Graph Neural Networks: Graph Generation Part II

Deep Graph Generative Models Continued

Christopher Risi

University of Waterloo

03 04 2024

Topics

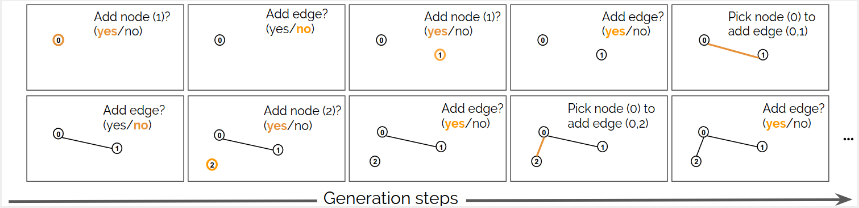
1. Deep Autoregressive Methods
    - GNN-based Autoregressive Model
    - Graph Recurrent Neural Networks (GraphRNN)
    - Graph Recurrent Attention Networks (GRAN)
2. Generative Adversarial Methods
    - Adjacency Matrix Based GAN
    - Random Walk Based GAN

Section 1:
Deep Autoregressive Methods

Deep Autoregressive Methods

The shared underlying idea of these autoregressive models is to characterize the graph generation process as:

- a sequential decision-making process
- make a new decision at each step, conditioning on all previously made decisions.



Conditional RNN - Background

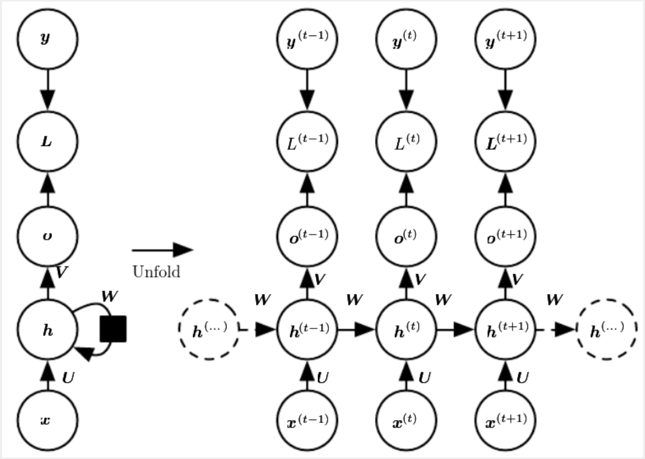
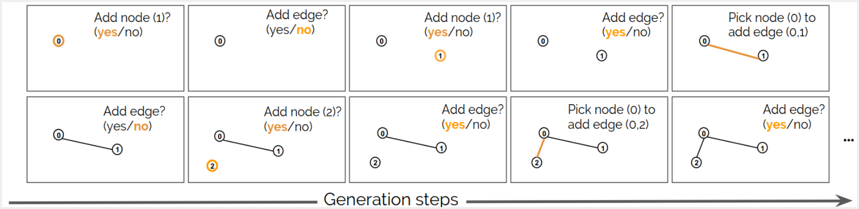


Figure: Figure 10.3 from Goodfellow et al. 2016 [**?**]. A recurrent neural network (RNN).

Autoregressive Decision Process

The graph generation is formulated as a sequential decision-making process.



At each step of the generation, the model needs to decide:

1. $p_{AddNode}$: add a new node or stop generation
2. $p_{AddEdge}$: add an edge that links any existing node to the new node.
3. p_{nodes}: choose an existing node to link to the newly added node.

