methods

Spectral methods encompass techniques that involve representing a graph in the spectral domain, which follows an intuitive concept. These methods draw upon graph signal processing principles and establish the convolution operator within the spectral domain by applying the Fourier transform **F**. To initiate the process, the graph signal **x** undergoes a transformation into the spectral domain using the graph Fourier transform **F**. Subsequently; the convolution operation takes place by performing element-wise multiplication. Once the convolution is completed, the resulting signal is converted back to its original form using the Fourier transform F inverse graph.



The Graph Laplacian

$$D_v = \sum_u A_{vu}.$$

Given a graph G , let us fix an arbitrary ordering of the n nodes of G. We denote the 0−1 adjacency matrix of G by A , we can construct the diagonal degree matrix D of G as :

where A_{vu} denotes the entry in the row corresponding to v and the column corresponding to uu in the matrix A, we will use this notation throughout this section. Then, the graph Laplacian L is the square $n \times n$ matrix defined as: $L=D-A$.

Polynomials of the Laplacian

Now that we have understood what the graph Laplacian is, we can build polynomials of the form:

$$p_w(L) = w_0 I_n + w_1 L + w_2 L^2 + \ldots + w_d L^d = \sum_{i=0}^{d} w_i L^i.$$

Each polynomial of this form can alternately be represented by its vector of coefficients w=[w0,…,wd] Note that for every ww, $pw(L)$ is an n×n matrix, just like L. These polynomials can be considered the equivalent of 'filters' in CNNs, and the coefficients ww as the weights of the 'filters'. For ease of exposition, we will focus on the case where nodes have one-dimensional features: each of the $xvxv$ for v∈V is just a real number. The same ideas hold when each of the $xvxv$ are higher-dimensional vectors. Using the previously chosen ordering of the nodes, we can stack all of the nodes' features xv to get a vector x∈R.

Once we have constructed the feature vector x, we can define its convolution with a polynomial filter p_w as:

$$x' = p_w(L)\ x$$

To understand how the coefficients *ww* affect the convolution, let us begin by considering the 'simplest' polynomial: when $w_0 = 1$ and all of the other coefficients are 0. In this case, x' is just x:

$$x' = p_w(L)\ x = \sum_{i=0}^{d} w_i L^i x = w_0 I_n x = x.$$

Now, if we increase the degree, and consider the case where instead $w_1 = 1$ and and all of the other coefficients are 0. Then, $x'$is just Lx, and so:

$$\begin{aligned} x'_v = (Lx)_v &= L_v x \\ &= \sum_{u \in G} L_{vu} x_u \\ &= \sum_{u \in G} (D_{vu} - A_{vu}) x_u \\ &= D_v\ x_v - \sum_{u \in \mathcal{N}(v)} x_u \end{aligned}$$

We see that the features at each node vv are combined with those of its immediate neighbors $u \in N(v)$. For readers familiar with Laplacian filtering of images, this is the same idea. When x is an image, $x'=Lx$ is precisely the result of applying a 'Laplacian filter' to x.
At this point, a natural question is: How does the degree dd of the polynomial influence the behavior of the convolution? Indeed, it is not too hard to show that.

$$\text{dist}_G(v,u) > i \quad \Longrightarrow \quad L^i_{vu} = 0.$$

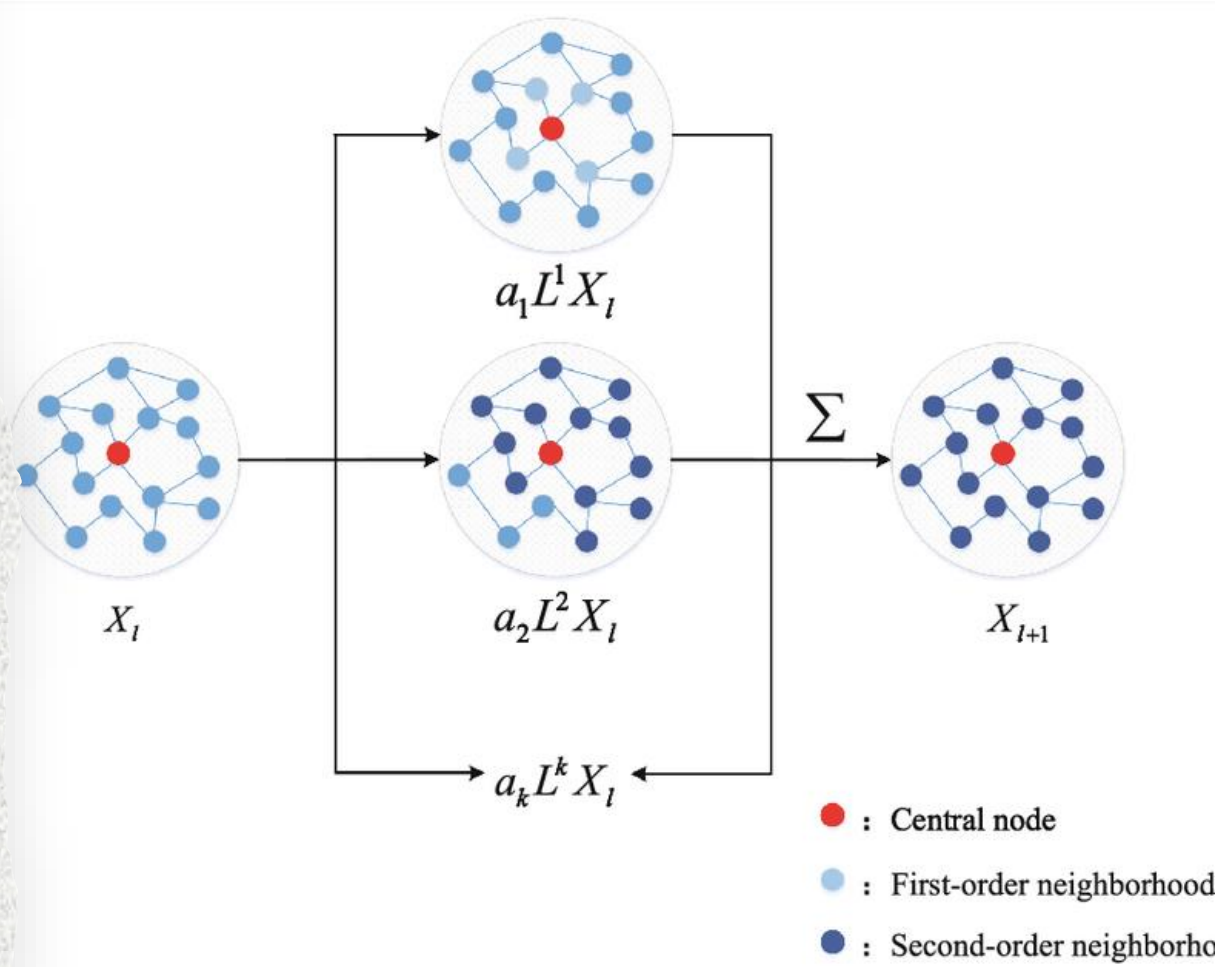
This implies, when we convolve x with $p_w(L)$ of degree d to get x':

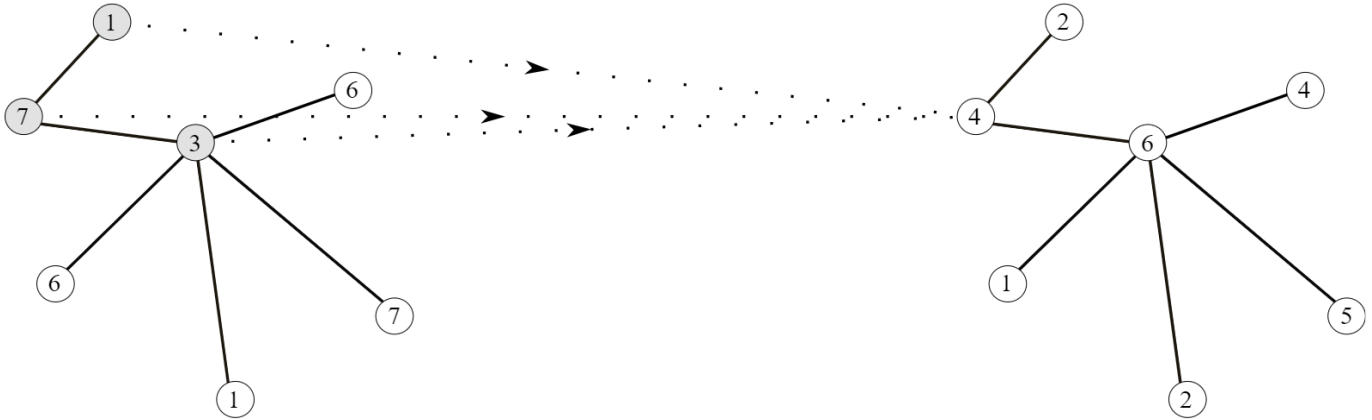
$$\begin{aligned} x'_v = (p_w(L)x)_v &= (p_w(L))_v x \\ &= \sum_{i=0}^{d} w_i L^i_v x \\ &= \sum_{i=0}^{d} w_i \sum_{u \in G} L^i_{vu} x_u \\ &= \sum_{i=0}^{d} w_i \sum_{\substack{u \in G \\ \text{dist}_G(v,u) \le i}} L^i_{vu} x_u. \end{aligned}$$

Effectively, the convolution at node v occurs only with nodes uu which are not more than d hops away.
Thus, these polynomial filters are localized.
The degree of the localization is governed completely by d.

ChebNets

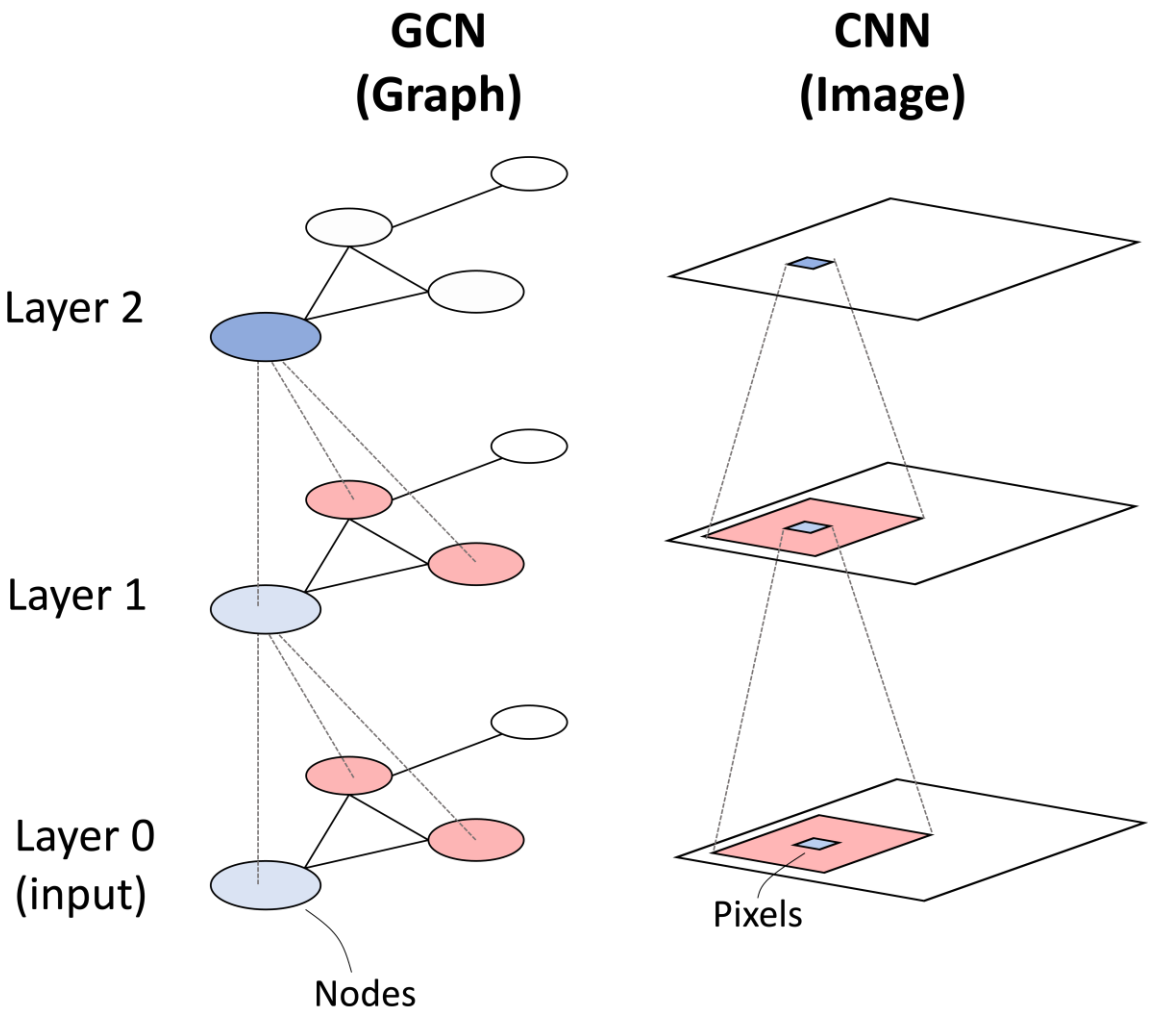
- ChebNets is a graph convolutional neural network (GCN) approach designed to address the issue of locality in graph data. They propose that a node's feature representation should only be influenced by its k-hop neighborhood, introducing the notion of k-localized convolutions.
- To achieve this, ChebNets utilize Chebyshev polynomials of order K for defining the localized convolution. These polynomials help capture information from a node's k-hop neighbors, enabling the network to focus on a limited region of the graph around each node, similar to the concept of receptive fields in traditional convolutional neural networks for images. By employing Chebyshev polynomials, the advantage of ChebNets lies in the reduced computational complexity compared to spectral networks. In contrast to computing the eigenvectors of the Laplacian, ChebNets only require the computation of Chebyshev polynomials, which is computationally more efficient and makes them practical for handling larger graphs. This makes ChebNets a popular choice for various graph-related tasks where locality plays a significant role in the data representation.


$a_1 L^1 X_l$
X_l
$a_2 L^2 X_l$
Σ
X_{l+1}
$a_k L^k X_l$
: Central node
: First-order neighborhood
: Second-order neighborho

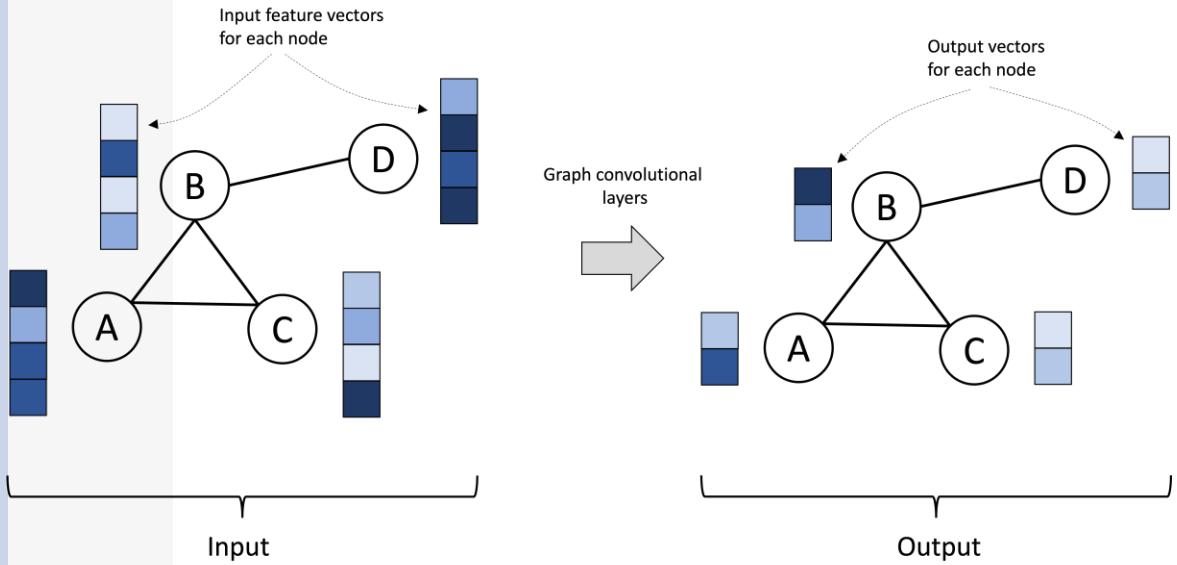


Understanding Convolutions on Graphs

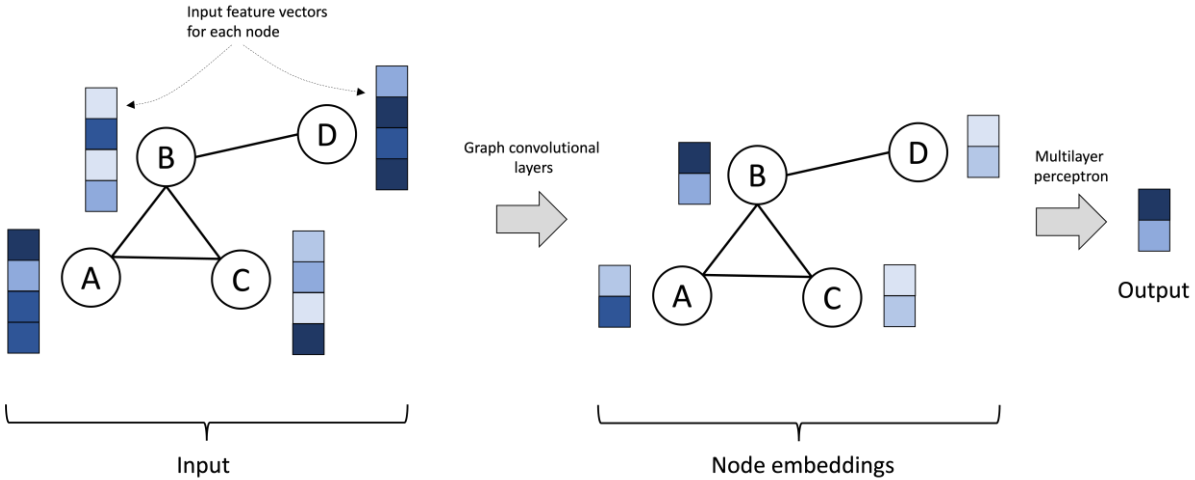
Convolutional Neural Networks have been seen to be quite powerful in extracting features from images. However, images can be seen as graphs with a regular grid-like structure, where the individual pixels are nodes, and the RGB channel values at each pixel are the node features.


GCN
(Graph)
CNN
(Image)
Layer 2
Layer 1
Layer 0
(input)
Nodes
Pixels

Inputs and outputs of a GCN



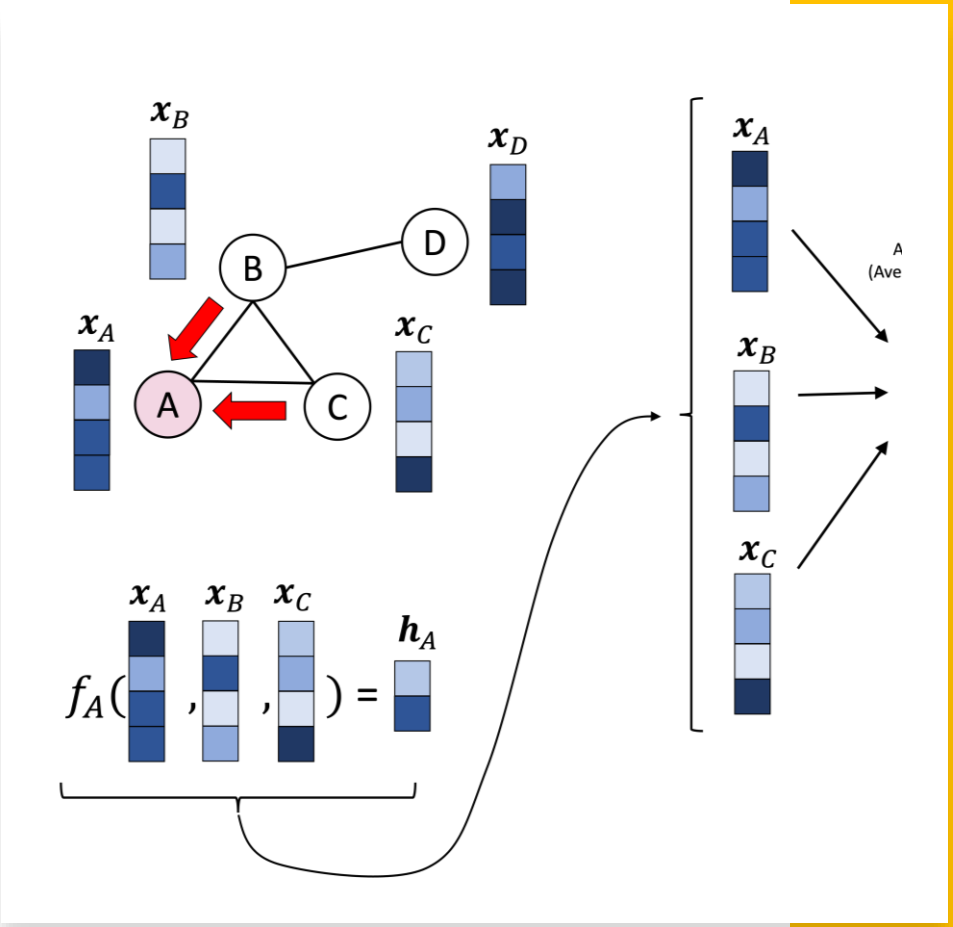
Fundamentally, a GCN takes as input a graph together with a set of feature vectors where each node is associated with its feature vector. The GCN is then composed of a series of graph convolutional layers (to be discussed in the next section) that iteratively transform the feature vectors at each node. The output is then the graph associated with output vectors associated with each node. These output vectors can be (and often are) of different dimensions than the input vectors.



