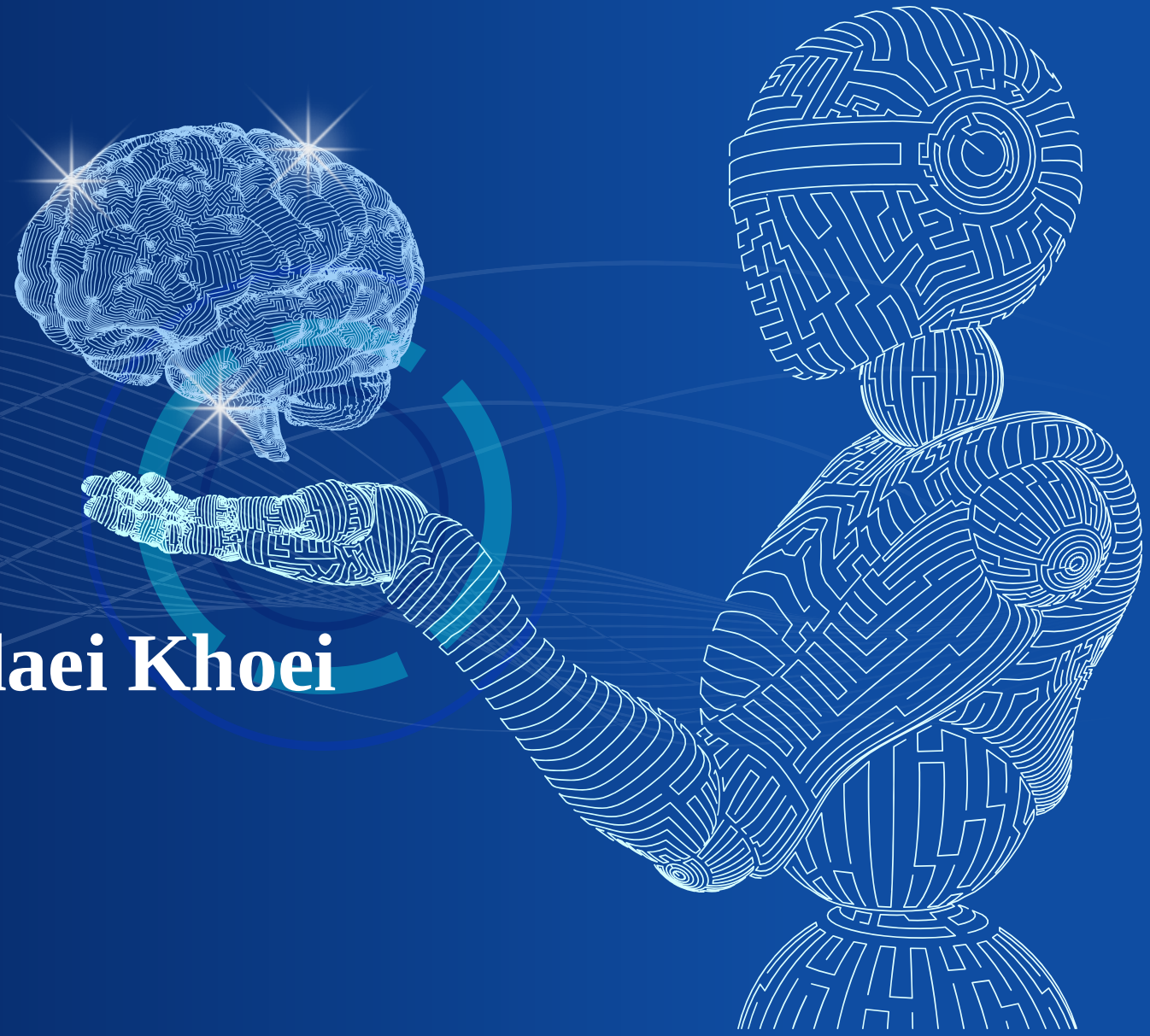
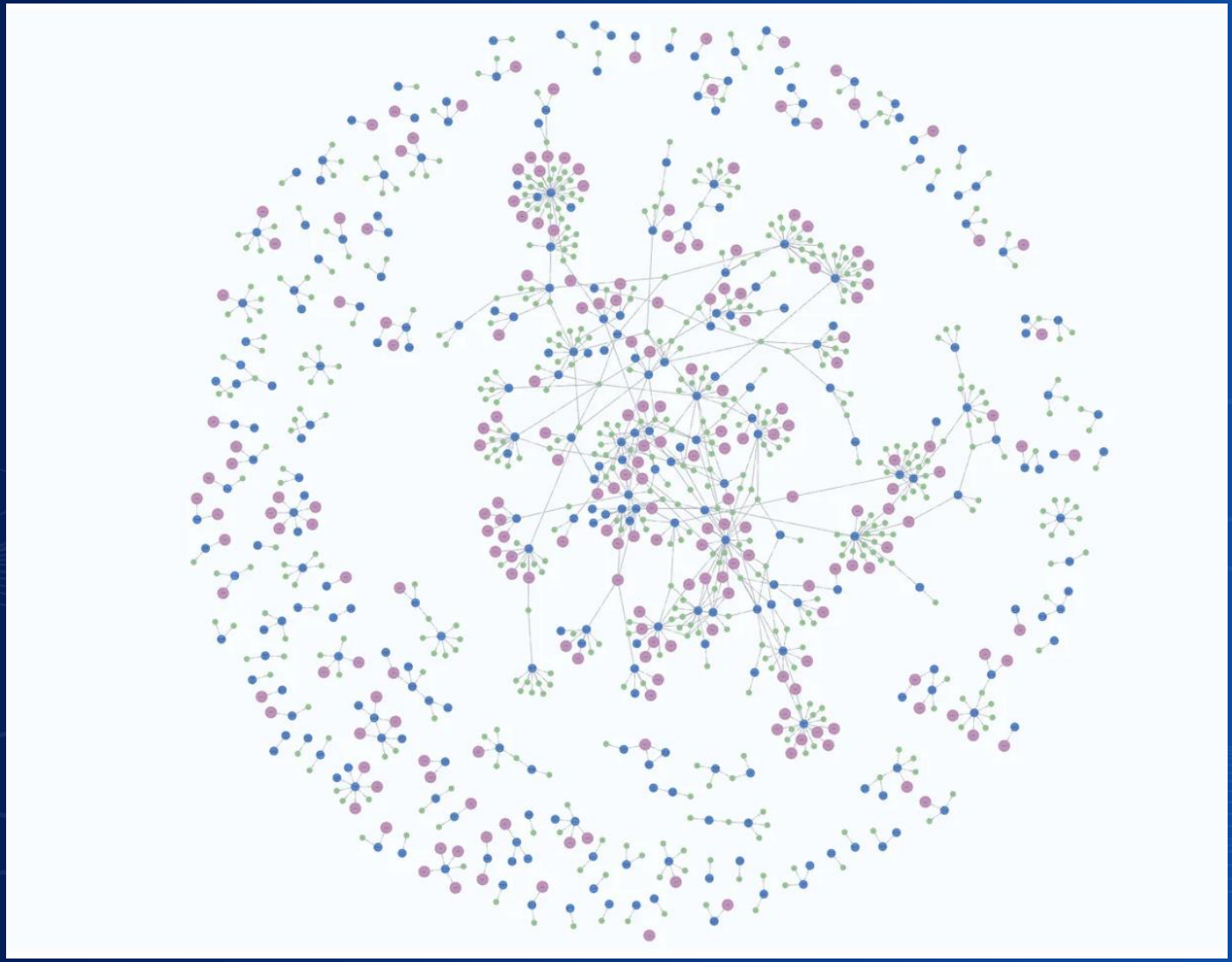
CS 7150: Deep Learning



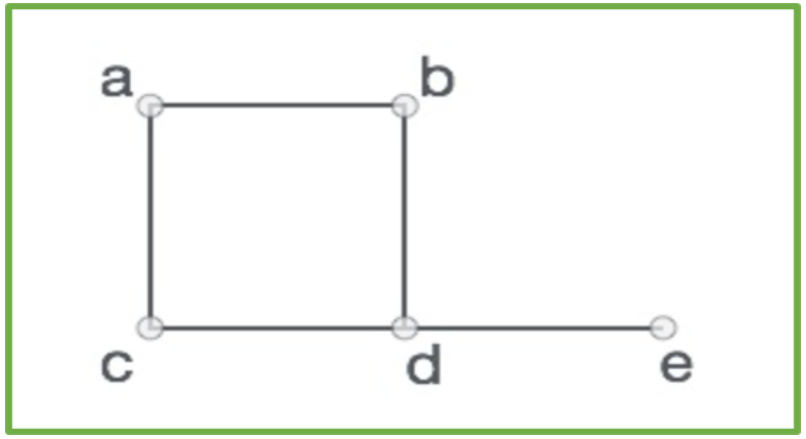
Presented by: Dr. Tala Talaei Khoei

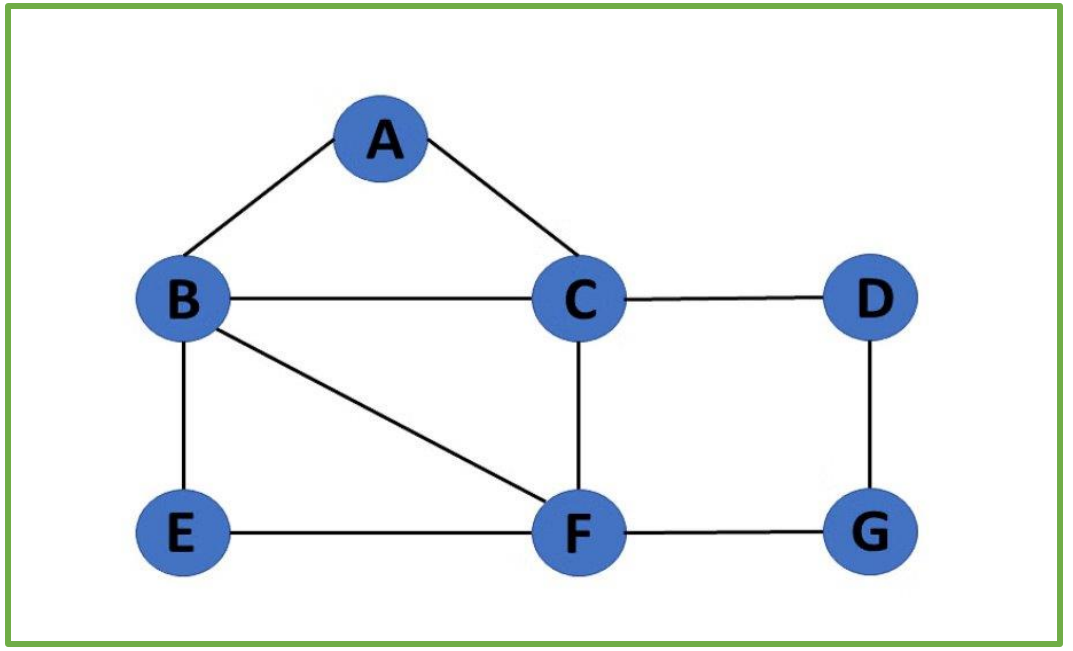


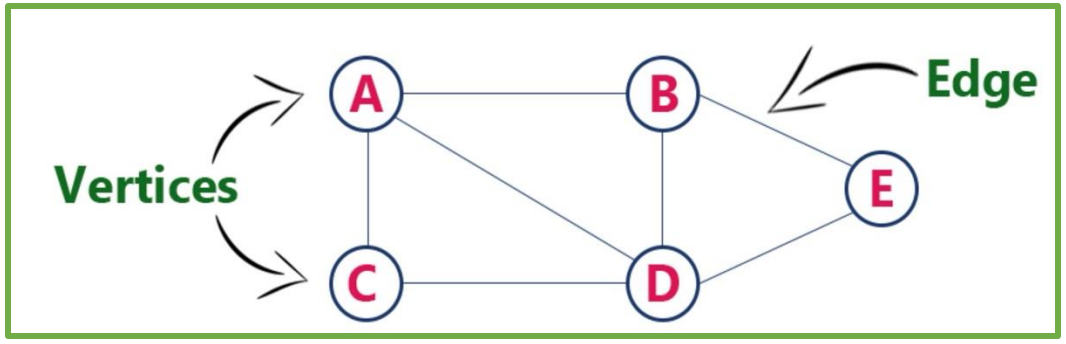
Graph Neural Networks

What is Graph?

- Graph data structures are data structures that consist of a collection of nodes or vertices connected by edges. Graphs are used to represent relationships or connections between objects and are widely used in various fields, including computer science, mathematics, social networks, and transportation systems.



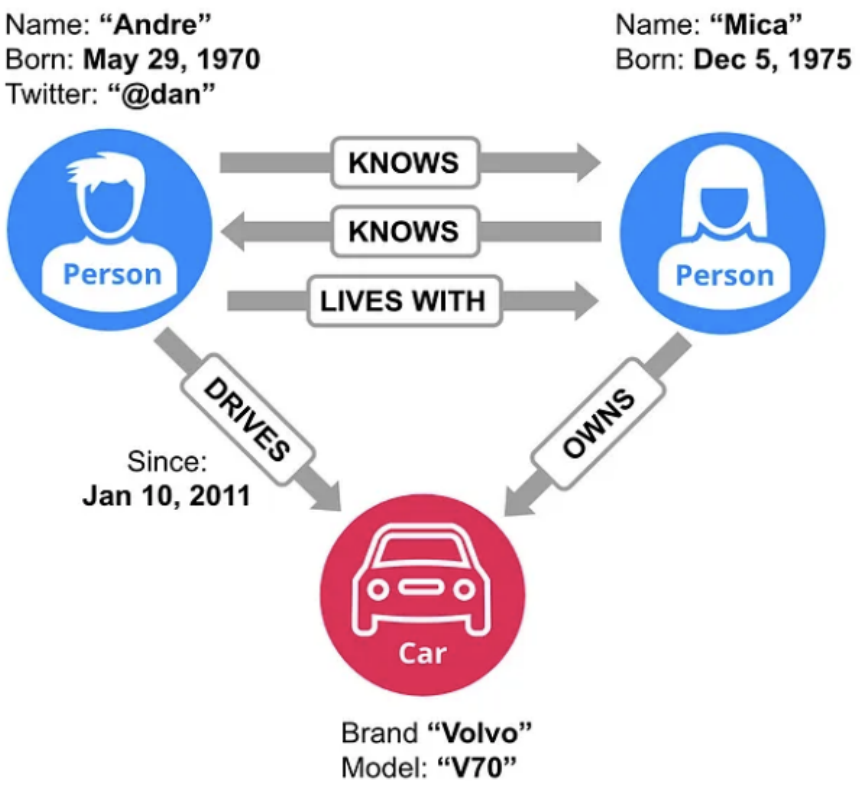




Nodes represent entities in the graph

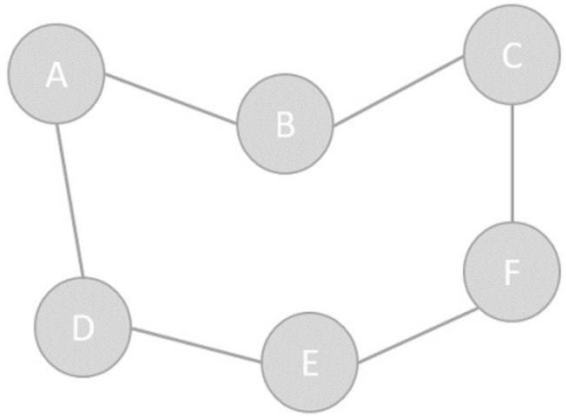
Relationships represent associations or interactions between nodes

Properties represent attributes of nodes or relationships



Graph Data Structure

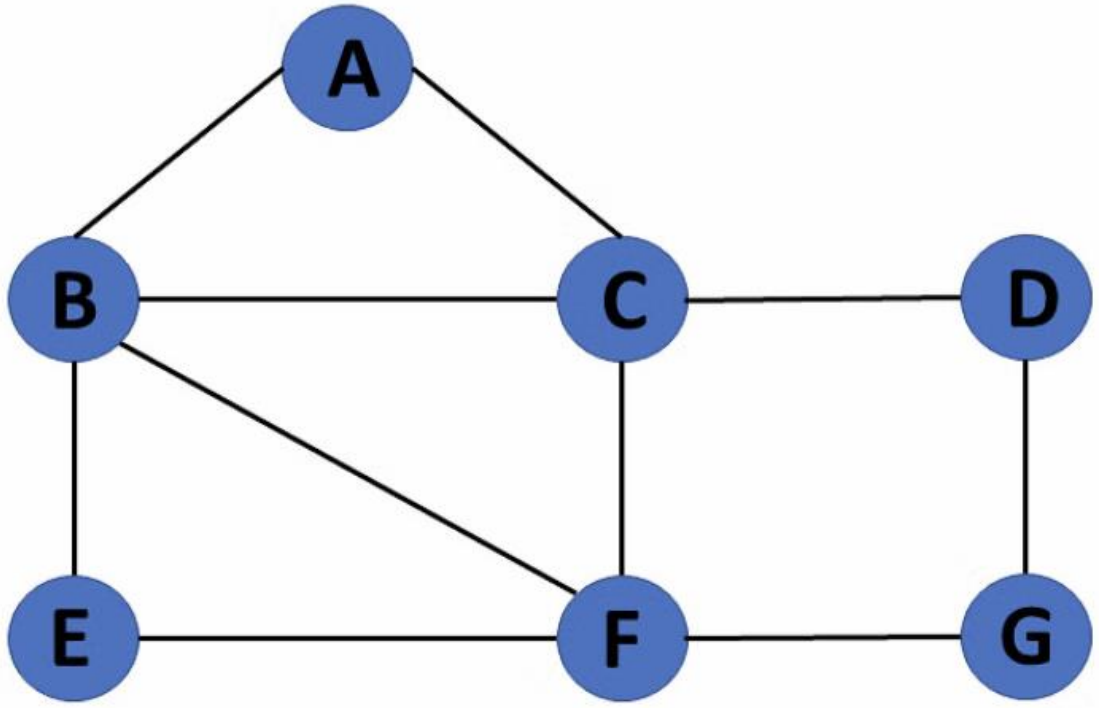
- Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges:
- **Vertex** − Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to F are vertices. We can represent them using an array. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **Path** − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



Other Types of Graphs

- **Finite Graph**

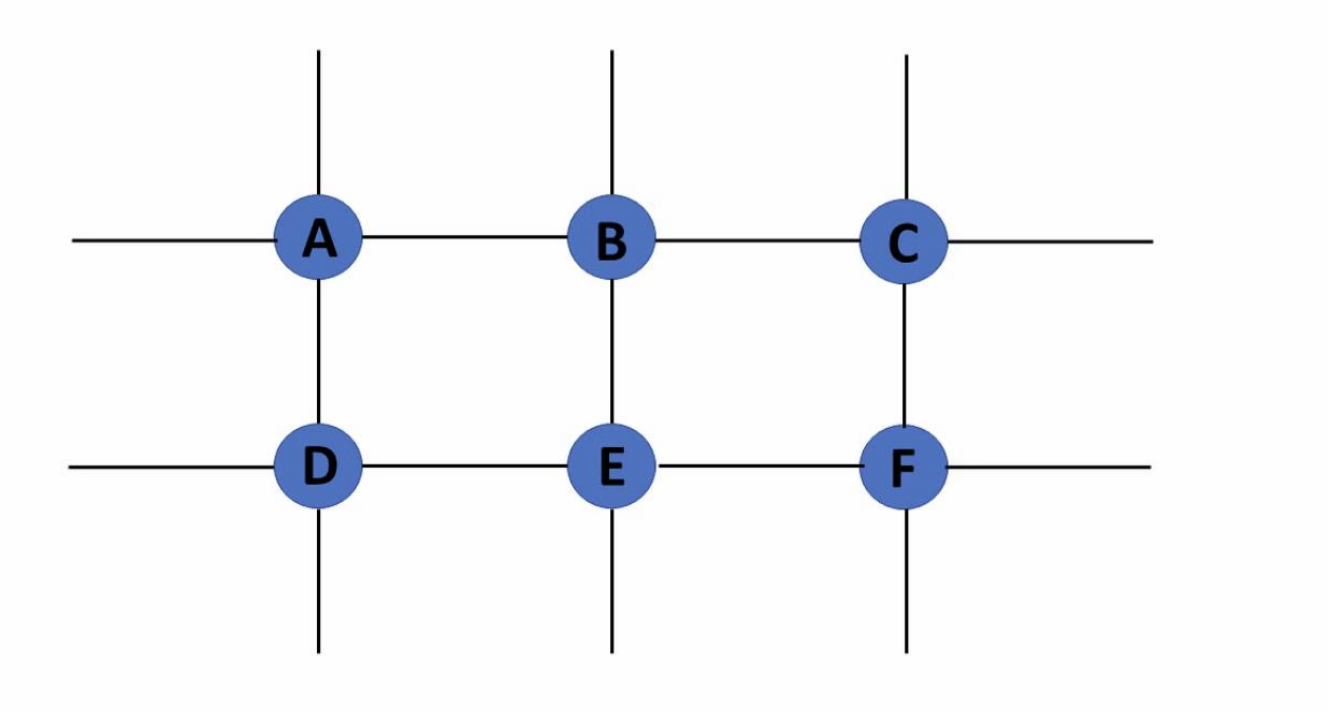
The graph G=(V, E) is called a finite graph if the number of vertices and edges in the graph is limited in number



Other Types of Graphs

- **Infinite Graph**

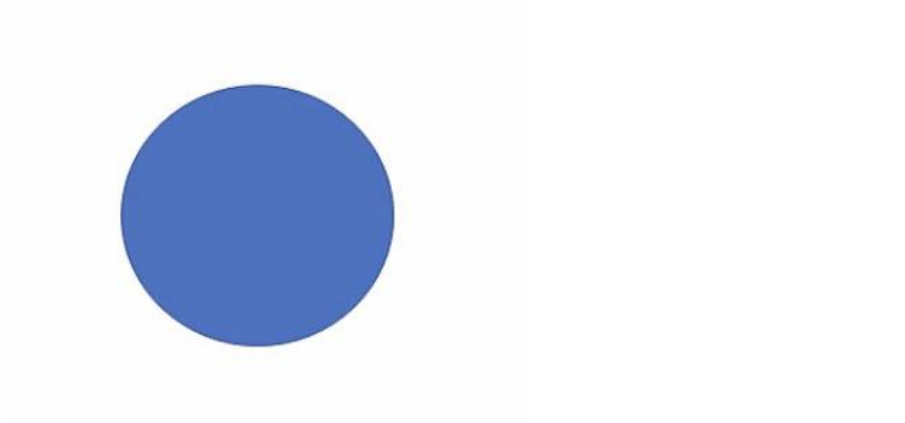
The graph G=(V, E) is called a finite graph if the number of vertices and edges in the graph is interminable.