GNN-based Autoregressive Model II

Message Passing Graph Neural Networks

- A partial graph $\mathcal{G}^{t-1} = (\mathcal{V}^{t-1}, \mathcal{E}^{t-1})$
- Adjacency matrix and feature node matrix: (A^{t-1}, X^{t-1})
- Input: At time step $t-1$ is (A^{t-1}, H^{t-1})
  - H^{t-1} is the node representation.
  - At $t = 0$ we probabilistically generate the initial node $p_{AddNode}$ based on randomly initialized hidden state.
- One step message passing:
  - $\mathbf{m}_{ij} = f_{Msg}(\mathbf{h}_i^{t-1}, \mathbf{h}_j^{t-1}) \quad \forall (i,j) \in \mathcal{E}$
  - $\bar{\mathbf{m}}_i = f_{Agg}(\{\mathbf{m}_{ij} | \forall j \in \Omega_i\}) \quad \forall i \in \mathcal{V}$
  - $\tilde{\mathbf{h}}_i^{t-1} = f_{Update}(\mathbf{h}_i^{t-1}, \bar{\mathbf{m}}_i) \quad \forall i \in \mathcal{V}$
- where:
  - f_{Msg} is the message function, (often an MLP)
  - f_{Agg} the aggregation function, (could be avg or sum operator)
  - f_{Update} the node update function (GRU or LSTM)

GNN-based Autoregressive Model II

- One step message passing:
  - $\mathbf{m}_{ij} = f_{Msg}(\mathbf{h}_i^{t-1}, \mathbf{h}_j^{t-1}) \quad \forall (i,j) \in \mathcal{E}$
  - $\bar{\mathbf{m}}_i = f_{Agg}(\{\mathbf{m}_{ij} | \forall j \in \Omega_i\}) \quad \forall i \in \mathcal{V}$
  - $\tilde{\mathbf{h}}_i^{t-1} = f_{Update}(\mathbf{h}_i^{t-1}, \bar{\mathbf{m}}_i) \quad \forall i \in \mathcal{V}$
- $\mathbf{h}_i^{t-1}$ is the input node representation at time step $t-1$.
- Ω_i denotes the set of neighbouring nodes of the node i.
- $\tilde{\mathbf{h}}_i^{t-1}$ is the updated node representation which serves as the input node representation for the next message passing step.

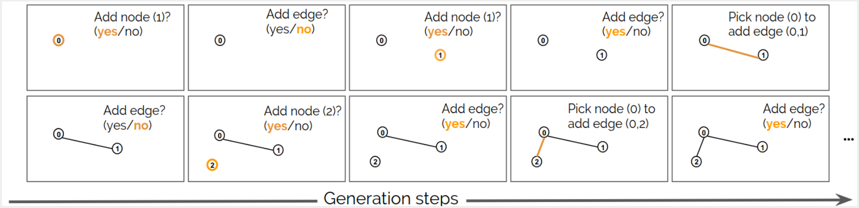
Note:

The message passing process is typically executed for a fixed number of steps. The generation step t is different from the message passing step.

GNN Autoregressive Model Output Probs. I

Output Probabilities After the message passing, we obtain the new node representations H^t, the output probabilities become:

1. $\mathbf{h}_{\mathcal{G}^{t-1}} = f_{ReadOut}(H^t)$
2. $p_{AddNode} = Bernoulli(\sigma(MLP_{AddNode}(\mathbf{h}_{\mathcal{G}^{t-1}})))$
3. $p_{AddEdge} = Bernoulli(\sigma(MLP_{AddEdge}(\mathbf{h}_{\mathcal{G}^{t-1}}, \mathbf{h}_v)))$
4. $s_{uv} = MLP_{Nodes}(\mathbf{h}_u^t, \mathbf{h}_v) \quad \forall u \in \mathcal{V}^{t-1}$
5. $p_{Nodes} = Categorical(softmax(\mathbf{s}))$