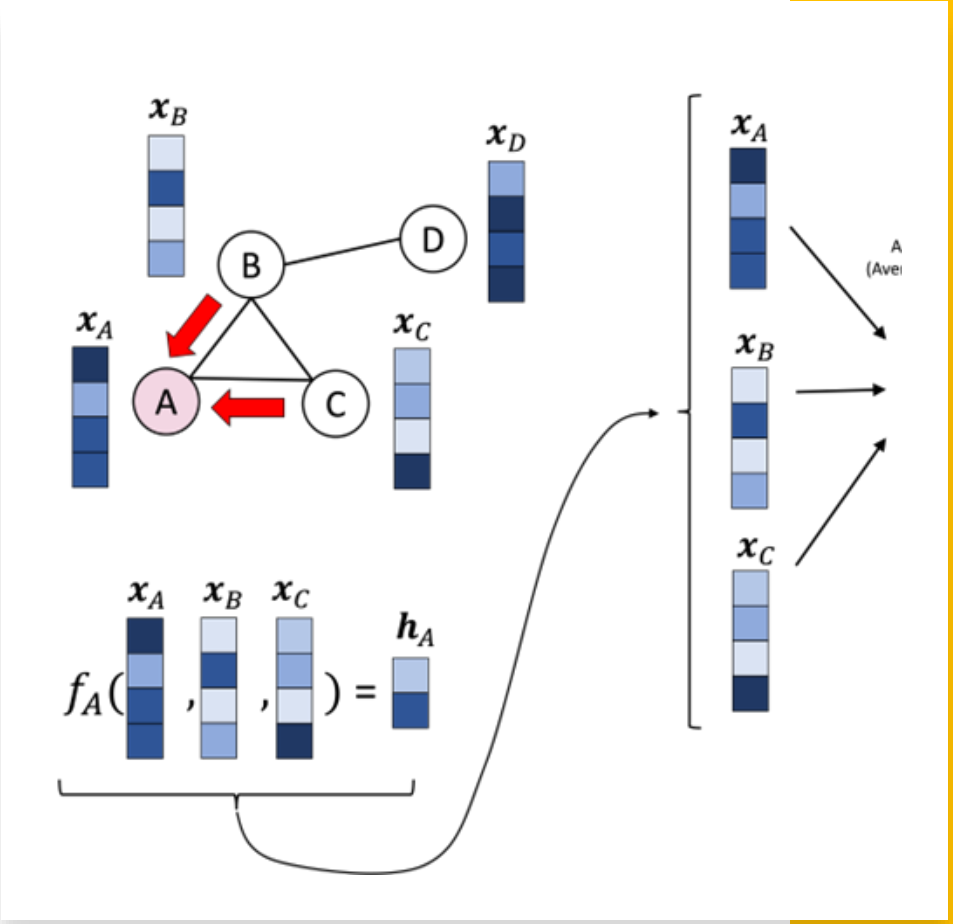
If the task at hand is a “node-level” task, such as performing classification on the nodes, then these per-node vectors can be treated as the model’s final outputs. For node-level classification, these output vectors could, for example, encode the probabilities that each node is associated with each class. Alternatively, we may be interested in performing a “graph-level” task, where instead of building a model that produces an output per node, we are interested in a task that requires an output over the graph as a whole. For example, we may be interested in classifying whole graphs rather than individual nodes. In this scenario, the per-node vectors could be fed collectively into another neural network (such as a simple multilayer perceptron) that operates on all of them to produce a single output vector.

The graph convolutional layer

GCNs are composed of stacked graph convolutional layers in a similar way that traditional CNNs are composed of convolutional layers. Each convolutional layer takes as input the nodes' vectors from the previous layer (for the first layer this would be the input feature vectors) and produces corresponding output vectors for each node. To do so, the graph convolutional layer pools the vectors from each node's neighbors

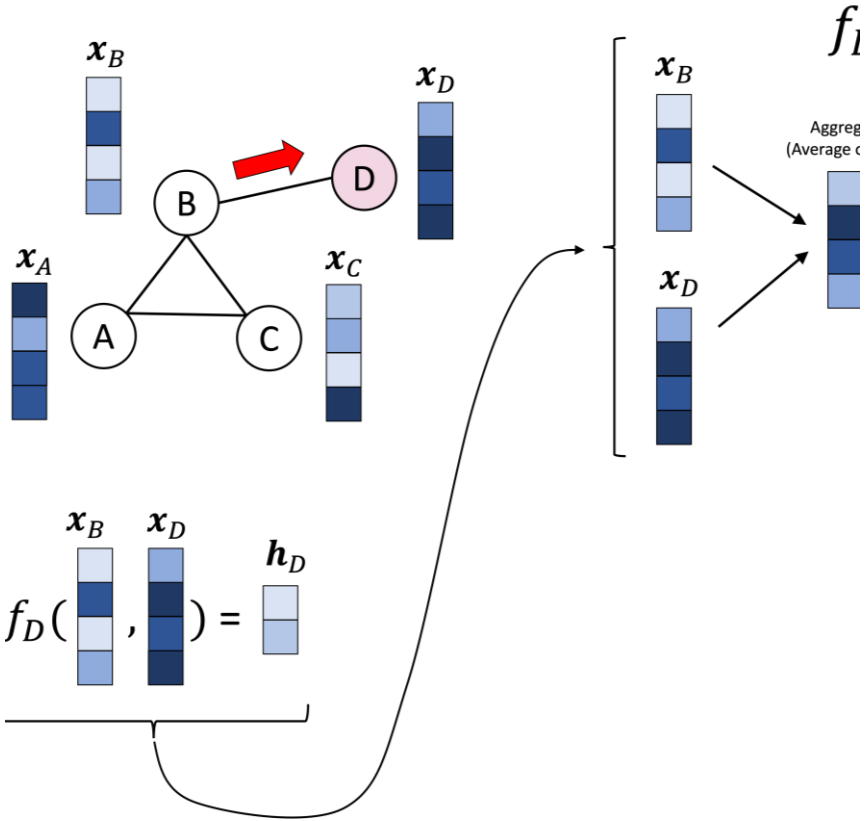


node A's vector, denoted x_A is pooled/aggregated with the vectors of its neighbors, x_B and x_C . This pooled vector is then transformed/updated to form node A's vector in the next layer, denoted h_A. This same procedure is carried out over every node. Below we show this same procedure, but on performed on node D, which entails aggregating x_D with its neighbor's vector x_B



Performing on node D, entails aggregating x_D with its neighbor's vector x_B

This procedure is often called message passing since each node is "passing" its vector to its neighbors to update their vectors. Each node's "message" is the vector associated with it.



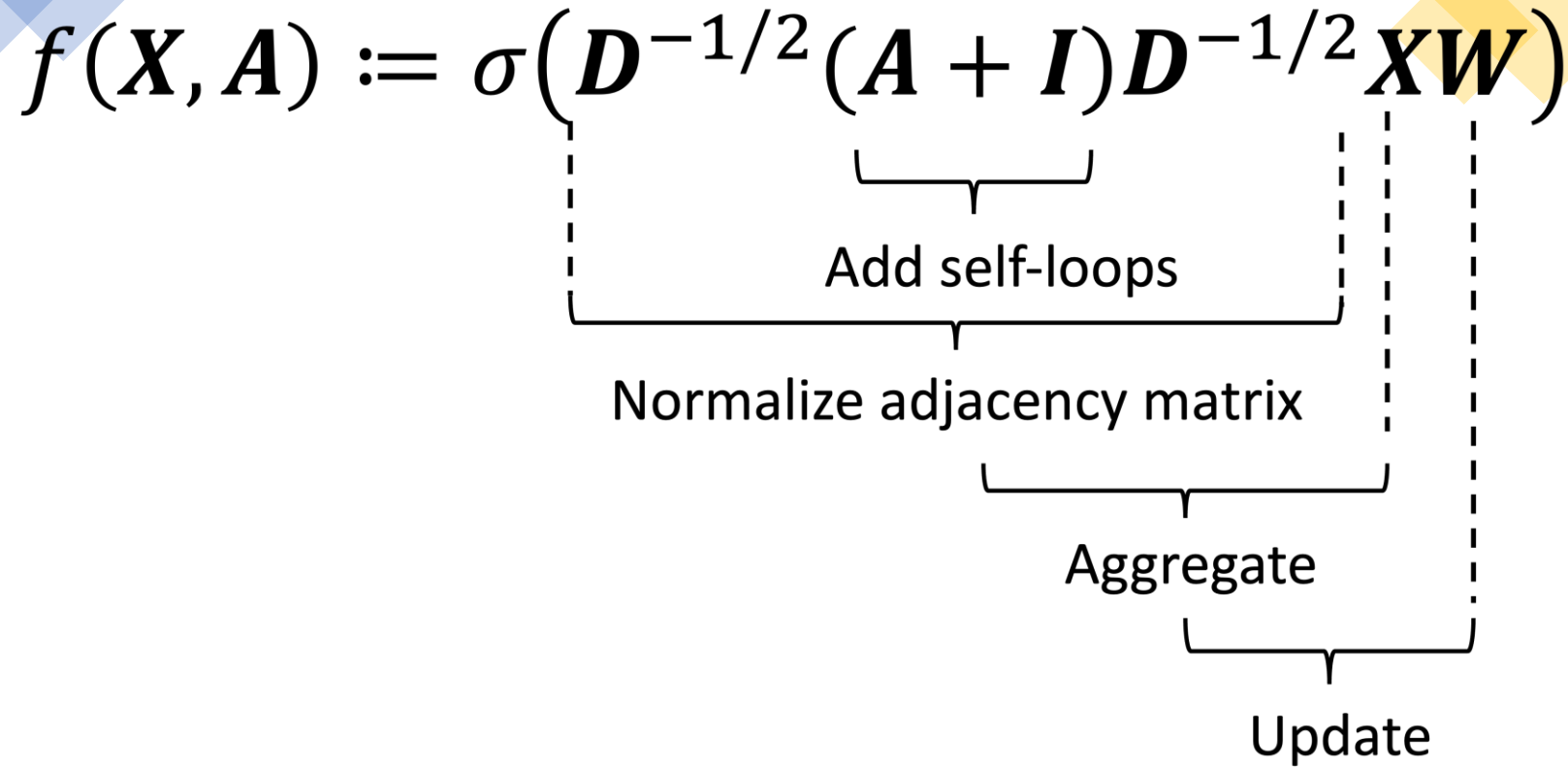
Let $\mathrm{X} \in \mathbb{R}^{n \times d}$ be the features corresponding to the nodes where n is the number of nodes and d is the number of features. That is, row i of X stores the features of node i . Let A be the adjacency matrix of this graph where :

$$\boldsymbol{A}_{i,j} := \begin{cases} 1, & \text{if there is an edge between node } i \text{ and } j \\ 0, & \text{otherwise} \end{cases}$$

Note the matrices X and A are the two pieces of data required as input to a GCN for a given graph. The graph convolutional layer can thus be expressed as function that accepts these two inputs and outputs a matrix representing the updated vectors associated with each node. This function is given by:

$$f(\boldsymbol{X}, \boldsymbol{A}) := \sigma\left(\boldsymbol{D}^{-1/2}(\boldsymbol{A}+\boldsymbol{I})\boldsymbol{D}^{-1/2}\boldsymbol{X}\boldsymbol{W}\right)$$

$$\begin{aligned} \boldsymbol{A} \in \mathbb{R}^{n \times n} &:= \text{The adjacency matrix} \\ \boldsymbol{I} \in \mathbb{R}^{n \times n} &:= \text{The identity matrix} \\ \boldsymbol{D} \in \mathbb{R}^{n \times n} &:= \text{The degree matrix of } \boldsymbol{A}+\boldsymbol{I} \\ \boldsymbol{X} \in \mathbb{R}^{n \times d} &:= \text{The input data (i.e., the per-node feature vectors)} \\ \boldsymbol{W} \in \mathbb{R}^{d \times w} &:= \text{The layer's weights} \\ \sigma(.) &:= \text{The activation function (e.g., ReLU)} \end{aligned}$$


$f(\boldsymbol{X}, \boldsymbol{A}) \coloneqq \sigma\left(\boldsymbol{D}^{-1/2}(\boldsymbol{A}+\boldsymbol{I})\boldsymbol{D}^{-1/2}\boldsymbol{X}\boldsymbol{W}\right)$
Add self-loops
Normalize adjacency matrix
Aggregate
Update

$\boldsymbol{A} + \boldsymbol{I}$: This operation is simply adding ones along the diagonal entries of the adjacency matrix. This is the equivalent of adding self-loops to the graph where each node has an edge pointing to itself. The reason we need this is because when we perform message passing, each node should pass its vector to itself (since each node aggregates its own vector together with its neighbors). The matrix D is the degree matrix of $\boldsymbol{A} + \boldsymbol{I}$. This is a diagonal matrix where element i, j stores the total number of neighboring nodes to node i (including itself). That is where $d_{i,j}$ is the number of adjacent nodes (i.e., direct neighbors) to node i .

$$\boldsymbol{D} := \begin{bmatrix} d_{1,1} & 0 & 0 & \dots & 0 \\ 0 & d_{2,2} & 0 & \dots & 0 \\ 0 & 0 & d_{3,3} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & d_{n,n} \end{bmatrix}$$

The matrix $D^{-1/2}$ is the matrix formed by taking the reciprocal of the square root of each entry in D . As we will discuss in the next section, left and right multiplying $A + I$ by $D^{-1/2}$ can be viewed as “normalizing” the adjacency matrix. We will discuss what we mean by “normalizing” in the next section and why this is an important step; however, for now, the important point is that, like A , the matrix $D^{-1/2}(A + I\ )\ D^{-1/2}$ will also only have a nonzero entry at element i, j only if nodes i and j are adjacent.
For ease of notation, let’s let $\tilde{A}$ denote this normalized matrix .

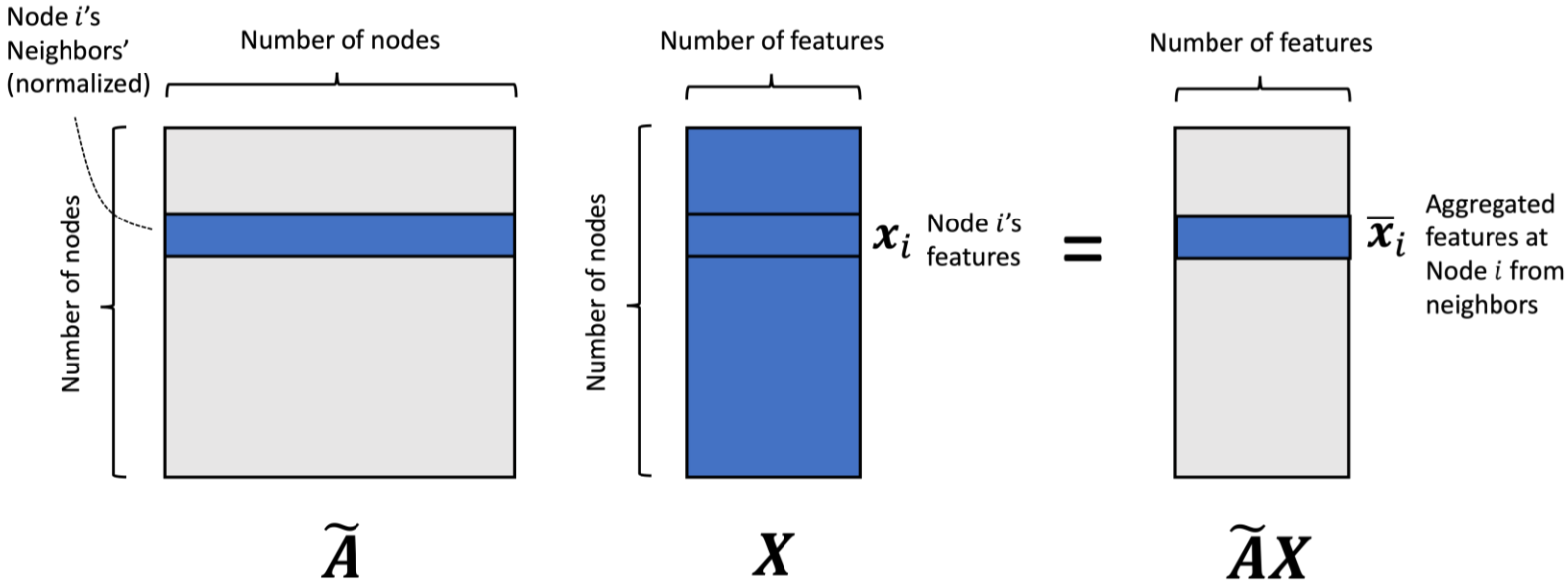
$$\tilde{\boldsymbol{A}} := \boldsymbol{D}^{-1/2}(\boldsymbol{A} + \boldsymbol{I})\boldsymbol{D}^{-1/2}$$

$$\tilde{\boldsymbol{A}}_{i,j} := \begin{cases} \frac{1}{\sqrt{d_{i,i}d_{j,j}}}, & \text{if there is an edge between node } i \text{ and } j \\ 0, & \text{otherwise} \end{cases}$$

$$\boldsymbol{D}^{-1/2} := \begin{bmatrix} \frac{1}{\sqrt{d_{1,1}}} & 0 & 0 & \dots & 0 \\ 0 & \frac{1}{\sqrt{d_{2,2}}} & 0 & \dots & 0 \\ 0 & 0 & \frac{1}{\sqrt{d_{3,3}}} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \frac{1}{\sqrt{d_{n,n}}} \end{bmatrix}$$

With this notation, we can simplify the graph convolutional layer function as follows:

$$f(\boldsymbol{X}, \boldsymbol{A}) := \sigma\left(\tilde{\boldsymbol{A}}\boldsymbol{X}\boldsymbol{W}\right)$$



let's turn to the matrix $\tilde{A}X$. This matrix-product is performing the aggregation function/message passing that we described previously. That is, for every feature, we take a weighted sum of the features of the adjacent nodes where the weights are determined by $\tilde{A}$.

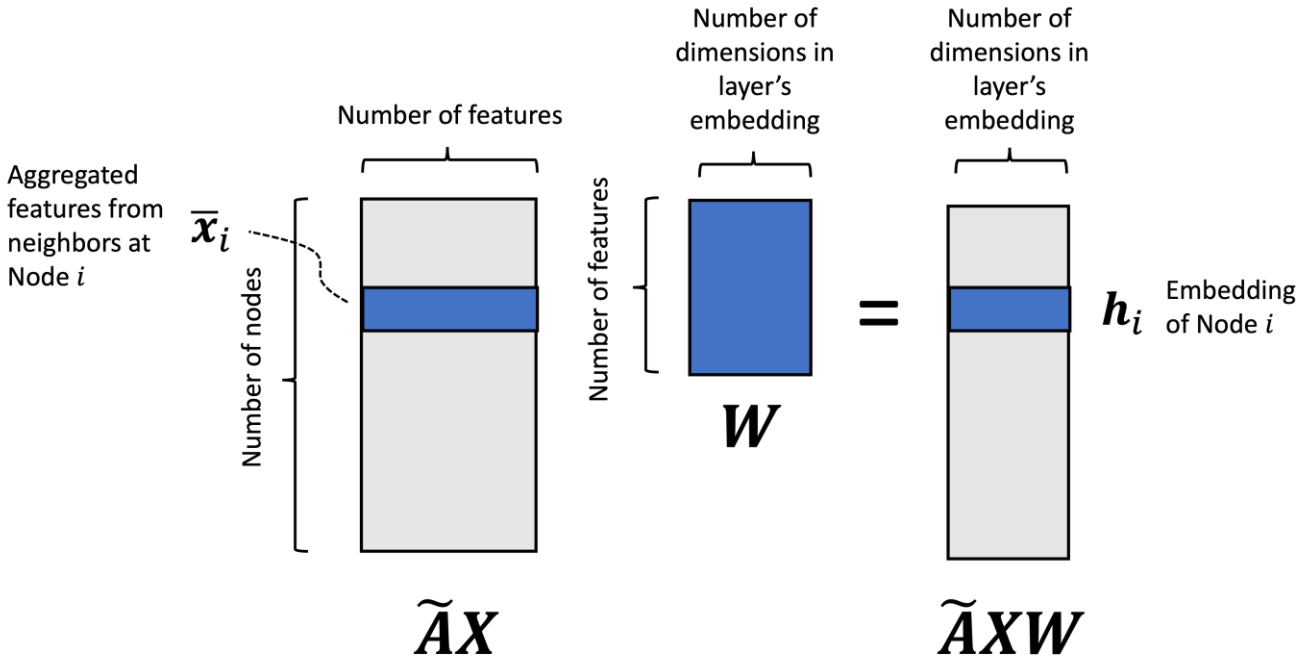
Let $\bar{x}_i$ denote the vector at node i representing the aggregated features. We see that this vector is given by :

It is computed by taking a weighted sum of the neighboring vectors where the weights are stored in the normalized adjacency matrix. We will discuss these neighbor weights in more detail in the next section, but it is important to note that these weights are not learned weights – that is, they are not model parameters. Instead, they are determined based only on the input graph itself.

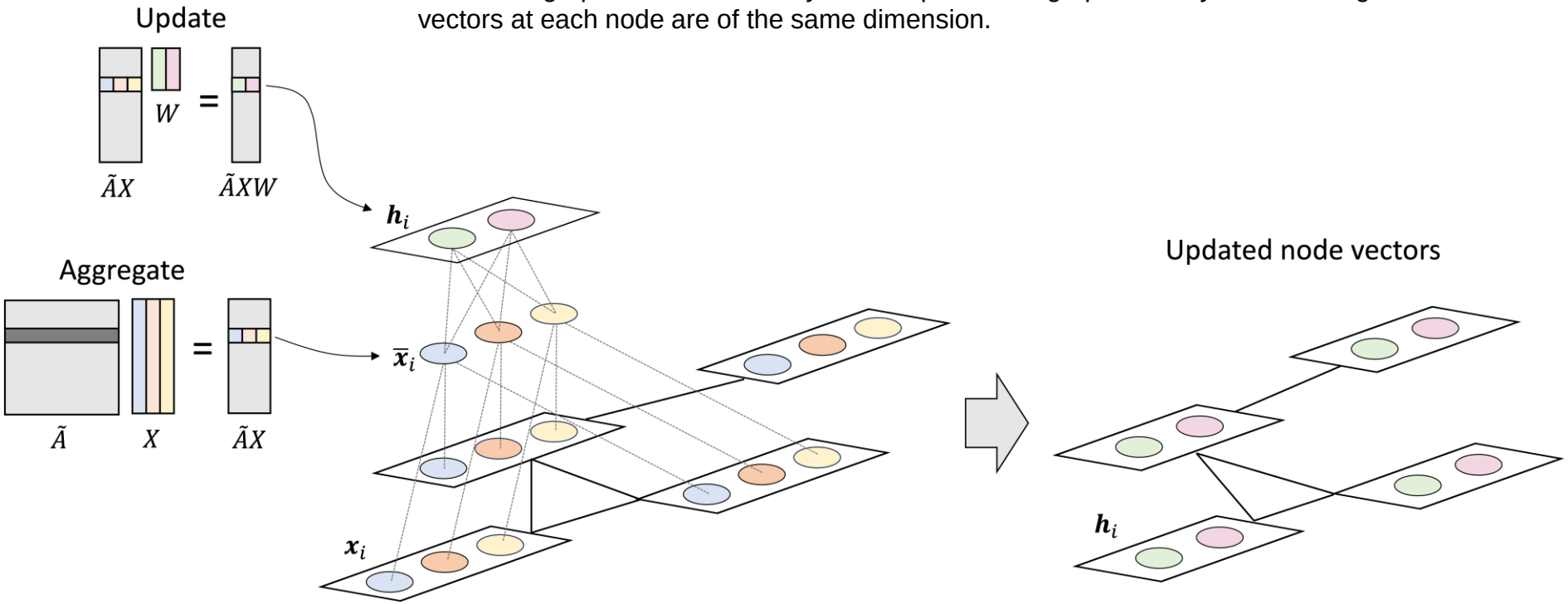
$$\begin{aligned}\bar{\boldsymbol{x}}_i &= \sum_{j=1}^{n} \tilde{a}_{i,j}\boldsymbol{x}_j \\ &= \sum_{j \in \text{Neigh}(i)} \tilde{a}_{i,j}\boldsymbol{x}_j \\ &= \sum_{j \in \text{Neigh}(i)} \frac{1}{\sqrt{d_{i,i}d_{j,j}}}\boldsymbol{x}_j\end{aligned}$$

So, where are the learned weights/parameters of the model?