Other Types of Graphs

- **Trivial Graph**

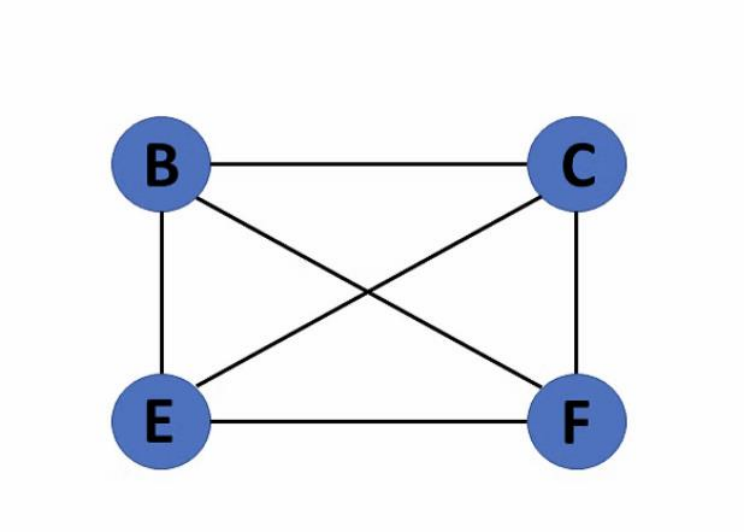
A graph G= (V, E) is trivial if it contains only a single vertex and no edges.



Other Types of Graphs

- **Simple Graph**

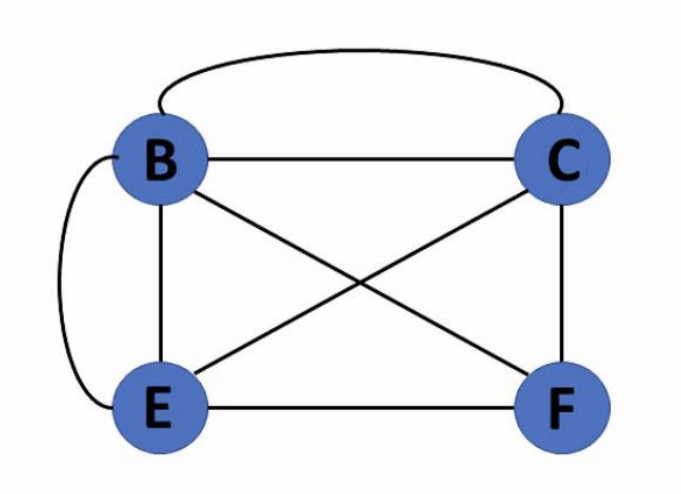
If each pair of nodes or vertices in a graph G=(V, E) has only one edge, it is a simple graph. As a result, there is just one edge linking two vertices, depicting one-to-one interactions between two elements.



Other Types of Graphs

- **Multi Graph**

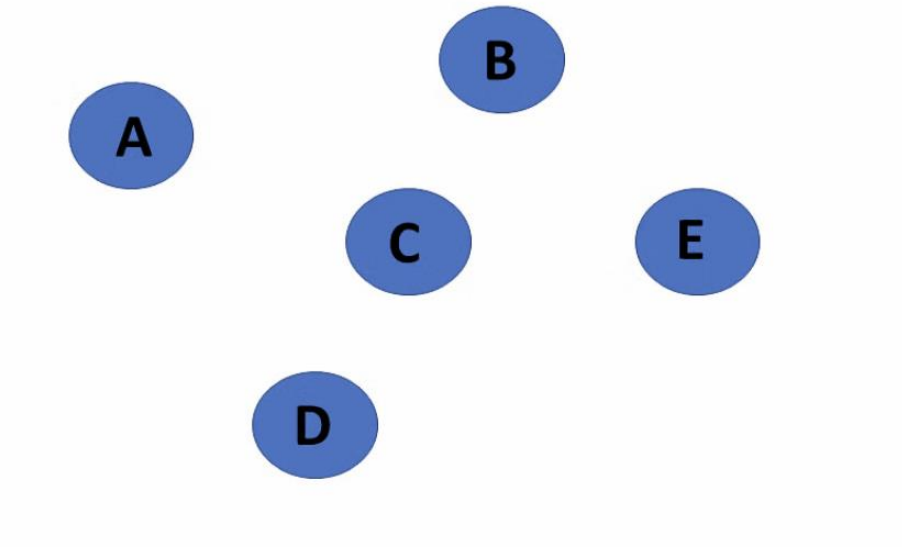
If there are numerous edges between a pair of vertices in a graph G= (V, E), the graph is referred to as a multigraph. There are no self-loops in a Multigraph.



Other Types of Graphs

- **Null Graph**

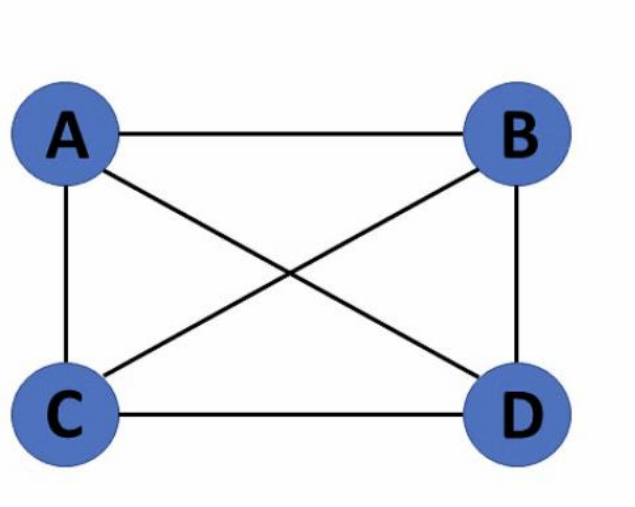
It's a reworked version of a trivial graph. If several vertices but no edges connect them, a graph G= (V, E) is a null graph



Other Types of Graphs

- **Complete Graph**

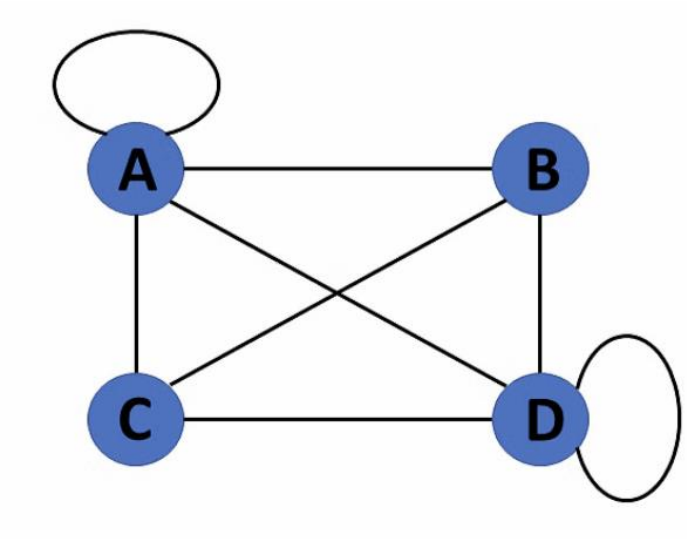
If a graph G= (V, E) is also a simple graph, it is complete. Using the edges, with n number of vertices must be connected. It's also known as a full graph because each vertex's degree must be n-1.



Other Types of Graphs

- **Pseudo Graph**

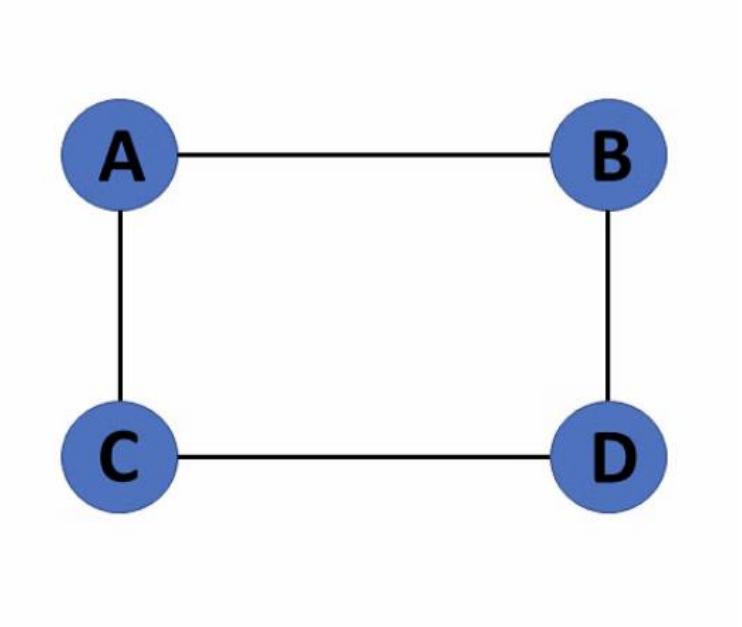
If a graph G= (V, E) contains a self-loop besides other edges, it is a pseudograph.



Other Types of Graphs

- **Regular Graph**

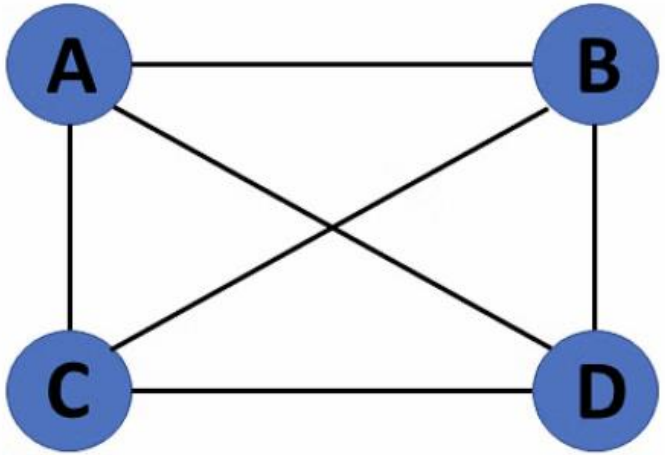
If a graph G= (V, E) is a simple graph with the same degree at each vertex, it is a regular graph. As a result, every whole graph is a regular graph.



Other Types of Graphs

- **Connected Graph**

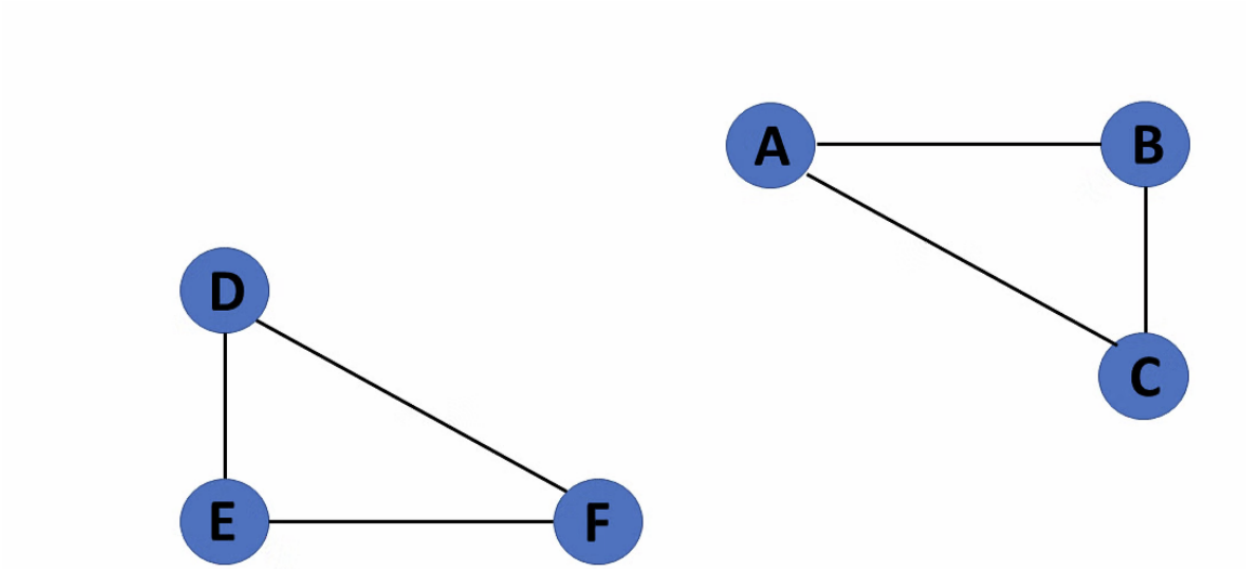
If there is a path between one vertex of a graph data structure and any other vertex, the graph is connected.



Other Types of Graphs

- **Disconnected Graph**

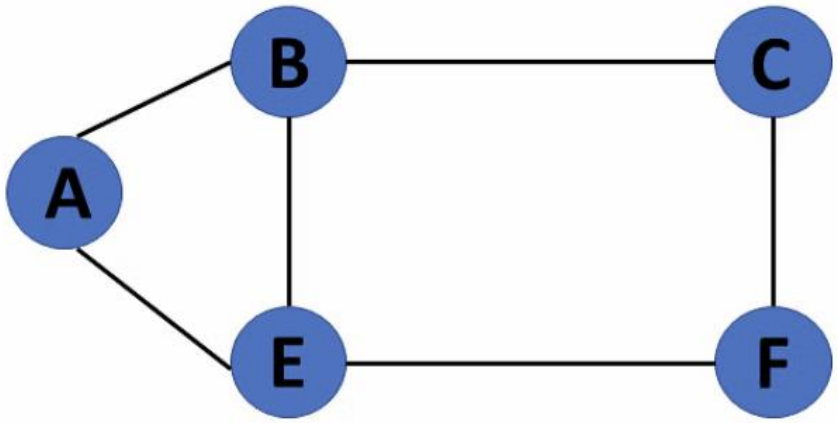
When there is no edge linking the vertices, you refer to the null graph as a disconnected graph.



Other Types of Graphs

- **Cyclic Graph**

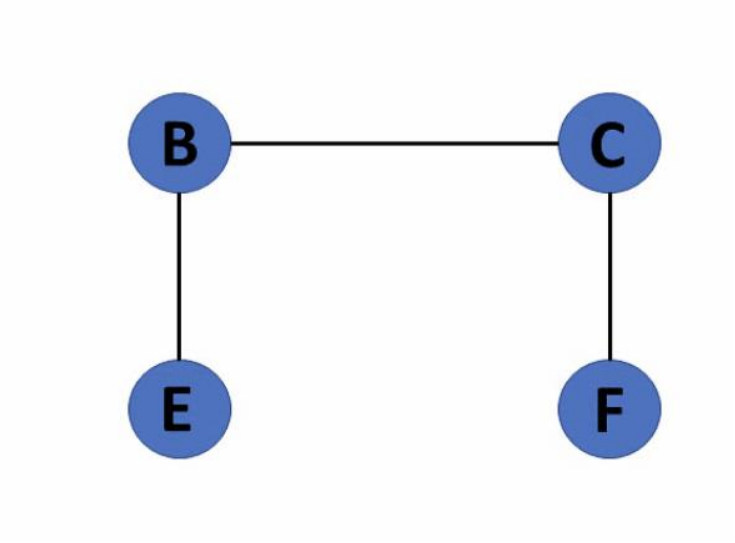
If a graph contains at least one graph cycle, it is considered to be cyclic.



Other Types of Graphs

- **Acyclic Graph**

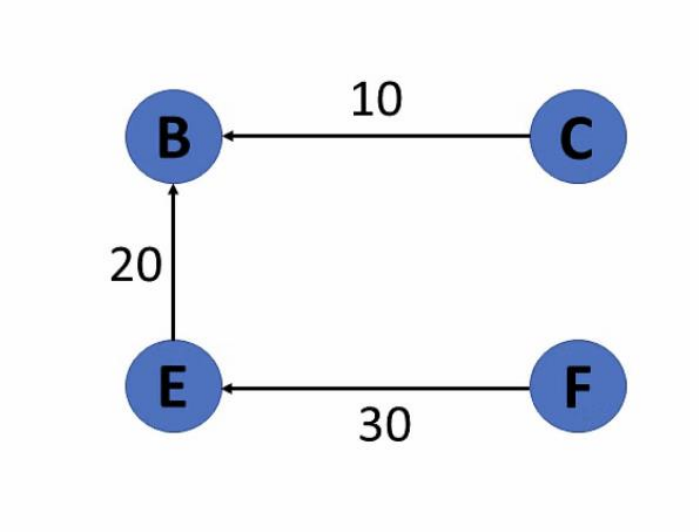
When there are no cycles in a graph, it is called an acyclic graph.



Other Types of Graphs

- **Directed Acyclic Graph**

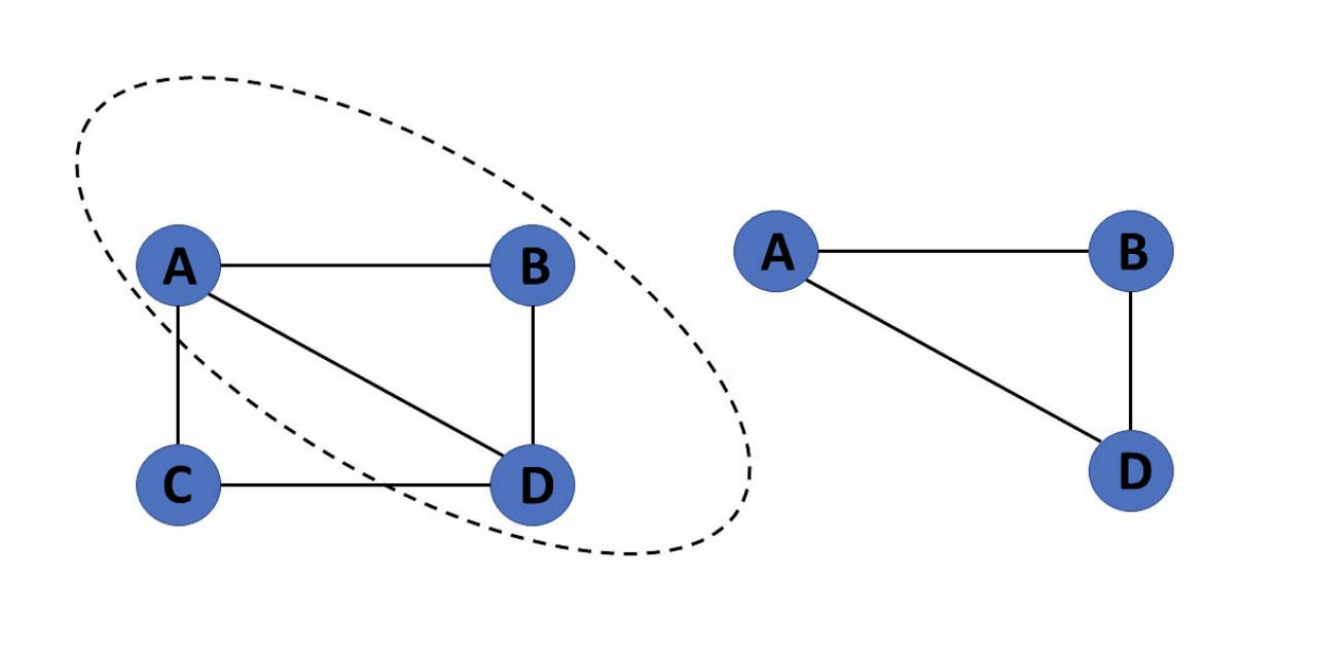
It's also known as a directed acyclic graph (DAG), and it's a graph with directed edges but no cycle. It represents the edges using an ordered pair of vertices since it directs the vertices and stores some data.



Other Types of Graphs

- **Sub Graph**

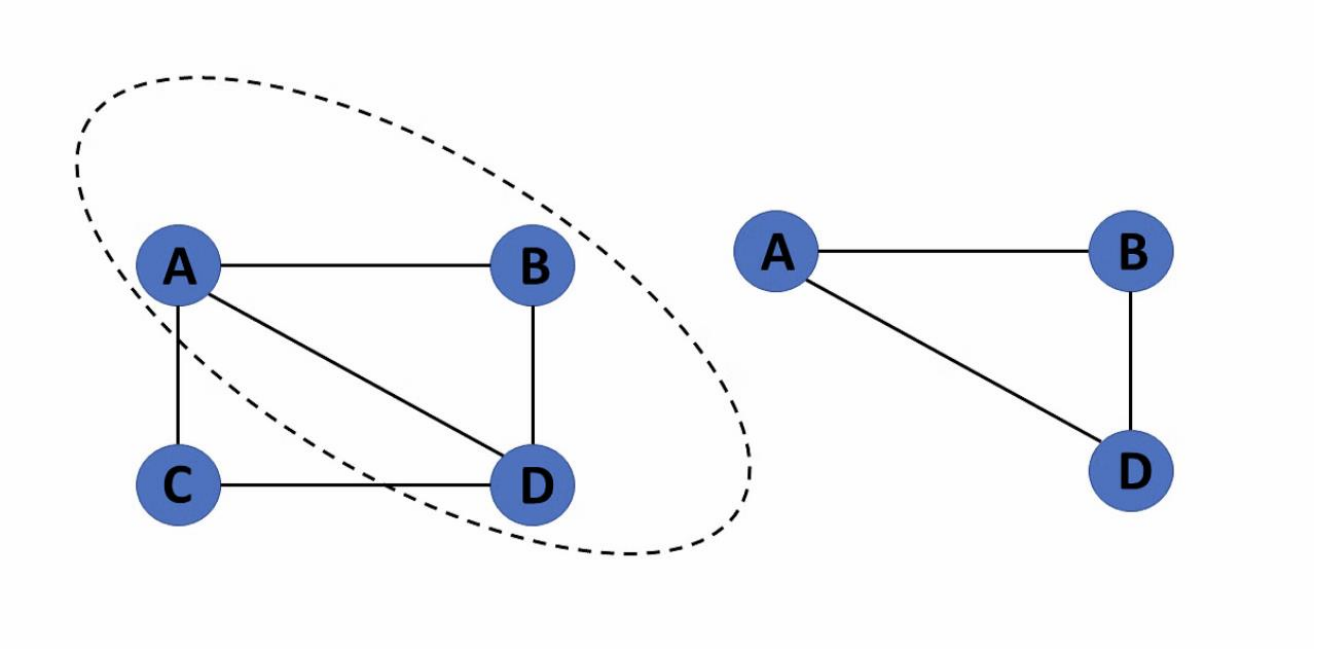
The vertices and edges of a graph that are subsets of another graph are known as a subgraph.



