GNN: Graph Generation

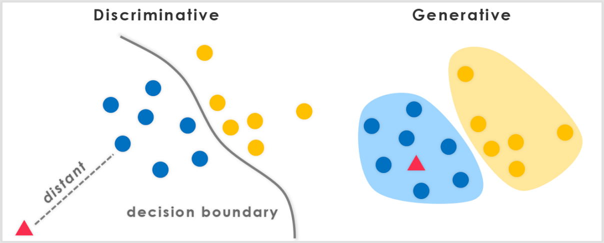
Harish Rajagopal

Seminar in Deep Neural Networks

11 May 2021

Generative Models

Generative models model the joint probability distribution $P(X, Y)$ over the input X and output Y.

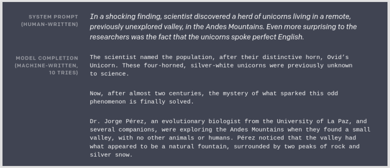


Source: https://duphan.wordpress.com/2016/10/27/gaussian-discriminant-analysis-and-logistic-regression/

Generative models can be used to **generate** examples.

Deep Generative Models

They are generative models that use deep learning, e.g. GPT-1/2/3, VAEs, GANs.



SYSTEM PROMPT (HUMAN-WRITTEN) *In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.*

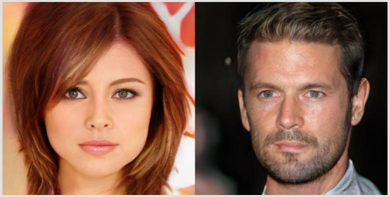
MODEL COMPLETION (MACHINE-WRITTEN, 10 TRIES)

```
The scientist named the population, after their distinctive horn, Ovid's
Unicorn. These four-horned, silver-white unicorns were previously unknown
to science.

Now, after almost two centuries, the mystery of what sparked this odd
phenomenon is finally solved.

Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and
several companions, were exploring the Andes Mountains when they found a small
valley, with no other animals or humans. Pérez noticed that the valley had
what appeared to be a natural fountain, surrounded by two peaks of rock and
silver snow.
```

(a) GPT2 [6] generates text from the given prompt



(b) Imaginary celebrities generated by Progressive GAN [3]

Graph Generation

Graphs are used to model data containing relations among distinct entities.

Graph generation aims to generate graphs with some desired properties.

Source:
https://news.mit.edu/2013/new-approach-to-vertex-connectivity-could-maximize-networks-bandwidth-1224

Example: Molecule Generation

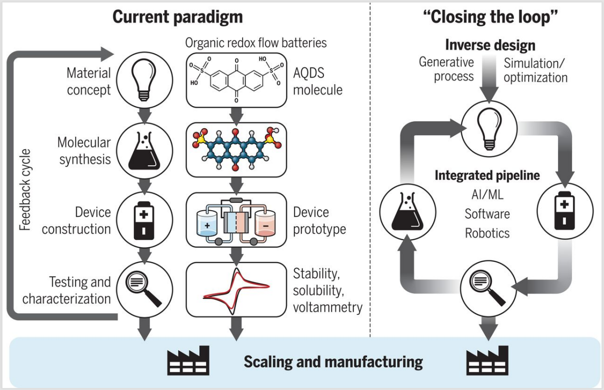


Figure: Schematic comparison of material discovery paradigms [7]

Challenges

- **Discreteness**: Graphs are discrete structures
- **Variability**: Graphs can be of different sizes
- **Ordering**: Graph nodes and edges are unordered

Approaches

Working with graphs as adjacency matrices helps tackle the **discreteness** problem.

This leads to two popular classes of deep graph generators:

- **Single-Shot**: Outputs the entire adjacency matrix at once
- **Autoregressive**: Sequentially outputs each row of the adjacency matrix

Single-Shot Models

Single-Shot Models

These tackle the **variability** problem by fixing a maximum size for the graph. Then they prune the adjacency matrix.

Different models tackle the **ordering** problem in different ways.