They are stored in the matrix W. In the next matrix multiplication, we "update" the aggregated feature vectors according to these weights via $(\widetilde{A}\,X)\mathrm{W}$

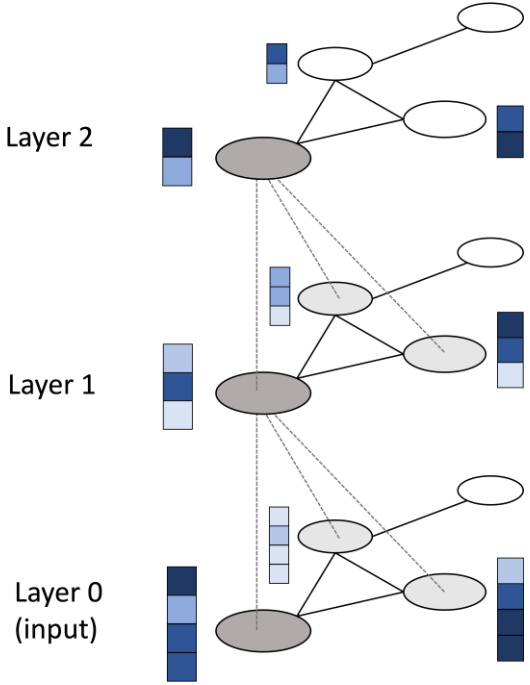


These vectors are then passed to the activation function σ before being output by the layer. This activation function injects non-linearity into the model. One key point to note is that the dimensionality of the weights vector W does not depend on the number of nodes in the graph. Thus, the graph convolutional layer can operate on graphs of any size as long as the feature vectors at each node are of the same dimension.



Now, so far we have discussed only a single graph convolutional layer. We can create a multi-layer GCN by stacking graph convolutional layers together where the output of one layer is fed as input to the next layer! That is, the embedded vector at each node h_i that is output by a graph convolutional layer can treated as input to the next layer . where H_1, H_2 and H_3 are the embedded node vectors at layers 1, 2 and 3 respectively. The matrices W_1 , W_2 and W_3 are the weight matrices that parameterize each layer A schematic illustration of stacked graph convolutional layers is depicted below:

$$\boldsymbol{H}_1 := f_{\boldsymbol{W}_1}(\boldsymbol{X}, \boldsymbol{A})$$
$$\boldsymbol{H}_2 := f_{\boldsymbol{W}_2}(\boldsymbol{H}_1, \boldsymbol{A})$$
$$\boldsymbol{H}_3 := f_{\boldsymbol{W}_3}(\boldsymbol{H}_2, \boldsymbol{A})$$

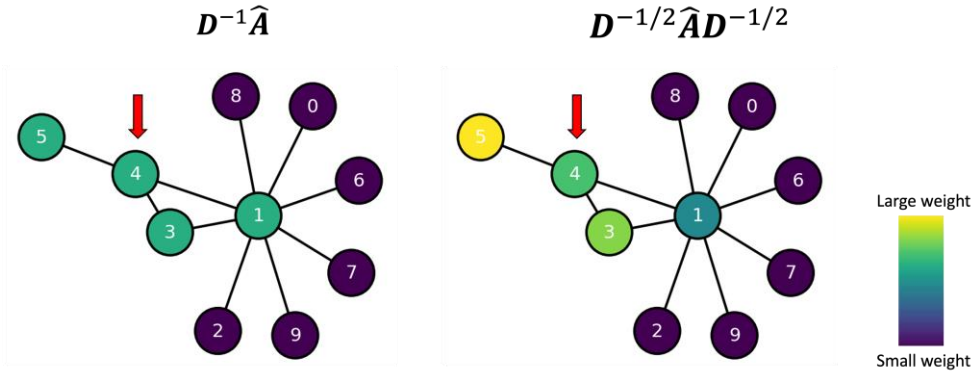


Normalizing the adjacency matrix

The normalized adjacency matrix $\tilde{A}$ is obtained by dividing each entry in the adjacency matrix A by the square root of the product of the degrees of the corresponding nodes.

Mathematically: $\tilde{A} = D^{-1/2}(A + I)\, D^{-1/2}$

The largest eigenvalue of $\tilde{A}$ is bounded by 1. This normalization ensures stable training and prevents numerical instability. Using the normalized adjacency matrix helps balance the influence of different nodes.



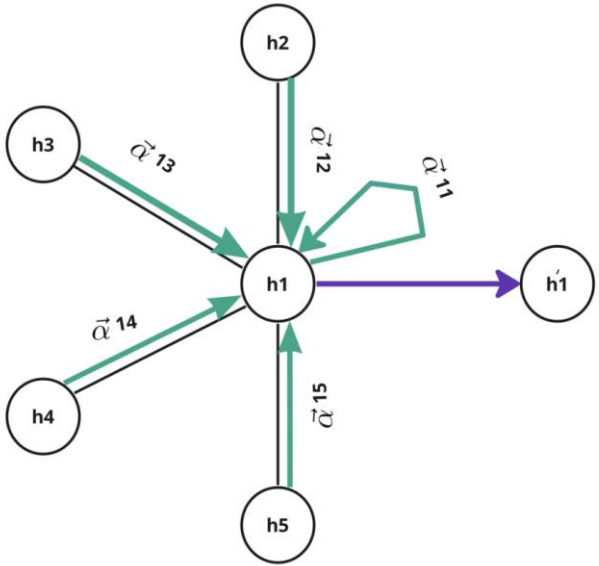
Graph Attention Networks

Graph Attention Networks (GATs) are a variant of GNNs that leverage attention mechanisms for feature learning on graphs. Compared to traditional GNNs, GATs offer a more nuanced approach to aggregating neighborhood information.

In standard GNNs, such as Graph Convolutional Networks (GCNs), the feature update of a node is typically the average of the features of its neighbors. This approach does not differentiate between the contributions of different neighbors.

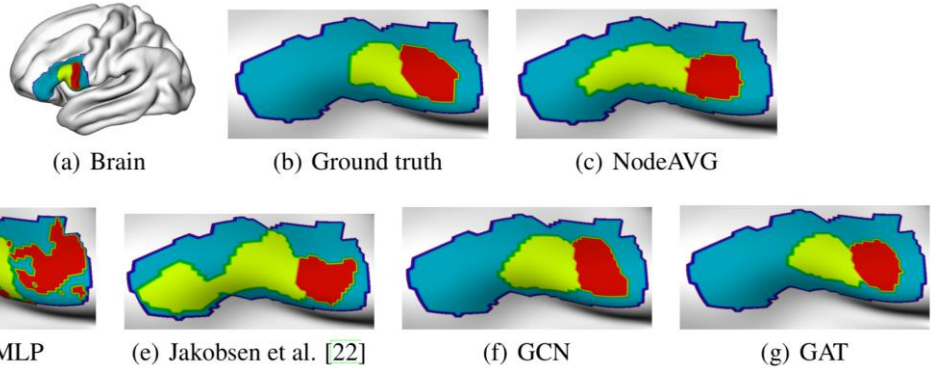
GATs, on the other hand, assign an attention coefficient to each neighbor, indicating the importance of that neighbor's features for the feature update of the node. These coefficients are computed using a shared self-attention mechanism, which calculates an attention score for each pair of nodes. The scores are then normalized across each node's neighborhood using a SoftMax function.

The attention-based approach allows GATs to assign different weights to different neighbors, providing a more flexible and potentially more expressive model. It also offers a level of interpretability, as the attention coefficients can be seen as indicating the importance of each neighbor.



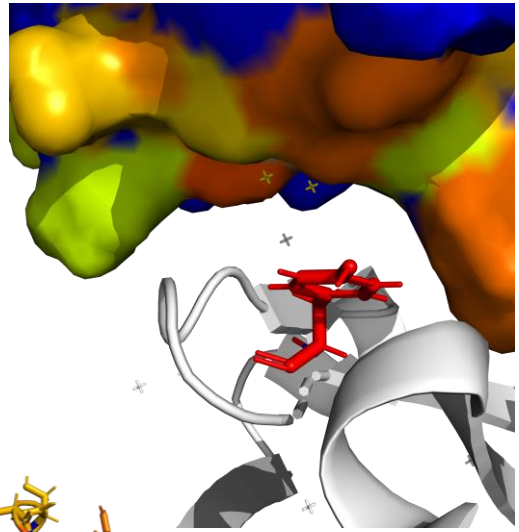
Mesh-based parcellation of the cerebral cortex

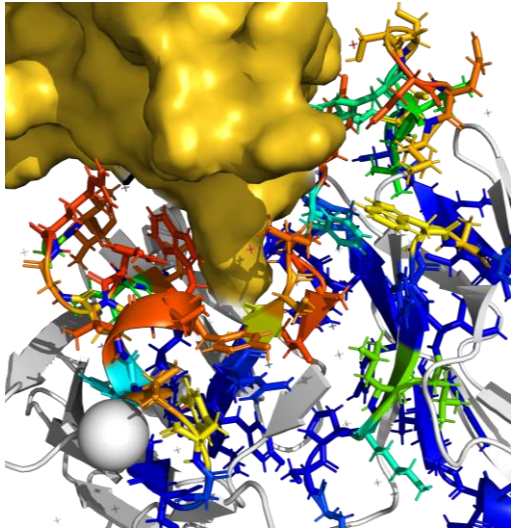
In this work (done in collaboration with the University of Cambridge Department of Psychiatry and the Montréal Neurological Institute), authors considered the task of cortical mesh segmentation (predicting functional regions for locations on a human brain mesh). They have leveraged functional MRI data from the Human Connectome Project (HCP). They found that graph convolutional methods (such as GCNs and GATs) can set state-of-the-art results, exploiting the underlying structure of a brain mesh better than all prior approaches, enabling more informed decisions.



(a) Brain (b) Ground truth (c) NodeAVG

(d) NodeMLP (e) Jakobsen et al. [22] (f) GCN (g) GAT





Neural paratope prediction

Antibodies are a critical part of the immune system, having the function of directly neutralising or tagging undesirable objects (the antigens) for future destruction. Here we consider the task of paratope prediction: predicting the amino acids of an antibody that participate in binding to a target antigen. A viable paratope predictor is a significant facilitator to antibody design, which in turn will contribute to the development of personalised medicine.

In this work, we build on Paraphed, the previous state of the art approach of Liberos et al. (2018), substituting its convolutional and recurrent layers with à torus convolutional and attentional layers, respectively. These layers have been shown to perform more favorably, and allowed us to, for the first time, positively exploit antigen data. We do this through cross-modal attention, allowing amino acids of the antibody to attend over the amino acids of the antigen (which could be seen as a GAT-like model applied to a bipartite antibody-antigen graph). This allowed us to set new state-of-the-art results on this task, along with obtaining insightful interpretations about the model's mechanism of action.

We instead decide to let a_{ij}be implicitly defined, employing self-attention over the node features to do so. This choice was with motivation, as self-attention was previously shown to be self-sufficient for state-of-the-art-level results on machine translation, as demonstrated by the Transformer architecture . Generally, we let αij be computed as a byproduct of an attentional mechanism $a{:}RN{\times}RN{\rightarrow}R$, which computes unnormalized coefficients eij across pairs of nodes i,j, based on their features:

$$e_{ij} = a(\overrightarrow{h_i}, \overrightarrow{h_j})$$

- e_{ij}: This typically represents an element, or a value associated with indices i and j. It could denote an interaction relationship, or some form of output between two entities indexed by i and j.
- $\overrightarrow{h_i}, \overrightarrow{h_j}$: These are likely representations of some features, attributes, or hidden states associated with the indices i and j. The bar over h often indicates that these are averaged or processed values.

The function a takes the two inputs $\overrightarrow{h_i}\ and\ \overrightarrow{h_j}$. This function could represent various operations, such as addition multiplication, or a more complex interaction (like a neural network activation function).

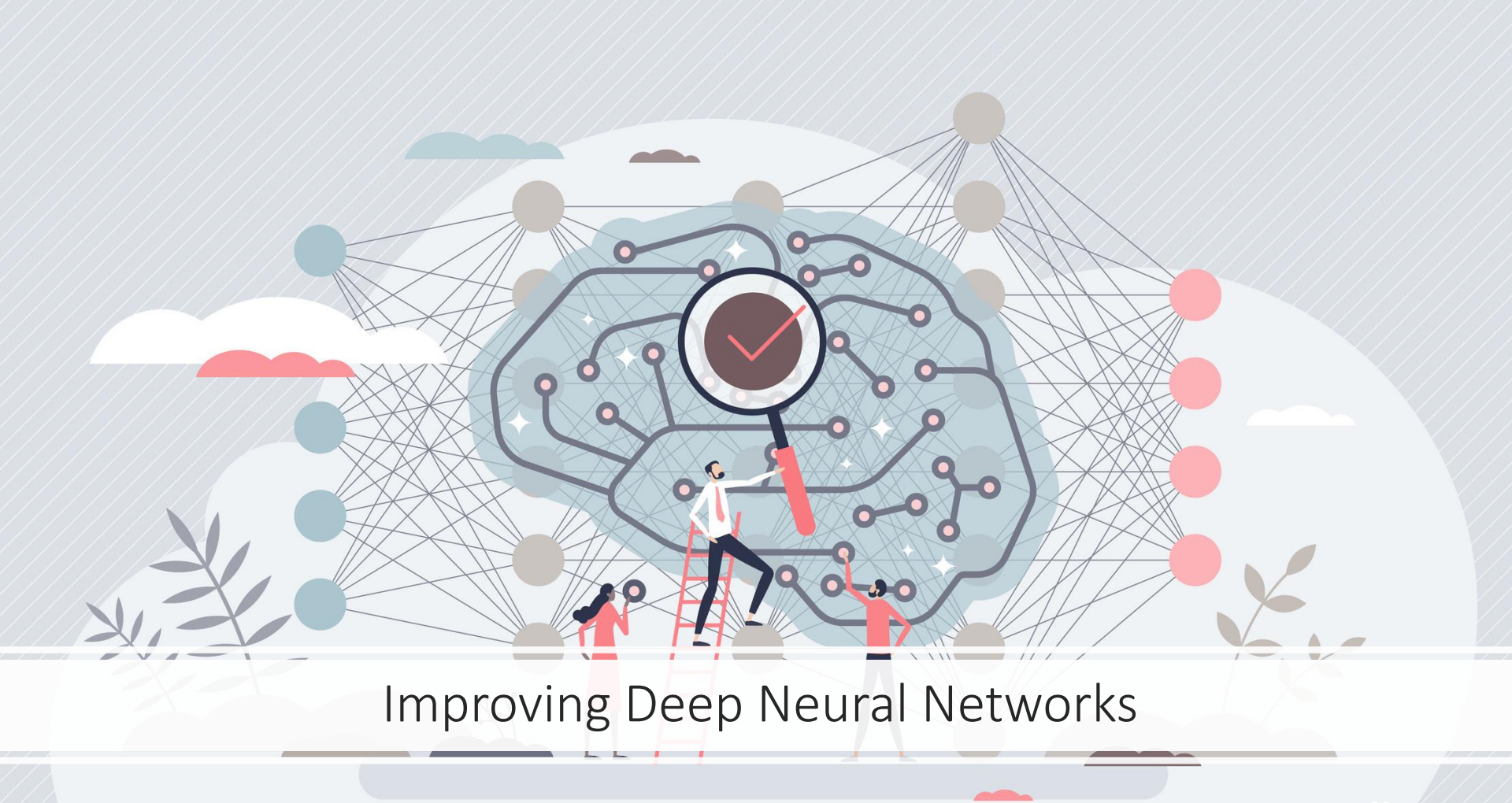
We inject the graph structure by only allowing node i to attend over nodes in its neighborhood $j \in \mathcal{N}_i$. These coefficients are then typically normalized using the softmax function, in order to be comparable across different neighborhoods:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} exp(e_{ik})}$$

- α_{ij} : This represents a normalized weight or attention score between entity i and entity j.
- e_{ij} : This is a value (often a score or energy) that measures the relationship between i and j.
- $\mathcal{N}_i$: This denotes a set or neighborhood of entities related to i which will be used to calculate the normalization.

Our framework is agnostic to the choice of attentional mechanism α in our experiments, we employed a simple single-layer neural network. The parameters of the mechanism are trained jointly with the rest of the network in an end-to-end fashion.

More information



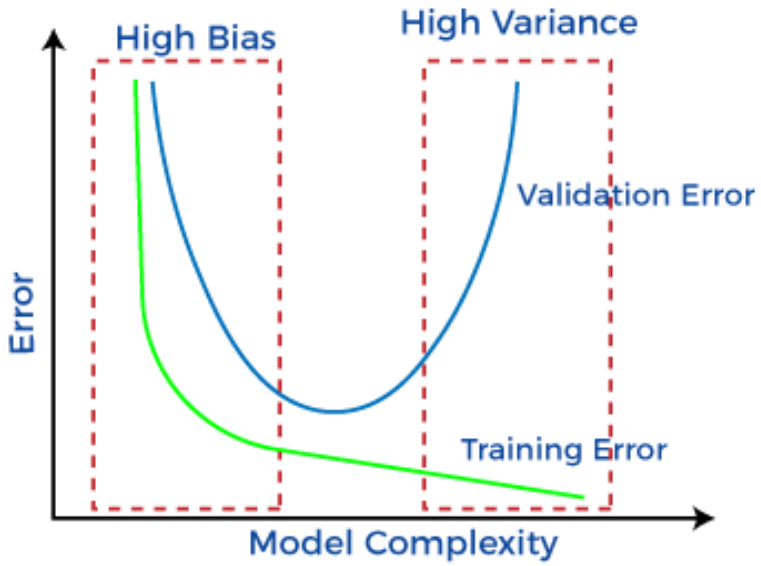
Improving Deep Neural Networks

Train / Valid / Test

- **Reduce Overfitting**: Separating data into these sets helps control overfitting and ensures the model does not merely memorize the training data.
- **Monitoring Performance**: The dev set allows for ongoing performance monitoring during training, facilitating adjustments that enhance performance without influencing the ultimate evaluation results.
- **Final Validation**: The test set provides a robust measure of how the model will perform in real-world scenarios, critical for deploying neural networks in practice.

Bias / Variance

- **Regularization Techniques**: Apply methods like L1 (Lasso) or L2 (Ridge) regularization to penalize excessive complexity in the model weights and prevent overfitting.
- **Dropout Regularization**: Introduce dropout layers in the neural network to randomly deactivate a subset of neurons during training, reducing reliance on any single feature and promoting broader learning.
- **Use More Data**: Larger training datasets can help the model generalize better, reducing the impact of noise and getting more representative learning of the underlying patterns.
- **Ensemble Methods**: Combine predictions from multiple models (like bagging or boosting) to reduce variance without significantly increasing bias.



Bias-Variance Tradeoff

- **Concept**: The bias-variance tradeoff is the balance between bias and variance that affects the model's overall prediction error. A model with low bias tends to have high variance and vice versa. The goal is to find a sweet spot where both bias and variance are minimized, leading to optimal generalization.

- **Adjusting for Tradeoff**: Monitor training and validation performances to understand where a model stands in terms of bias and variance. For instance, if the validation error is significantly higher than the training error, it indicates high variance, prompting you to apply regularization techniques.

If your algorithm has a high bias:

- Try to make your NN bigger (size of hidden units, number of layers)
- Try a different model that is suitable for your data.
- Try to run it longer.
- Different (advanced) optimization algorithms.

If your algorithm has a high variance:

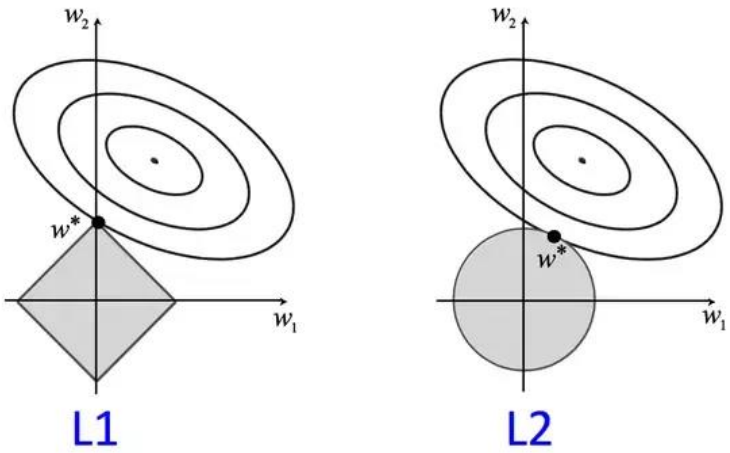
- More data.
- Try regularization.
- Try a different model that is suitable for your data.

Regularization

Regularization is a set of techniques used in machine learning, including neural networks, to prevent overfitting and improve the model's generalization ability.

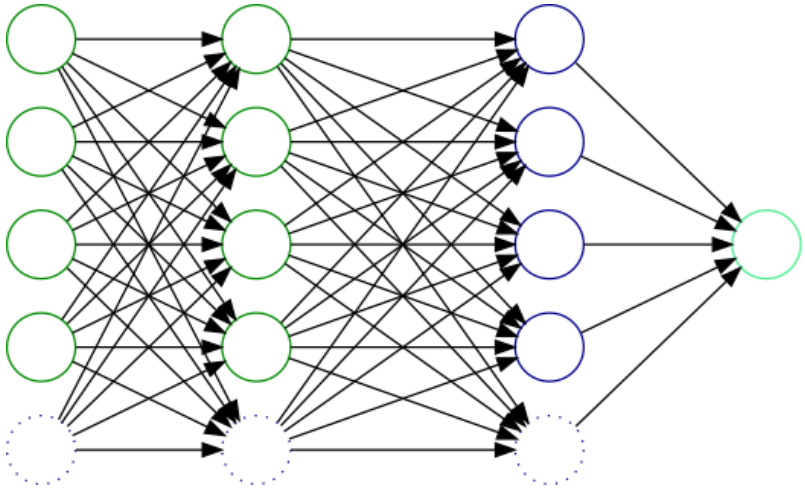
Overfitting occurs when a model learns not only the underlying patterns in the training data but also the noise and outliers, resulting in poor performance on unseen data.