Other Types of Graphs

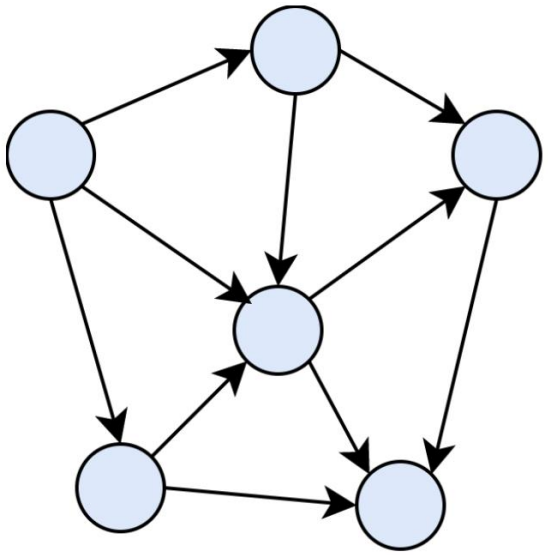
- **Sub Graph**

The vertices and edges of a graph that are subsets of another graph are known as a subgraph.

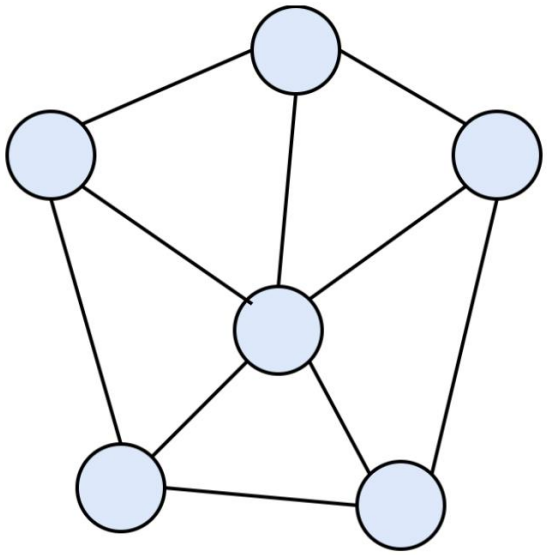


Directed(Di-graph) vs Undirected Graph

- **Directed (Digraph) –** A directed graph is a set of vertices (nodes) connected by edges, with each node having a direction associated with it. Edges are usually represented by arrows pointing in the direction the graph can be traversed.
- **Undirected Graphs –** In an undirected graph the edges are bidirectional, with no direction associated with them. Hence, the graph can be traversed in either direction. The absence of an arrow tells us that the graph is undirected.



Directed Graph



Undirected Graph

Weighted vs Unweighted Graphs:

- A weight is a numerical value attached to each individual edge in the graph.
  Weighted Graph will contains weight on each edge whereas unweighted does not.

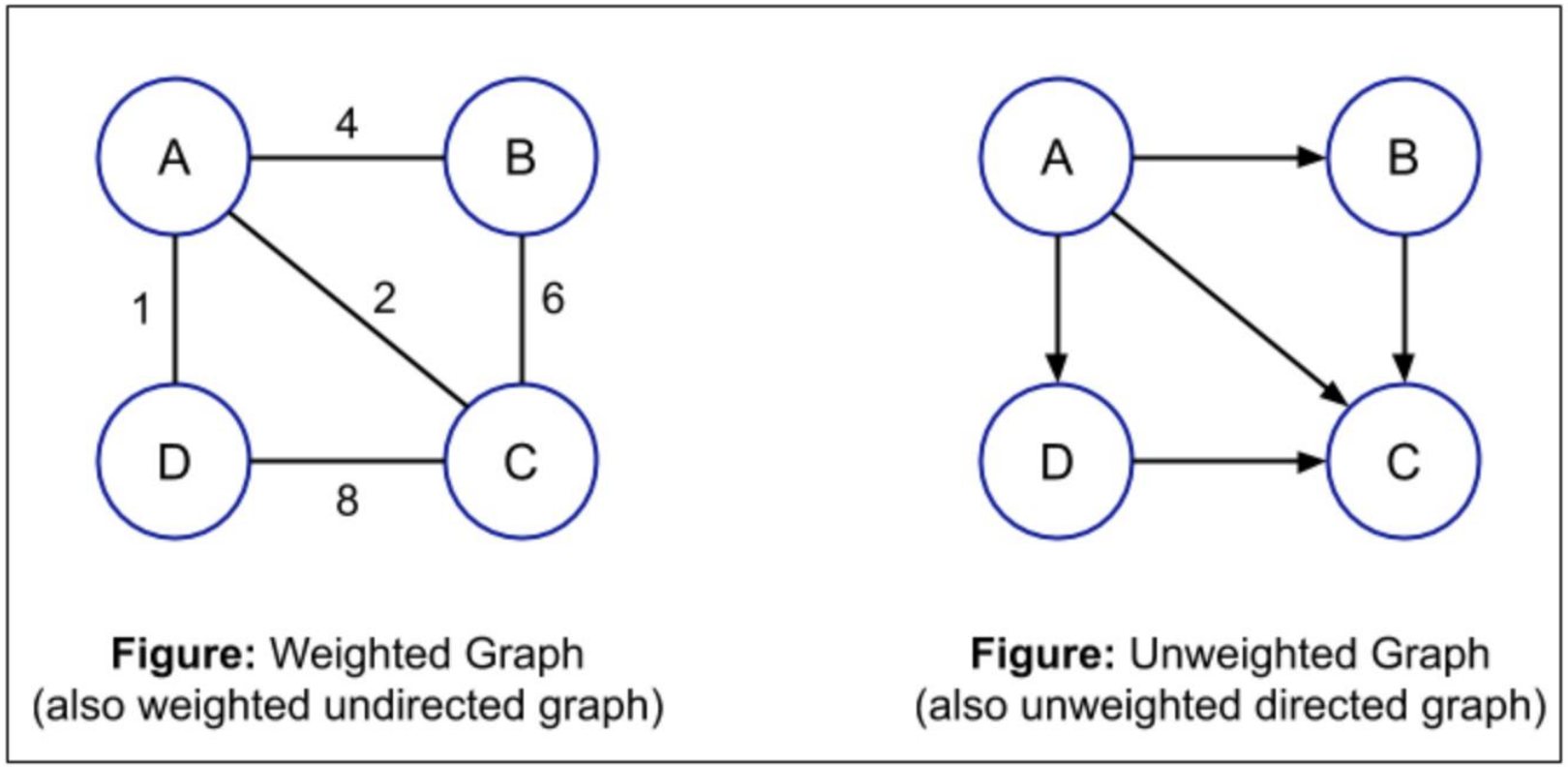
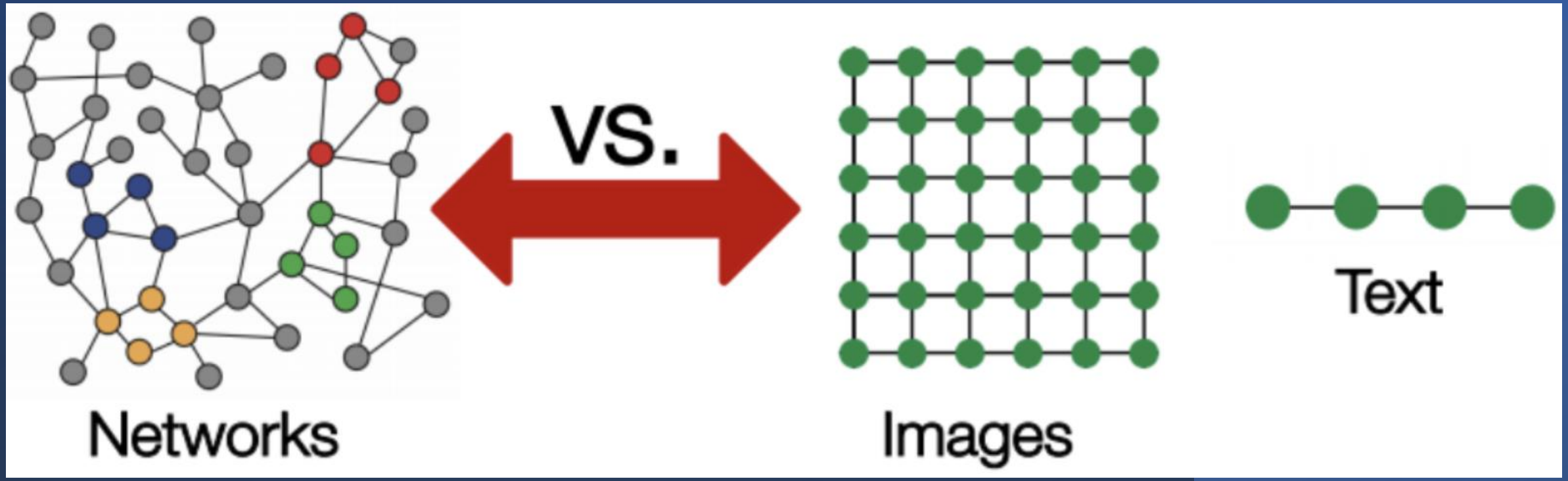


Figure: Weighted Graph
(also weighted undirected graph)

Figure: Unweighted Graph
(also unweighted directed graph)


VS.
Networks
Images
Text

Where to find graphs?

- **Images as graphs**

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 exactly 8 neighbors, and the information stored at each node is a 3-dimensional vector representing the RGB value of the pixel. 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 exactly 8 neighbors, and the information stored at each node is a 3-dimensional vector representing the RGB value of the pixel.

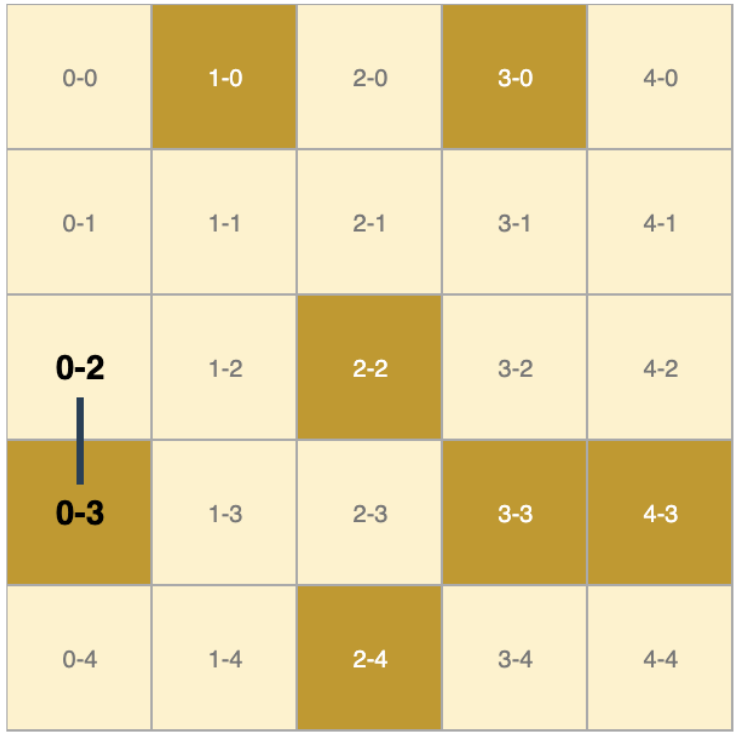
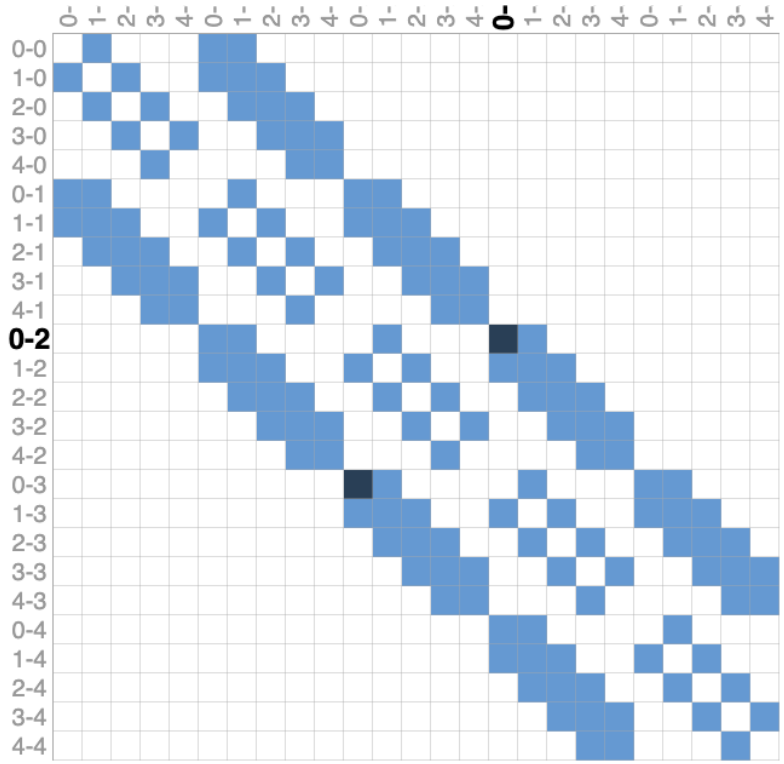
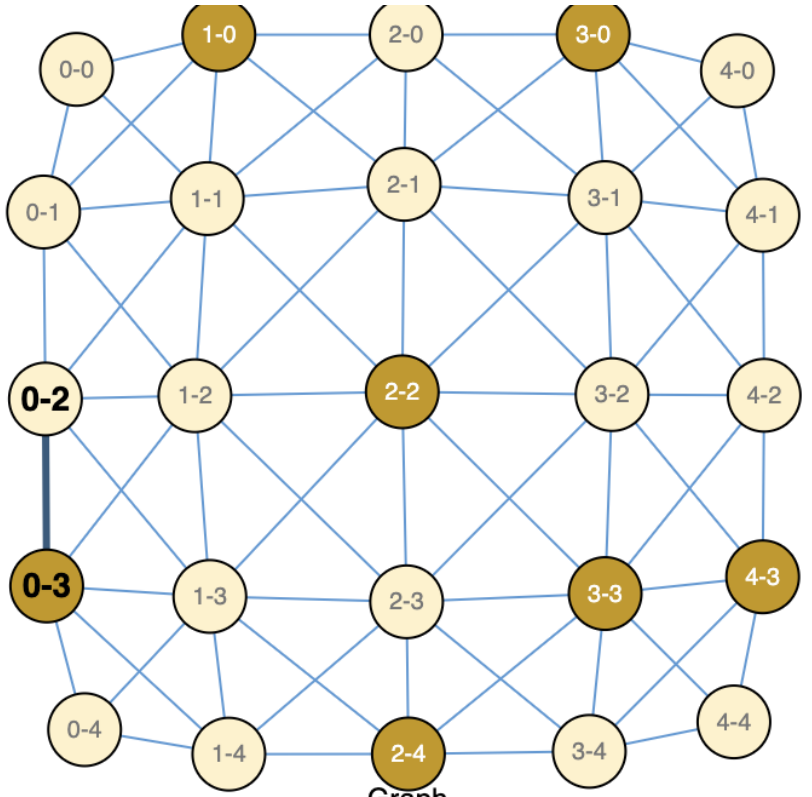


Image Pixels



Adjacency Matrix

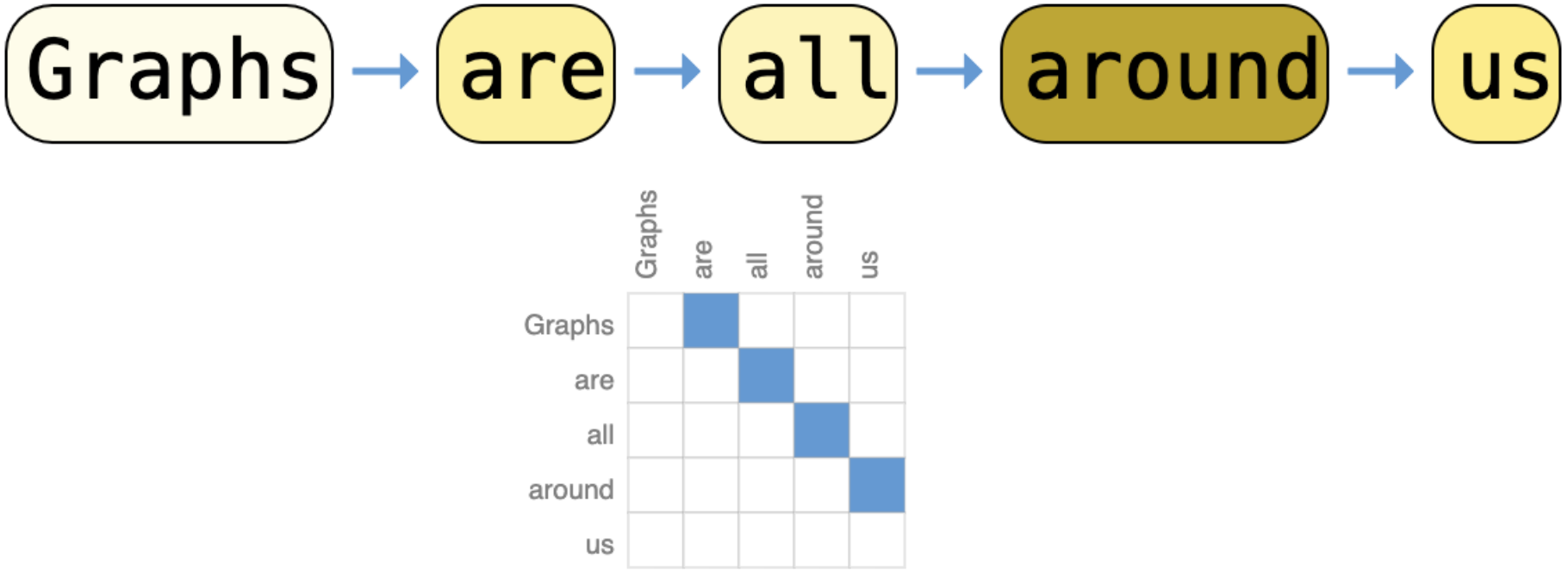


Graph

Where to find graphs?

- **Text as graphs**

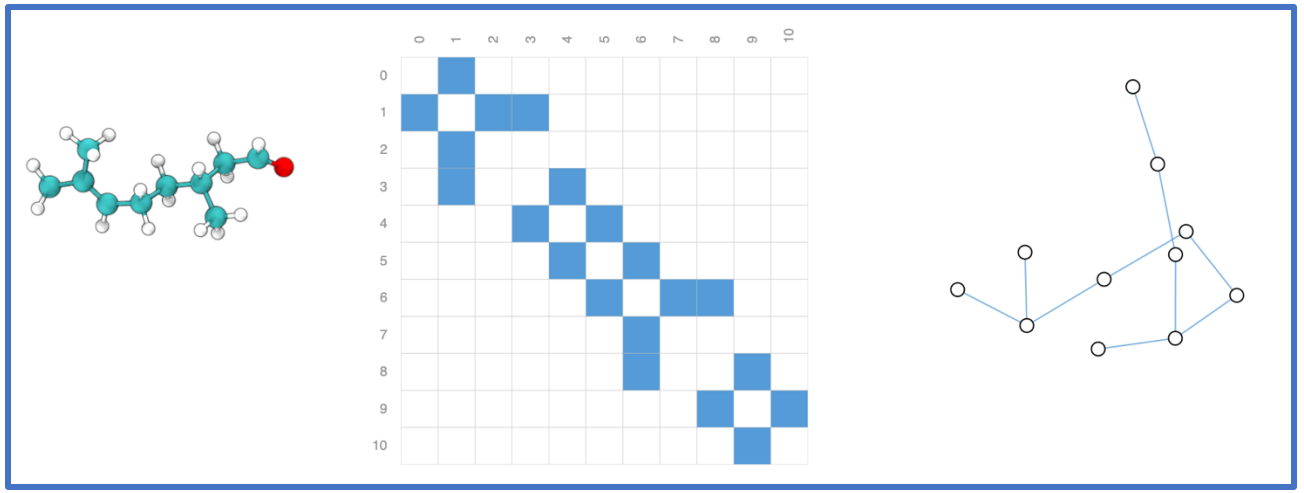
We can digitize text by associating indices to each character, word, or token, and representing text as a sequence of these indices. This creates a simple directed graph, where each character or index is a node and is connected via an edge to the node that follows it.