We study the following single-shot models:

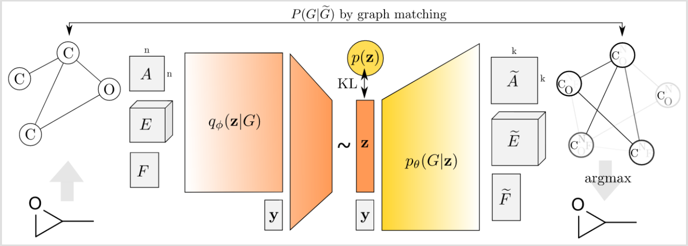
- GraphVAE [8]
- MolGAN [1]

Single-Shot Models

GraphVAE

GraphVAE

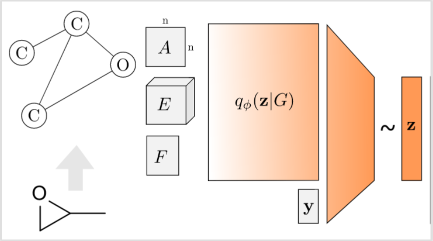
GraphVAE uses a Variational Autoencoder [4] (VAE) setup.



GraphVAE Encoder

It is a GNN that takes $G = (A, E, F)$ and the graph properties y.

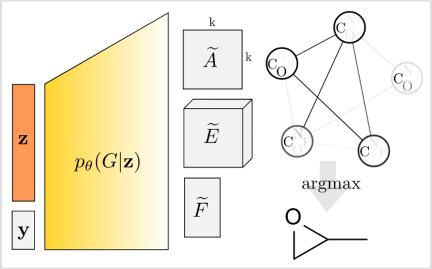
It models $q_\phi(z|G)$ for the latent vector z.



GRAPHVAE DECODER

It is an MLP that takes a latent vector z and the graph properties y.

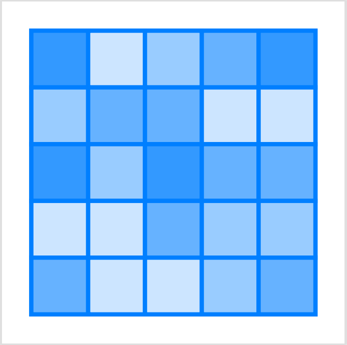
It models $p_\theta(G|z)$ with the **probabilistic adjacency matrix** $\tilde{A}$ and the class probabilities $\tilde{E}$, $\tilde{F}$.



The Probabilistic Adjacency Matrix

The PAM $\tilde{A}$ is of size $k \times k$, where k is the maximum graph size.

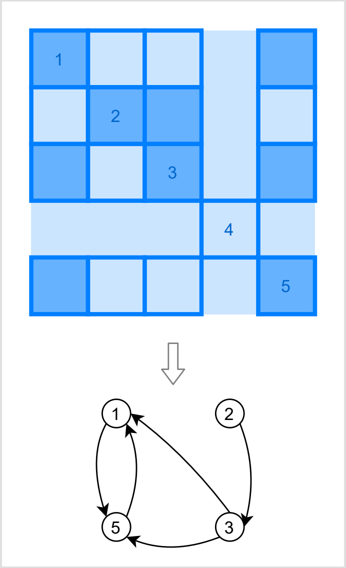
Each element is a sigmoid probability, which is **thresholded** during inference.



The Probabilistic Adjacency Matrix

The diagonal element $\tilde{A}_{ii}$ shows whether to keep node i.

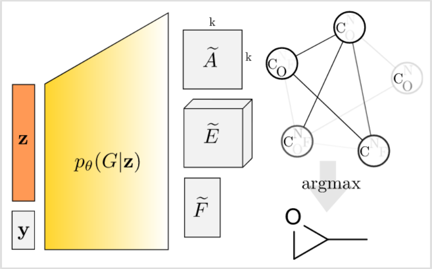
The off-diagonal element $\tilde{A}_{ij}$ shows whether to keep edge $i \to j$.



Graph Generation

During training, the decoder will be fed $z \sim q_\phi(z|G)$.