- L1 Regularization (Lasso Regularization)
- L2 Regularization (Ridge Regularization)
- Dropout
- Early Stopping
- Data Augmentation
- Batch Normalization
- Weight Constraints
- Ensemble Methods

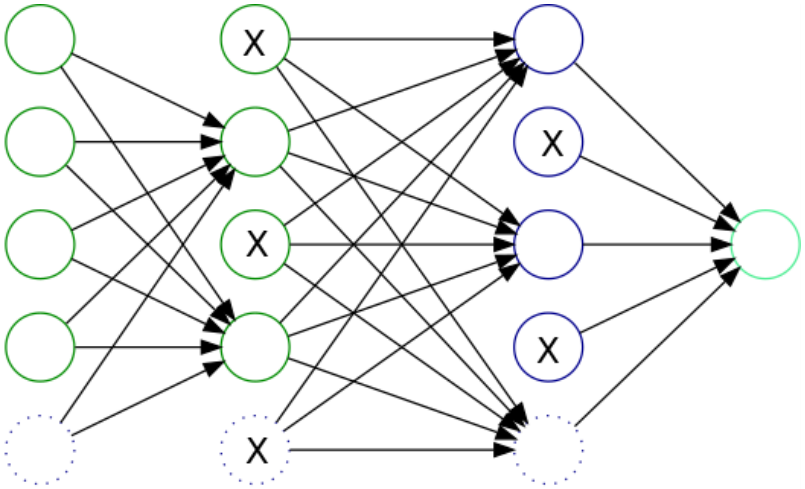


L1 Regularization (Lasso Regularization): Adds the absolute value of the magnitude of coefficients as a penalty term to the loss function. It can lead to sparse weight matrices by driving some weights to zero, effectively performing feature selection.

L2 Regularization (Ridge Regularization): Adds the square of the magnitude of coefficients as a penalty term. It encourages smaller weights but does not necessarily drive them to zero, which can help in cases where all features may have some contribution.



Without dropout



With dropout

Dropout : Dropout is a regularization technique that randomly "drops out" a fraction of neurons during training. By preventing the network from becoming too reliant on any one neuron, it helps to reduce overfitting and promotes a more robust feature representation.

What you should remember about dropout

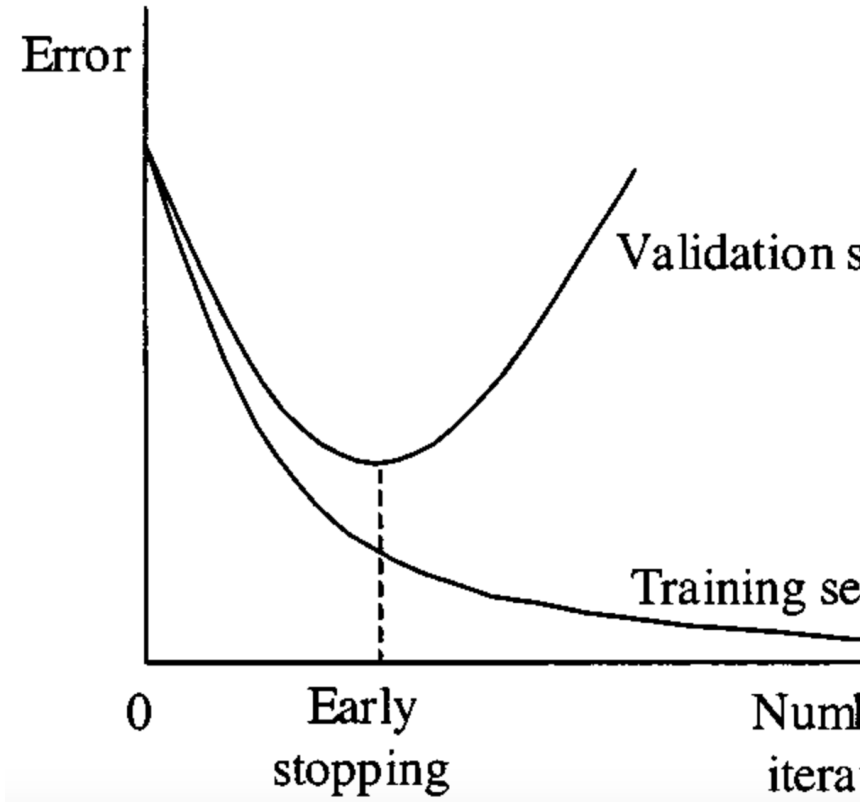
Dropout is a regularization technique.

You only use dropout during training. Don't use dropout (randomly eliminate nodes) during test time.

Apply dropout both during forward and backward propagation.

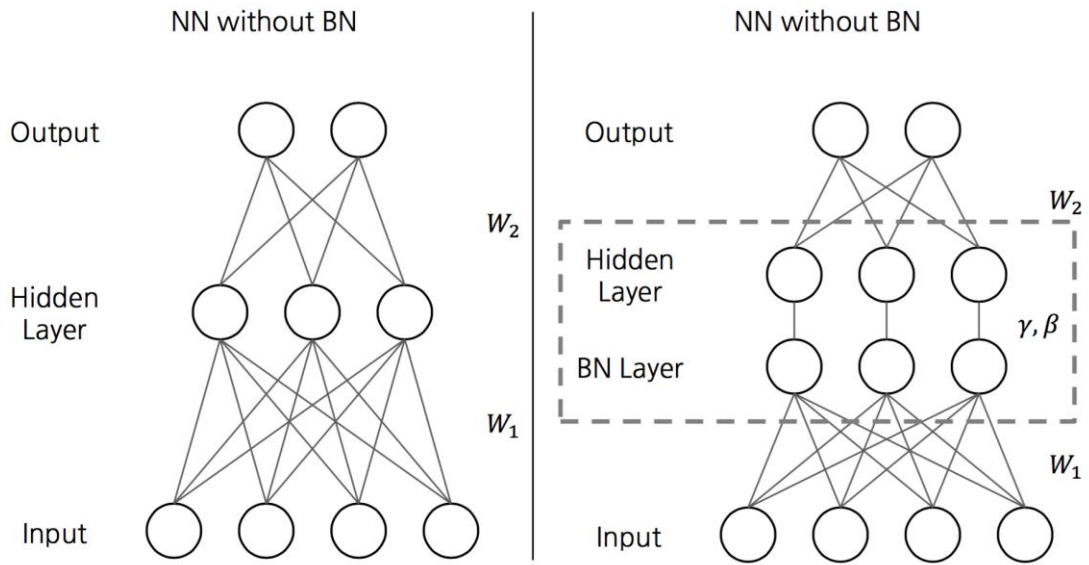
During training time, divide each dropout layer by keep_prob to keep the same expected value for the activations. For example, if keep_prob is 0.5, we will, on average, shut down half the nodes, so the output will be scaled by 0.5 since only the remaining half contribute to the solution. Dividing by 0.5 is equivalent to multiplying by 2. Hence, the output now has the same expected value. You can check that this works even when keep_prob has values other than 0.5.

Early Stopping: Early stopping involves monitoring the model's performance on a validation set during training. Training can be halted when the performance starts to degrade (indicating overfitting). This way, the best model can be saved before it begins to memorize the training data.



Data Augmentation: This technique involves artificially increasing the size of the training dataset by creating modified versions of existing data. For example, in image recognition tasks, images can be rotated, flipped, or altered in color and brightness. Exposing the model to a wider variety of inputs helps it generalize better.

Batch Normalization: Batch normalization normalizes the inputs to each layer within a mini-batch. This can improve training speed and stability and has a slight regularization effect, as it introduces some noise during training.



- **Weight Constraints**: Techniques like weight clipping or norm constraints prevent the weights from growing too large, which can help control overfitting. These approaches enforce certain limits on the magnitude of the weights.

- **Ensemble Methods**: Enssembling involves training multiple models and combining their predictions. Techniques like bagging and boosting create a more robust overall predictive model and can reduce variance.

Benefits of Regularization

Improved Generalization: Regularization techniques help ensure that the model performs well not just on training data but also on unseen data.

Reduced Complexity: By discouraging overly complex models (e.g., models with too many parameters), regularization can lead to simpler models that are easier to interpret.

Stability: Regularized models can be more stable in the presence of noisy data, making them more robust in practice.

Normalizing inputs

Common techniques for normalization include

Min-Max scaling

(scaling values to a specified range, often [0, 1]),

Z-score normalization

(standardizing values to have a mean of 0 and a standard deviation of 1),

Robust scaling

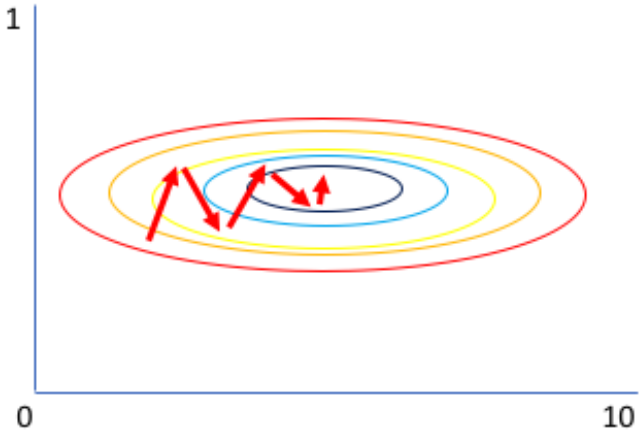
(using median and interquartile range to reduce sensitivity to outliers).

Faster Convergence : Normalization helps in speeding up convergence during training. When inputs are on the same scale, the optimization algorithm can move more smoothly through the parameter space. This can reduce the number of epochs needed to train the model effectively.

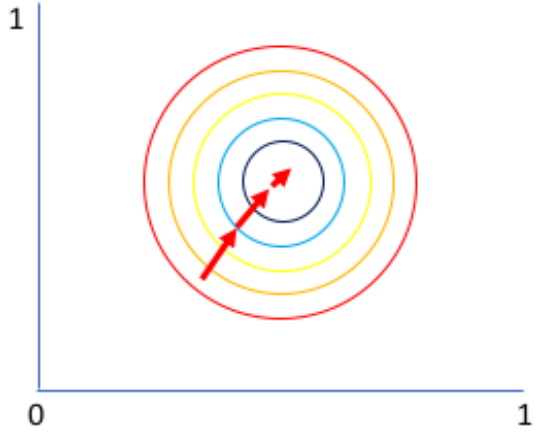
Reduced Sensitivity to Feature Scale : Different features may have different ranges (e.g., age [0-100] vs. income [0-100,000]). Normalization ensures that all features contribute equally to the distance calculations, preventing large-scale features from dominating the training process.

Helps with Gradient Descent : Normalized inputs lead to more stable and efficient gradient descent. When gradients are derived from input values that are on different scales, they can result in erratic updates to the model weights. Normalizing inputs can produce more consistent gradients.

Why normalize?



Gradient of larger parameter dominates the update



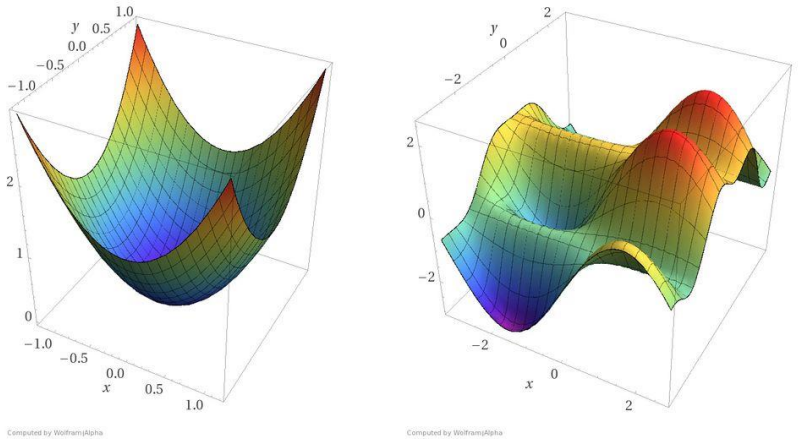
Both parameters can be updated in equal proportions

Improved Model Performance : Normalization often leads to better accuracy and reduces overfitting. When the model can learn more efficiently from the data scaled equally, it can generalize better on unseen data.

Mitigating Vanishing and Exploding Gradients : Normalizing input can help mitigate issues like vanishing and exploding gradients, especially in deep neural networks. By keeping the values in a more consistent range, the model can be more stable throughout training.

Facilitating Regularization : Some regularization techniques assume inputs are standardized. With normalized inputs, regularization methods like L1 or L2 norm can become more effective, leading to better generalization.

Easier to Diagnose : When inputs are normalized, individuals can more easily explore the model's behavior, understand its predictions, and diagnose any issues without the complexity added by input scales.



Optimization algorithms

Training an NN with large data is slow. So, it's a good idea to find an optimization algorithm that runs faster.

mini-batch gradient descent

Mini-batch gradient descent is an optimization algorithm used to train neural networks and other machine learning models. It is a variation of the standard gradient descent algorithm that improves the efficiency and convergence speed of the training process.

- **Gradient Descent**: In traditional gradient descent, we calculate the gradient of the loss function using the entire training dataset to update the model parameters. This can be computationally expensive and slow, especially for large datasets.
- **Mini-Batch**: Instead of using the entire dataset, mini-batch gradient descent splits the training set into smaller subsets called mini-batches. Each mini-batch contains a fixed number of samples, typically ranging from 32 to 256 samples (though the size can be adjusted based on the dataset and model).

Mini-batch gradient descent leads to several variants based on how the updates are processed:

Stochastic Gradient Descent (SGD): A specific case where mini-batches contain only one sample, leading to frequent updates with high variance.

Batch Gradient Descent: The case where the mini-batch size is equal to the size of the training dataset.

Adaptive Methods: Algorithms like Adam, RMSprop, and AdaGrad build upon mini-batch gradient descent by adapting learning rates based on past gradients.

Exponentially weighted averages

Exponentially weighted averages (EWA) are a statistical technique used to compute the average of a sequence of values while giving greater importance to more recent values. This method is particularly useful in scenarios where the most recent data points are more relevant, allowing for responsiveness to changes over time.

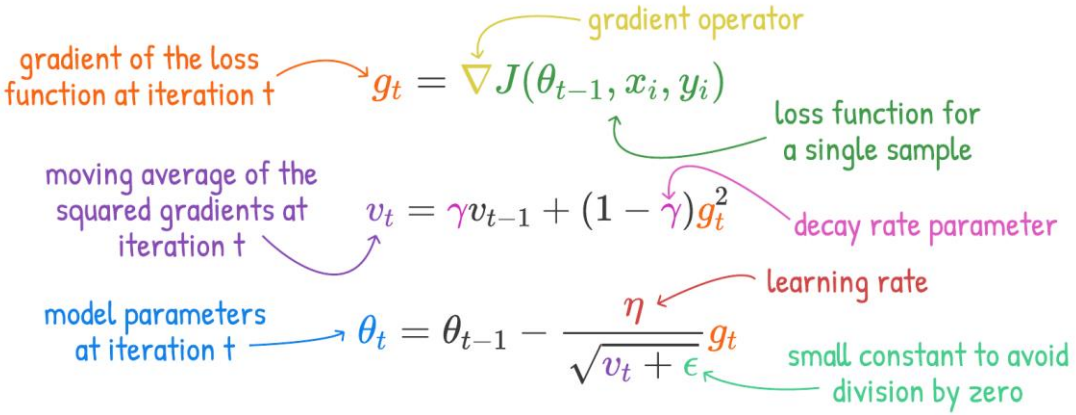
Benefits of Exponentially Weighted Averages in Neural Networks

- **Adaptive Learning Rates** : EWA is utilized in various optimization algorithms (like Adam) to create adaptive learning rates based on the history of gradients. By maintaining a weighted average of past gradients and their squared values, these algorithms adjust learning rates dynamically, which can lead to faster convergence and improved performance.
- **Improving Convergence** : By incorporating EWA into the optimization process, models can converge more smoothly and efficiently, especially in complex or noisy environments. The averaging process can help the model escape local minima by stabilizing the updates.
- **Regularization Effect** : The inherent smoothing property of EWA can act as a form of regularization, preventing overfitting to the training data. As the model learns, the averaging effect can help ensure that it captures broader trends rather than noise in the data.
- **Monitoring Performance** : Exponentially weighted averages can also be used to monitor the performance of the model over training epochs. By applying EWA to loss or accuracy metrics, you can derive a smoother curve that is easier to analyze and interpret.

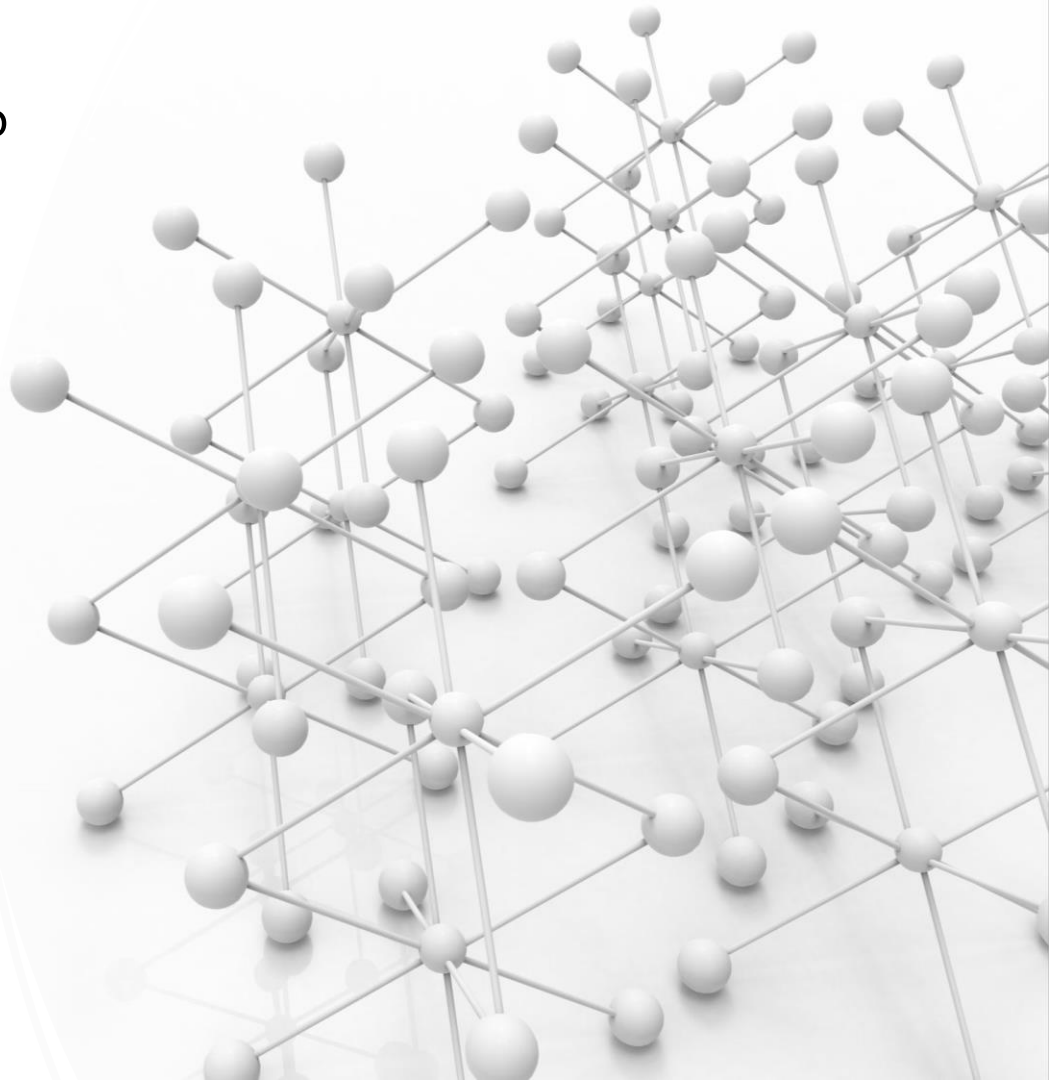
RMSprop

RMSPROP

RMSprop optimizer adjusts the learning rates of each parameter based on the root mean square of the past gradients. It uses a moving average of the squared gradients to normalize the gradient updates and prevent oscillations in optimization.



When do we use RMSprop?



- **Non-Stationary Objectives**: When the loss surface is changing or the data distribution itself is shifting, RMSprop's adaptive learning rate helps in managing the updates more effectively. For example, this can occur in reinforcement learning scenarios or when working with time-series data.
- **Deep Neural Networks**: In deeper architectures, issues with vanishing and exploding gradients can arise. RMSprop helps mitigate these issues by adapting to the scale of the gradients, making it suitable for deep feedforward networks, convolutional neural networks (CNNs), and recurrent neural networks (RNNs).
- **Sparse Gradients**: For networks dealing with high-dimensional data or tasks where only a few weights are updated significantly (e.g., natural language processing tasks or models with large embedding layers), RMSprop can effectively handle sparse gradients by scaling updates based on recent gradient behavior.

When do we use RMSprop?

RNNs and LSTMs: Due to the typically complex and varying nature of the loss landscapes when training recurrent models, RMSprop is frequently used in training RNNs and long short-term memory (LSTM) networks. Its ability to stabilize updates can significantly enhance training stability.

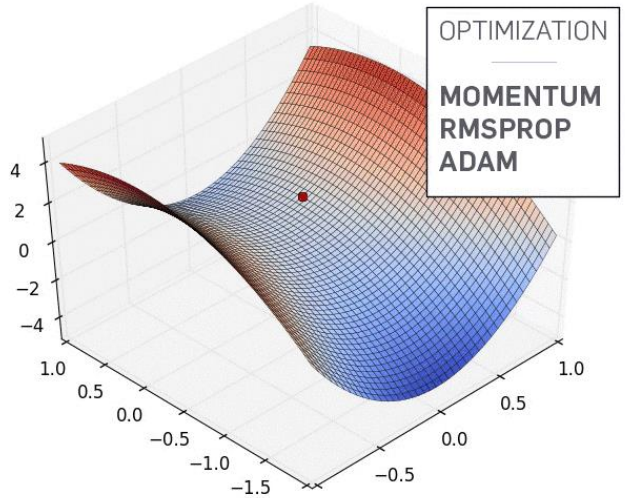
Transfer Learning: When fine-tuning pre-trained models on a new task, RMSprop can help adjust learning rates for parameters that have already been optimized historically, facilitating quicker adaptation to new tasks.