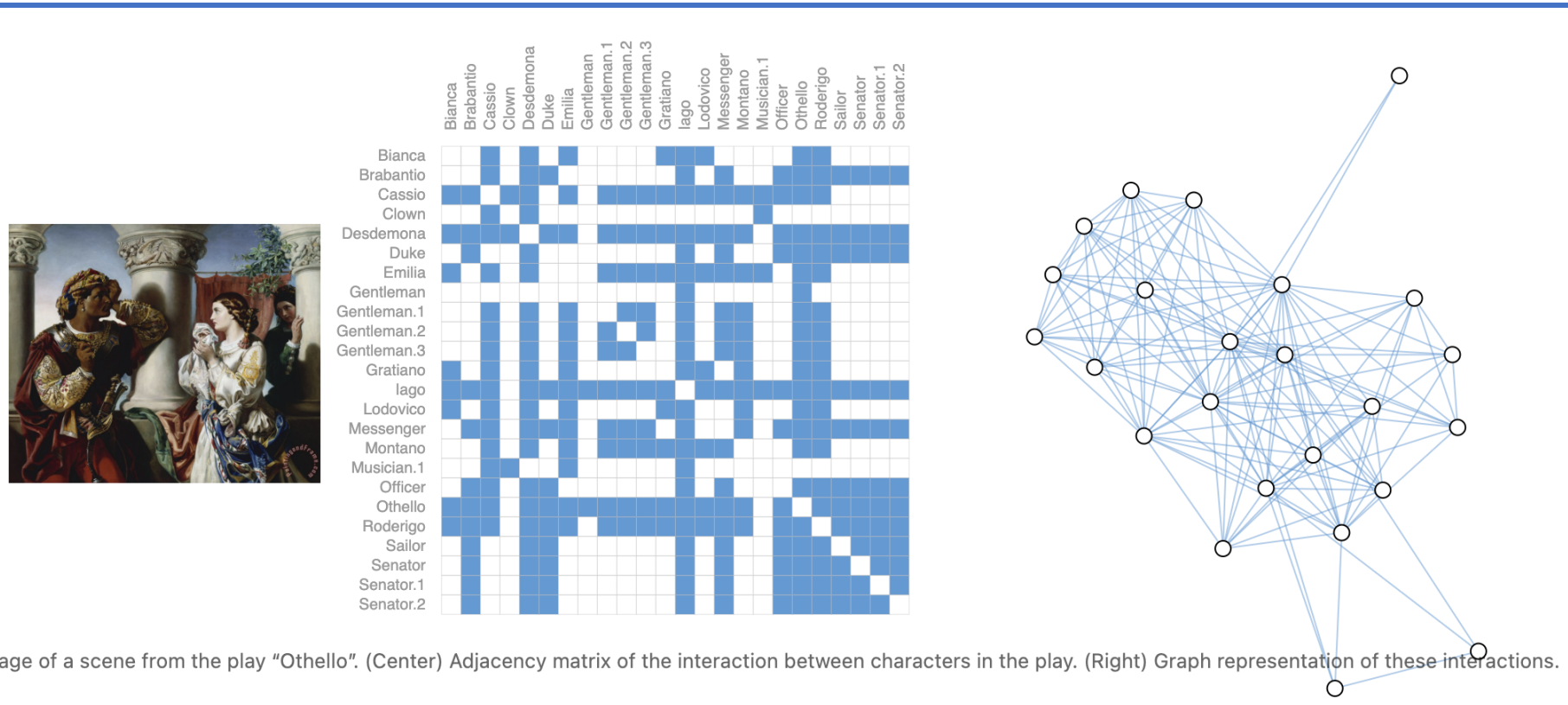


Note: This representation (a sequence of character tokens) refers to the way text is often represented in RNNs; other models, such as Transformers, can be considered to view text as a fully connected graph where we learn the relationship between tokens.

Graph Applications

- Molecules as graphs
- Social networks as graphs
- Citation networks as graphs
- A lot of tasks in computer vision

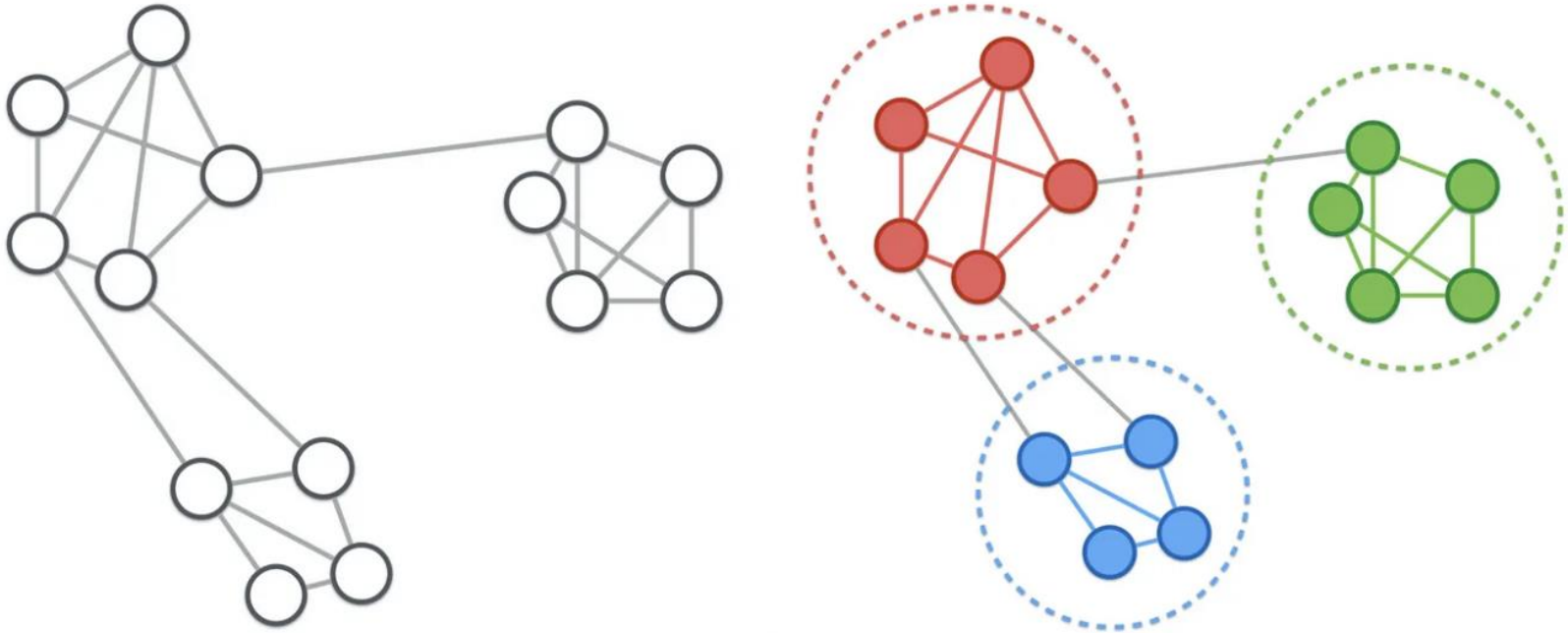




(Left) Image of a scene from the play "Othello". (Center) Adjacency matrix of the interaction between characters in the play. (Right) Graph representation of these interactions.

Introduction to Graph Machine Learning

At its core, Graph machine learning (GML) is the application of machine learning to graphs specifically for predictive and prescriptive tasks. GML has a variety of use cases across supply chain, fraud detection, recommendations, customer 360, drug discovery, and more.

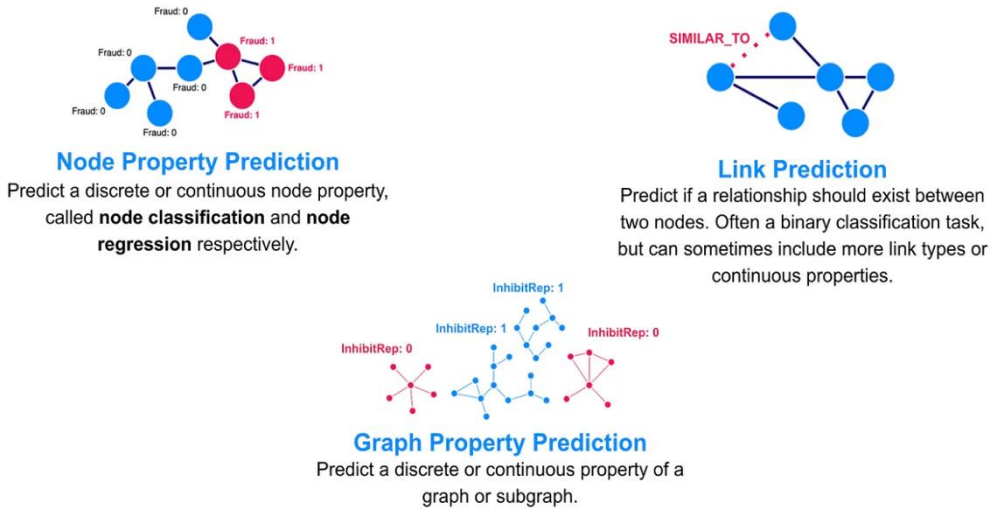


Supervised Machine Learning

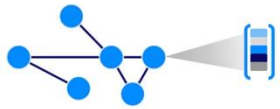
1.Node property prediction: Predicting a discrete or continuous node property. One can think of node property prediction as ***predicting an adjective about a thing***, such as whether an account on a financial services platform should be classified as fraud or how to categorize a product on an online retail store.

2.Link prediction: Predicting whether or not a relationship should exist between two nodes and potentially some properties about the relationship. Link prediction is helpful for applications like entity resolution, where we want to predict whether two nodes reflect the same underlying entity; recommendation systems where we want to predict what a user will want to purchase or interact with next; and bioinformatics, for predicting things like protein and drug interactions. For each case, we care about ***predicting an association, similarity, or potential action or interaction between entities***.

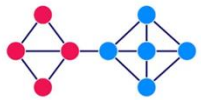
3.Graph property prediction: Predicting a discrete or continuous property of a graph or subgraph. Graph property prediction is useful in domains where you want to ***model each entity as an individual graph for prediction*** rather than modeling entities as nodes within a larger graph representing a complete dataset. Use cases include material sciences, bioinformatics, and drug discovery, where individual graphs can represent molecules or proteins that you want to make predictions about.



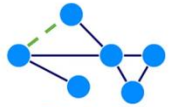
Unsupervised GML Tasks



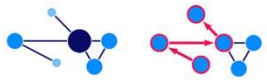
Representation Learning
Automatically generate features based on graph structure for downstream ML and EDA.



Clustering / Community Detection
Identify groups of nodes that have higher connectivity between each other than the rest of the graph.



Similarity
Find and measure similar pairs of nodes in the graph. Use for recommendation, entity resolution, and more.

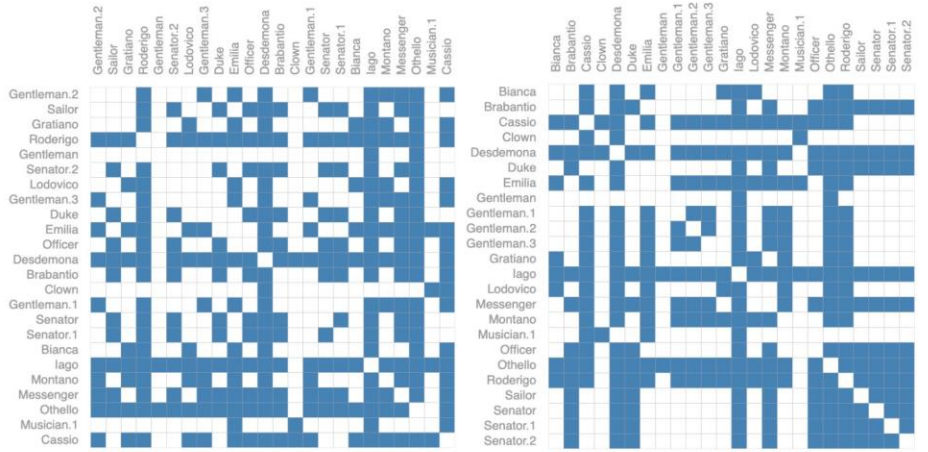


Centrality & Pathfinding
Find important and influential entities in the graph. Identify and evaluate more efficient paths and trees.

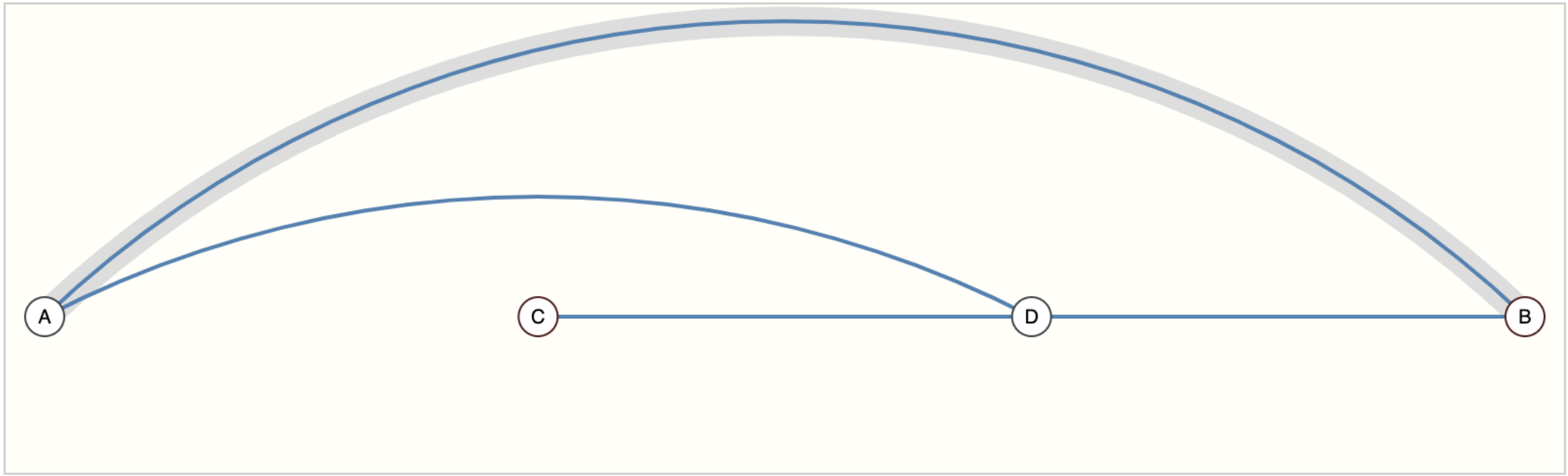
- **Representation Learning**: Reducing dimensionality while maintaining important signals is a central theme for GML applications. Graph representation learning does this explicitly by generating low-dimensional features from graph structures, usually to use them for downstream exploratory data analysis (EDA) and ML.
- **Community Detection (clustering for relationships):** Community detection is a clustering technique for identifying groups of densely interconnected nodes within a graph. Community detection has various practical applications in anomaly detection, fraud and investigative analytics, social network analysis, and biology.
- **Similarity:** Similarity in GML refers to finding and measuring similar pairs of nodes in a graph. Similarity is applicable to many use cases, including recommendation, entity resolution, and anomaly and fraud detection. Common Similarity techniques include node similarity algorithms, topological link prediction, and K-Nearest-Neibor (KNN).
- **Centrality & Pathfinding:** I'm grouping these together as they tend to be less associated with ML tasks and more with analytical measures. However, they still technically fit here, so I will cover them for completeness. Centrality finds important or influential nodes in a graph. Centrality is ubiquitous throughout many use cases, including fraud and anomaly detection, recommendation, supply chain, logistics, and infrastructure problems. Pathfinding is used to find the lowest cost paths in a graph or to evaluate the quality and availability of paths. Pathfinding can benefit many use cases dealing with physical systems such as logistics, supply chain, transportation, and infrastructure.

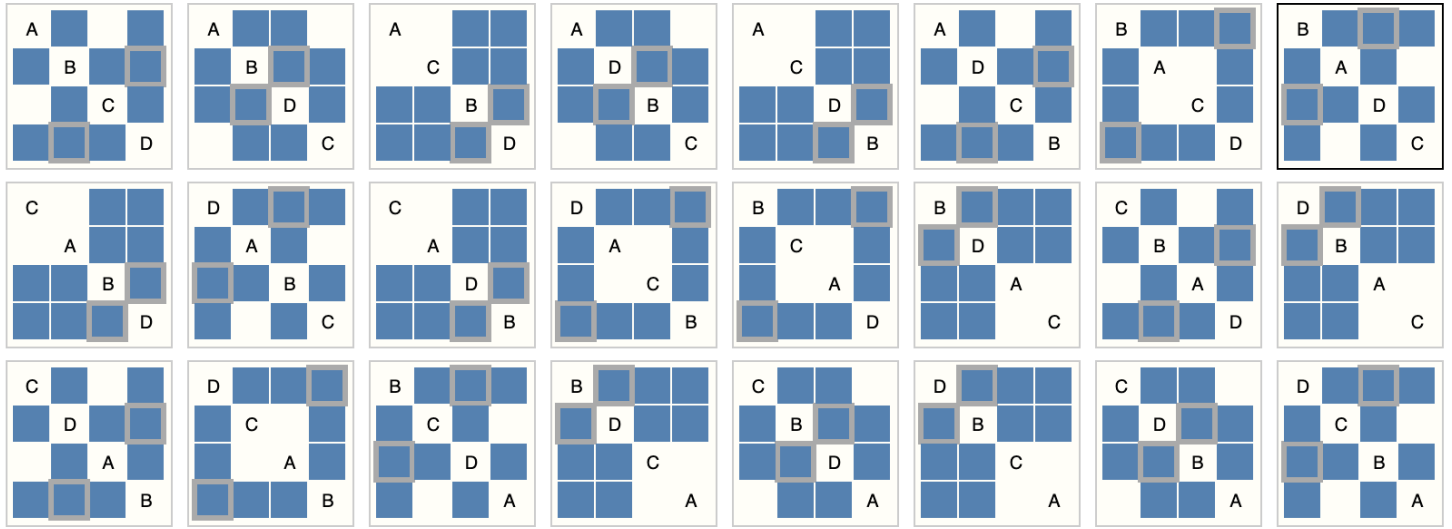
Challenges of GML

- **Node Representation**: Representing nodes with feature matrices is a straightforward approach. Each node is assigned an index, and its features are stored in a matrix. This matrix can then be processed by neural networks.
- **Edge Representation**: Edges can be represented similarly to nodes, by storing edge features in matrices or tensors. These features might include edge weights, types, or any other relevant information.
- **Global Context Representation**: Representing global context involves capturing information about the entire graph. This could include graph-level features such as size, density, or other properties that characterize the overall structure of the graph.
- **Connectivity Representation**: Representing graph connectivity is more complex. One common approach is to use an adjacency matrix, where each entry indicates whether there is a connection between two nodes. However, as you mentioned, adjacency matrices can be very sparse and space-inefficient, especially for large graphs.
- **Permutation Invariance**: There is no guarantee that different adjacency matrices encoding the same connectivity will produce the same result in a neural network. Learning permutation-invariant operations is an active area of research. Techniques such as graph isomorphism networks or graph neural networks (GNNs) aim to address this challenge by learning representations that are invariant to permutations of node indices.



Two adjacency matrices representing the same graph.


A
C
D
B

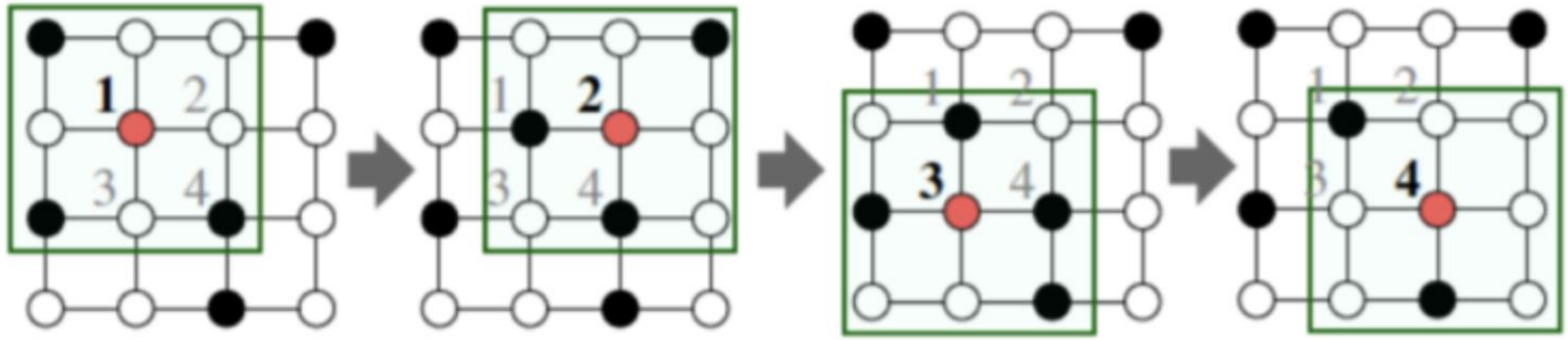

A B C D
A B D C
A C B D
A D B C
A C D B
A D C B
B A C D
B A D C
C A B D
D A B C
C A D B
D A C B
B C A D
B D A C
C B A D
D B A C
C D A B
D C A B
B C D A
B D C A
C B D A
D B C A
C D B A
D C B A

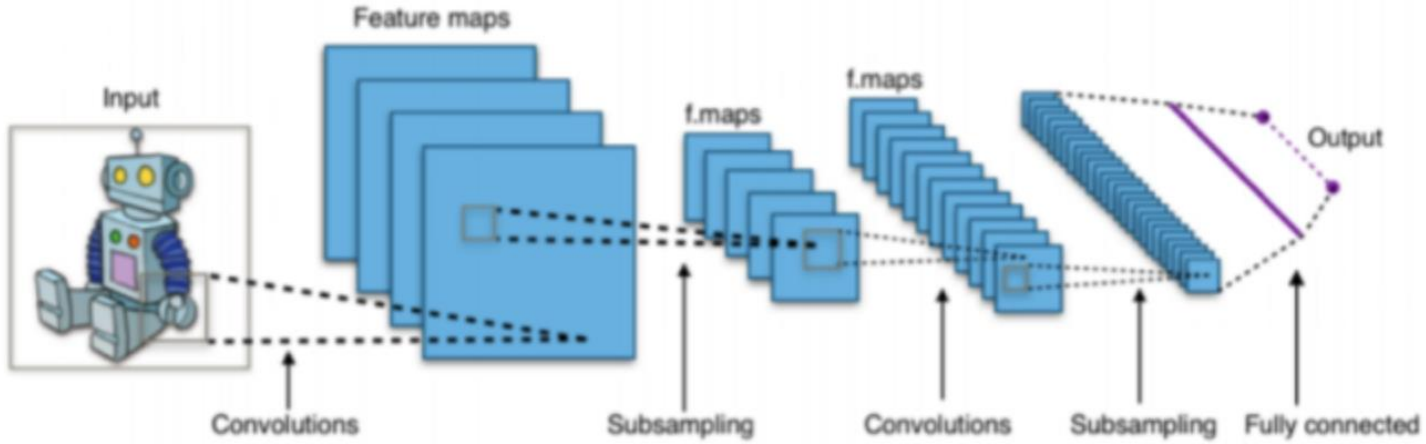
Graph Neural Networks

- **Graph Neural Networks (GNNs) are a class of deep learning methods designed to perform inference on data described by graphs.**
- **GNNs are neural networks that can be directly applied to graphs, and provide an easy way to do node-level, edge-level, and graph-level prediction tasks.**
- **GNNs can do what Convolutional Neural Networks (CNNs) failed to do.**

Why do Convolutional Neural Networks (CNNs) fail on graphs?