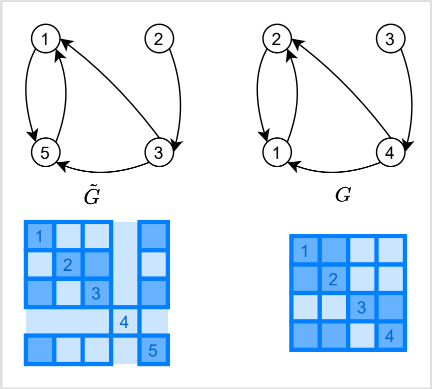
During inference, it will be fed $z \sim \mathcal{N}(0, I)$.



The Ordering Problem

The decoder loss consists of the cross-entropy loss for $\tilde{A}$, $\tilde{E}$ and $\tilde{F}$.

However, due to the **ordering** problem, the node orders between $\tilde{G}$ and G can differ.

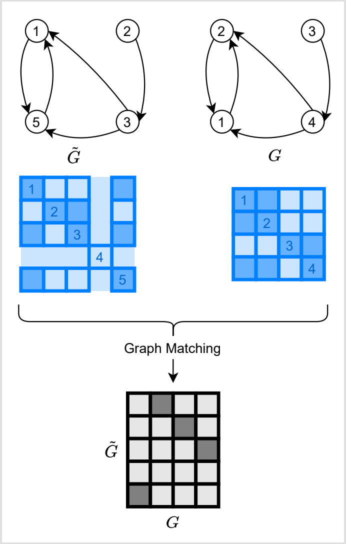


Approximate Graph Matching

Approximate graph matching is used to **assign** nodes from $\tilde{G}$ to nodes in G.

This gives us $X \in \{0, 1\}^{k \times n}$, where $X_{ij} = 1$ iff node $i \in \tilde{G}$ is assigned to node $j \in G$.

However, it is very **slow**.



GraphVAE Decoder Loss

The cross-entropy losses are now calculated for the following:

$$
\begin{aligned}
{A'}_{k\times k} &= XAX^{\mathsf{T}} - \tilde{A}_{k\times k} \\
E_{n\times n} - \tilde{E}'_{n\times n} &= X^{\mathsf{T}}\tilde{E}X \\
F_n - \tilde{F}'_n &= X^{\mathsf{T}}\tilde{F}
\end{aligned}
$$

The final decoder loss is a weighted sum of these loss terms.

GraphVAE Results

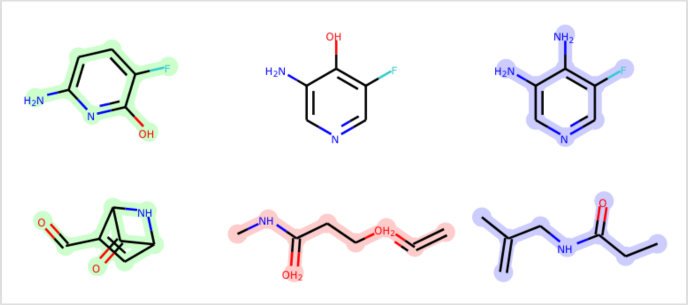


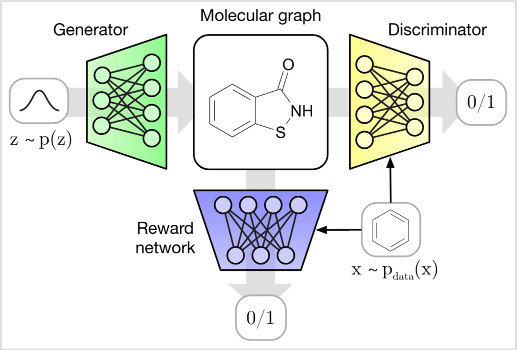
Figure: GraphVAE inputs (in green) and outputs

Single-Shot Models

MolGAN

MolGAN

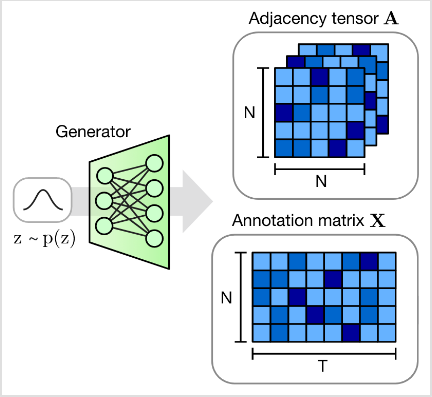
MolGAN uses a Generative Adversarial Network [2] (GAN) setup.