When You Have Limited Hyperparameter Time: RMSprop tends to be less sensitive to the choice of the learning rate compared to other optimizers like standard SGD, so it can be a good choice when you want to optimize performance without extensive hyperparameter tuning.

General Optimization Challenges: In cases where you face difficulties optimizing the loss landscape due to sharp cliffs or plateaus, RMSprop may provide a more robust path for convergence by dynamically adjusting the learning rates.

Adam

Adam (Adaptive Moment Estimation)
is one of the most popular optimization algorithms used in training neural networks due to its adaptive learning rate capabilities and efficiency.

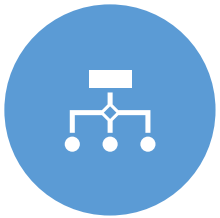


When do we use Adam?

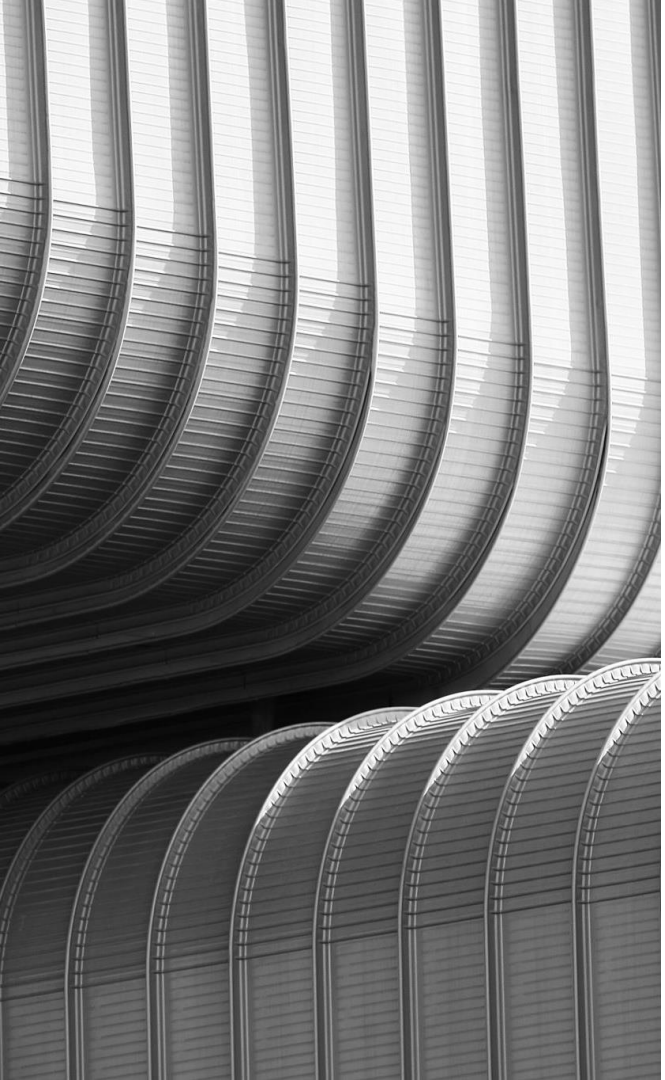
Deep Learning Models: Adam is well-suited for training deep neural networks, including convolutional neural networks (CNNs) and recurrent neural networks (RNNs). Its ability to adaptively adjust learning rates helps in efficiently navigating the complex loss landscapes that often accompany deep architectures.

Non-Stationary Objectives: Similar to RMSprop, Adam is effective in scenarios where the data distribution may change over time or where the loss surface is not static, making it useful for applications such as reinforcement learning or online learning.

High-Dimensional Spaces: Adam is particularly useful for optimizing in high-dimensional parameter spaces, where the gradient can be sparse or noisy. Its moments-based approach helps stabilize the updates and improve convergence speed.



Sparse Gradients: Adam performs well with data that exhibits sparsity (e.g., natural language processing tasks). The algorithm's ability to maintain separate adaptive learning rates for each parameter helps manage the noise and irregularity in gradient updates.



When do we use Adam?

Ideal for Transfer Learning: Adam can be an excellent choice for fine-tuning pre-trained models on new datasets, as it adapts to previously trained weights and can quickly adjust during training on the new task.

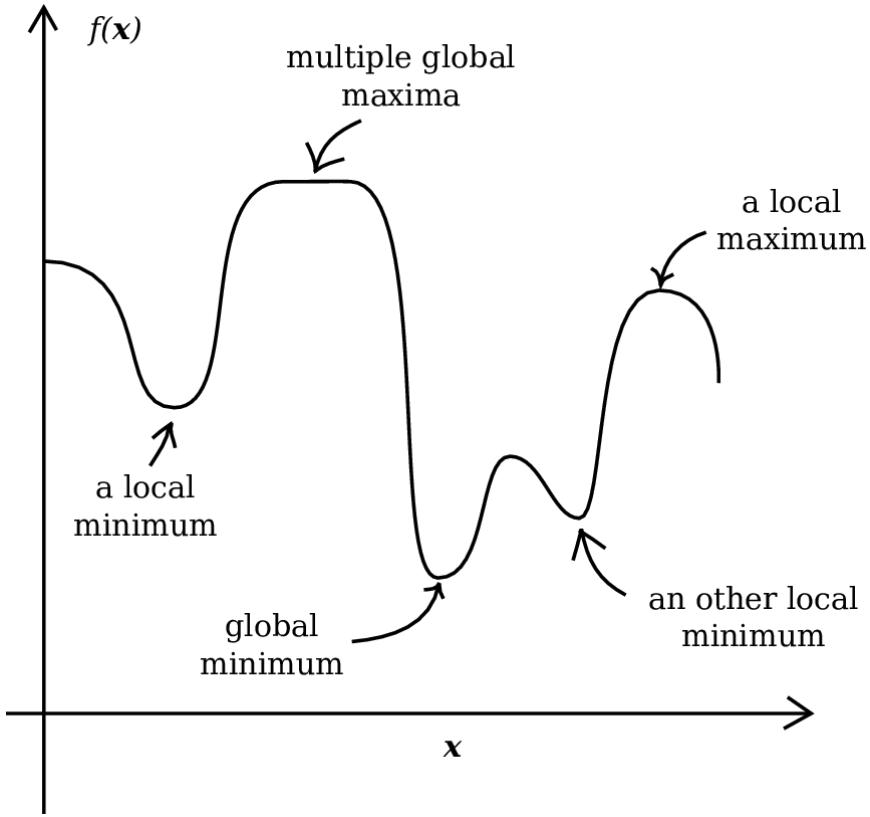
Training GANs and Other Complex Models: In generative models like Generative Adversarial Networks (GANs), Adam is commonly used due to its ability to stabilize training with fluctuating objectives.

Requires Less Tuning: Adam generally requires minimal tuning of hyperparameters. The default settings (learning rate, beta values) tend to work well in many cases, which is beneficial for practitioners looking for a good starting point.

Robustness to Hyperparameter Settings: Adam's adaptive nature makes it less sensitive to the choice of learning rate compared to plain stochastic gradient descent (SGD). Therefore, it can be a practical choice when one seeks robustness in convergence across different models and datasets.

The problem of local optima

- The normal local optima is not likely to appear in a deep neural network because data is usually high dimensional. For point to be a local optima it has to be a local optima for each of the dimensions which is highly unlikely.
- It's unlikely to get stuck in a bad local optima in high dimensions, it is much more likely to get to the saddle point rather to the local optima, which is not a problem.
- Plateaus can make learning slow:
- 1-Plateau is a region where the derivative is close to zero for a long time.
- 2-This is where algorithms like momentum, RMSprop or Adam can help.



Hyperparameter

Network hyperparameters

Hyperparameter	min	max	default
init. learning rate (log)	1×10^{-7}	0.5	0.001
learning rate schedule (choice)	{inv, fixed}		fixed
inv schedule[2]: lr. half-life (cond)	1	50	25
inv schedule[2]: p (cond)	0.5	1.	0.71
momentum	0	0.99	0.6
weight decay (log)	5×10^{-7}	0.05	0.0005
batch size B	10	1000	100
number of layers	1	6	1
input dropout (Boolean)	{true, false}		false
input dropout rate (cond)	0.05	0.8	0.4

Fully connected layer hyperparameters

Hyperparameter	min	max	default
number of units	128	6144	1024
weight filler type (choice)	{Gaussian, Xavier}		Gaussian
Gaussian weight init σ (log; cond)	1×10^{-6}	0.1	0.005
bias init (choice)	{const-zero, const-value}		const-zero
constant value bias filler (cond)	0	1	0.5
dropout enabled (Boolean)	{true, false}		true
dropout ratio (cond)	0.05	0.95	0.5

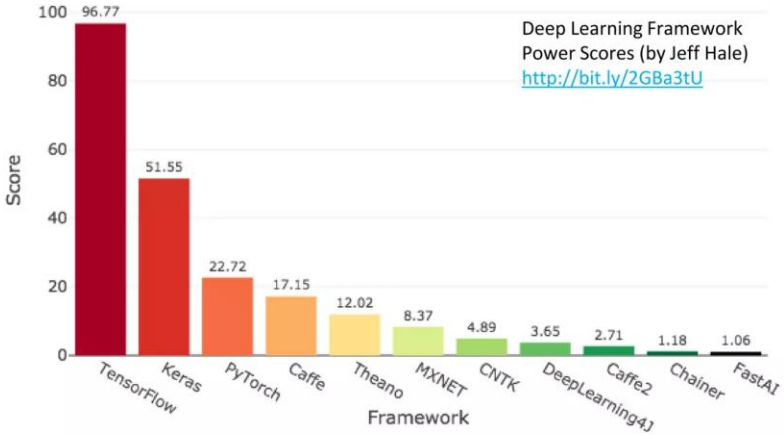
Hyperparameters	**Symbol**
Number of convolutional layers	N_c
Number of kernels of each convolutional layer	$kN_{i,i\in N_c}$
Kernel size in each convolutional layer	$kS_{i,i\in N_c}$
Activation function in each convolutional layer	$aF_{i,i\in N_c}$
Pooling size (if any) after each convolutional layer	$pS_{i,i\in N_c}$
Number of dense layers	N_d
Connectivity pattern of each dense layer	$P_{i,i\in N_d}$
Number of neurons of each dense layer	$nN_{i,i\in N_d}$
Weight regularization in each dense layer	$R_{i,i\in N_d}$
Dropout (none or 50%)	$R_{dropout}$
Batch size	S_{batch}
Learning rule	L_{rule}
Learning rate	L_{rate}

As a data scientist, machine learning engineer, or AI practitioner, your role in setting the best hyperparameters for a neural network is crucial. No one-size-fits-all method for tuning hyperparameters exists, but your expertise and the strategies outlined here can help you find optimal settings.

Grid Search: This method systematically explores a manually specified subset of the hyperparameter space. You define a grid of hyperparameter values to try for parameters like learning rate, batch size, and architecture configurations. This approach can be computationally expensive but is thorough.
Random Search: Instead of testing every combination of parameters, random search selects combinations at random. Research shows that random search can outperform grid search, especially when only a few hyperparameters significantly affect the model's performance.
Bayesian Optimization: This probabilistic model builds a surrogate model based on past evaluations of hyperparameters. It tries to predict the outcomes of untested sets of hyperparameters and chooses the next set to evaluate based on these predictions, seeking to find the optimal parameters efficiently.
•**Hyperband : Hyperband is a bandit-based approach to hyperparameter optimization that adaptively allocates resources to promising configurations. It uses a combination of random search and early stopping to optimize the hyperparameter search process, making it more efficient and giving you confidence in its ability to find the best parameters**.
Cross-Validation: Use cross-validation to evaluate the model's performance on different subsets of the data. This helps identify hyperparameters that generalize well rather than those that overfit the training set.
Learning Rate Schedulers: Experiment with different learning rates and implement learning rate schedules (e.g., step decay, exponential decay, or cyclical learning rates). Finding the correct learning rate can significantly affect model performance.
Early Stopping: Based on validation performance, use early stopping. This stops the training process once the validation loss increases, helping to avoid overfitting and providing insight into the optimal training duration.
Visualization: Utilize tools like TensorBoard, Weights & Biases, or other visualization libraries to visualize training metrics like loss and accuracy. This can help identify patterns or trends, suggesting hyperparameter adjustments may be beneficial.

Deep learning frameworks

Deep Learning Frameworks



Factors to consider:

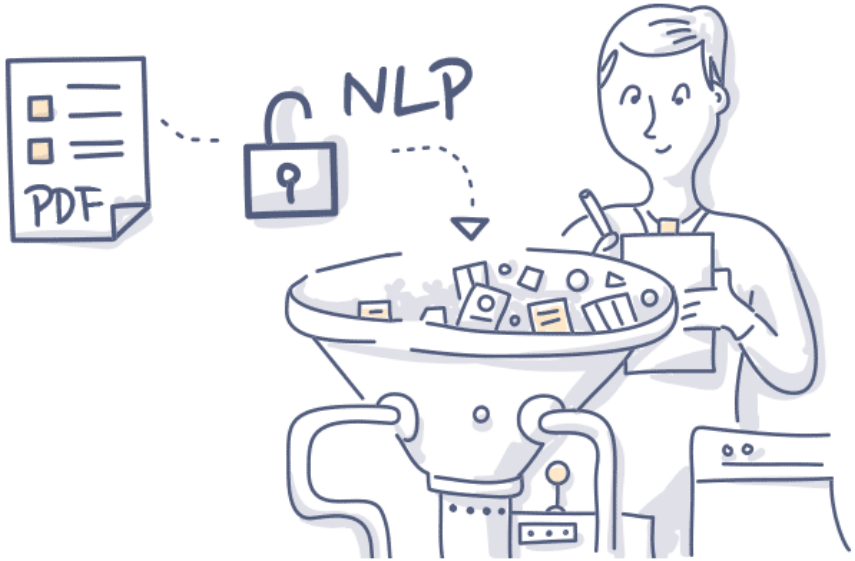
- Learning curve
- Speed of development
- Size and passion of community
- Number of papers implemented in framework
- Likelihood of long-term growth and stability
- Ecosystem of tooling

1. TensorFlow
2. Keras
3. PyTorch
4. Caffe
5. theano
6. APACHE mxnet
7. Microsoft CNTK
8. DL4J
9. Caffe2
10. Chainer
11. fast.ai

Natural Language Processing (NLP)

Natural language processing (NLP) is a field of computer science and a subfield of AI that aims to make computers understand human language.

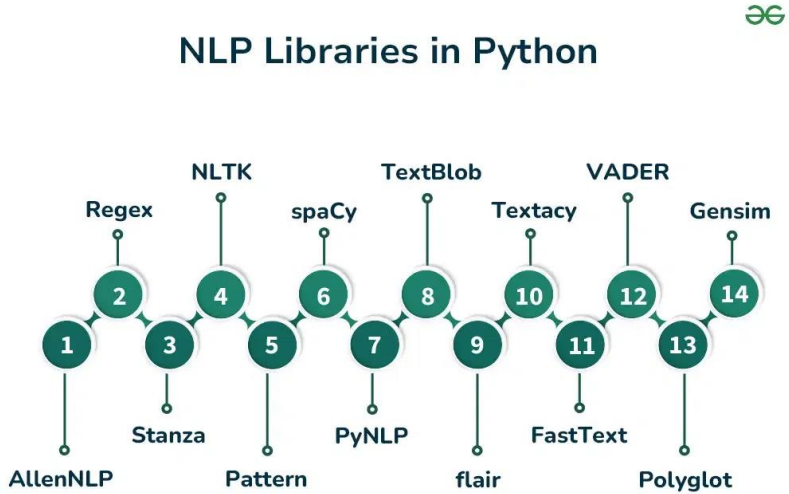
NLP uses computational linguistics, which is the study of how language works, and various models based on statistics, machine learning, and deep learning. These technologies allow computers to analyze and process text or voice data and to grasp their whole meaning, including the speaker's or writer's intentions and emotions. NLP powers many applications that use language, such as text translation, voice recognition, text summarization, and chatbots. You may have used some of these applications yourself, such as voice-operated GPS systems, digital assistants, speech-to-text software, and customer service bots. NLP also helps businesses improve their efficiency, productivity, and performance by simplifying complex tasks that involve language.



R packages for NLP

From sources across the web

OpenNLP	Quanteda	Stringr
TM	Korpus	Spacyr
Text2vec	Tidytext	LSA
Rcmdrplugin.temis		

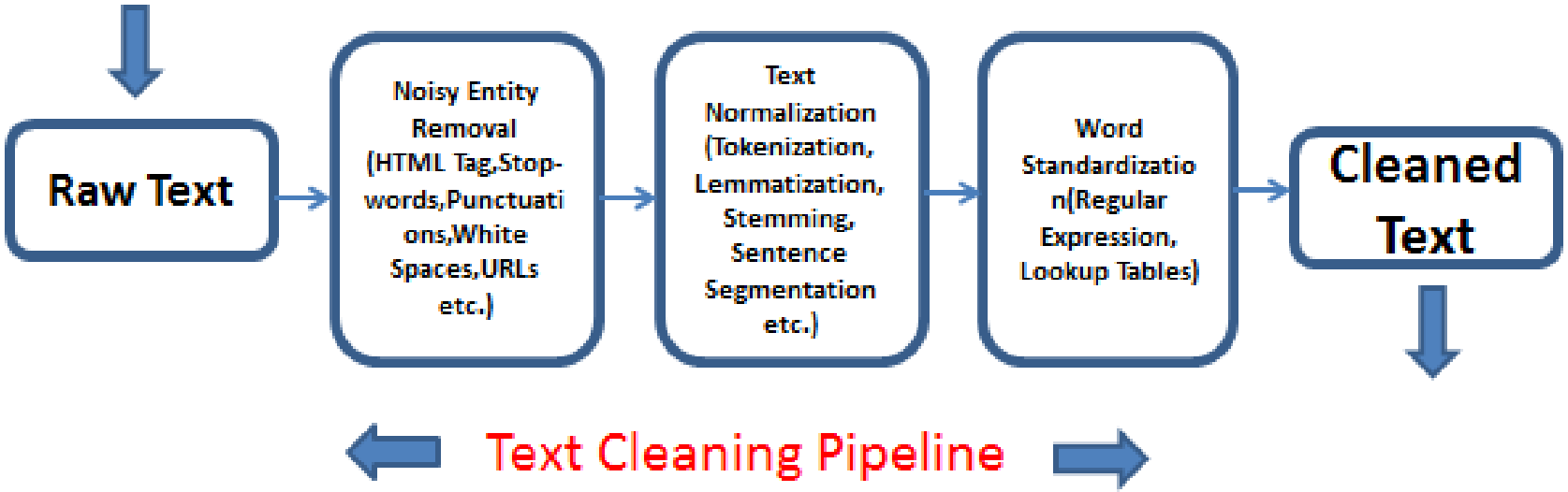


Basic concepts of NLP

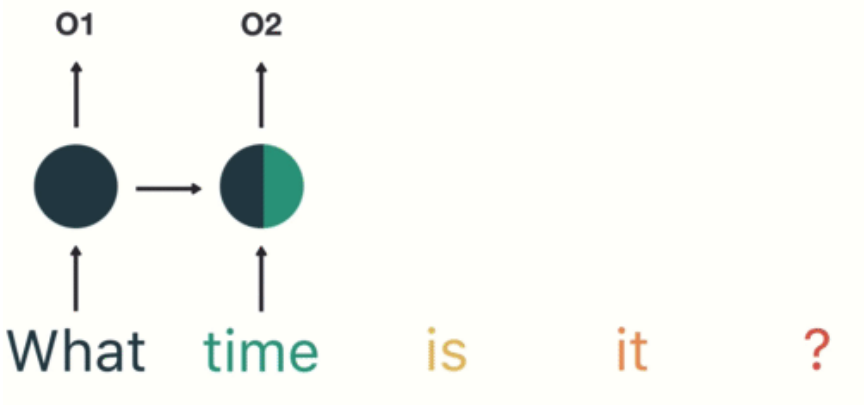
- Tokenization
- Morphological Analysis
- Part-of-Speech Tagging
- Named Entity Recognition (NER)
- Parsing
- Sentiment Analysis
- Word Embeddings
- Language Modeling
- Machine Translation
- Text Classification
- Information Retrieval
- Text Summarization
- Dialogue Systems (Chatbots)
- Contextual Models
- Preprocessing

Text Cleaning

Text cleaning is a fundamental preprocessing step in Natural Language Processing (NLP) that involves transforming raw text into a clean and standardized format. This process helps improve the performance of NLP models by ensuring the input data is relevant, consistent, and free of noise.



Lowercasing	Lowercasing: Converts all characters to lowercase to maintain consistency.
Removing	Removing Punctuation: Eliminates punctuation marks that may not be useful.
Removing	Removing Special Characters: Removes unwanted characters that can interfere with text processing.
Removing	Removing Numbers: Eliminates numerical values that may not add meaning.
Removing	Removing Extra Whitespace: Consolidates multiple spaces between words into a single space.
Expanding	Expanding Contractions: Converts contractions into their full forms (e.g., "can't" to "cannot").
Removing	Removing Stop Words: Filters out common words like "and," "the," which do not carry significant meaning.
Stemming	Stemming and Lemmatization: Reduces words to their root forms. Stemming is more aggressive, while lemmatization retains context.



Tokenization

This is the process of breaking down text into smaller units called tokens, which can be words, phrases, or symbols. Tokenization is a crucial first step in most NLP tasks.

Types of Tokenization in NLP

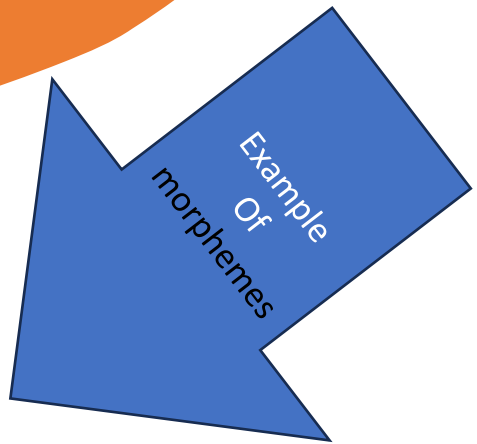
Word Tokenization: Common for languages with clear separation (e.g., English).

Character Tokenization: Useful for languages without clear separation or for very detailed analysis.

Subwords Tokenization: Smaller than words but bigger than characters (useful for complex languages or unknown words).