GNN Autoregressive Model Output Probs. II

1. $\mathbf{h}_{\mathcal{G}^{t-1}} = f_{ReadOut}(H^t)$
   - $f_{ReadOut}$ could be an average operator or attention-based.
2. $p_{AddNode} = Bernoulli(\sigma(MLP_{AddNode}(\mathbf{h}_{\mathcal{G}^{t-1}})))$
   - $\mathbf{h}_{\mathcal{G}^{t-1}}$ is used to predict the probability of adding a new node.
   - σ is the sigmoid function.
   - A new node v is added if we sample 1 from the Bernoulli distribution $p_{AddNode}$.
3. $p_{AddEdge} = Bernoulli(\sigma(MLP_{AddEdge}(\mathbf{h}_{\mathcal{G}^{t-1}}, \mathbf{h}_v)))$
   - h_v is initialized with by sampling $\mathcal{N}(0, I)$ or a learned distribution.
4. $s_{uv} = MLP_{Nodes}(\mathbf{h}_u^t, \mathbf{h}_v) \quad \forall u \in \mathcal{V}^{t-1}$
   - computed similarity scores between every existing node u in $\mathcal{G}^{t-1}$.
5. $p_{Nodes} = Categorical(softmax(\mathbf{s}))$
   - $\mathbf{s}$ is the concatenated vector of all similarity scores.
   - the normalize the scores using softmax to form the categorical distribution. Every sampled node indicates a new edge.
6. Repeat the procedure carrying the node representations and the generated graphs, until the model generates a stop signal from $p_{AddNode}$.

GNN Autoregressive - Training

Training to train the model, we need to maximize the likelihood of the observed graphs. $\mathcal{G} \equiv \{PAP^\intercal | P \in \prod_{\mathcal{G}}\}$, where $\prod_{\mathcal{G}}$ is the maximal subset of $\prod$ so that $P_1 A P_2^\intercal \neq P_2 A P_2^\intercal$ holds for any $P_1, P_2 \in \prod_{\mathcal{G}}$

$$\max \quad \log p(\mathcal{G}) \Leftrightarrow \max \quad \log \left(\sum_{P \in \prod_{\mathcal{G}}} p(PAP^\intercal) \right)$$

We've omitted $\mathbf{m}_{ij} = f_{Msg}(\mathbf{h}_i^{t-1}, \mathbf{h}_j^{t-1})$ and $\mathbf{h}_{\mathcal{G}^{t-1}} = f_{ReadOut}(H^t)$. Because $\prod_{\mathcal{G}}$ is nearly factorial in size, in practise, we use random samples to create $\tilde{\prod}_{\mathcal{G}}$:

$$\max \quad \log \left(\sum_{P \in \tilde{\prod}_{\mathcal{G}}} p(PAP^\intercal) \right)$$

GNN Autoregessive - Discussion

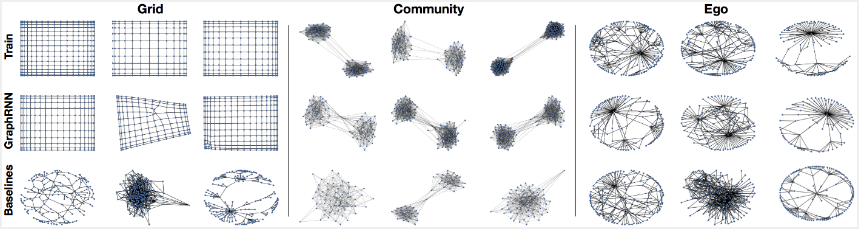
Discussion this model formulates the graph generation as a sequential decision-making process and provides a GNN-based autoregressive model to construct probabilities of possible decisions at each step.

- The overall model design is well-motivated.
- It achieves good empirical performances in generating small graphs, like molecules.
- The model only generates at most one new node and one new edge per step; the total number of generation steps scales with the number of nodes quadratically for dense graphs
- it is inefficient to generate moderately large graphs

Graph Recurrent Neural Networks (GraphRNN)

Similar to the previous formulation but an RNN is used to construct the conditional probabilities.

- The GNN-based autoregressive model generates one entry of the adjacency matrix per step.
- GraphRNN generates one column of entries per step.
- GRAN (next) generates a block of columns/rows per step.