- CNNs can be used to make machines visualize things, and perform tasks like image classification, image recognition, or object detection. This is where CNNs are the most popular.
- The core concept behind CNNs introduces hidden convolution and pooling layers to identify spatially localized features via a set of receptive fields in kernel form.

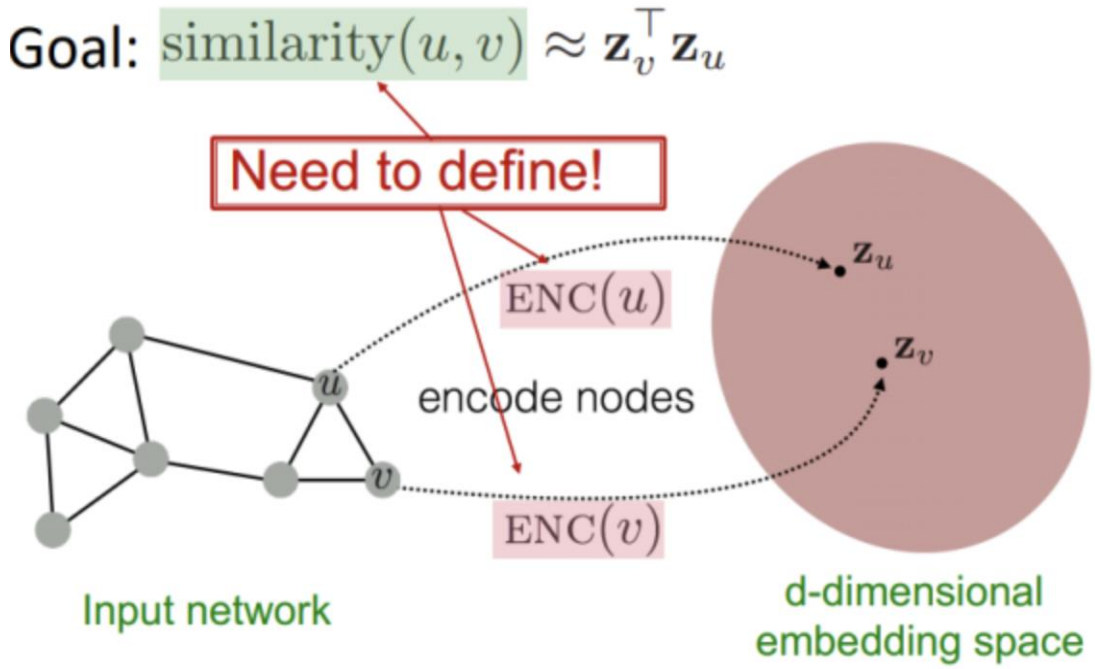




- How does convolution operate on images that are regular grids? We slide the convolutional operator window across a two-dimensional image, and we compute some function over that sliding window. Then, we pass it through many layers.
- Our goal is to generalize the notion of convolution beyond these simple two-dimensional lattices.
- The insight allowing us to reach our goal is that **convolution takes a little sub-patch of the image (a little rectangular part of the image), applies a function to it, and produces a new part (a new pixel).**
- What happens is that the center node of that center pixel aggregates information from its neighbors, as well as from itself, to produce a new value.
- It's very difficult to perform CNN on graphs because of the arbitrary size of the graph, and the complex topology, which means there is no spatial locality.
- There's also unfixed node ordering. If we first labeled the nodes A, B, C, D, E, and the second time we labeled them B, D, A, E, C, then the inputs of the matrix in the network will change. Graphs are invariant to node ordering, so we want to get the same result regardless of how we order the nodes.

Basic of Deep Learning on Graphs

In graph theory, we implement the concept of Node Embedding. It means mapping nodes to a d- dimensional embedding space (low dimensional space rather than the actual dimension of the graph), so that similar nodes in the graph are embedded close to each other.
Our goal is to map nodes so that similarity in the embedding space approximates similarity in the network.
Let's define ***u*** and ***v*** as two nodes in a graph.
x_u and x_v are two feature vectors.
Now we'll define the encoder function ***Enc(u)*** and ***Enc(v)***, which convert the feature vectors to z_u and z_v.
Note: the similarity function could be Euclidean distance.

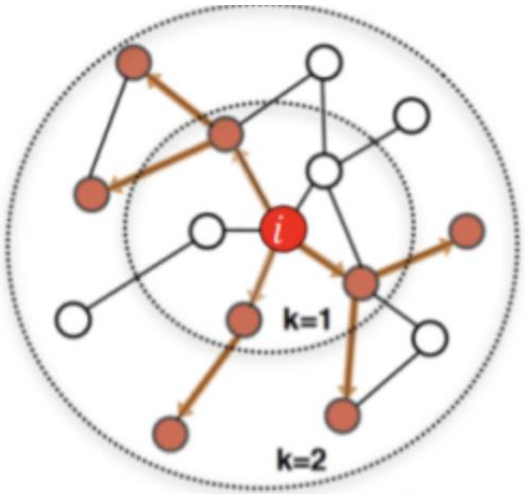


Basic of Deep Learning on Graphs

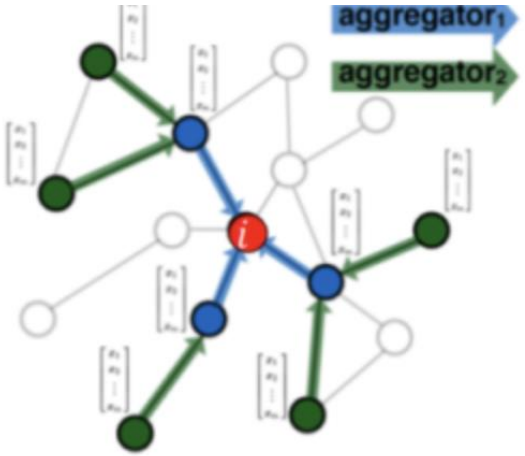
So the challenge now is how to come up with the encoder function?

The encoder function should be able to perform :

- Locality (local network neighborhoods)
- Aggregate information
- Stacking multiple layers (computation)
- Locality information can be achieved by using a computational graph. As shown in the graph below, ***i*** is the red node where we see how this node is connected to its neighbors and those neighbors' neighbors. We'll see all the possible connections, and form a computation graph.
- By doing this, we're capturing the structure, and also borrowing feature information at the same time

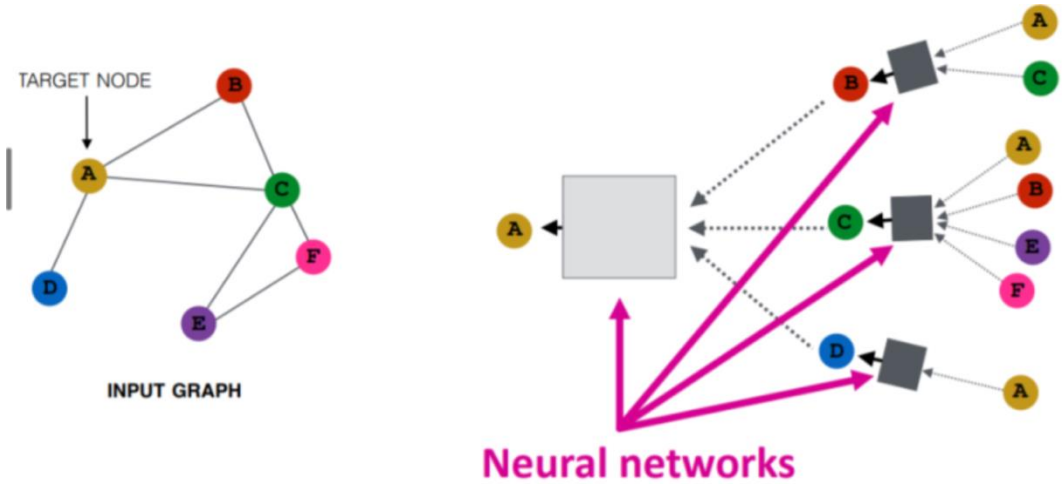


Determine node computation graph

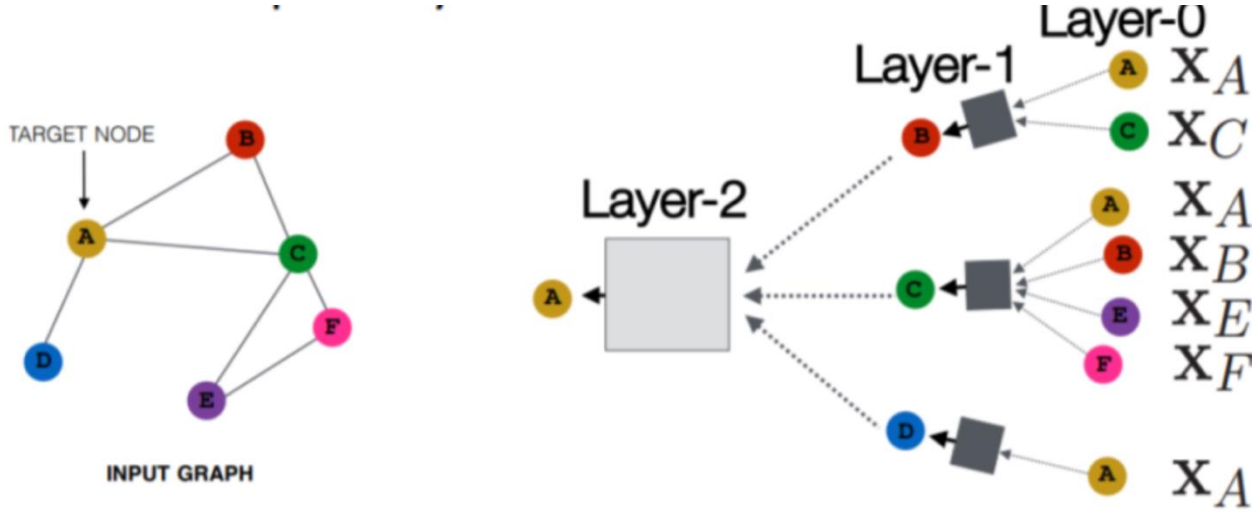


Propagate and transform information

- Once the locality information preserves the computational graph, we start aggregating. This is basically done using neural networks.



- Neural Networks are presented in grey boxes. They require aggregations to be order-invariant, like sum, average, maximum, because they are permutation-invariant functions. This property enables the aggregations to be performed.
- Let's move on to the **forward propagation rule** in GNNs. It determines how the information from the input will go to the output side of the neural network.



Every node has a feature vector.
For example, (X_A) is a feature vector of node ***A***.
The inputs are those feature vectors, and the box will take the two feature vectors (X_A and X_c), aggregate them, and then pass on to the next layer.
Notice that, for example, the input at node ***C*** are the features of node ***C***, but the representation of node ***C*** in layer 1 will be a hidden, latent representation of the node, and in layer 2 it'll be another latent representation.
So in order to perform forward propagation in this computational graph, we need 3 steps:

1. Initialize the activation units:

$$h_v^0 = X_v \ (feature\ vector)$$

2. Every layer in the network:

$$h_v^k = \sigma(W_k \sum \frac{h_u^{k-1}}{|\ N(v)} + B_k h_v^{k-1}) \text{ where } k = 1, ..., k-1$$

We can notice that there are two parts for this equation:
•The first part is basically averaging all the neighbors of node ***v***.

$$W_k \sum \frac{h_u^{k-1}}{|\ N(v)}$$

•The second part is the previous layer embedding of node ***v*** multiplied with a bias B_k, which is a trainable weight matrix and it's basically a self-loop activation for node ***v***.

$$B_k h_v^{k-1}$$

•**σ**: the non-linearity activation that is performed on the two parts.

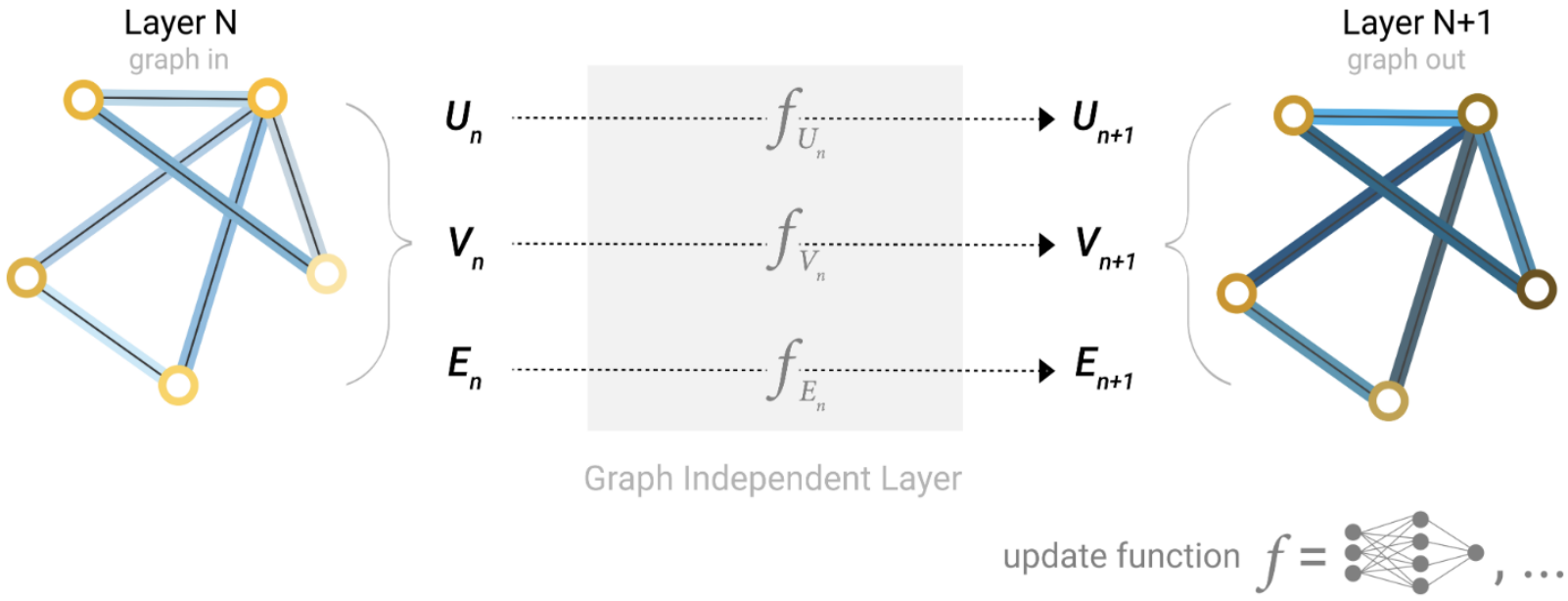
3. The last equation (at the final layer):

It's the embedding after ***K*** layers of neighborhood aggregation. Now, to train the model we need to define a loss function on the embeddings. We can feed the embeddings into any loss function and run stochastic gradient descent to train the weight parameters.

$$z_v = h_v^K$$

Simple GNN

- With the numerical representation of graphs, we are now ready to build a GNN. We will start with the simplest GNN architecture, one where we learn new embeddings for all graph attributes (nodes, edges, global), but where we do not yet use the connectivity of the graph.
- This GNN uses a separate multilayer perceptron (MLP) (or your favorite differentiable model) on each component of a graph; we call this a GNN layer. For each node vector, we apply the MLP and get back a learned node-vector. We do the same for each edge, learning a per-edge embedding, and also for the global-context vector, learning a single embedding for the entire graph



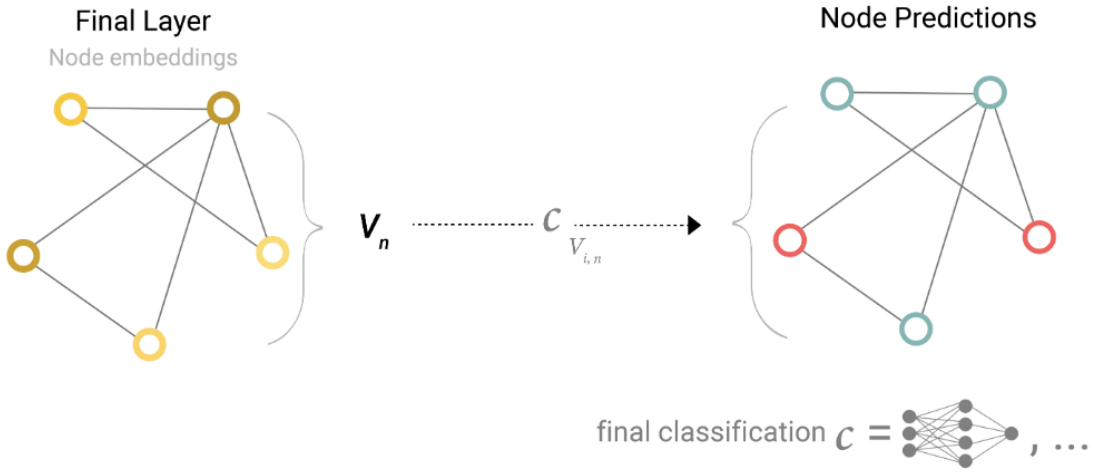
A single layer of a simple GNN. A graph is the input, and each component (V,E,U) gets updated by a MLP to produce a new graph. Each function subscript indicates a separate function for a different graph attribute at the n-th layer of a GNN model.

Simple GNN

Note: As is common with neural networks modules or layers, we can stack these GNN layers together.
Because a GNN does not update the connectivity of the input graph, we can describe the output graph of a GNN with the same adjacency list and the same number of feature vectors as the input graph. But, the output graph has updated embeddings, since the GNN has updated each of the node, edge and global-context representations.

GNN Predictions by Pooling Information

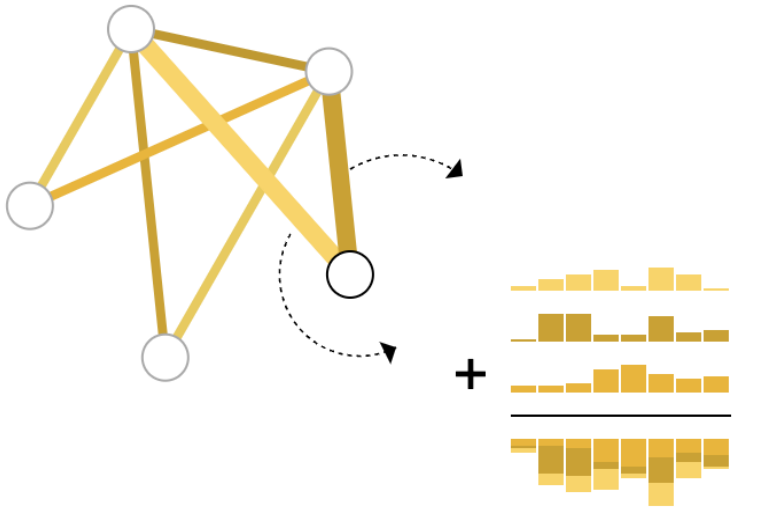
We will consider the case of binary classification, but this framework can easily be extended to the multi-class or regression case. If the task is to make binary predictions on nodes, and the graph already contains node information, the approach is straightforward — for each node embedding, apply a linear classifier.



Simple GNN

However, it is not always so simple. For instance, you might have information in the graph stored in edges, but no information in nodes, but still need to make predictions on nodes. We need a way to collect information from edges and give them to nodes for prediction. We can do this by *pooling*. Pooling proceeds in two steps:

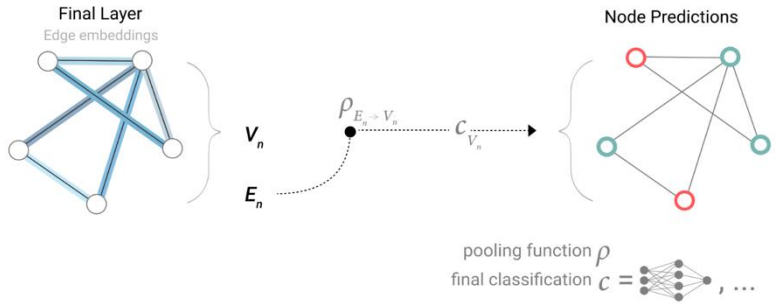
- For each item to be pooled, *gather* each of their embeddings and concatenate them into a matrix.
- The gathered embeddings are then *aggregated*, usually via a sum operation



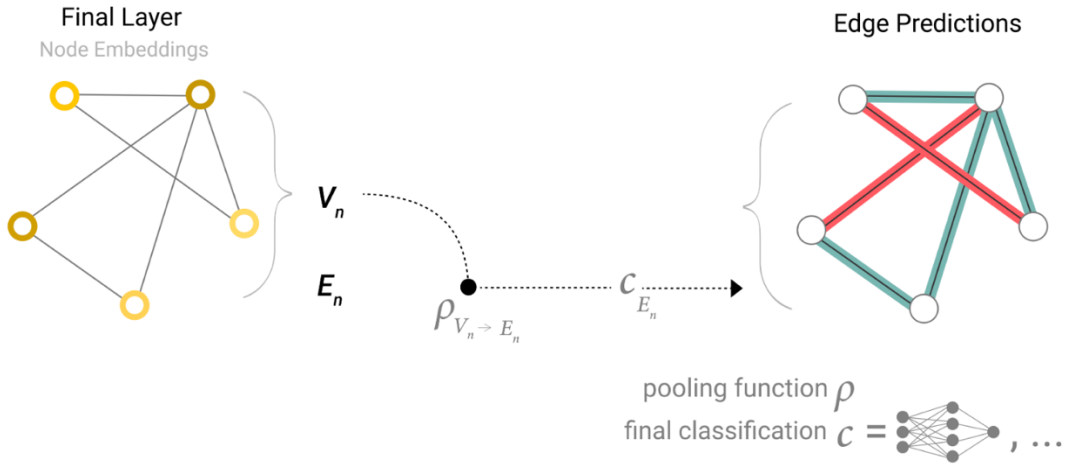
Aggregate information
from adjacent edges

Simple GNN

- So If we only have edge-level features, and are trying to predict binary node information, we can use pooling to route (or pass) information to where it needs to go. The model looks like this.