MolGAN Generator

It is an MLP that takes a latent vector $z \sim \mathcal{N}(0, 1)$.

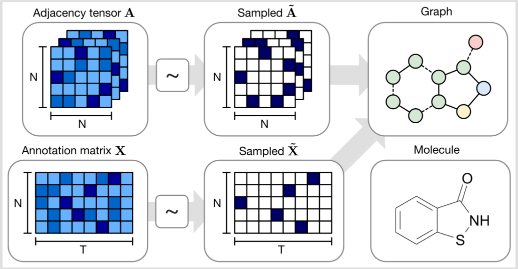
It generates the PAM A and the node attributes X.



The Probabilistic Adjacency Matrix

MolGAN tackles the **variability** problem using the PAM.

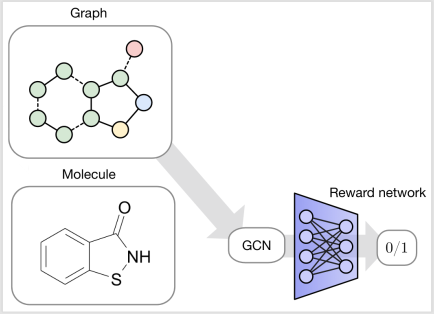
During inference, instead of pruning the PAM, they **sample** from the probabilities.


Adjacency tensor $\mathbf{A}$
N
N
~
Sampled $\tilde{\mathbf{A}}$
N
N
Graph
Annotation matrix $\mathbf{X}$
N
T
~
Sampled $\tilde{\mathbf{X}}$
N
T
Molecule
O
NH
S

MolGAN Reward Network

It is a GNN that takes the PAM A and the node attributes X.

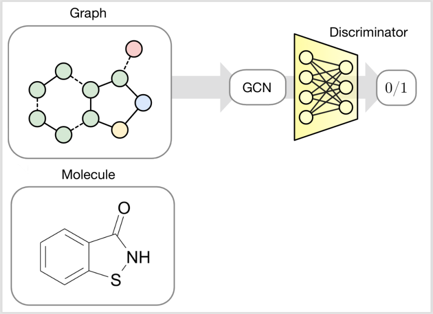
It returns the **reward** for the input molecule's properties.



MolGAN Discriminator

It is another GNN that takes the PAM A and the node attributes X.

It predicts whether its inputs are from the dataset or generated by the generator.



The GAN in MolGAN

The generator aims to fool the discriminator, while the discriminator aims to catch the generator.

$$\mathcal{L}_{GAN}(\mathrm{Disc}(\boldsymbol{G}), \mathrm{Disc}(\mathrm{Gen}(\boldsymbol{z})))$$

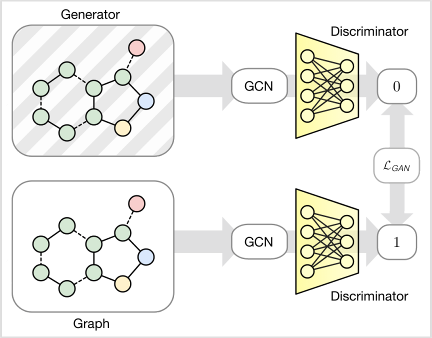
The generator aims to minimize the GAN loss, while the discriminator aims to maximize it.

The GAN Loss

The GAN loss is:

$$\mathcal{L}_{GAN}(\mathrm{Disc}(G), \mathrm{Disc}(\mathrm{Gen}(z)))$$

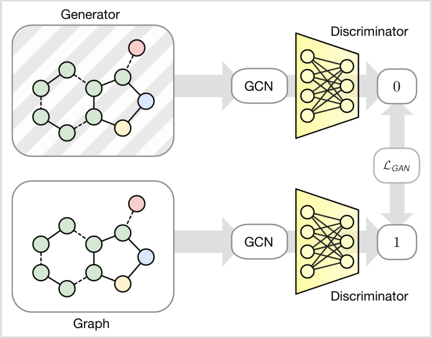
The generator's outputs must pass **through the discriminator** before interacting with the ground-truth.