GraphRNN - Simple Variant I

Again we have $\mathcal{G} \equiv \{PAP^{\intercal} | P \in \prod_{\mathcal{G}}\}$ but we assume $P = I$.
The **simple variant of GraphRNN** is then:

$$p(A) = \prod_{t=1}^{n} p(A_t | A_{<t})$$

where A_t is the t-th column of the adjacency matrix A and $A_{<t}$ is a matrix formed by columns $A_1, A_2, ..., A_{t-1}$, n is the maximum number of nodes.

$$p(A_t | A_{<t}) = \text{Bernoulli}(\Theta_t) = \prod_{i=1}^{n} \Theta_{t,i}^{\mathbf{1}[A_{i,t}=1]} (1 - \Theta_{t,i})^{\mathbf{1}[A_{i,t}=0]}$$

GraphRNN - Simple Variant II

$$p(A_t|A_{<t}) = \text{Bernoulli}(\Theta_t) = \prod_{i=1}^{n} \Theta_{t,i}^{\mathbf{1}[A_{i,t}=1]}(1-\Theta_{t,i})^{\mathbf{1}[A_{i,t}=0]}$$

- $\Theta_t = f_{out}(\mathbf{h}_t)$
  - Θ_t is a size-n vector of Bernoulli parameters.
  - $\Theta_{t,i}$ denotes its i-th element.
  - f_{out} could be an MLP which takes the hidden state $\mathbf{h}_t$ as input and outputs Θ_t.
  - $\mathbf{h}_t = f_{trans}(\mathbf{h}_{t-1}, A_{t-1})$
    - f_{trans} is the RNN cell function which takes the $(t-1)$-th column of the adjacency matrix A_{t-1} and the hidden state $\mathbf{h}_{t-1}$ as input and outputs the current hidden state $\mathbf{h}_t$.
- $A_{i,t}$ denotes the i-th element of the column vector A_t.
- We can use LSTM or GRU as the RNN cell function.

GraphRNN Objective

Objective to train the GraphRNN we again use the maximum log likelihood similarly to the previous.
(You et al 2018b) proposes to use a random-breadth-first-search ordering.

- Treat the first sampled node as a root starting from this root node to generate the final node ordering.
- The corresponding premutation matrix is denoted P_{BFS}.

The final objective is:

$$\max \quad \log\left(p(P_{BFS} A P_{BFS}^{\intercal})\right)$$

GraphRNN Discussion - Pros

Pros

- The implementation is straightforward.
- The simple variant is more efficient that the previous autoregressive GNN model.
- The **simple variant** performs comparable with the **full version** in the experiments.

GraphRNN Discussion - Cons

Cons

- GraphRNN is limited in that it **highly depends on the node ordering** since different node orderings would result in very different hidden states.
- Sequential ordering could make two nearby (even neighbouring) nodes far away in the generation sequence (i.e. far away in the generation time step).
- Typically, hidden states of an RNN that are far away regarding the generation time step tend to be quite different,
  - thus making it hard for the model to learn that these nearby nodes should be connected.
  - This is called the ***sequential ordering bias***.

Graph Recurrent Attention Networks (GRAN)

GRAN greatly improves the previous GNN-based models in terms of capacity and efficiency.

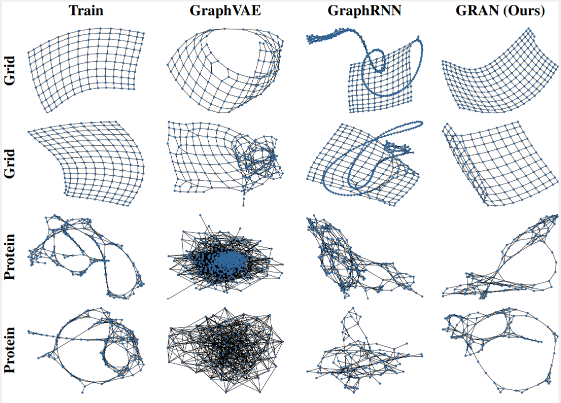


Figure 2: Visualization of sample graphs generated by different models.

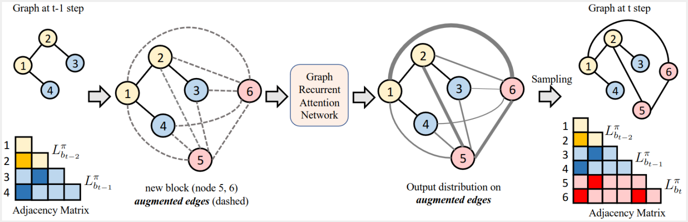
It also alleviates the *sequential ordering bias* of GraphRNN.