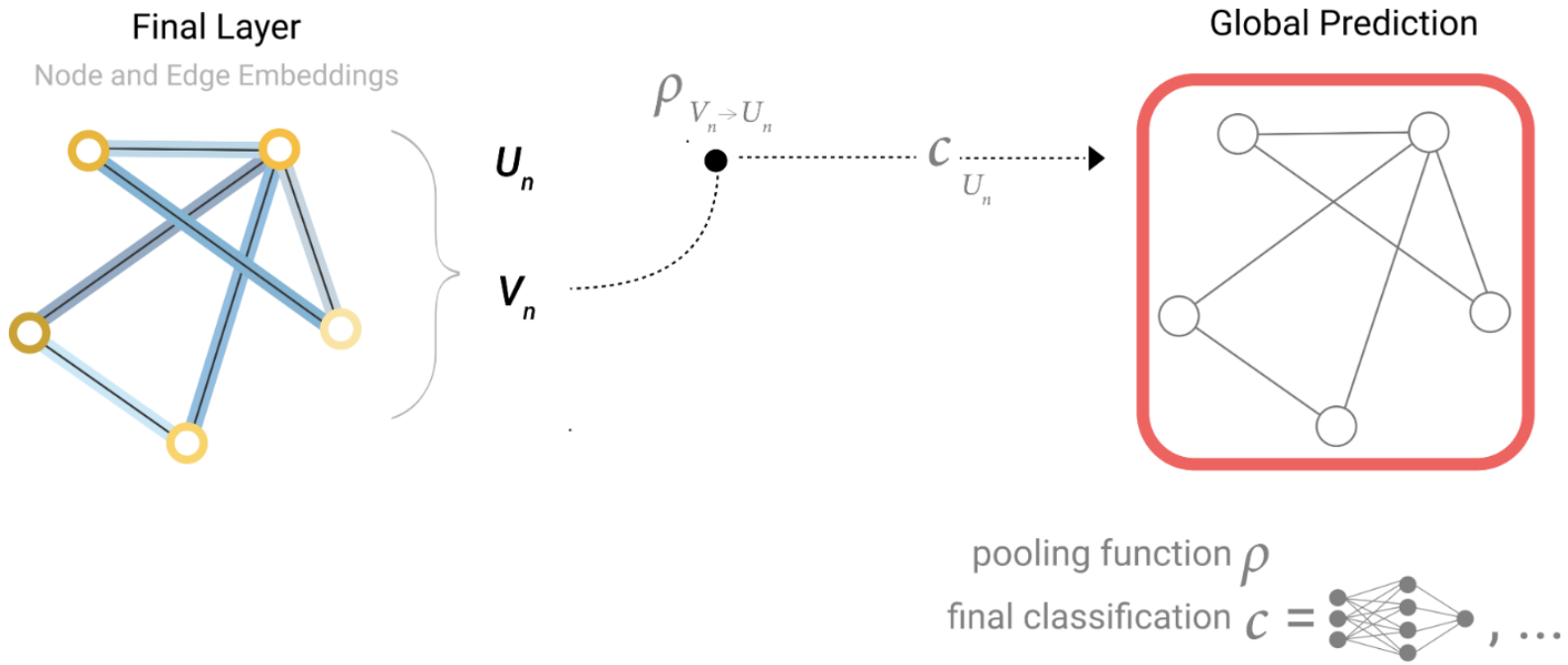


- If we only have node-level features, and are trying to predict binary edge-level information, the model looks like this.



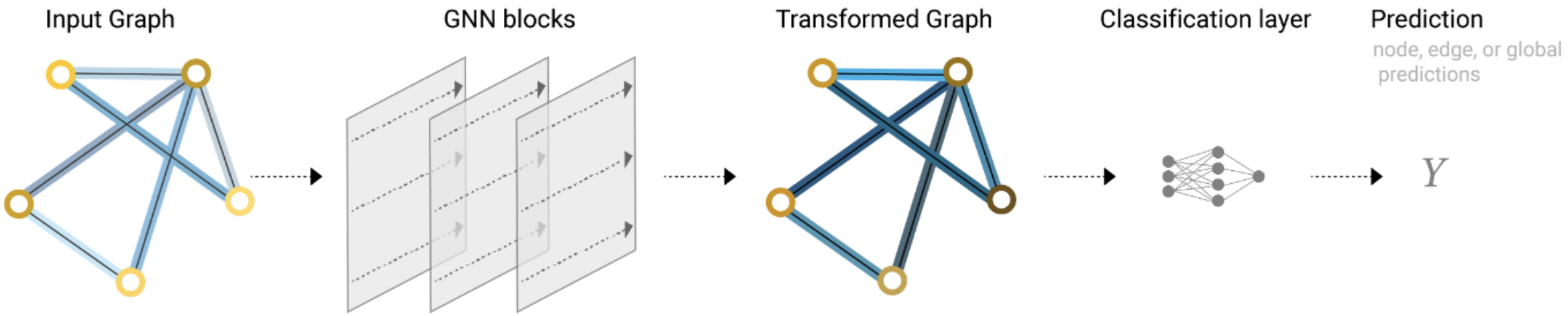
Simple GNN

- If we only have node-level features, and need to predict a binary global property, we need to gather all available node information together and aggregate them. This is similar to *Global Average Pooling* layers in CNNs. The same can be done for edges.



Simple GNN

- In our examples, the classification model *c* can easily be replaced with any differentiable model, or adapted to multi-class classification using a generalized linear model.



An end-to-end prediction task with a GNN model.

Passing messages between parts of the graph

We could make more sophisticated predictions by using pooling within the GNN layer, in order to make our learned embeddings aware of graph connectivity. We can do this using *message passing*. where neighboring nodes or edges exchange information and influence each other's updated embeddings.

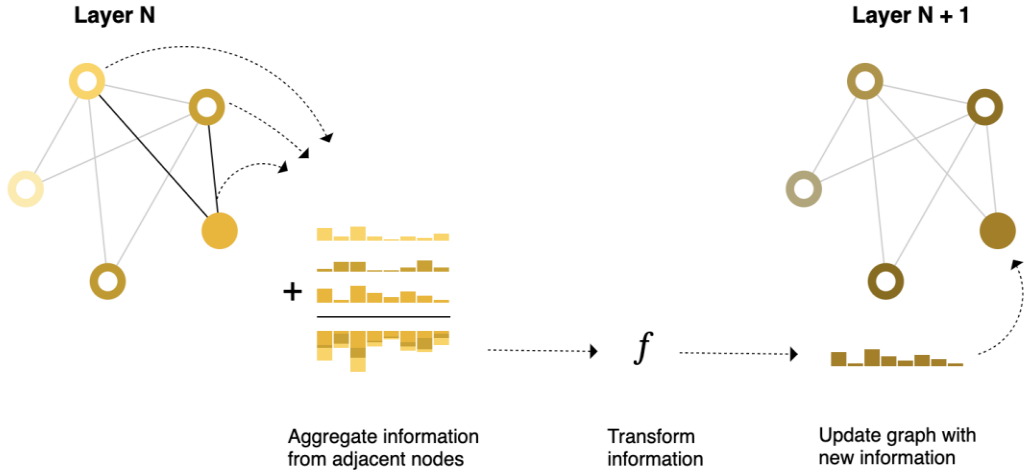
Message passing works in three steps:

1. For each node in the graph, *gather* all the neighboring node embeddings (or messages), which is the g function described above.
2. Aggregate all messages via an aggregate function (like sum).
3. All pooled messages are passed through an *update function*, usually a learned neural network.

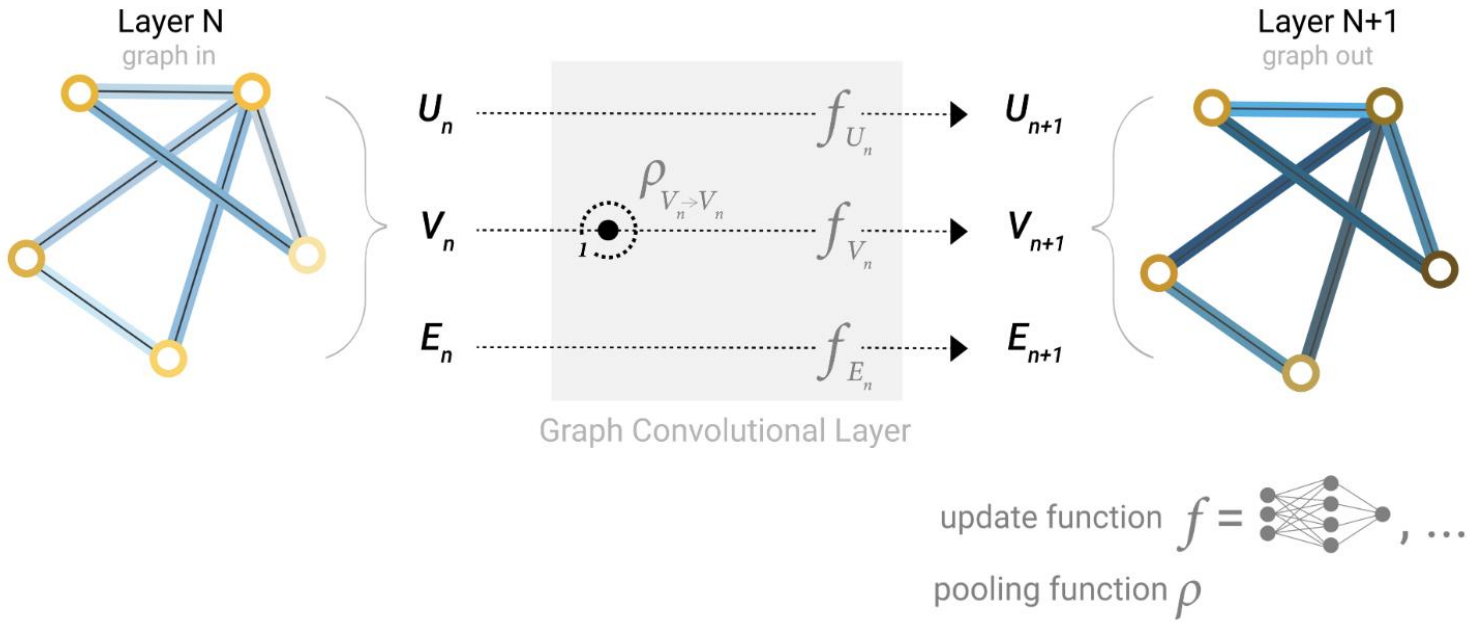
You could also 1) gather messages, 3) update them and 2) aggregate them and still have a permutation invariant operation.

Just as pooling can be applied to either nodes or edges, message passing can occur between either nodes or edges.

These steps are key for leveraging the connectivity of graphs. We will build more elaborate variants of message passing in GNN layers that yield GNN models of increasing expressiveness and power.



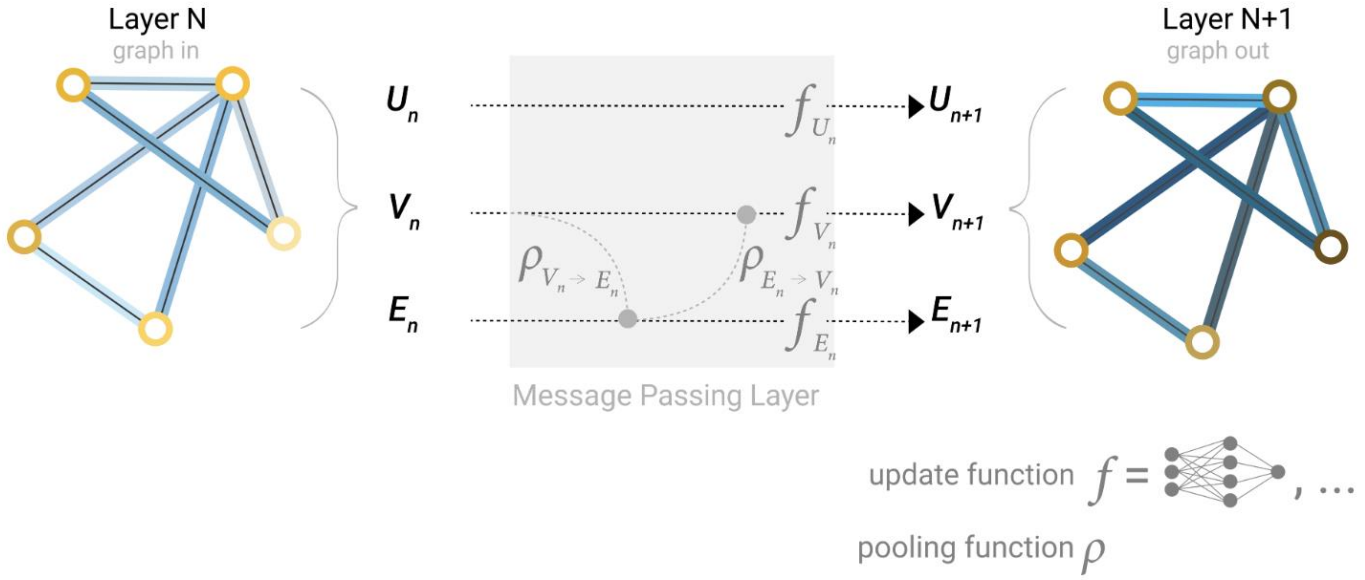
This sequence of operations, when applied once, is the simplest type of message-passing GNN layer.
This is reminiscent of standard convolution: in essence, message passing and convolution are operations to aggregate and process the information of an element's neighbors in order to update the element's value. In graphs, the element is a node, and in images, the element is a pixel. However, the number of neighboring nodes in a graph can be variable, unlike in an image where each pixel has a set number of neighboring elements.
By stacking message passing GNN layers together, a node can eventually incorporate information from across the entire graph: after three layers, a node has information about the nodes three steps away from it.
We can update our architecture diagram to include this new source of information for nodes:



Schematic for a GCN architecture, which updates node representations of a graph by pooling neighboring nodes at a distance of one degree.

Learning edge representations

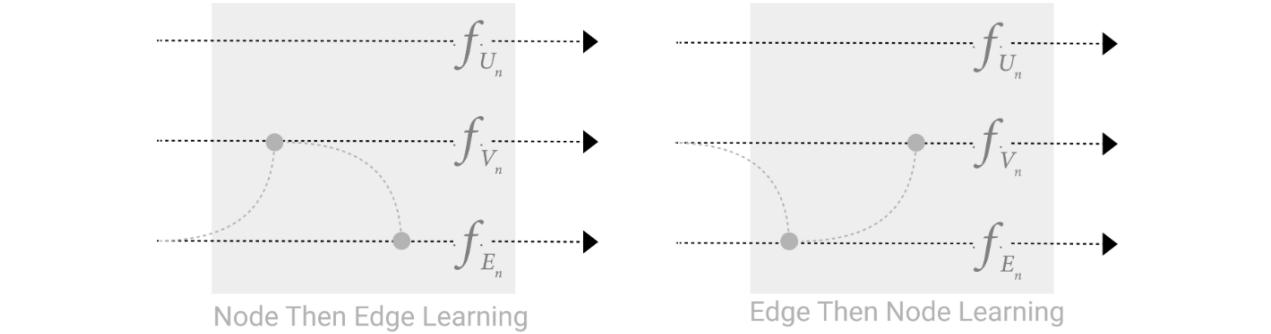
- Our dataset does not always contain all types of information (node, edge, and global context). When we want to make a prediction on nodes, but our dataset only has edge information, we showed above how to use pooling to route information from edges to nodes, but only at the final prediction step of the model. We can share information between nodes and edges within the GNN layer using message passing.
- We can incorporate the information from neighboring edges in the same way we used neighboring node information earlier, by first pooling the edge information, transforming it with an update function, and storing it.
- However, the node and edge information stored in a graph are not necessarily the same size or shape, so it is not immediately clear how to combine them. One way is to learn a linear mapping from the space of edges to the space of nodes, and vice versa. Alternatively, one may concatenate them together before the update function.

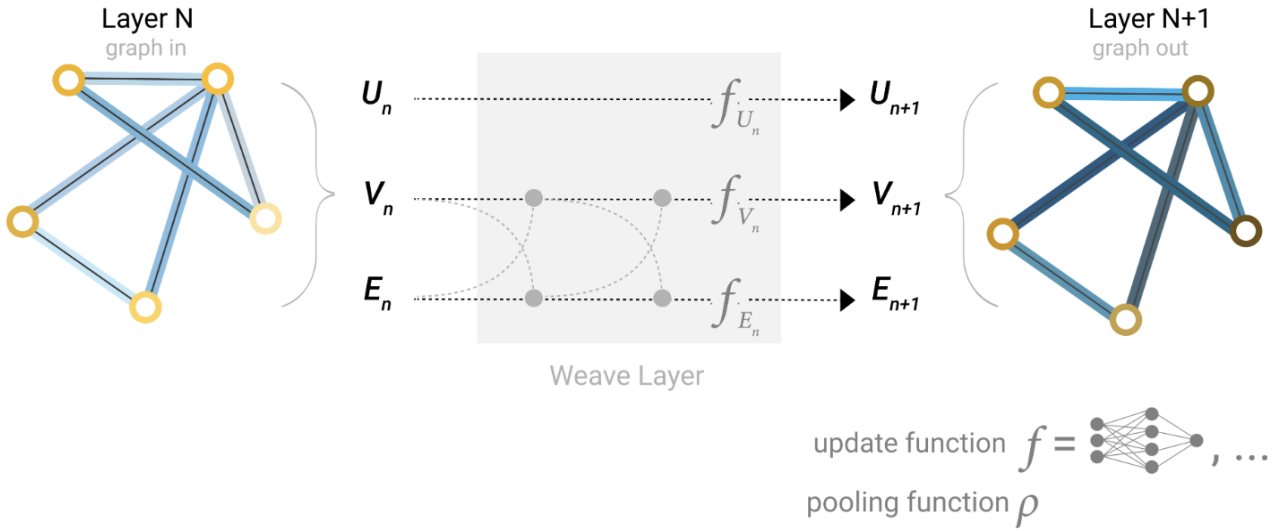


Architecture schematic for Message Passing layer. The first step "prepares" a message composed of information from an edge and it's connected nodes and then "passes" the message to the node.

Which graph attributes we update and in which order we update them is one design decision when constructing GNNs. We could choose whether to update node embeddings before edge embeddings, or the other way around. This is an open area of research with a variety of solutions– for example we could update in a ‘weave’ fashion.

where we have four updated representations that get combined into new node and edge representations: node to node (linear), edge to edge (linear), node to edge (edge layer), edge to node (node layer).

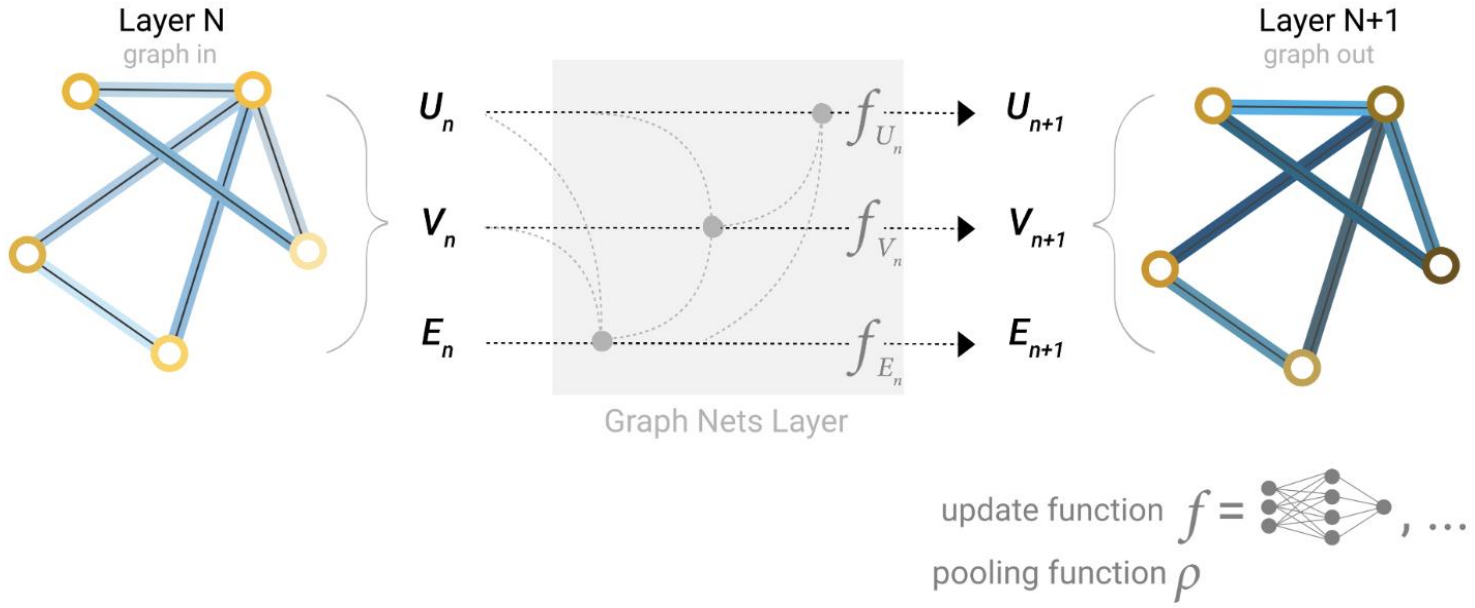




Some of the different ways we might combine edge and node representation in a GNN layer.