The Ordering Problem

Since the discriminator is a GNN, it is invariant to node ordering.

Thus, the **ordering** problem does not affect MolGAN.



MolGAN Results

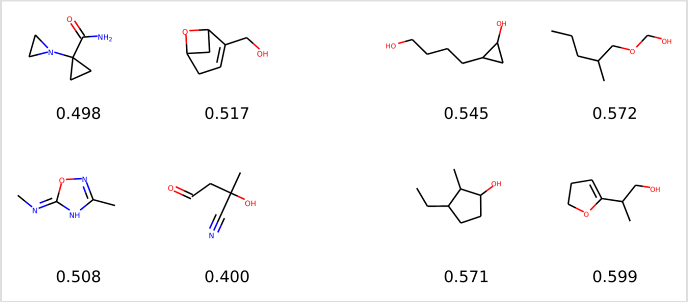


Figure: QM9 samples vs MolGAN outputs

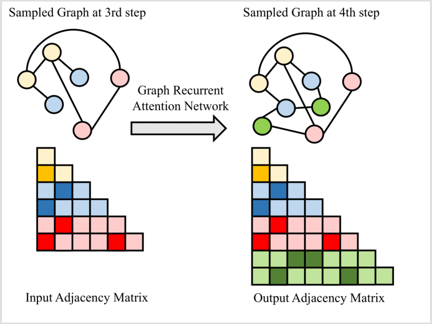
GraphVAE vs MolGAN

	GraphVAE	**MolGAN**
Architecture	Encoder-decoder	Generator-discriminator
PAM	Thresholding	Sampling
Graph-matching	Required, expensive	None
Convergence	VAEs are easier to train	GANs are hard to train

Autoregressive Models

Autoregressive Models

Autoregressive models tackle the **variability** problem by generating the rows of the adjacency matrix **sequentially**.


Sampled Graph at 3rd step
Sampled Graph at 4th step
Graph Recurrent Attention Network
Input Adjacency Matrix
Output Adjacency Matrix

They can decide to stop generating by outputting a special token.

Autoregressive Models

These models usually deal with the **ordering** problem by considering **all** node orders from a set of canonical orderings.

We study the following autoregressive models:

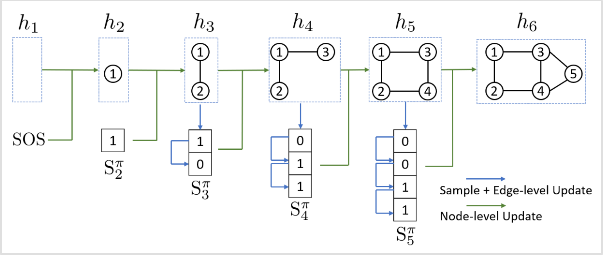
- GraphRNN [9]
- GRAN [5]

Autoregressive Models

GraphRNN

GraphRNN

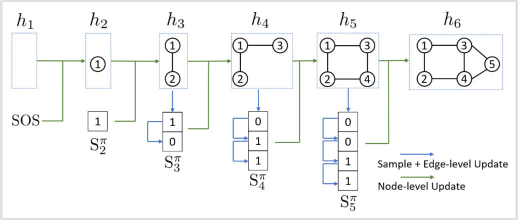
GraphRNN uses Gated Recurrent Units (GRUs) in a hierarchical setup.



Graph-level RNN

Let the sequence of rows of the adjacency matrix be S^π.

A **graph-level** RNN generates nodes by modelling $p(S^\pi_i | S^\pi_{<i})$.



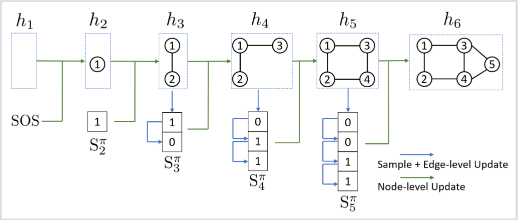
These variable-length sequences help solve the **variability** problem.