Morphological Parsing

Morphological parsing involves analyzing the structure of words to identify their morphemes (roots, prefixes, suffixes). It requires knowledge of morphological rules and patterns. Finite-state transducers (FSTs) are used as a tool for morphological parsing.



- **Morphological Analysis**: Parsing words into their morphemes.
- **Morphological Generation**: Generating word forms from morphemes.

Example 1: "Unhappiness"

Word: Unhappiness

Analysis:

- **Un-**: A prefix meaning "not"
- **Happy**: A root word referring to a state of joy
- **-ness**: A suffix that turns an adjective into a noun (indicating a state or quality)

Morphemes: Un- + Happy + -ness

Meaning: The state of not being happy.

Example 2: "Rewriting"

Word: Rewriting

Analysis:

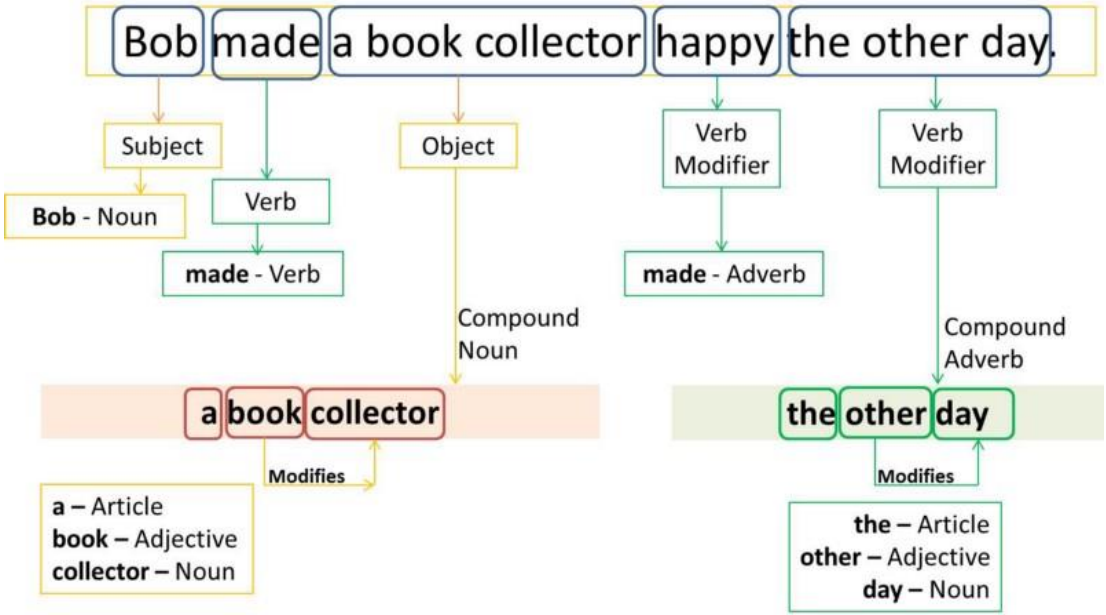
- **Re-**: A prefix meaning "again" or "back."
- **Write**: A root word meaning to inscribe or compose text.
- **-ing**: A suffix indicating the present participle or gerund form.

Morphemes: Re- + Write + -ing

Meaning: The act of writing again.

Part-of-Speech Tagging

This process involves identifying the grammatical categories of words (e.g., noun, verb, adjective) in a given text. It helps in understanding the roles that different words play in sentences.

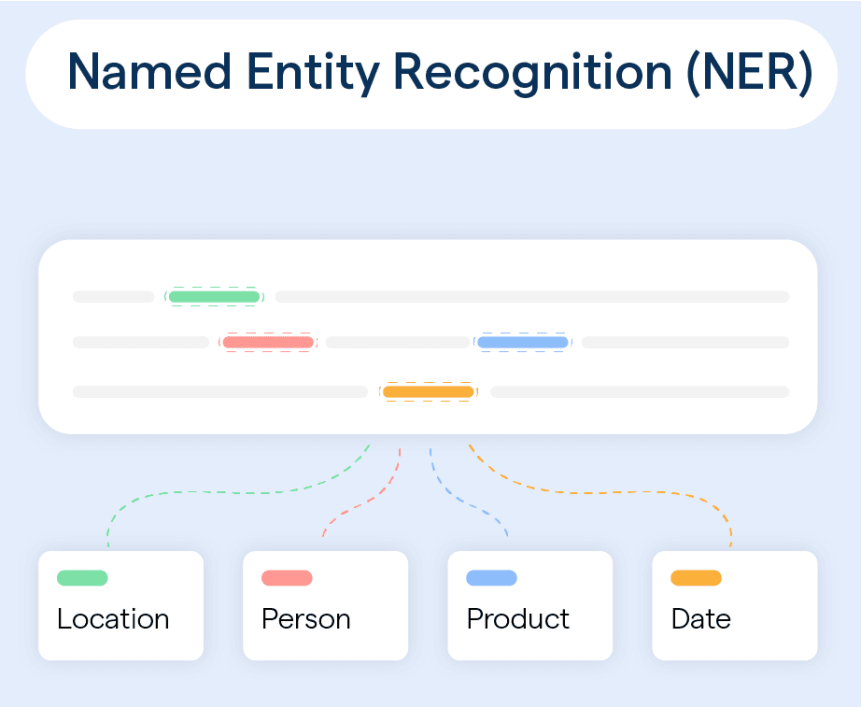


Named Entity Recognition (NER)

This technique identifies and classifies key entities in text into predefined categories (such as names of people, organizations, locations, dates, etc.).

Types of NER

- Supervised machine learning-based systems
- Rules-based systems
- Dictionary-based systems
- Unsupervised machine learning systems
- Bootstrapping systems
- Neural network systems
- Statistical systems
- Semantic role labeling systems
- Hybrid systems

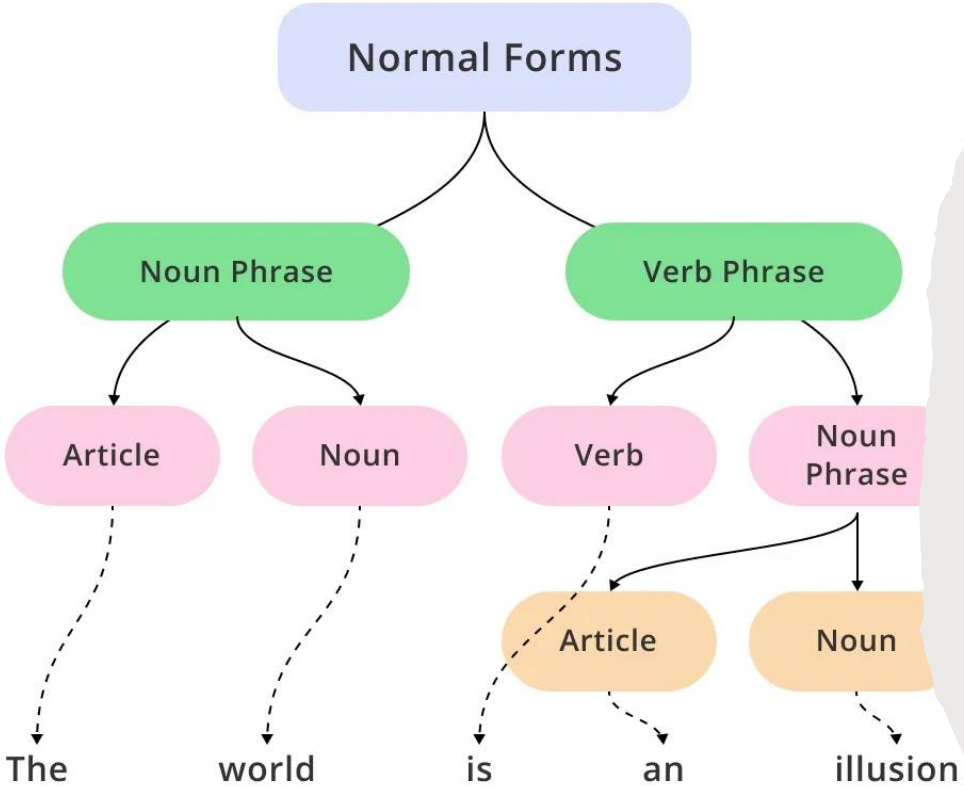


- **Supervised machine learning-based systems** use ML models trained on texts humans have pre-labeled with named entity categories. Supervised machine learning approaches use algorithms such as conditional random fields and maximum entropy, two complex statistical language models. This method is effective for parsing semantic meanings and other complexities, though it requires large volumes of training data.
- **Rules-based systems** use rules to extract information. Rules can include capitalizations or titles, such as "Dr." This method requires much human intervention to input, monitor, and tweak the rules. It might miss textual variations not included in its training annotations. It's thought that rules-based systems don't handle complexity as well as machine learning models.
- **Dictionary-based systems** use a dictionary with an extensive vocabulary and synonym collection to cross-check and identify named entities. This method might have trouble classifying named entities with variations in spellings.

Applications of NER

- **Chatbots**
- **Customer support**
- **Finance**
- **Healthcare**
- **Higher education**
- **HR**
- **News providers**
- **Recommendation engines**
- **Search engines**
- **Sentiment analysis**
- **QA**

Parsing



Parsing

Parsing is the process of examining the grammatical structure and relationships inside a given sentence or text NLP.

It involves analyzing the text to determine the roles of specific words, such as nouns, verbs, and adjectives, and their interrelationships. This analysis produces a structured representation of the text, allowing NLP computers to understand how words in a phrase connect to one another. Parsers expose the structure of a sentence by constructing parse trees or dependency trees that illustrate the hierarchical and syntactic relationships between words. This essential NLP stage is crucial for various language understanding tasks, which allow machines to extract meaning, provide coherent answers, and execute tasks such as machine translation, sentiment analysis, and information extraction.

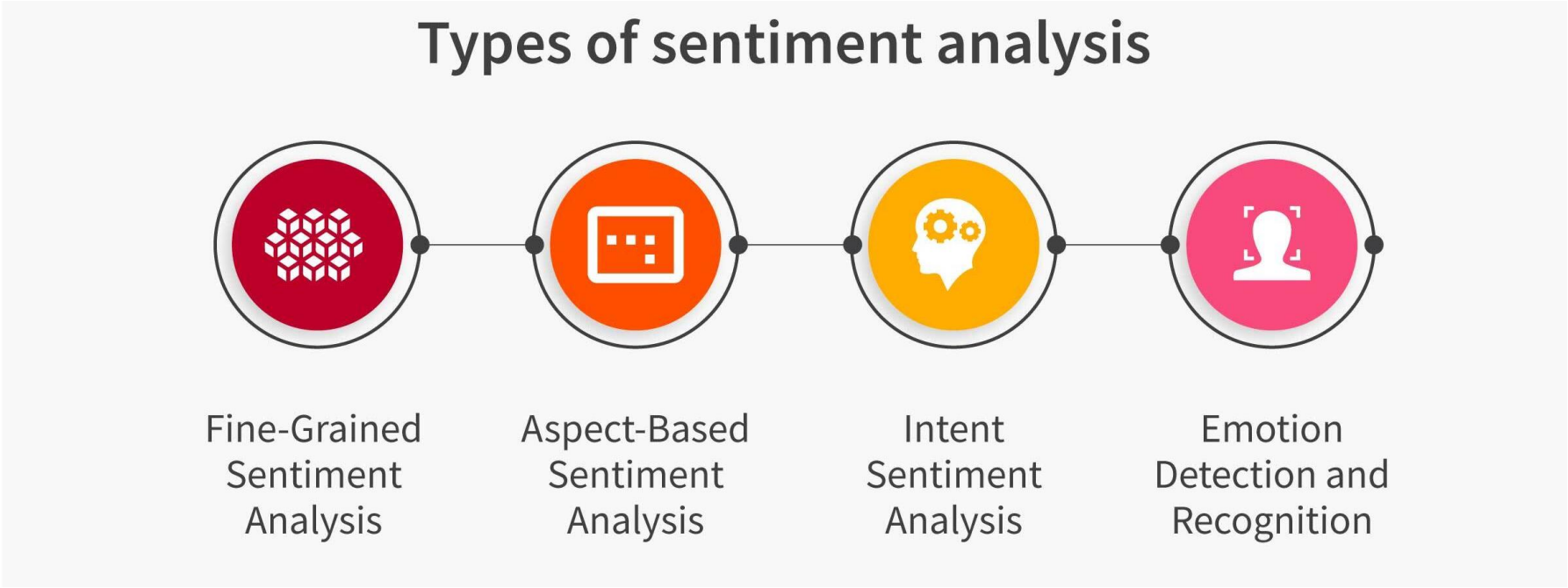
Types of Parsing in NLP

- **Syntactic Parsing**: Syntactic parsing deals with a sentence's grammatical structure. It involves looking at the sentence to determine parts of speech, sentence boundaries, and word relationships. The two most common approaches included are as follows:
  - **Constituency Parsing**: Constituency Parsing builds parse trees that break down a sentence into its constituents, such as noun phrases and verb phrases. It displays a sentence's hierarchical structure, demonstrating how words are arranged into bigger grammatical units.
  - **Dependency Parsing**: Dependency parsing depicts grammatical links between words by constructing a tree structure in which each word in the sentence is dependent on another. It is frequently used in tasks such as information extraction and machine translation because it focuses on word relationships such as subject-verb-object relations.
- **Semantic Parsing**: Semantic parsing goes beyond syntactic structure to extract a sentence's meaning or semantics. It attempts to understand the roles of words in the context of a certain task and how they interact with one another. Semantic parsing is utilized in a variety of NLP applications, such as question answering, knowledge base populating, and text understanding. It is essential for activities requiring the extraction of actionable information from text.

Sentiment Analysis

Sentiment Analysis is a subfield of Natural Language Processing (NLP) that involves determining the emotional tone behind a body of text. It aims to classify the sentiment expressed in the text as positive, negative, neutral, or even more nuanced emotions such as joy, anger, sadness, etc. This analysis is often applied to understand opinions, emotions, and attitudes expressed in various forms of written communication, such as social media posts, reviews, and articles.

Types of Sentiment Analysis



- Fine-grained (graded)

Fine-grained, or graded, sentiment analysis is a type of sentiment analysis that groups text into different emotions and the level of emotion being expressed. The emotion is then graded on a scale of zero to 100, similar to how consumer websites deploy star ratings to measure customer satisfaction.

- Aspect-based (ABSA)

Aspect-based sentiment analysis (ABSA) narrows the scope of what's being examined in a body of text to a singular aspect of a product, service, or customer experience a business wishes to analyze. For example, a budget travel app might use ABSA to understand how intuitive a new user interface is or to gauge the effectiveness of a customer service chatbot. ABSA can help organizations better understand how their products succeed or fall short of customer expectations.

- Emotional detection

Emotional detection sentiment analysis seeks to understand the psychological state of the individual behind a body of text, including their frame of mind when writing it and their intentions. It is more complex than fine-grained or ABSA and is typically used to better understand a person's motivation or emotional state. Rather than using polarities, like positive, negative, or neutral, emotional detection can identify specific emotions in a body of text, such as frustration, indifference, restlessness, and shock.

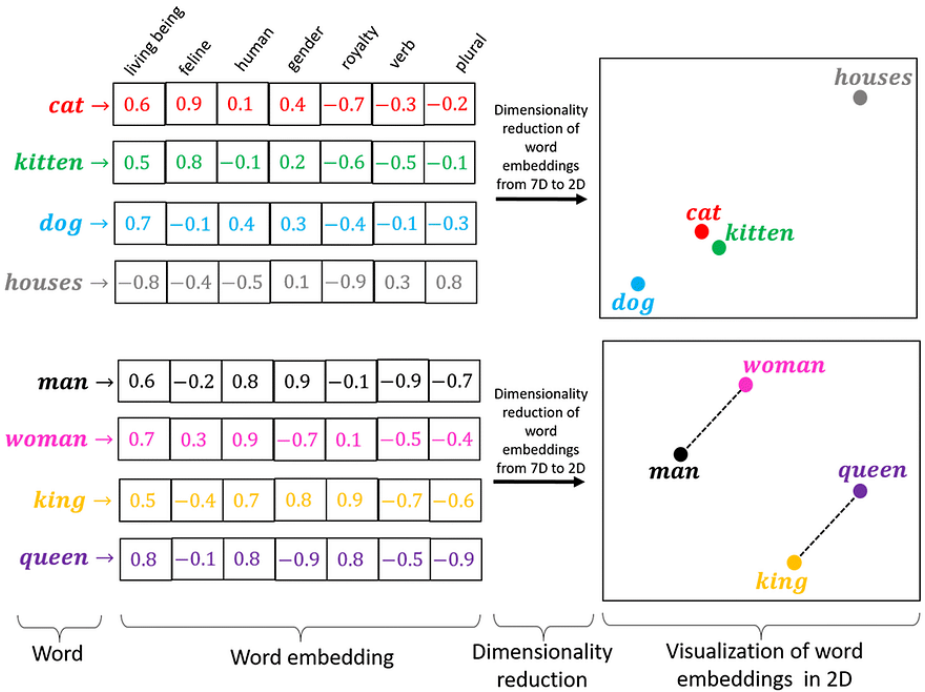
- Intent Sentiment Analysis

Intent Sentiment Analysis is a specialized form of sentiment analysis that assesses the emotional tone of a piece of text and identifies the underlying intention or purpose behind that sentiment. This dual analysis can be particularly useful in applications like customer service, chatbots, and voice assistants, where understanding both the user's emotion and intent can lead to more effective responses.

An Example of Sentiment Analysis

- **Review:** "The food was fantastic, but the service was terrible."
- **Analysis Steps:**
- **Tokenization:** Split the text into words or phrases.
- **Sentiment Classification:**
- "The food was fantastic" → Positive sentiment
- “The service was terrible" → Negative sentiment
- **Overall Sentiment:** The overall sentiment could be classified as mixed, indicating that while one aspect (food) is positively received, another (service) is negatively perceived.

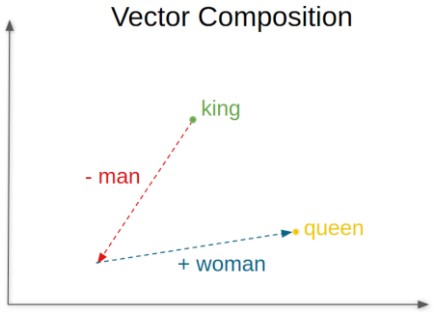
Word Embedding



Word embeddings are numerical representations of words in a continuous vector space where semantically similar words are mapped to nearby points.

Unlike traditional one-hot encoding—which represents words as sparse vectors—word embeddings capture the meanings of words based on their context.

Key Concepts

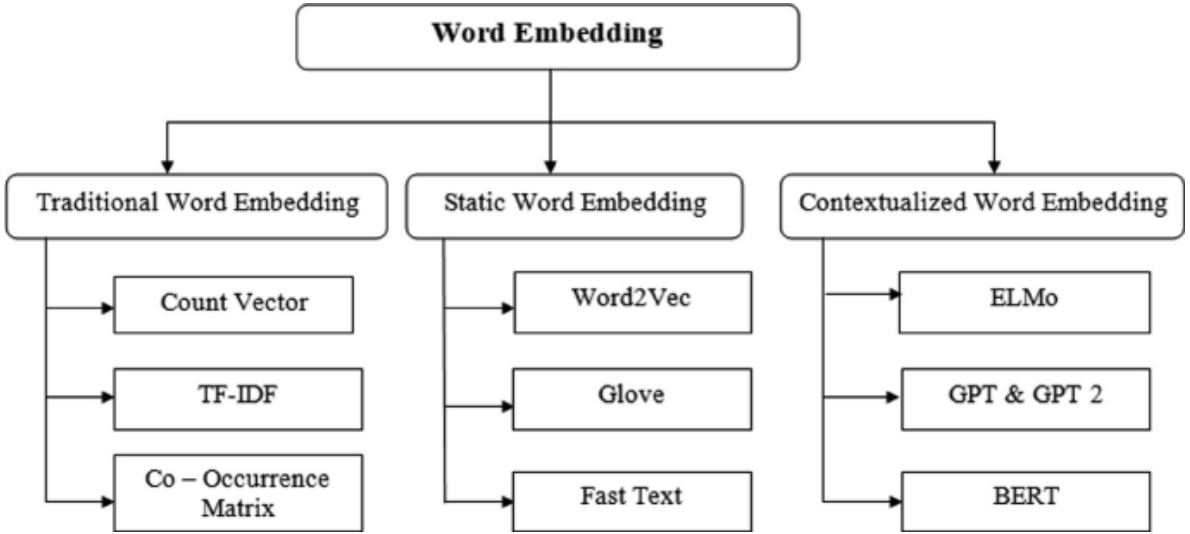


Dimensionality : Word embeddings typically exist in a lower-dimensional space (e.g., 50 to 300 dimensions), significantly reducing the size compared to one-hot encoding.

Distributed Representation : Each word is represented by a dense vector of real numbers. The position of the vector in the space reflects the word's meaning, allowing for various mathematical operations to be performed.

Semantic Similarity : Words with similar meanings have similar vector representations. For example, "king" and "queen" are closer to each other in the vector space compared to "king" and "car."

Techniques for Creating Word Embeddings



Traditional Word Embedding

Count Vector: Represents words based on their frequency in a document, creating a sparse vector for each word.

TF-IDF (Term Frequency-Inverse Document Frequency): Adjusts the count of words by considering their importance across multiple documents, reducing the weight of common words.

Co-occurrence Matrix: Captures how often words appear together in a given context, forming a matrix that can be used for further analysis.

Static Word Embedding

- **Word2Vec**: A model that generates word vectors based on the context of words in a corpus using the CBOW or Skip-Gram approach.
- **GloVe**: This method utilizes global word co-occurrence statistics to create word vectors, focusing on the relationships between words in a corpus.
- **FastText**: Extends Word2Vec by considering subword information, allowing it to generate embeddings for out-of-vocabulary words.

Contextualized Word Embedding

ELMo (Embeddings from Language Models)

Produces embeddings that vary based on the context of the word in a sentence, using a bidirectional LSTM.

GPT & GPT-2 (Generative Pre-trained Transformer):

Focuses on generating text and understanding context, providing embeddings that adapt based on the surrounding text.

BERT (Bidirectional Encoder Representations from Transformers):

Generates contextual embeddings by analyzing the entire sentence, capturing nuanced meanings based on context.

Language Modeling

Language modeling is a fundamental task in NLP that involves predicting the probability of a sequence of words.

A language model is designed to understand a language's structure and semantics, enabling various applications, including text generation, machine translation, speech recognition, and more.