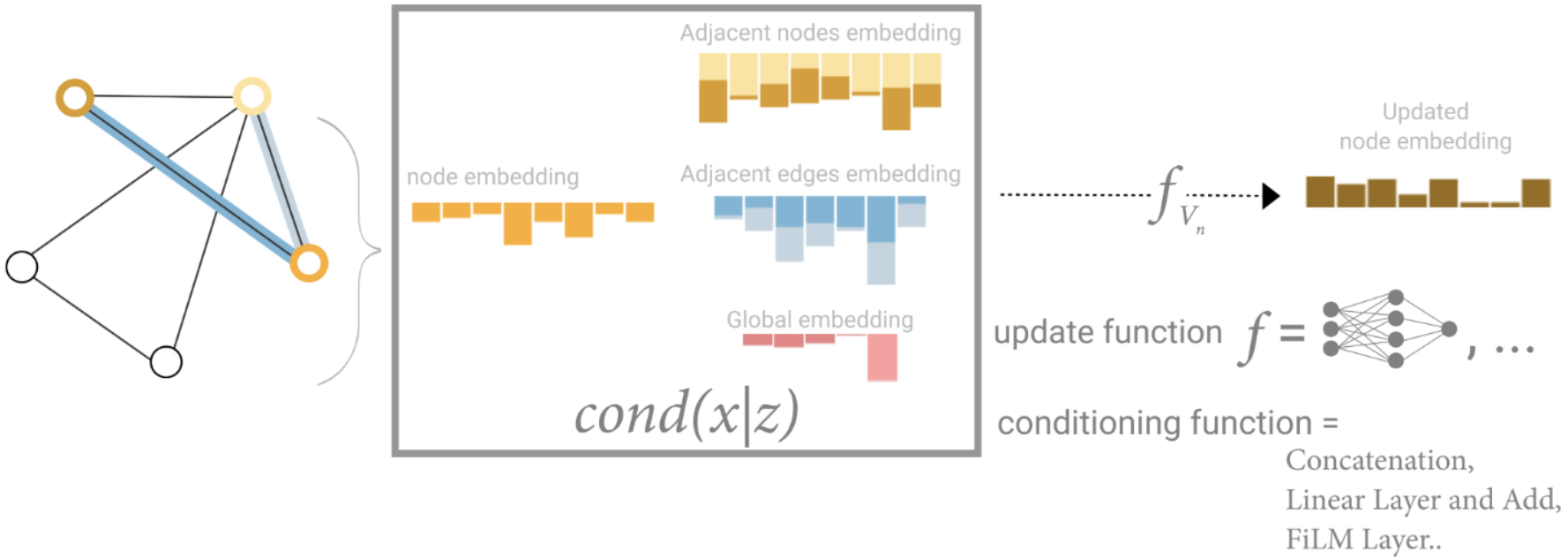
Adding global representations

There is one flaw with the networks we have described so far: nodes that are far away from each other in the graph may never be able to efficiently transfer information to one another, even if we apply message passing several times. For one node, If we have k-layers, information will propagate at most k-steps away. This can be a problem for situations where the prediction task depends on nodes, or groups of nodes, that are far apart. One solution would be to have all nodes be able to pass information to each other. Unfortunately for large graphs, this quickly becomes computationally expensive (although this approach, called 'virtual edges', has been used for small graphs such as molecules). One solution to this problem is by using the global representation of a graph (U) which is sometimes called a **master node** or context vector. This global context vector is connected to all other nodes and edges in the network, and can act as a bridge between them to pass information, building up a representation for the graph as a whole. This creates a richer and more complex representation of the graph than could have otherwise been learned.



Schematic of a Graph Nets architecture leveraging global representations.

- In this view all graph attributes have learned representations, so we can leverage them during pooling by conditioning the information of our attribute of interest with respect to the rest. For example, for one node we can consider information from neighboring nodes, connected edges and the global information. To condition the new node embedding on all these possible sources of information, we can simply concatenate them. Additionally, we may also map them to the same space via a linear map and add them or apply a feature-wise modulation layer, which can be considered a type of featurize-wise attention mechanism.

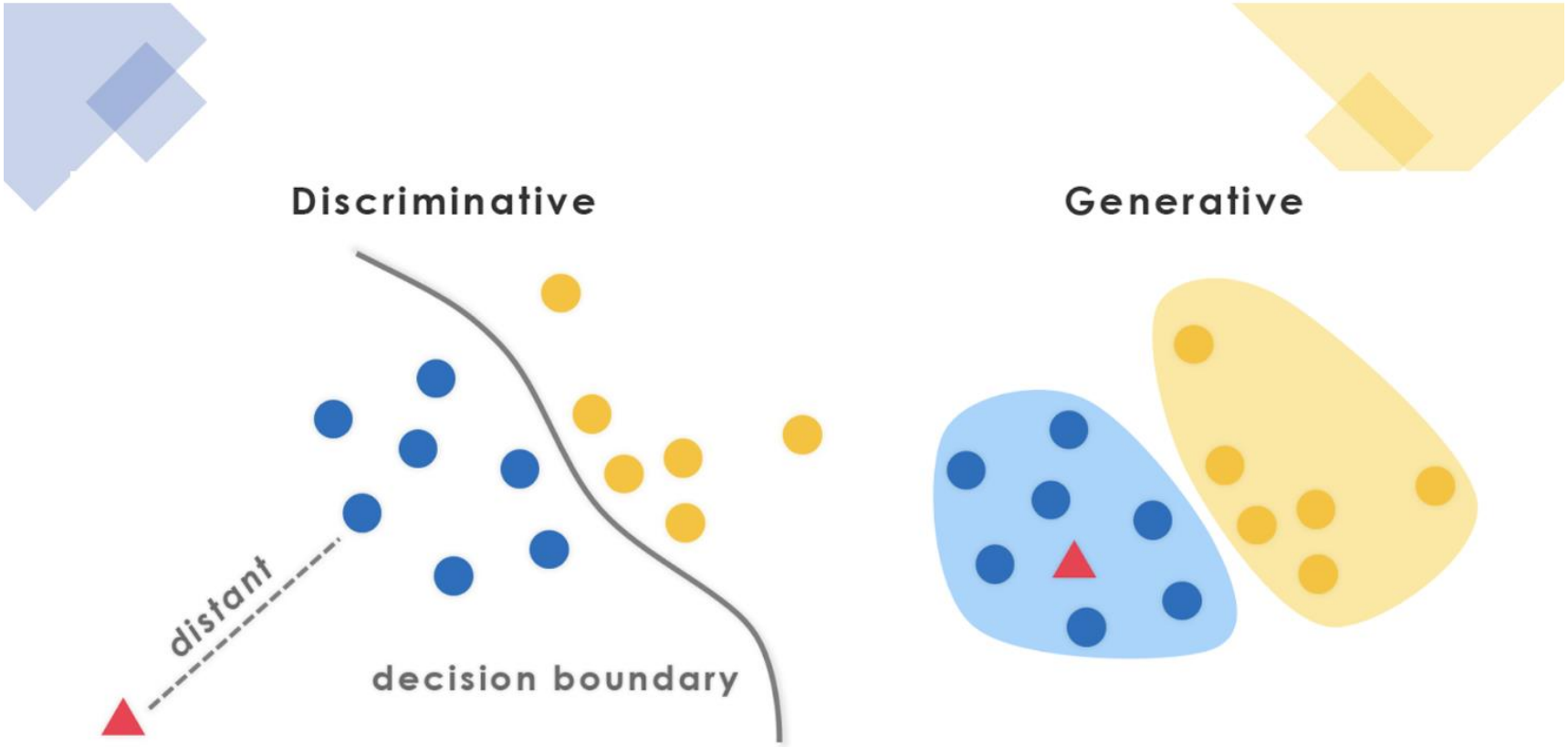


Schematic for conditioning the information of one node based on three other embeddings (adjacent nodes, adjacent edges, global). This step corresponds to the node operations in the Graph Nets Layer.

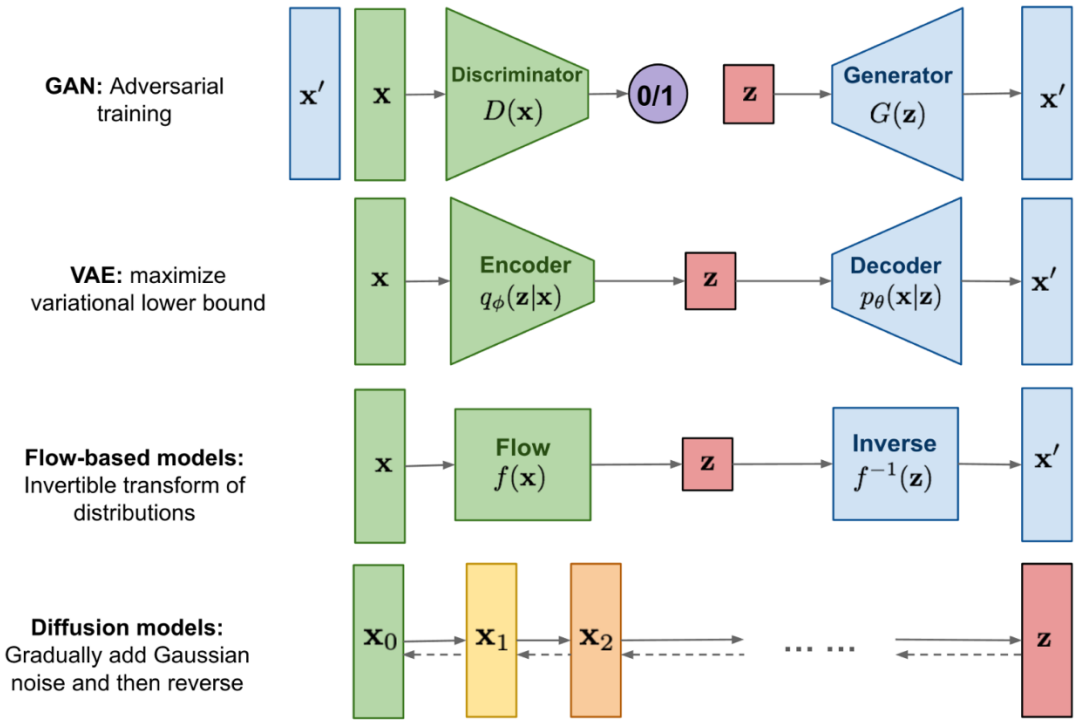
Unfortunately, GNN is a wide topic and I cannot cover everything here, So:
I will go through the topic in " Advanced Machine Learning Course" in details, which will be on Fall.

Generative Models

- **Generative** models can generate new data instances.
- **Discriminative** models discriminate between different kinds of data instances.



Generative Models



- **Diffusion models** will add noise to the data samples in the forward process and in the reverse process they will reconstruct the data samples from the noise.
- **Generative adversarial networks(GAN)** use Generator to generate the data samples and a Discriminator to check whether the generated data samples belong to the original dataset.
- **Variational autoencoders** use encoder-decoder architecture, In this encoder encodes the data samples into latent space and the decoder reconstructs the data samples from the latent space.
- **Flow-based models** use a sequence of inverse transformations to generate the image unlike other flow-based models and learn the probability distribution.

Linear Algebra Basics Recap

We should understand two key concepts before getting into the flow-based generative model: the Jacobian determinant and the change of variable rule. Pretty basic, so feel free to skip.

Jacobian Matrix and Determinant

Given a function of mapping a n-dimensional input vector $\mathbf{x}$ to a m-dimensional output vector, $\mathbf{f} : \mathbb{R}^n \mapsto \mathbb{R}^m$, the matrix of all first-order partial derivatives of this function is called the **Jacobian matrix**, $\mathbf{J}$ where one entry on the i-th row and j-th column is $\mathbf{J}_{ij} = \frac{\partial f_i}{\partial x_j}$.

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

The determinant is one real number computed as a function of all the elements in a squared matrix. Note that the determinant *only exists for **square** matrices*. The absolute value of the determinant can be thought of as a measure of *"how much multiplication by the matrix expands or contracts space"*.

The determinant of a nxn matrix M is:

$$\det M = \det \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} = \sum_{j_1 j_2 \dots j_n} (-1)^{\tau(j_1 j_2 \dots j_n)} a_{1 j_1} a_{2 j_2} \dots a_{n j_n}$$

where the subscript under the summation $j_1 j_2 \dots j_n$ are all permutations of the set {1, 2, ..., n}, so there are $n!$ items in total; $\tau(\,.\,)$ indicates the signature of a permutation.

The determinant of a square matrix M detects whether it is invertible: If $\det(M) = 0$ then M is not invertible (a *singular* matrix with linearly dependent rows or columns; or any row or column is all 0); otherwise, if $\det(M) \neq 0$, M is invertible.

The determinant of the product is equivalent to the product of the determinants: $\det(AB) = \det(A)\det(B)$. (proof)

Change of Variable Theorem

Let's review the change of variable theorem specifically in the context of probability density estimation, starting with a single variable case.

Given a random variable z and its known probability density function $z \sim \pi(z)$, we would like to construct a new random variable using a 1-1 mapping function $x = f(z)$. The function f is invertible, so $z = f^{-1}(x)$. Now the question is *how to infer the unknown probability density function of the new variable*, $p(x)$?

$$\int p(x)dx = \int \pi(z)dz = 1 \text{ ; Definition of probability distribution.}$$
$$p(x) = \pi(z)\left|\frac{dz}{dx}\right| = \pi(f^{-1}(x))\left|\frac{df^{-1}}{dx}\right| = \pi(f^{-1}(x))|(f^{-1})'(x)|$$

By definition, the integral $\int \pi(z)dz$ is the sum of an infinite number of rectangles of infinitesimal width Δz. The height of such a rectangle at position z is the value of the density function $\pi(z)$. When we substitute the variable, $z = f^{-1}(x)$ yields $\frac{\Delta z}{\Delta x} = (f^{-1}(x))'$ and $\Delta z = (f^{-1}(x))'\Delta x$. Here $|(f^{-1}(x))'|$ indicates the ratio between the area of rectangles defined in two different coordinate of variables z and x respectively.

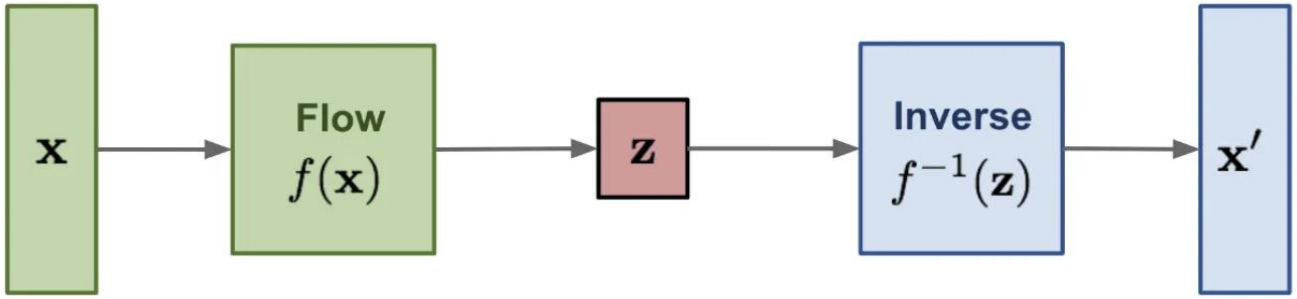
The multivariable version has a similar format:

$$\mathbf{z} \sim \pi(\mathbf{z}), \mathbf{x} = f(\mathbf{z}), \mathbf{z} = f^{-1}(\mathbf{x})$$
$$p(\mathbf{x}) = \pi(\mathbf{z})\left|\det \frac{d\mathbf{z}}{d\mathbf{x}}\right| = \pi(f^{-1}(\mathbf{x}))\left|\det \frac{df^{-1}}{d\mathbf{x}}\right|$$

where $\det \frac{\partial f}{\partial \mathbf{z}}$ is the Jacobian determinant of the function f. The full proof of the multivariate version is out of the scope of this post; ask Google if interested ;)

Flow-Based Models

- Normalizing flows were proposed to solve problems faced by GANs and VAE by using inverse transformation functions.
- Normalizing flows does not require putting noise to generate the images and the training process of Normalizing flows is much more stable than GAN.
- Normalizing flows models the true data distribution and provides us with the exact likelihood of the data hence the flow-based models use negative log-likelihood as the loss function.
- Flow-based generative models use invertible transformation functions to map data x to the latent representations z. Here z must be the same shape as x.
- A flow-based generative model is a generative model used in machine learning that explicitly models a probability distribution by leveraging normalizing flow,which is a statistical method using the change-of-variable law of probabilities to transform a simple distribution into a complex one.

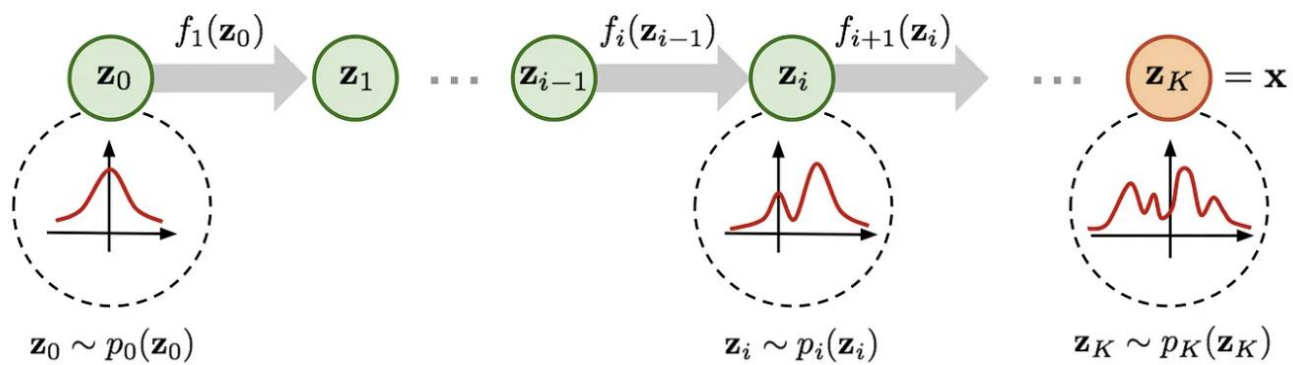


- Here invertible functions mean for every data point x, we have a corresponding latent representation z which allows us to perform lossless reconstruction (z to x).
- Suppose we have a prior density pz(z) (e.g. Gaussian) and an invertible function f, we can determine px(x) as follows:

$$\int p_x(x)dx = \int p_z(z)dz = 1 \qquad \text{(by definition of a probability distribution)}$$

$$\Leftrightarrow p_x(x) = p_z(z)\left|\frac{dz}{dx}\right| = p_z(f(x))\left|\frac{df(x)}{dx}\right|$$

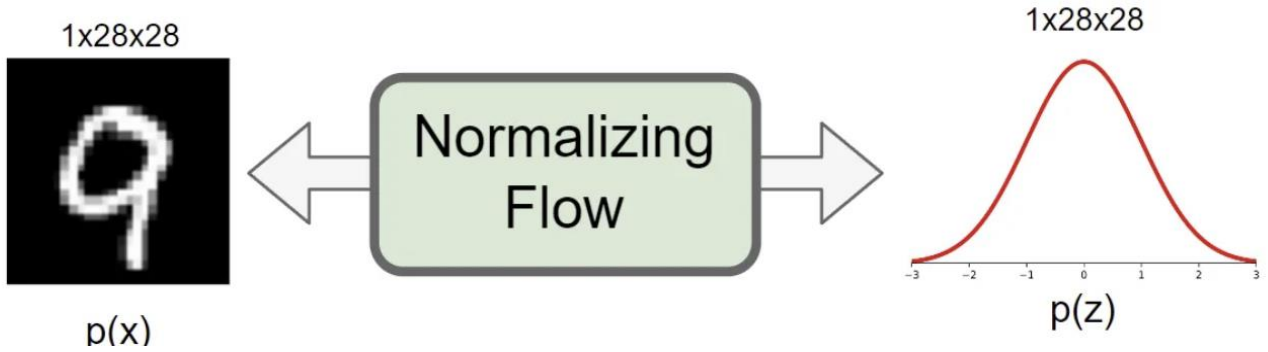
- Normalizing flow transforms a simple distribution into a complex one by applying a sequence of inverse transformation functions.



In the above starting from z0, which follows the prior Gaussian distribution, we sequentially apply the invertible functions f1,f2,…,fK, until zK represents x. The "**f** "should satisfy two conditions:

1. It is easily invertible.
2. Its Jacobian determinant is easy to compute.

During training, flow-based models map input images into latent space z as shown below and it generates the images by sampling the latent space z and applying that through inverse transformation functions f.



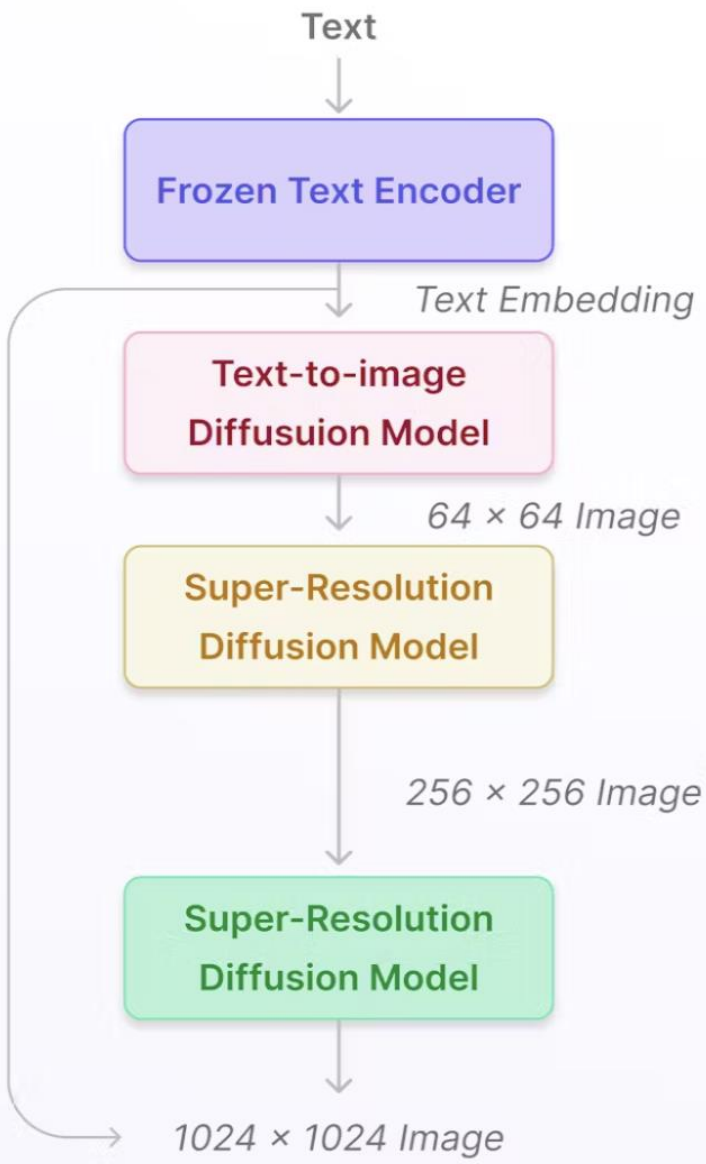
- Even though flow-based models use true data distribution to generate the data but training these models requires heavy computation power and also the data generated by flow-based models are not so clear compare to GANs, VAE, and diffusion models.