Edge-level RNN

To capture complex edge dependencies, $p(S_i^\pi | S_{<i}^\pi)$ is decomposed as:

$$p(S_i^\pi | S_{<i}^\pi) = \prod_{j=1}^{i-1} p(S_{i,j}^\pi | S_{i,<j}^\pi, S_{<i}^\pi)$$

This is done using an **edge-level** RNN to generate edges of each node.



GraphRNN Orderings

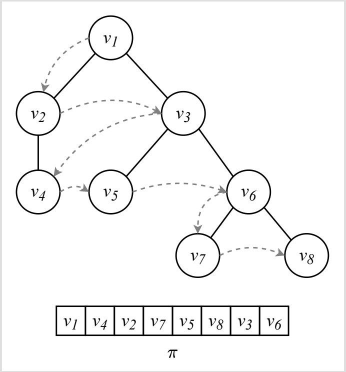
GraphRNN is optimized using SGD to maximize $p(G)$:

$$p(G) = \sum_{\pi \in \Pi} p(S^{\pi})$$

Π is the set of **all** orderings. Thus, it solves the **ordering** problem.

However, $|\Pi| = \mathcal{O}(N!)$. Hence, GraphRNN **restricts** it to a set of canonical orderings based on BFS.

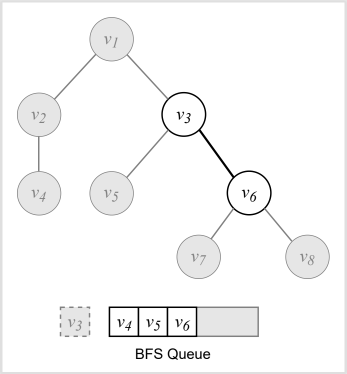
BFS Canonical Orderings



Multiple node orderings can map to the same BFS ordering.

Considering only **unique** BFS orderings, $|\Pi_{BFS}|$ can drop substantially.

More Benefits of BFS

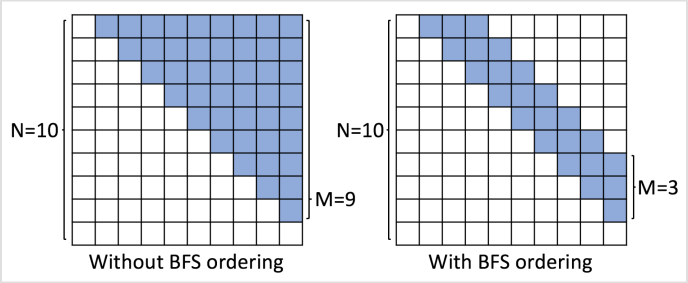


v_6 will be added to the BFS queue **just after** v_3 is removed.

Thus, the gap between v_3 and v_6 in the BFS order **cannot exceed** the max size of the BFS queue.

More Benefits of BFS

If we know the max size M of the BFS queue, then the edge-level RNN can **skip** $(0, \ldots, i - M - 1)$.



GraphRNN Results

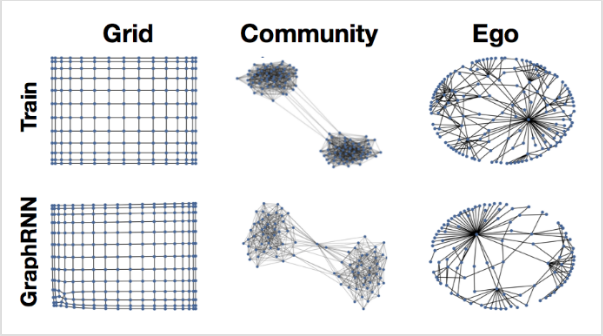


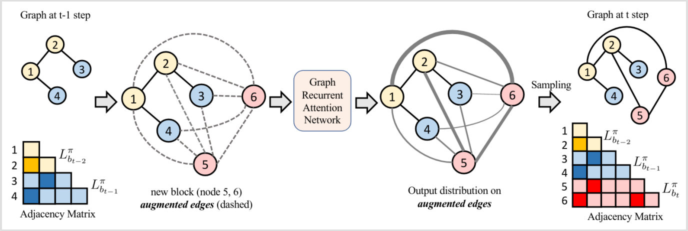
Figure: GraphRNN results on various datasets

Autoregressive Models

GRAN

GRAN

Graph Recurrent Attention Networks (GRANs) are a family of RNN-based models with attention.