Types of Language Models

- **Statistical Language Models**
- N-Gram
- Unigram
- Bidirectional
- Exponential
- Continuous Space
- **Neural Language Models**

Statistical Language Models

Statistical models include the development of probabilistic models that can predict the next word in the sequence, given the words that precede it.

Neural Language Models

These language models are based on neural networks and are often considered an advanced approach to execute NLP tasks. Neural language models overcome the shortcomings of classical models such as n-gram and are used for complex tasks such as speech recognition or machine translation. Language is significantly complex and keeps on evolving. Therefore, the more complex the language model is, the better it will be at performing NLP tasks. Compared to the n-gram model, an exponential or continuous space model is a better option for NLP tasks because they are designed to handle ambiguity and language variation. Meanwhile, language models should be able to manage dependencies. For example, a model should be able to understand words derived from different languages.

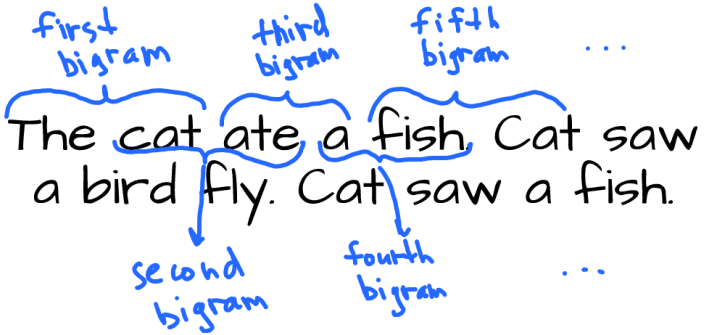
N-Gram: This is one of the simplest approaches to language modeling. Here, a probability distribution for a sequence of ‘n’ is created, where ‘n’ can be any number and defines the size of the gram (or sequence of words being assigned a probability). If n=4, a gram may look like: “can you help me”. Basically, ‘n’ is the amount of context that the model is trained to consider. There are different types of N-Gram models such as unigrams, bigrams, trigrams, etc.

Example with an n-gram Model

Let's consider a simple example using a bigram model (where n=2), which predicts a word based on the previous one.

"I love natural language processing."

"Natural language processing is fun.“



Building the Bigram Model

From the corpus, extract bigrams (pairs of consecutive words):

("I", "love")

("love", "natural")

("natural", "language")

("language", "processing")

("processing", "is")

("is", "fun")

- **Unigram**: The unigram is the simplest type of language model. It doesn't look at any conditioning context in its calculations. It evaluates each word or term independently. Unigram models commonly handle language processing tasks such as information retrieval. The unigram is the foundation of a more specific model variant called the query likelihood model, which uses information retrieval to examine a pool of documents and match the most relevant one to a specific query.

- **Bidirectional**: Unlike n-gram models, which analyze text in one direction (backwards), bidirectional models analyze text in both directions, backwards and forwards. These models can predict any word in a sentence or body of text by using every other word in the text. Examining text bidirectionally increases result accuracy. This type is often utilized in machine learning and speech generation applications. For example, Google uses a bidirectional model to process search queries.

- **Exponential**: This type of statistical model evaluates text by using an equation, which is a combination of n-grams and feature functions. Here the features and parameters of the desired results are already specified. The model is based on the principle of entropy, which states that probability distribution with the most entropy is the best choice. Exponential models have fewer statistical assumptions, meaning the chances of accurate results are higher.

- **Continuous Space**: In this type of statistical model, words are arranged as a non-linear combination of weights in a neural network. The process of assigning weight to a word is known as word embedding. This type of model proves helpful in scenarios where the data set of words continues to become large and include unique words.

Machine Translation (MT)

MT is a subfield of NLP that automatically converts text or speech from one language to another using algorithms and computational methods.

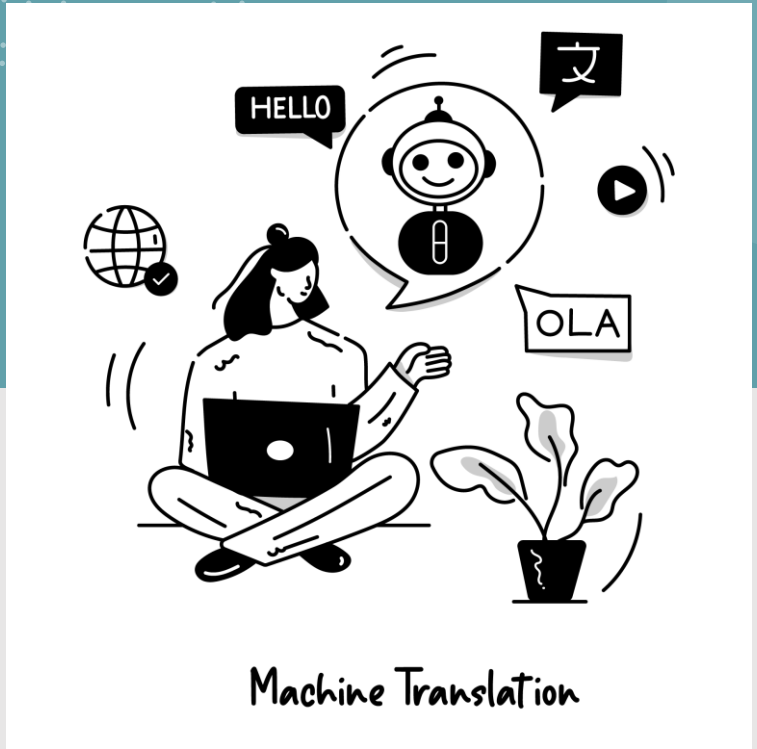
Categories of Machine Translation :

Rule-Based Machine Translation (RBMT)

Statistical Machine Translation (SMT)

Neural Machine Translation (NMT)

Hybrid Systems



Rule-Based Machine Translation (RBMT):

Uses linguistic rules and grammar to translate.

Requires extensive knowledge of both source and target languages.

Example: Systran applied rules developed by linguists.

Statistical Machine Translation (SMT):

Utilizes statistical methods based on large corpora to build translations.

Estimates probabilities of translations based on existing data.

Example: Google Translate before incorporating neural networks.

Neural Machine Translation (NMT):

Uses deep learning models, particularly recurrent neural networks (RNNs) and transformer models, to produce translations.

Allows for better contextual understanding and fluency in the translation.

Example: Google Translate transitioned to NMT for improved accuracy.

Hybrid Systems:

Combine multiple approaches (e.g., rule-based with statistical methods) to enhance translation quality.

Key Components

Source Language: The language to be translated from.

Target Language: The language to be translated to.

Corpora: Large sets of bilingual text used to train and evaluate translation models.

Preprocessing: Steps such as tokenization and normalization applied to the input text to prepare it for translation.

Post-editing: Human intervention to correct and refine machine-generated translations.

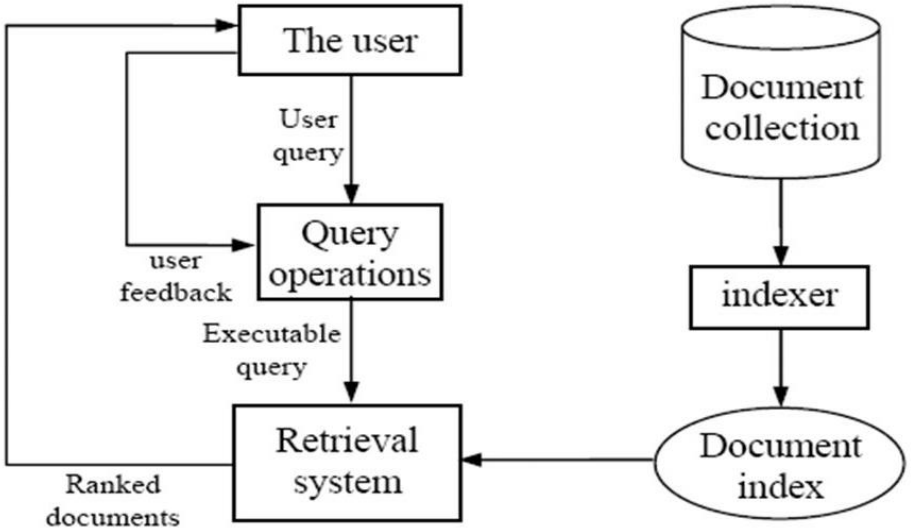
Challenges

- **Ambiguity**: Words or phrases with multiple meanings can lead to incorrect translations.
- **Context**: Translating idiomatic expressions or nuanced phrases requires understanding context, which can be difficult.
- **Domain-Specific Language**: Specific jargon or technical terms might not accurately translate without specialized training.
- **Cultural Nuances**: Language carries cultural context that may not translate well.

Information retrieval

Information retrieval (IR) may be defined as a software program that deals with the organization, storage, retrieval, and evaluation of information from document repositories, mainly textual information. The system assists users in finding the information they require, but it does not explicitly return the answers to the questions. It informs the existence and location of documents containing the required information. The documents that satisfy the user's requirement are called relevant documents. A perfect IR system will retrieve only relevant documents. **Google Search** is the most famous example of information retrieval.

IR architecture



Applications of Information Retrieval

Search Engines: Google, Bing, and other search engines rely heavily on IR techniques to return relevant web pages based on user queries.

Digital Libraries: Systems for academic papers or books that allow efficient searching and retrieval of information.

E-commerce: Product search engines on e-commerce platforms where users can find products based on specific queries.

Recommendation Systems: Suggesting articles or products based on user interactions and searches.

Key Concepts

- **Documents:** Any information unit that can be retrieved, including text documents, web pages, images, audio files, etc. Documents, the building blocks of information retrieval, keep the audience engaged in the process.
- **Query:** A user's request for information, typically expressed in natural language. Queries can be in various forms, from simple keywords to complex phrases.
- **Corpus:** A collection of documents (corpus) from which relevant information is retrieved. This can include everything from databases, websites, or document collections.
- **Indexing:** The process of organizing data to allow for efficient retrieval. This often involves creating an inverted index, which maps terms to their locations in the documents.
- **Ranking:** The process of determining the order of results based on their relevance to the query. Ranking algorithms use factors such as term frequency, document length, etc.

Types of Information Retrieval (IR) Model

- **Classical IR Model**

It is the simplest and easiest to implement IR model. This model is based on mathematical knowledge that is easily recognized and understood. Boolean, Vector, and Probabilistic are the three classical IR models.

- **Non-Classical IR Model**

It is completely opposite to the classical IR model. Such IR models are based on principles other than similarity, probability, and Boolean operations. Information logic models, situation theory models, and interaction models are examples of non-classical IR models.

- **Alternative IR Model**

It enhances the classical IR model by using specific techniques from other fields. Cluster, fuzzy, and latent semantic indexing (LSI) models are examples of alternative IR models.

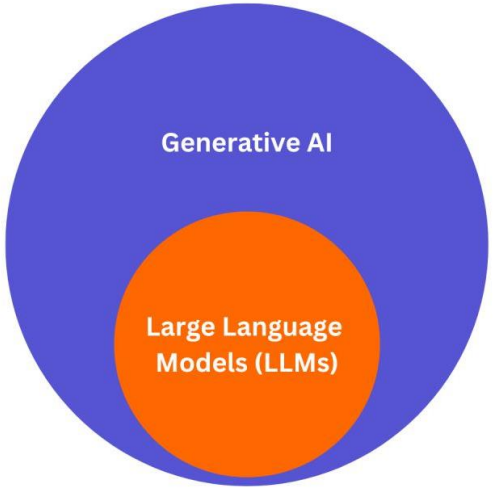
More Info

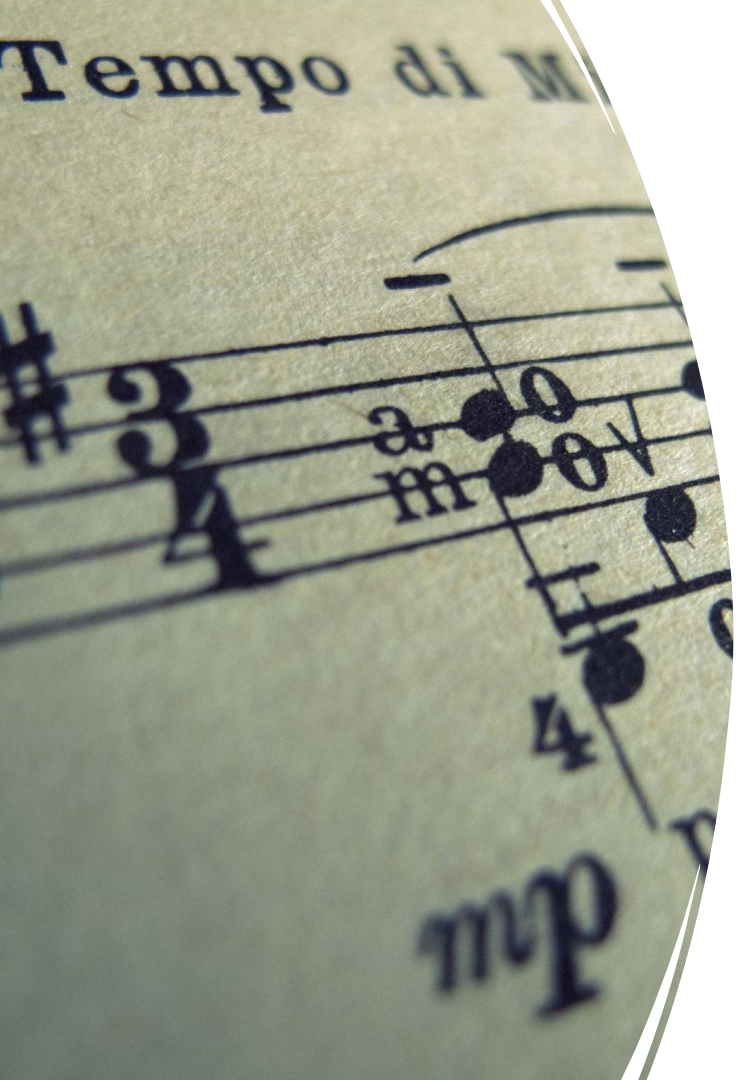

LLMs

What is Generative AI?

Generative AI is a subset of traditional machine learning.

The machine learning models that underpin generative AI have learned these abilities by finding statistical patterns in massive datasets of content originally generated by humans.





Applications of Generative AI

- **Text Generation**: Creating articles, stories, or poetry (e.g., GPT-3).
- **Image Generation**: Producing images from textual descriptions (e.g., DALL-E, Midjourney).
- **Music Composition**: Composing original music tracks (e.g., OpenAI's MuseNet).
- **Video Generation**: Creating video content based on scripts or prompts.
- **Game Development**: Generating game levels, characters, or narratives.
- **Drug Discovery**: Designing new molecules for pharmaceuticals.
- **Personalization**: Tailoring content for users based on preferences.

Generative AI Use Cases

Healthcare

Medical Image Synthesis for Diagnostics

Drug Discovery and Development

Personalized Medicine and Treatment Plans

Finance and Trading

Algorithmic Trading Strategies

Predictive Market Analysis

High-Frequency Trading Algorithms

Content Creation

Art and Music Generation

Automated Content Creation for Marketing

Virtual Assistants and Chatbots

Natural Language Processing (NLP)

Text Generation for Content Creation

Sentiment Analysis

Language Translation and Sentiment Analysis

SoluLab

Key Concepts

Neural Networks: The backbone of most generative AI models.

Deep Learning: A subset of machine learning that uses neural networks with many layers.

Training Data: The dataset used to train generative models.

Latent Space: A representation of compressed data used in generating new content.

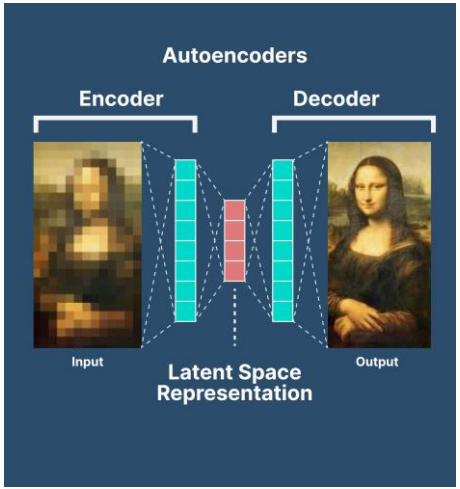
Variational Autoencoders (VAEs): A type of generative model that learns to encode data into a latent space.

Generative Adversarial Networks (GANs) : A framework where two neural networks compete to improve the quality of generated content.

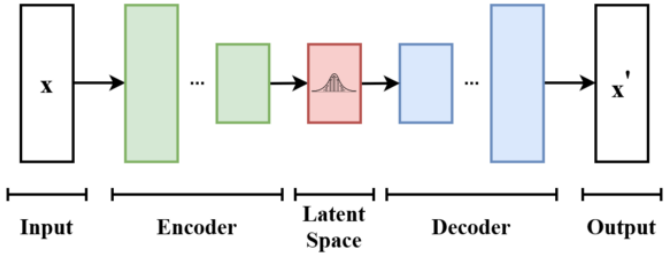
Prompt Engineering: Crafting inputs to guide the output of generative models effectively.

Variational Autoencoders (VAEs)

Variational Autoencoders (VAEs) are a type of generative model that combines principles from both traditional autoencoders and Bayesian inference. They are particularly useful for generating new data points that are similar to a given dataset.



- **Encoder**: Compresses input data into a lower-dimensional latent representation, focusing on extracting meaningful features.
- **Decoder**: Takes the latent representation and reconstructs the original input or generates new data, aiming for high fidelity in the output.



Applications of VAEs

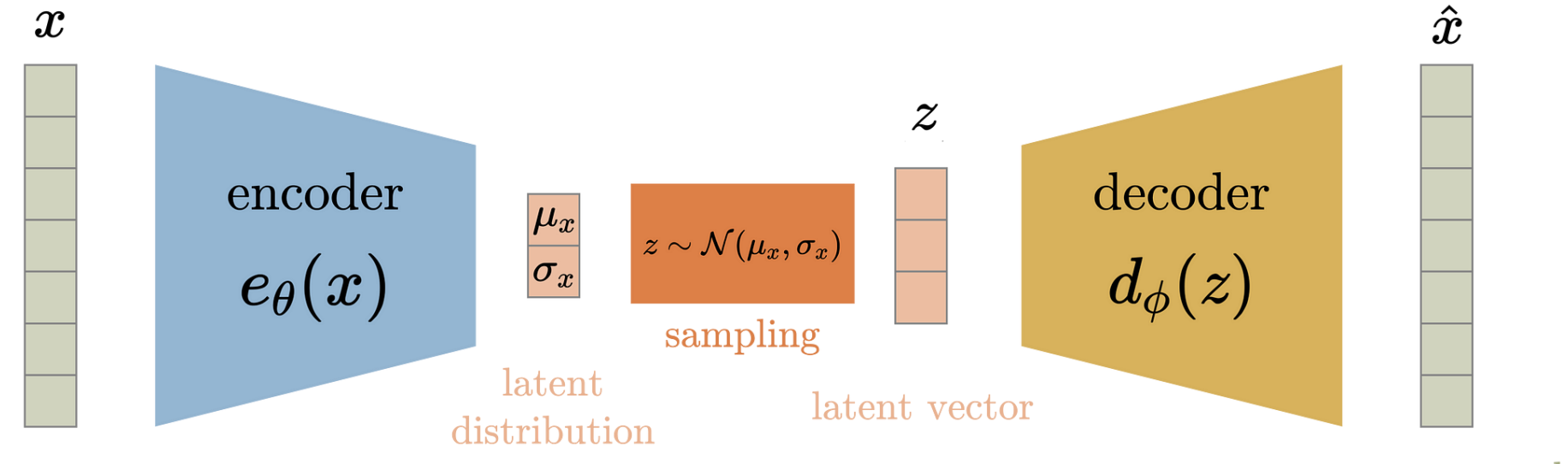
Image Generation: VAEs can generate new images that resemble the training dataset, making them useful in creative applications.

Data Imputation: They can fill in missing data points by sampling from the learned distribution.

Anomaly Detection: By learning the distribution of normal data, VAEs can identify outliers or anomalies.

Semi-Supervised Learning: VAEs can leverage both labeled and unlabeled data to improve learning performance.

Representation Learning: VAEs can learn meaningful representations of data that can be used for downstream tasks.



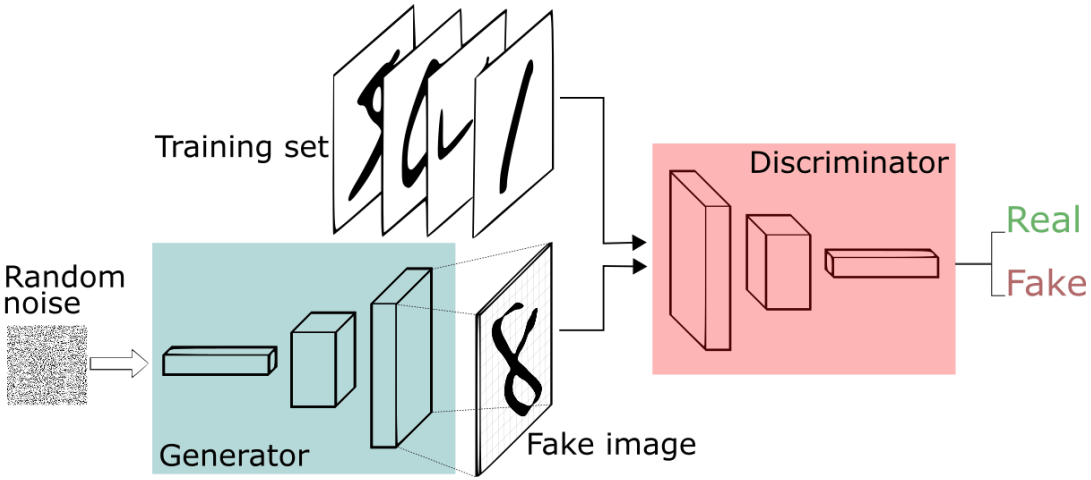
$$\text{reconstruction loss} = \|x - \hat{x}\|_2 = \|x - d_\phi(z)\|_2 = \|x - d_\phi(\mu_x + \sigma_x \epsilon)\|_2$$

$$\mu_x, \sigma_x = e_\theta(x), \quad \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

$$similarity\ loss = KL\ Divergence = D_{KL}(\mathcal{N}(\mu_x, \sigma_x) \parallel \mathcal{N}(\mathbf{0}, \mathbf{I}))$$

$$loss = reconstruction\ loss + similarity\ loss$$

Generative Adversarial Networks (GANs)



Generative Adversarial Networks (GANs) are a class of machine learning frameworks designed to generate new data samples that resemble a given training dataset. Introduced by Ian Goodfellow and his colleagues in 2014, GANs consist of two neural networks, the generator and the discriminator, that compete against each other in a game-theoretic setting.