GRAN - Model I

Model Again we start with the adjacency matrix representation of graphs, i.e., $\mathcal{G} \equiv \{PAP^\intercal | P \in \Pi_\mathcal{G}\}$ GRAN aims at directly building a probabilistic model over the adjacency matrix similarly to GraphRNN.

- GNN-based autoregressive model generates **one entry** of the adjacency matrix at a step.
- GraphRNN generates **one column** of entries at a step.
- GRAN generates **a block of columns/rows** of the adjacency matrix at a step which greatly improves the generation speed.

GRAN - Model Overview



- At each step we add a new block of nodes
  - block size is 2 and color indicates the membership of individual group in the visualization and augmented edges (dashed lines).
- Apply GRAN to this graph to obtain the output distribution over augmented edges
  - shown is an edge-independent Bernoulli where the line width indicates the probability of generating individual augmented edges.
- Finally, we sample from the output dist. to obtain a new graph.

GRAN - Autoregressive Decomposition

GRAN generates a block of columns/rows of the adjacency matrix at a step, denoting the submatrix with first k rows of the adjacency matrix A as $A_{1:k,:}$ giving the following autoregressive decomposition of the probability:

$$p(A) = \prod_{t=1}^{\lceil n/k \rceil} p(A_{(t-1)k:tk,:} | A_{:(t-1)k,:})$$

where:

- $A_{:(t-1)k,:}$ indicates the adjacency matrix that has been generated before the t-th step (i.e., $t-1$ blocks with block size k.
- $A_{(t-1)k:tk,:}$ denotes the to-be-generated block at t-th time step.

GRAN - Message Passing

The GNN used has the following one-step msg. passing process:

- $\mathbf{m}_{ij} = f_{msg}(\mathbf{h}_i - \mathbf{h}_j) \quad \forall (i,j) \in \tilde{\mathcal{E}}^t$
  - $\mathbf{m}_{ij}$ is the message over edge (i,j)
- $\tilde{\mathbf{h}}_i = [\mathbf{h}_i \| \mathbf{x}_i] \quad \forall i \in \mathcal{V}^t$
  - $\mathbf{h}'_i$ serves as the input to the next message passing step.
  - $[\mathbf{h}_i \| \mathbf{x}_i]$ means the concatenation of two vectors.
- $a_{ij} = sigmoid(g_{att}(\tilde{\mathbf{h}}_i - \tilde{\mathbf{h}}_j)) \quad \forall (i,j) \in \tilde{\mathcal{E}}^t$
  - a_{ij} are the attention weights
  - g_{att} could be MLPs.
- $\mathbf{h}'_i = GRU(\mathbf{h}_i, \sum_{j \in \Omega(i)} a_{ij}\mathbf{m}_{ij}) \quad \forall i \in \mathcal{V}^t$
  - Ω_i is the set of neighboring nodes of node i

The message passing is unrolled for a fixed number of steps which is set as a hyperparameter.

GRAN - Attention Weighted Output Distribution

Based on the final node representations returned by the message passing, we can construct the output distribution as follows:

- $p(A_{(t-1)k:tk,:} | A_{:(t-1)k,:}) = \sum_{c=1}^{C} \alpha_c \prod_{i=(t-1)k+1}^{tK} \prod_{j=1}^{n} \Theta_{c,i,j}$
- $\alpha = \text{softmax}\left(\sum_{i=(t-1)k+1}^{tK} \sum_{j=1}^{n} MLP_\alpha(\mathbf{h}_i^R - \mathbf{h}_j^R) \right)$
- $\Theta_{c,i,j} = \text{sigmoid}(MLP_\Theta(\mathbf{h}_i^R - \mathbf{h}_j^R))$

GRAN - Objective

Objective to train the model, there are different canonical orderings:

- breadth-first-search,
- depth-first-search,
- node-degree-descending,
- node-degree-ascending, and
- the k-core ordering.