GRAN Canonical Orderings

GRANs use the same loss and setup as GraphRNN:

$$p(G) = \sum_{\pi \in \mathcal{Q}} p(L^\pi) \geq \sum_{\pi \in \tilde{\mathcal{Q}}} p(L^\pi)$$

$$\text{where } \tilde{\mathcal{Q}} \subseteq \mathcal{Q}$$

However, instead of using BFS for $\tilde{\mathcal{Q}}$, they use a combination of various techniques.

GRAN Architecture

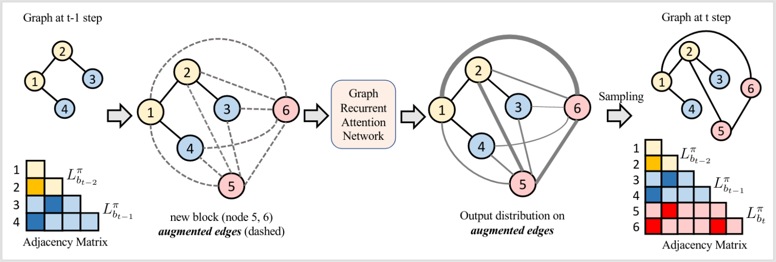
Downsides of hierarchical RNNs:

- RNNs suffer from vanishing gradients.
- Each graph-level RNN step cannot be run in parallel.

Thus, GRANs use a **GNN** at the graph-level to generate edges.

GNN Setup

The GNN uses the graph generated in the previous step to generate *B* new nodes.



Graph at t-1 step
Adjacency Matrix
$L^{\pi}_{b_{t-2}}$
$L^{\pi}_{b_{t-1}}$
new block (node 5, 6) ***augmented edges*** (dashed)
Graph Recurrent Attention Network
Output distribution on ***augmented edges***
Sampling
Graph at t step
$L^{\pi}_{b_t}$
Adjacency Matrix

GNN Initialization

The initial node representations of the GNN are:

$$h_i^0 = \begin{cases} WL_i^{\pi} + b & i \leq B(t-1) \\ 0 & \text{otherwise} \end{cases}$$

Here, $L_i^{\pi} \in \mathbb{R}^N$, where N is the **maximum** size of the graph.

GNN Updates

The GNN update step uses a GRU (**RNN**) cell:

$$h_i^{r+1} = \mathrm{GRU}(h_i^r, \sum_{j \in \mathcal{N}(i)} a_{ij}^r m_{ij}^r)$$

Here, m_{ij}^r's are a transformation of (h_i^r, h_j^r), while a_{ij}^r's are **attention** weights.

GNN Output

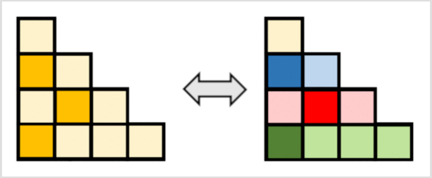
After R message-passing rounds, $p(L^{\pi}_{b_t} | L^{\pi}_{b_{<t}})$ is modelled as a **mixture model**:

$$p(L^{\pi}_{b_t} | L^{\pi}_{b_{<t}}) = \sum_{k=1}^{K} \alpha_k \prod_{i \in b_t} \prod_{1 \leq j \leq i} \theta_{kij}$$

Here, θ^r_{kij}'s are another transformation of (h^r_i, h^r_j), while α_k's are mixture probabilities.

Effect of B

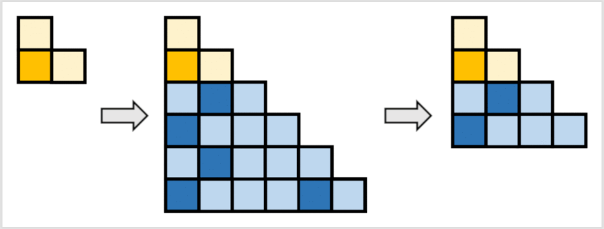
A higher value of B improves generation speed, while a lower value of B improves accuracy.



