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

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.

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

(b) Imaginary celebrities generated by Progressive GAN [3]
Graphs are used to model data containing relations among distinct entities.

Graph generation aims to generate graphs with some desired properties.

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

Each element is a sigmoid probability, which is \textit{thresholded} during inference.
The diagonal element $\tilde{A}_{ii}$ shows whether to keep node $i$.

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

\[ A'_{k \times k} = XAX^T - \tilde{A}_{k \times k} \]
\[ E_{n \times n} - \tilde{E}'_{n \times n} = X^T \tilde{E}X \]
\[ F_n - \tilde{F}'_n = X^T \tilde{F} \]

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

It generates the PAM $A$ and the node attributes $X$. 
MolGAN tackles the **variability** problem using the PAM.

During inference, instead of pruning the PAM, they **sample** from the probabilities.
It is a GNN that takes the PAM $A$ and the node attributes $X$.

It returns the **reward** for the input molecule’s properties.
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 generator aims to fool the discriminator, while the discriminator aims to catch the generator.

\[ \mathcal{L}_{GAN}(\text{Disc}(G), \text{Disc}(\text{Gen}(z))) \]

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

\[ \mathcal{L}_{GAN}(\text{Disc}(G), \text{Disc}(\text{Gen}(z))) \]

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

<table>
<thead>
<tr>
<th></th>
<th>GraphVAE</th>
<th>MolGAN</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Architecture</strong></td>
<td>Encoder-decoder</td>
<td>Generator-discriminator</td>
</tr>
<tr>
<td><strong>PAM</strong></td>
<td>Thresholding</td>
<td>Sampling</td>
</tr>
<tr>
<td><strong>Graph-matching</strong></td>
<td>Required, expensive</td>
<td>None</td>
</tr>
<tr>
<td><strong>Convergence</strong></td>
<td>VAEs are easier to train</td>
<td>GANs are hard to train</td>
</tr>
</tbody>
</table>
Autoregressive Models
Autoregressive models tackle the **variability** problem by generating the rows of the adjacency matrix **sequentially**.

They can decide to stop generating by outputting a special token.
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 uses Gated Recurrent Units (GRUs) in a hierarchical setup.
Let the sequence of rows of the adjacency matrix be $S^\pi$. A **graph-level** RNN generates nodes by modelling $p(S^\pi_i | S^\pi_{<i})$.

These variable-length sequences help solve the **variability** problem.
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 is optimized using SGD to maximize $p(G)$:

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

$\Pi$ is the set of all orderings. Thus, it solves the ordering problem.

However, $|\Pi| = O(N!)$. Hence, GraphRNN restricts it to a set of canonical orderings based on BFS.
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
Graph Recurrent Attention Networks (GRANs) are a family of RNN-based models with attention.
GRANs use the same loss and setup as GraphRNN:

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

where \( \tilde{Q} \subseteq Q \)

However, instead of using BFS for \( \tilde{Q} \), 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.
The initial node representations of the GNN are:

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

Here, \( L_i^\pi \in \mathbb{R}^N \), where \( N \) is the maximum size of the graph.
The GNN update step uses a GRU (RNN) cell:

\[ h_{i}^{r+1} = \text{GRU}(h_{i}^{r}, \sum_{j \in \mathcal{N}(i)} a_{ij}^{r} m_{ij}^{r}) \]

Here, \( m_{ij}^{r} \)'s are a transformation of \( (h_{i}^{r}, h_{j}^{r}) \), while \( a_{ij}^{r} \)'s are attention weights.
After $R$ message-passing rounds, $p(L_{b_t}^{\pi} | L_{b<t}^{\pi})$ is modelled as a mixture model:

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

Here, $\theta_{kij}$’s are another transformation of $(h_i^r, h_j^r)$, while $\alpha_k$’s are mixture probabilities.
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.
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 $S = 1$. 
Figure: GRAN for Protein Graphs
<table>
<thead>
<tr>
<th></th>
<th>GraphRNN</th>
<th>GRAN</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Architecture</strong></td>
<td>Hierarchical RNNs</td>
<td>GNN with attention and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RNN updates</td>
</tr>
<tr>
<td><strong>Edge Updates</strong></td>
<td>Single-row updates</td>
<td>Strided sampling</td>
</tr>
<tr>
<td><strong>Graph Size</strong></td>
<td>Variable</td>
<td>Fixed maximum size</td>
</tr>
<tr>
<td><strong>Ordering</strong></td>
<td>BFS-based</td>
<td>Mixture of orderings</td>
</tr>
</tbody>
</table>
SUMMARY

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

<table>
<thead>
<tr>
<th></th>
<th>Single-Shot</th>
<th>Autoregressive</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variability</strong></td>
<td>PAM Quantization</td>
<td>Sequential generation</td>
</tr>
<tr>
<td><strong>Ordering</strong></td>
<td>Varies</td>
<td>Canonical Orderings</td>
</tr>
<tr>
<td><strong>Graph Size</strong></td>
<td>Fixed maximum size</td>
<td>Usually variable</td>
</tr>
<tr>
<td><strong>Speed</strong></td>
<td>High</td>
<td>Low</td>
</tr>
</tbody>
</table>
Model Comparison

Grid

Train

GraphVAE

GraphRNN

GRAN (Ours)

Protein
Thank You!