The final training objective is:

$$\max \quad \log\left(\sum_{P\in\tilde{\Pi}_{\mathcal{G}}} p(PAP^{\top})\right)$$

GRAN - Pros

Discussion GRAN improves over the GNN-based autoregressive model and GraphRNN in the following ways:

Pros

- It generates a block of rows of the adjacency matrix per step, which is more efficient than generating an entry per step and then generating a row per step.
- GRAN uses a GNN to construct the conditional probability which helps alleviate the sequential ordering bias in GraphRNN since GNN is permutation equivariant. (node order does not affect the conditional probability per step).
- the output distribution in GRAN is more expressive and more efficient for sampling.

GRAN - Cons

GRAN still has issues:

Cons

- it suffers from the fact that the overall model depends on the particular choices of node orderings.
- depending on the application it may be challenging to find good orderings.
- It is an open question on how to to build an order-invariant deep graph generative model.

Papers Discussed I

- PixelRNNs (van den Oord et al, 2016a)
- PixelCNNs (van den Oord et al, 2016b)
- GNN-based Autoregressive Model
  - First GNN-based autoregressive model was proposed in (Li et al, 2018d)
  - Message passing graph neural network (Scarselli et al, 2008; Li et al, 2016b; Gilmer et al, 2017) to learn node representations.
  - gated recurrent units (GRUs) (Cho et al, 2014a)
  - long-short term memory (LSTM) (Hochreiter and Schmidhuber, 1997).
- Graph Recurrent Neural Networks (GraphRNN)
  - Graph Recurrent Neural Networks (GraphRNN) (You et al, 2018b)

Papers Discussed II

- Instead of randomly sampling a few orderings like (Li et al, 2018d, You et al, 2018b) propose to use a random-breadth-first-search ordering.

- Graph Recurrent Attention Networks (GRAN)
    - (Li et al, 2018d, You et al, 2018b, Liao et al 2019a) propose to use a set of canonical orderings to deal with the permutations when maximizing the log likelihood (starting with largest node degree).
    - The k-core graph decomposition has been shown to be very useful for modeling cohesive groups in social networks (Seidman, 1983).
    - Importantly, the core decomposition, i.e., all cores ranked based on their orders, can be found in linear time (w.r.t. the number of edges) (Batagelj and Zaversnik, 2003).

Section 2: Generative Adversarial Methods

Generative Adversarial Methods - Background

Generative Adversarial Networks (GANs) are a generative modeling approach based on a game theoretic scenario.

There is a **generator network** that directly produces samples and a **discriminator network** that attempts to distinguish between samples drawn from the training data and samples drawn from the generator:

- The **generator network** directly produces samples: $\mathbf{x} = g(z; \theta^{(g)})$
- The **discriminator network** outputs a probability value: $d(\mathbf{x}; \theta^{(d)})$
- The discriminator receives: $v(\theta^{(g)}, \theta^{(d)})$ and,
- The generator receives: $-v(\theta^{(g)}, \theta^{(d)})$

For more details see: (Goodfellow et al, 2016, Ch 20) [**?**]

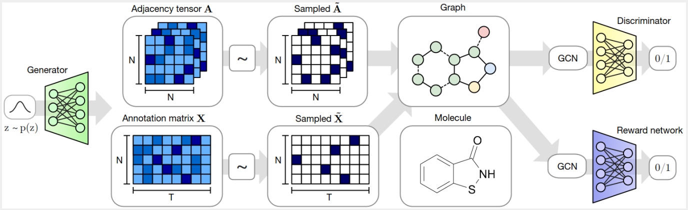
Graph GANs I

The GAN framework for graphs divides into two broad categories:

- **Adjacency matrix based GANs**
  - Two examples:
    - MolGAN
    - Graph Convolutional Policy Network (GCPN)
  - Both are molecular graph examples of Wasserstein GANs.
- **Random walk based GANs**
  - One example:
    - NetGAN
  - Essentially maps a graph to a set of random walks and learns a generator/discriminator in the space of random walks.
    - **Generator**: generates random walks similar to those sampled from observed graphs.
    - **Discriminator**: should correctly distinguish whether a random walk comes from a data distribution or the distribution corresponding to the generator.