Thus, the authors propose “strided sampling” to balance these.

Strided Sampling

After generating B rows, they **only keep** the first S rows. The next block is generated from the $(S+1)$-th row.



However, during training, they fix $S = 1$.

GRAN Results

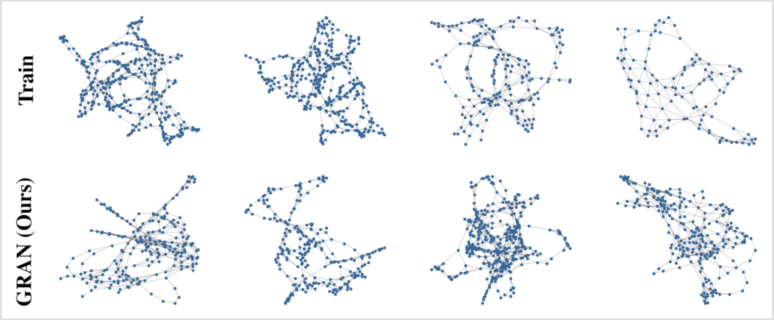


Figure: GRAN for Protein Graphs

GraphRNN vs GRAN

	GraphRNN	**GRAN**
Architecture	Hierarchical RNNs	GNN with attention and RNN updates
Edge Updates	Single-row updates	Strided sampling
Graph Size	Variable	Fixed maximum size
Ordering	BFS-based	Mixture of orderings

Summary

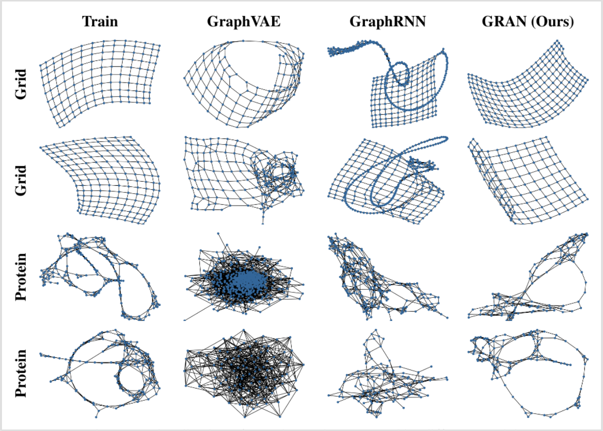
Summary

- Major challenges — discreteness, variability, & ordering
- Working with the adjacency matrix — tackles discreteness
- Two popular approaches — single-shot & autoregressive

Approach Comparison

	Single-Shot	Autoregressive
Variability	PAM Quantization	Sequential generation
Ordering	Varies	Canonical Orderings
Graph Size	Fixed maximum size	Usually variable
Speed	High	Low

Model Comparison



Thank You!

References I

[1] Nicola De Cao and Thomas Kipf.
Molgan: An implicit generative model for small molecular graphs, 2018.

[2] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio.
Generative adversarial networks, 2014.

[3] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen.
Progressive growing of gans for improved quality, stability, and variation, 2018.

[4] Diederik P Kingma and Max Welling.
Auto-encoding variational bayes, 2014.

[5] Renjie Liao, Yujia Li, Yang Song, Shenlong Wang, Charlie Nash, William L. Hamilton, David Duvenaud, Raquel Urtasun, and Richard S. Zemel.
Efficient graph generation with graph recurrent attention networks, 2020.

References II

[6] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever.
Language models are unsupervised multitask learners.
2019.

[7] Benjamin Sanchez-Lengeling and Alán Aspuru-Guzik.
Inverse molecular design using machine learning: Generative models for matter engineering.
Science, 361(6400):360–365, 2018.

[8] Martin Simonovsky and Nikos Komodakis.
Graphvae: Towards generation of small graphs using variational autoencoders, 2018.

[9] Jiaxuan You, Rex Ying, Xiang Ren, William L. Hamilton, and Jure Leskovec.
Graphrnn: Generating realistic graphs with deep auto-regressive models, 2018.