Advantages of flows Models

- Normalizing flows have a number of advantages over other types of probabilistic models, including:
- Flexibility: Normalizing flows can be used to learn a wide variety of probability distributions.
- Ease of training: Normalizing flows are relatively easy to train, compared to other types of probabilistic models, such as variational autoencoders.
- Simplicity: Normalizing flows are relatively simple to understand and implement.

Disadvantages of flows Models

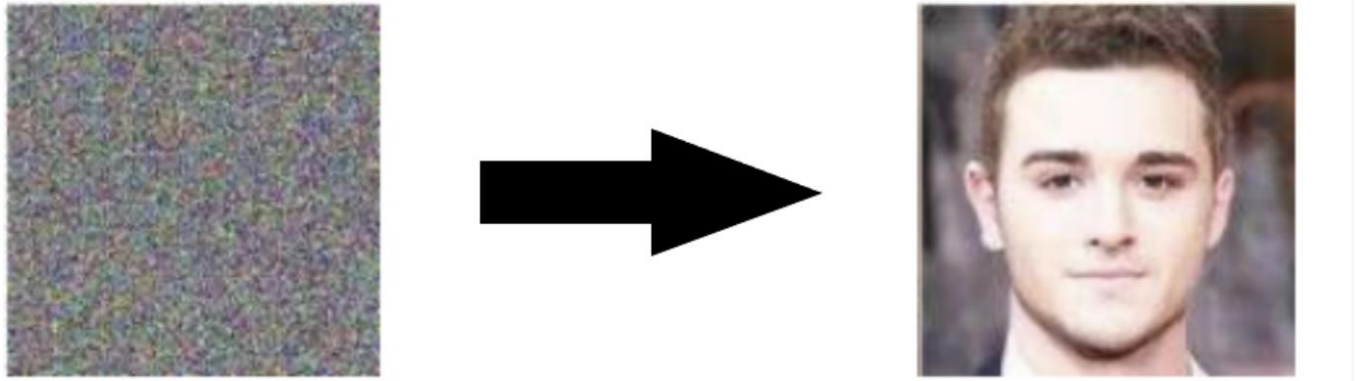
- Normalizing flows also have a few disadvantages, including:
- Computational complexity: Normalizing flows can be computationally expensive to train, especially for large data sets.
- Stability: Normalizing flows can be unstable to train, especially for complex distributions.
- Interpretability: Normalizing flows can be difficult to interpret, compared to other types of probabilistic models, such as Bayesian networks



"A Pig wearing a top hat and a pink bow tie."

Introduction to Diffusion Models

- In AI, diffusion models, also known as diffusion probabilistic models or score-based generative models, are a class of generative models.
- A diffusion model consists of three major components: the *forward process*, the *reverse process*, and the *sampling procedure*.
- The goal of diffusion models is to learn a diffusion process that generates a probability distribution for a given dataset from which we can then sample new images. They learn the latent structure of a dataset by modeling the way in which data points diffuse through their latent space.



Diffusion Models can be used to generate images from noise (adapted from source)

How diffusion models work?

- They work by gradually adding Gaussian noise to the original data in the forward diffusion process and then learning to remove the noise in the reverse diffusion process. They are latent variable models referring to a hidden continuous feature space and are loosely based on non-equilibrium thermodynamics.

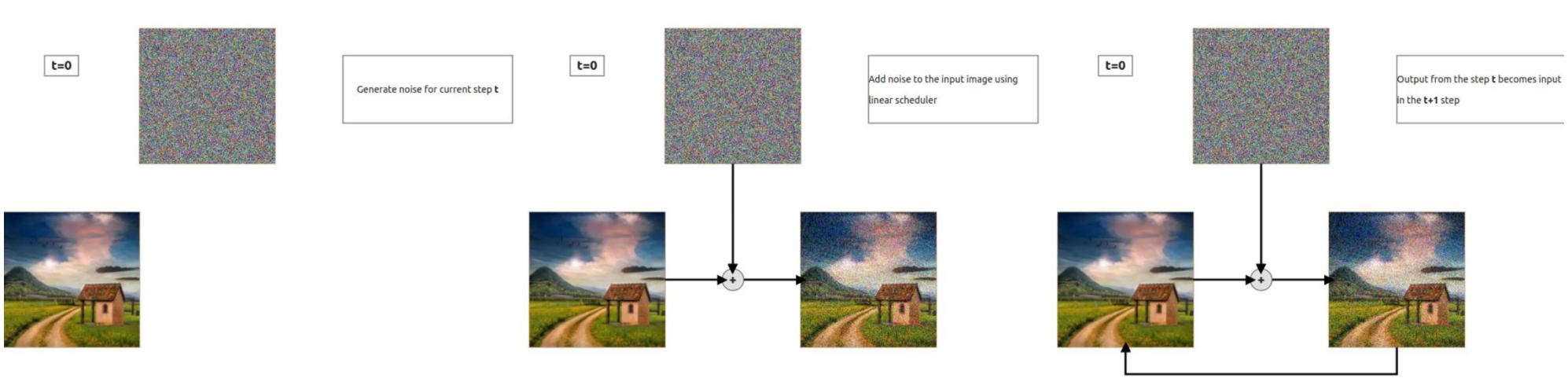
- A Diffusion Model consists of a forward process (or diffusion process), in which a datum (generally an image) is progressively noised, and a reverse process (or reverse diffusion process), in which noise is transformed back into a sample from the target distribution.

Data Pre-Processing

- Before the diffusion process begins, data needs to be appropriately formatted for model training.
- This process involves data cleaning to remove outliers, data normalization to scale features consistently, and data augmentation to increase dataset diversity, especially in the case of image data.
- Standardization is also applied to achieve normal data distribution, which is important for handling noisy image data. Different data types, such as text or images, may require specific preprocessing steps, like addressing class-imbalance issues.
- Well-executed data processing ensures high-quality training data and contributes to the model's ability to learn meaningful patterns and generate high-quality images (or other data types) during inference.

Introducing noise (Forward Diffusion Process)

- The forward diffusion process begins by sampling from a basic, usually Gaussian, distribution.
- This initial simple sample undergoes a series of reversible, incremental modifications, where each step introduces a controlled amount of complexity. It gradually layers on complexity, often visualized as the addition of structured noise.
- This diffusion of the initial data through successive transformations allows the model to capture and reproduce the complex patterns and details inherent in the target distribution.
- The ultimate goal of the forward diffusion process is to evolve these simple beginnings into samples that closely mimic the desired complex data distribution, showcasing how starting with minimal information can lead to rich, detailed outputs.

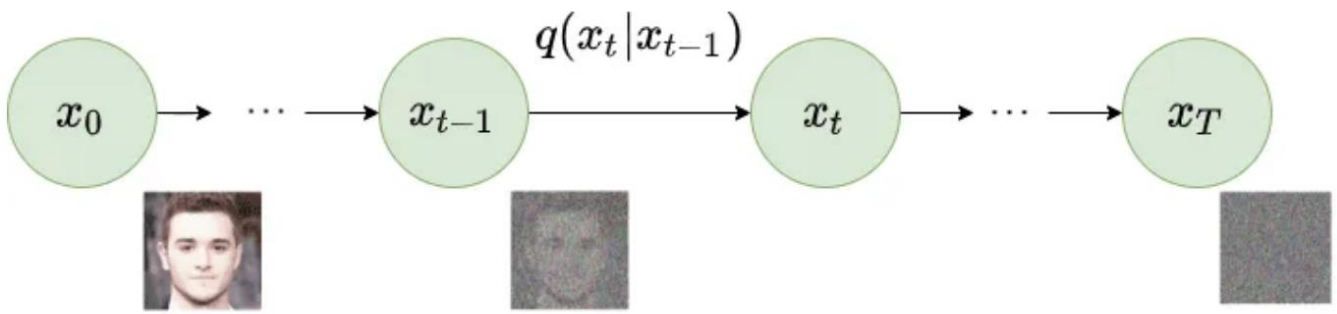


We start with sampling a data point x_0 from the real data distribution $q(x)$ like ($x_0 \sim q(x)$) and then adding some Gaussian noise with variance β_t to x_{t-1}, producing a new latent variable x_t with distribution $q(x_t|x_{t-1})$

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \quad q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1})$$

Here, $q(x_t|x_{t-1})$ is defined by the mean $\boldsymbol{\mu}$ as:

$$\boldsymbol{\mu}_t = \sqrt{1-\beta_t}\mathbf{x}_{t-1}$$



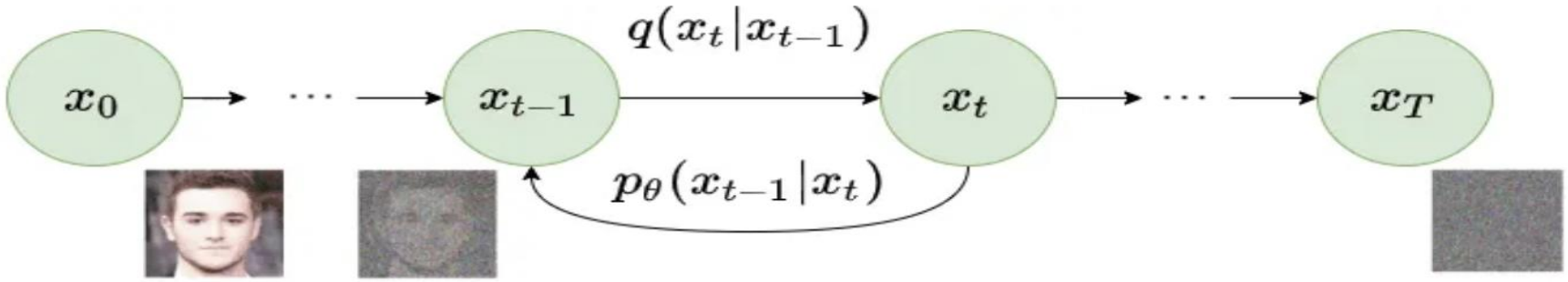
Reverse Diffusion Process

- This phase separates diffusion models from other generative models, like GANs. The reverse diffusion process involves recognizing the specific noise patterns introduced at each step and denoising the data accordingly.

- This isn't a simple process but rather involves complex reconstruction. Converting some random noise into a meaningful image is a complex task. The model uses its acquired knowledge to predict the noise at each step and then carefully removes it.

- It is the process of training a neural network to recover the original data by reversing the noising process applied in the forward pass. Estimating $q(x_{t-1}|x_t)$ is difficult as it can require the whole dataset. That's why a parameterized model p_θ(Neural Network) can be used to learn the parameters. For small enough β_t, it will be a Gaussian and can be obtained by just parameterizing the mean and variance.

$$p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T) \prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) \quad p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t))$$



Diffusion Models

- **Denoising Diffusion Probabilistic Models (DDPMs)**

DDPMs are a type of diffusion model that specializes in generating probabilistic data. They simulate a process to transform noisy data into clean samples. During training, they learn the parameters of this transformation process.
When generating data, they use their learned knowledge to produce denoised and highly realistic samples. DDPMs are especially effective in tasks like image denoising, inpainting, and super-resolution.

- **Score-Based Generative Models (SGMs)**

SGMs belong to the category of diffusion models that focus on generating new data based on patterns learned from existing data. They estimate data likelihood using a score function and can create entirely new samples following the same patterns as the original data. SGMs find applications in tasks such as deepfakes and generating additional data in scenarios where data is scarce.

- **Stochastic Differential Equations (Score SDEs)**

Score SDEs are mathematical equations employed in generative modeling to parameterize score-based models. They describe how systems change over time when subjected to deterministic and random forces.
Essentially, Score SDEs use stochastic processes to model unpredictable situations, proving valuable in addressing randomness in fields like physics and financial markets.

Denoising Diffusion Probabilistic Models (DDPMs)

- DDPMs are a type of diffusion model used for probabilistic data generation. As mentioned earlier, diffusion models generate data by applying a sequence of transformations to random noise. DDPMs, in particular, operate by simulating a diffusion process that transforms noisy data into clean data samples.
- Training DDPMs entails acquiring knowledge of the diffusion process's parameters, effectively capturing the relationship between clean and noisy data during each transformation step.
- During inference (generation), DDPMs start with noisy data (e.g., noisy images) and iteratively apply the learned transformations in reverse to obtain denoised and realistic data samples.

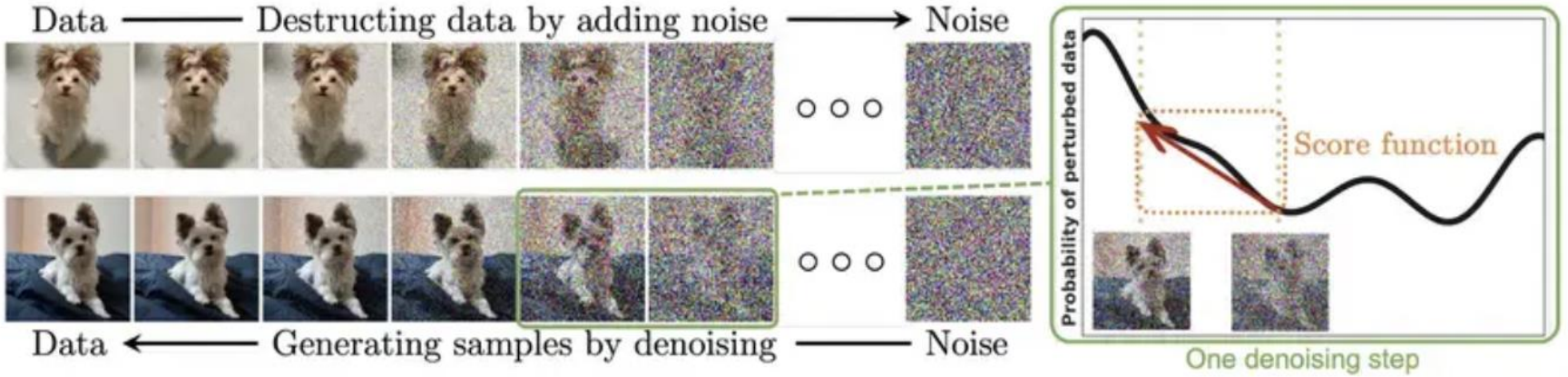


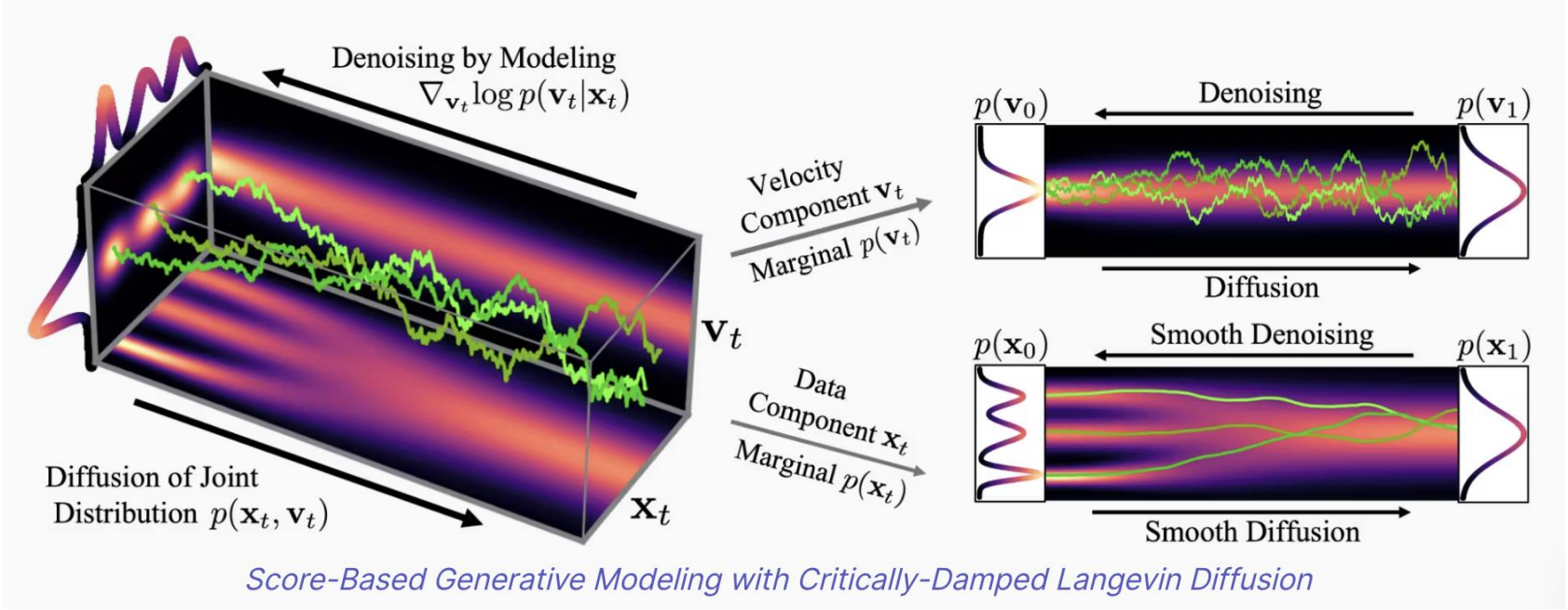
Fig. 2. Diffusion models smoothly perturb data by adding noise, then reverse this process to generate new data from noise. Each denoising step in the reverse process typically requires estimating the score function (see the illustrative figure on the right), which is a gradient pointing to the directions of data with higher likelihood and less noise.

- DDPMs are particularly effective for image-denoising tasks. They can effectively remove noise from corrupted images and produce visually appealing denoised versions. Moreover, DDPMs can also be used for image inpainting and super-resolution, among other applications.

Score-Based Generative Models (SGMs)

- Score-Based Generative Models are a class of generative models that use the score function to estimate the likelihood of data samples. The score function, also known as the gradient of the log-likelihood with respect to the data, provides essential information about the local structure of the data distribution.

- SGMs use the score function to estimate the data's probability density at any given point. This allows them to effectively model complex and high-dimensional data distributions. Although the score function can be computed analytically for some probability distributions, it is often estimated using automatic differentiation and neural networks.



Score-Based Generative Modeling with Critically-Damped Langevin Diffusion

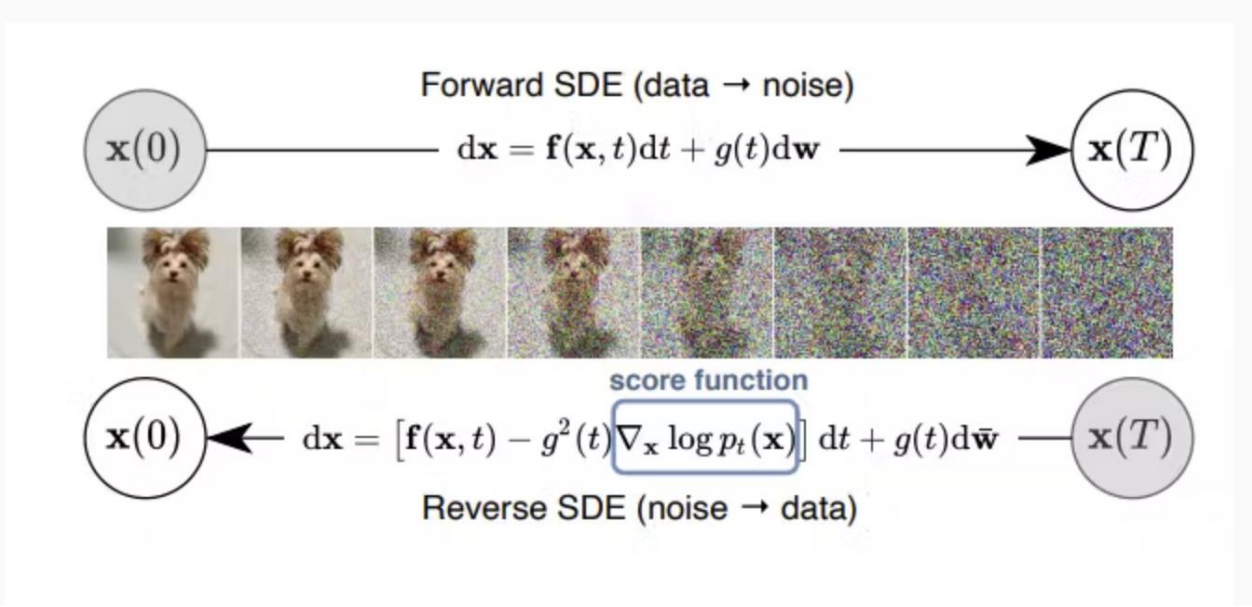
- Using the score function, SGMs can generate data samples that resemble the training data distribution. This is achieved by iteratively updating them toward the log-likelihood's negative gradient.

Stochastic Differential Equations (Score SDEs)

- Stochastic Differential Equations (SDEs) are mathematical equations describing how a system changes over time when subject to deterministic and random forces. In generative modeling, Score SDEs can parameterize the score-based models.

- In Score SDEs, the score function is a solution to a stochastic differential equation. By solving this differential equation, the model can learn a data-driven score function that adapts to the data distribution.

- In essence, Score SDEs use stochastic processes to model the evolution of data samples and guide the generative process toward generating high-quality data samples



- Score SDEs and score-based modeling can be combined to create powerful generative models capable of handling complex data distributions and generating diverse and realistic samples.

Please refer to Codes on Canvas


THANK YOU
Any Questions?