Applications of GANs

- **Image Generation**: GANs can create high-quality images from random noise, making them popular for generating realistic images in various domains (e.g., faces, landscapes).
- **Image-to-Image Translation**: GANs can transform images from one domain to another (e.g., converting sketches to photographs, changing seasons in images).
- **Super Resolution**: GANs can enhance the resolution of images, generating high-resolution images from low-resolution inputs.
- **Text-to-Image Synthesis**: GANs can generate images based on textual descriptions, allowing for creative applications in art and design.
- **Video Generation**: GANs can be used to generate realistic video sequences, which can be applied in entertainment and simulation.
- **Data Augmentation**: GANs can create synthetic data to augment training datasets, especially in scenarios with limited data availability.
- **Anomaly Detection**: By learning the distribution of normal data, GANs can help identify outliers or anomalies in datasets.

Key Concepts of GANs

1.Generator:

The generator is responsible for creating new data samples from random noise (usually sampled from a Gaussian or uniform distribution).

Its goal is to produce data that is indistinguishable from real data in the training set.

2.Discriminator:

The discriminator's role is to differentiate between real data (from the training set) and fake data (produced by the generator).

It outputs a probability score indicating whether the input data is real or generated.

Key Concepts of GANs

3.Adversarial Training:

GANs operate on a minimax game framework where the generator tries to maximize the probability of the discriminator making a mistake, while the discriminator tries to minimize its error.

The generator and discriminator are trained simultaneously: as the generator improves, the discriminator must also improve to keep up, leading to a dynamic and competitive training process.

4.Loss Functions:

The loss function for the generator encourages it to produce data that the discriminator classifies as real.

The loss function for the discriminator penalizes it for misclassifying real and fake data.

The overall objective is to reach a point where the discriminator can no longer distinguish between real and generated data.

Prompt engineering

Prompt engineering is designing and refining input prompts to effectively guide the behavior of generative AI models, particularly those based on natural language processing (NLP) like GPT-3 and similar models. The goal is to elicit the desired output from the model by carefully crafting the input it receives.

DEEP (LEARNING) FOCUS

Advanced Prompt Engineering

Forward Generation Template

I gave a friend an instruction and five inputs. The friend read the instruction and wrote an output for every one of the inputs. Here are the input-output pairs:

Input: $[Q_1]$ **Output**: $[A_1]$
Input: $[Q_2]$ **Output**: $[A_2]$
...

The instruction was **<COMPLETE>**

Reverse Generation Template

I instructed my friend to **<INSERT>**.

The friend read the instruction and wrote an output for every one of the inputs. Here are the input-output pairs:

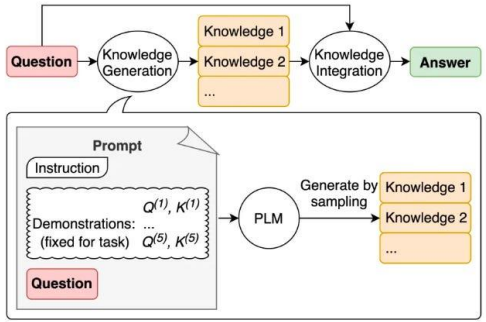
Input: $[Q_1]$ **Output**: $[A_1]$
Input: $[Q_2]$ **Output**: $[A_2]$
...

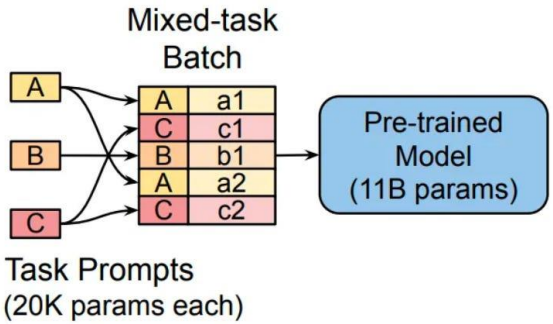
Template for TruthfulQA

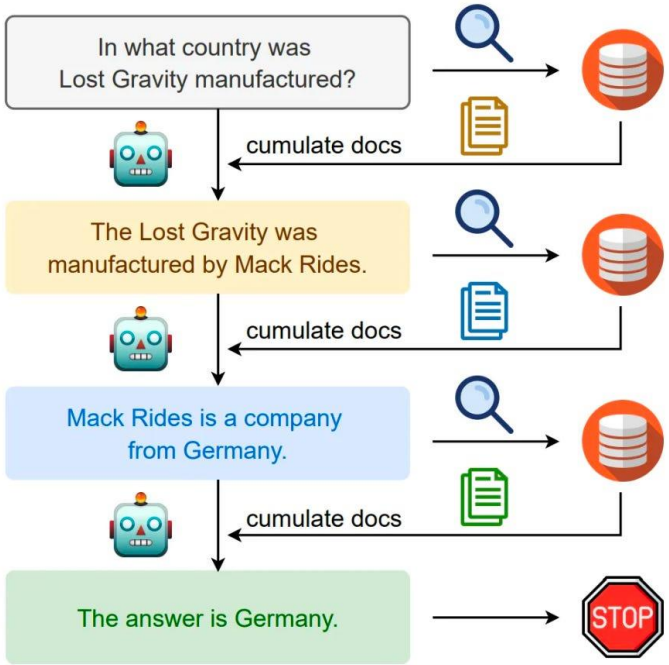
Professor Smith was given the following instructions: **<INSERT>**

Here are the Professor's responses:

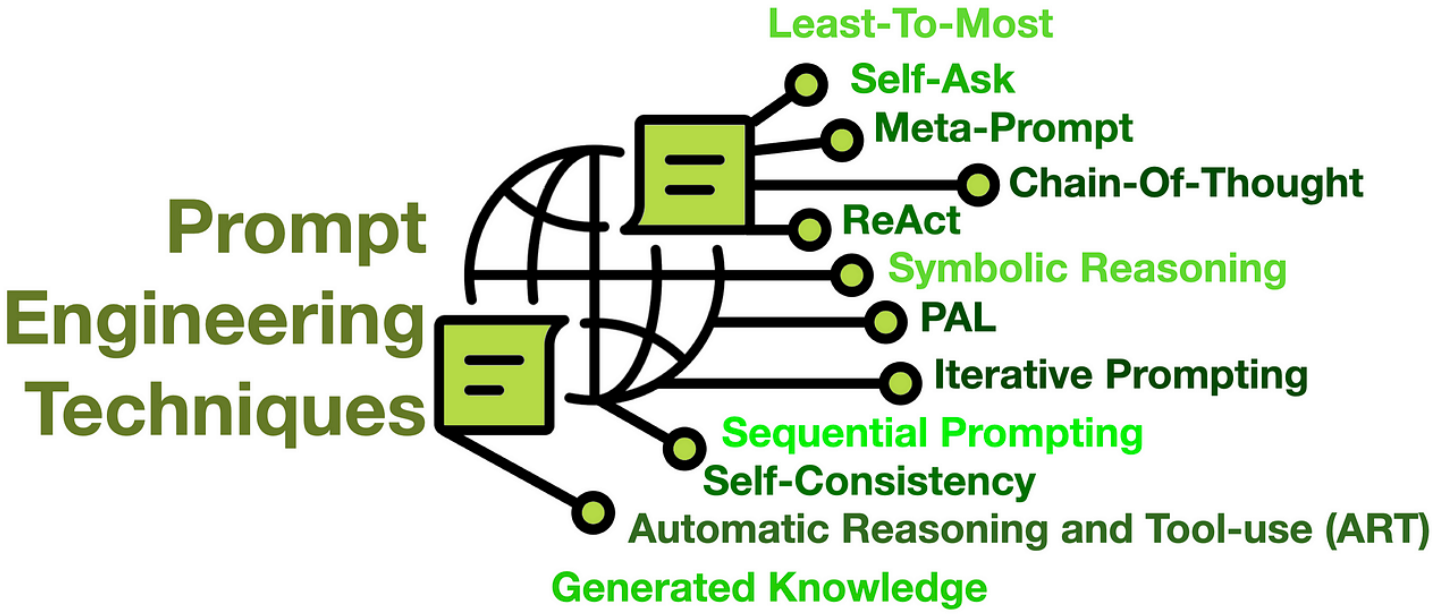
Input: $[Q_1]$ **Output**: $[A_1]$
Input: $[Q_2]$ **Output**: $[A_2]$
...







12 Prompt Engineering Techniques

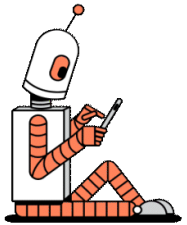


Large Language Models (LLMs)

A large language model is an advanced language model trained using deep learning techniques on massive amounts of text data. These models can generate human-like text and perform various natural language processing tasks. In contrast, the definition of a language model refers to the concept of assigning probabilities to sequences of words, based on the analysis of text corpora. A language model can be of varying complexity, from simple n-gram models to more sophisticated neural network models.

However, the term “large language model” usually refers to models that use deep learning techniques and have many parameters, ranging from millions to billions. These AI models can capture complex patterns in language and produce text that is often indistinguishable from that written by humans.

WIRED



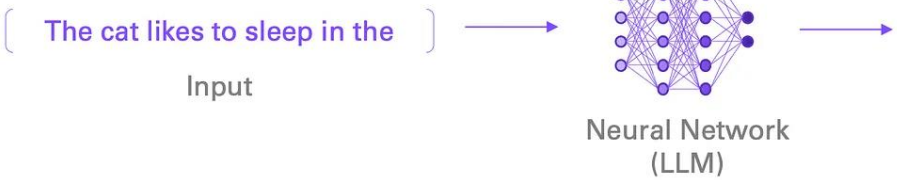
Language modeling

Imagine the following task: Predict the next word in a sequence

The cat likes to sleep in the ____ → What **word** comes next?

Now we have (say) ~50,000 classes (i.e. words)

Can we frame this as a ML problem? Yes, it's a classification task.



Word	Probability
ability	0.002
bag	0.071
box	**0.085**
...	...
zebra	0.001

Output

Generative AI vs. Large Language Models (LLMs)

Generative AI

- Creates many outputs
- Myriad of generative AI tools
- Built on LLMs but also other types of machine learning models

vs.

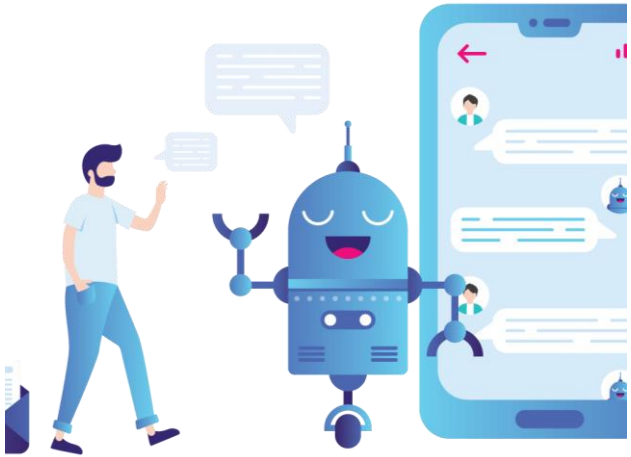
Large Language Models

- Create text-only outputs
- Myriad of parameters to understand and produce text
- Foundation for text-based generative AI tools like ChatGPT

appian

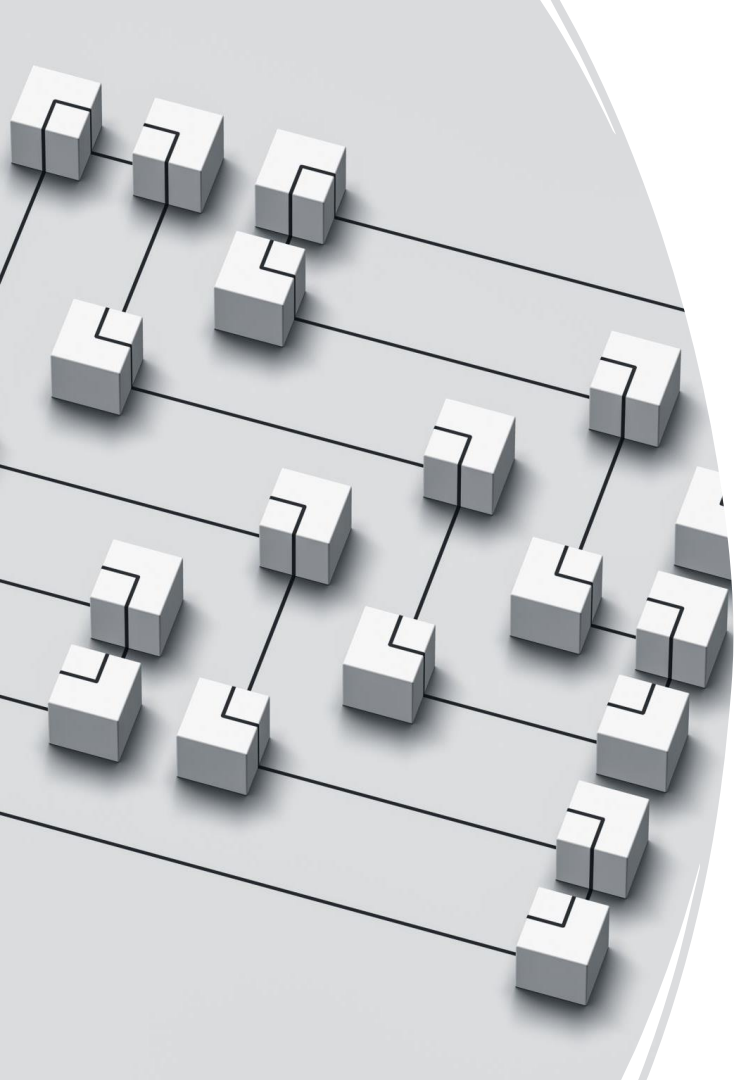
Applications

- **Chatbots and Virtual Assistants**
- **Content Generation**
- **Text Summarization**
- **Translation Services**
- **Sentiment Analysis**
- **Personalization**
- **Education and Tutoring**
- **Research Assistance**
- **Programming Help**
- **Health Care**
- **Gaming**
- **Social Media Management**



Examples of popular LLMs

- PaLM: Google's Pathways Language Model (PaLM) is a transformer language model capable of common-sense and arithmetic reasoning, joke explanation, code generation, and translation.
- BERT: Google also developed the Bidirectional Encoder Representations from Transformers (BERT) language model. This transformer-based model can understand natural language and answer questions.
- XLNet: A permutation language model, XLNet generated output predictions in a random order, which distinguishes it from BERT. It assesses the pattern of tokens encoded and then predicts tokens in random order, instead of a sequential order.
- GPT: Generative pre-trained transformers are perhaps the best-known large language models. Developed by OpenAI, GPT is a popular foundational model whose numbered iterations are improvements on their predecessors (GPT-3, GPT-4, etc.). It can be fine-tuned to perform specific tasks downstream. Examples are EinsteinGPT, developed by Salesforce for CRM, and Bloomberg's BloombergGPT for finance.



How LLMs work

LLMs operate by leveraging deep learning techniques and vast amounts of textual data. These models are typically based on a transformer architecture, like the generative pre-trained transformer, which excels at handling sequential data like text input. LLMs consist of multiple layers of neural networks, each with parameters that can be fine-tuned during training. These layers are enhanced further by a multitude of layers known as the attention mechanism, which dials in on specific parts of data sets.

During the training process, these models learn to predict the next word in a sentence based on the context provided by the preceding words. The model does this by attributing a probability score to the recurrence of words that have been tokenized—broken down into smaller sequences of characters. These tokens are then transformed into embeddings, numeric representations of this context.

To ensure accuracy, this process involves training the LLM on a massive corpora of text (in the billions of pages), allowing it to learn grammar, semantics and conceptual relationships through zero-shot and self-supervised learning. Once trained on this training data, LLMs can generate text by autonomously predicting the next word based on the input they receive, and drawing on the patterns and knowledge they've acquired. The result is coherent and contextually relevant language generation that can be harnessed for a wide range of NLU and content generation tasks. Model performance can also be increased through prompt engineering, prompt-tuning, fine-tuning and other tactics like reinforcement learning with human feedback (RLHF) to remove the biases, hateful speech and factually incorrect answers known as “hallucinations” that are often unwanted byproducts of training on so much unstructured data. This is one of the most important aspects of ensuring enterprise-grade LLMs are ready for use and do not expose organizations to unwanted liability, or cause damage to their reputation.

LLM Training

Training LLMs involves several key components:

Architectures: This refers to the underlying structure of the model, such as transformer architecture, which is commonly used in LLMs. The architecture determines how the model processes and generates text.

Objectives: These are the goals set during training, such as predicting the next word in a sentence or understanding context. The objectives guide how the model learns from the data.

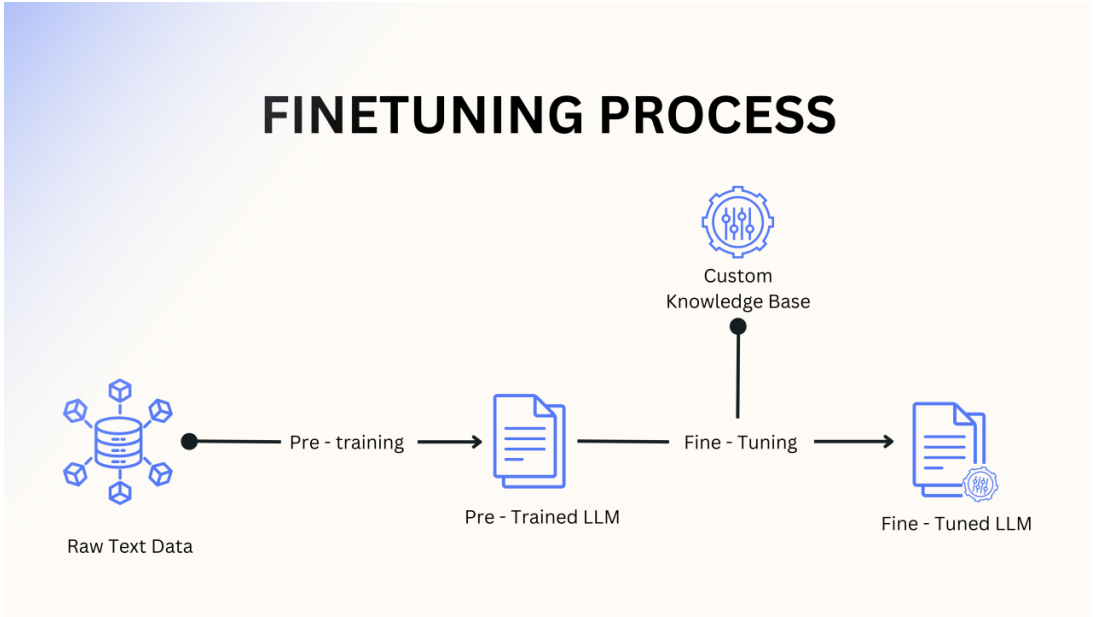
LLM Finetuning

Finetuning is the process of adapting a pre-trained model to specific tasks or datasets:

Instruction Finetuning: This involves training the model on specific instructions to improve its ability to follow user commands or queries.

RLHF (Reinforcement Learning from Human Feedback): This technique uses feedback from human evaluators to refine the model's outputs, making them more aligned with human preferences.

Bootstrapping: This refers to the process of using the model's own outputs to further train and improve itself, often in a self-supervised manner.



The image illustrates fine-tuning a large language model to learn a specific style of answering and writing. Here's a breakdown of the components:

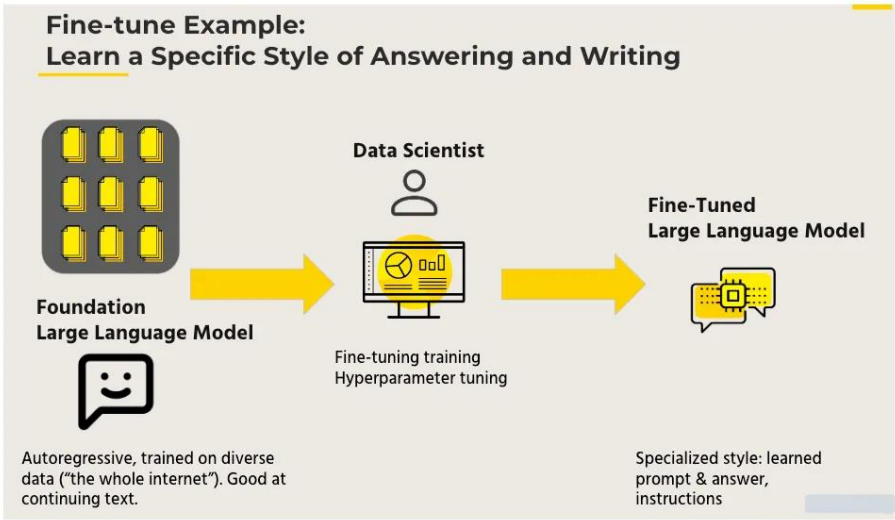
1.Foundation Large Language Model: This is the initial model trained on a vast and diverse dataset (referred to as "the whole internet"). It is autoregressive, which predicts the next word in a sequence based on the previous words, making it effective at generating coherent text.

2.Data Scientist: This individual is responsible for fine-tuning the model's settings for optimal performance. They utilize techniques such as hyperparameter tuning.

3.Fine-Tuning Process: This involves training the foundation model further on a specific dataset or style, allowing it to adapt and specialize in particular types of prompts and responses.

4.Fine-Tuned Large Language Model: The result of the fine-tuning process is a model that can generate text in a specialized style tailored to specific instructions or prompts.

Overall, the image conveys the workflow from a general-purpose language model to a specialized one through the intervention of a data scientist.



LLM Risks

Bias: LLMs can inadvertently learn and perpetuate biases in the training data, leading to unfair or harmful outputs.

Misinformation: They can generate plausible-sounding but incorrect or misleading information.

Privacy Concerns: LLMs can inadvertently reveal sensitive information from their training data if not handled properly.


10000
7500
5000
2500
0
Numbers of Parameters (in Millions)
April 2018
July 2018
October 2018
January 2019
April 2019
July 2019
ELMo
94
OpenAI
GPT
110
Google AI
BERT-Large
340
Transformer
ELMo
465
OpenAI
GPT-2
1500
MT-DNN
330
UNIVERSITY of WASHINGTON
Grover-
Mega
1500
XLM
665
XLNET
340
Carnegie
Mellon
University
RoBERTa
355
DistilBERT
66
MegatronLM
8300
NVIDIA

LLM

R Packages	Python Libraries
tidychatmodels	OpenAI
Gptstudio	LangChain
rollama	Hugging Face

Let's have some examples with R and Python