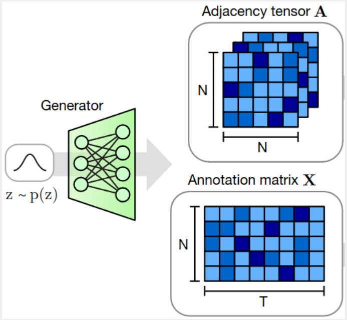
Adjacency Matrix Based GAN - Overview

1. Draw a latent variable $Z \sim p(Z)$.
2. Feed $Z \sim p(Z)$ to a generator $\mathcal{G}_\theta(Z)$, which produces a **probabilistic (continuous) adjacency tensor** A and a **probabilistic (continuous) node type matrix** X.
3. Draw a discrete adjacency matrix $\tilde{A} \sim A$ and a discrete node type matrix $\tilde{X} \sim X$, which together specify a molecule graph.
4. During training, simultaneously feed the generated graph to a **discriminator** $\mathcal{D}_\phi(A, X)$, and a **reward network** $\mathcal{R}_\psi(A, X)$, to obtain the:
   - **adversarial loss**: measuring how similar the generated and the observed graphs are, and
   - **negative reward**: measuring how likely the generated graphs satisfy the certain chemical constraints.



Adjacency Matrix Based GAN - Model 1

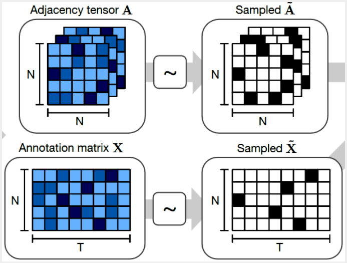
1. Sample a latent variable $Z \in \mathbb{R}^d$ from some prior $Z \sim \mathcal{N}(0, \mathcal{I})$
2. Use an MLP $\bar{\mathcal{G}}_\theta(Z)$ to directly map the sampled Z to:
   - a continuous adjacency tensor A
   - a continuous node type matrix X.



- Each $A_{i,j,c}$ is the probability of connecting the atom i to the atom j using the chemical bond type c
- Each $X_{i,t}$ is the probability of assigning the t-th atom type to the i-atom.

Adjacency Matrix Based GAN - Model 2

1. The discrete graph data $(\tilde{A}, \tilde{X})$ is sampled from (A, X)
2. The sampling procedure can be implemented using a Gumbel softmax:
   - one_hot $\left(\arg\max_i [g_i + log \pi_i]\right)$

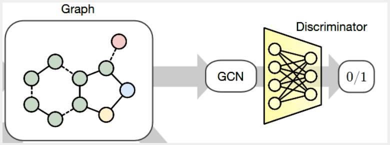


Adjacency tensor $\mathbf{A}$
Sampled $\tilde{\mathbf{A}}$
Annotation matrix $\mathbf{X}$
Sampled $\tilde{\mathbf{X}}$
N
N
T
~

- The discrete graph data $(\tilde{A}, \tilde{X})$ specify a molecule graph and complete generation process.

Adjacency Matrix Based GAN - Model 3

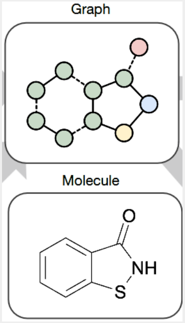
To evaluate how similar the generated graphs and observed graphs are, we need to build a **discriminator** $\mathcal{D}_\phi(A, X)$. This is where GNNs come in, typically a variant of a GCN is used here:



- $$\mathbf{h}_i' = \tanh\left(f_s(\mathbf{h}_i, \mathbf{x}_i) + \sum_{j=1}^{N} \sum_{y=1}^{Y} \frac{\tilde{A}_{i,j,y}}{|\Omega_i|} f_y(\mathbf{h}_i, \mathbf{x}_i) \right)$$
  - $\mathbf{h}_i$ is the input node representation.
  - $\mathbf{h}_i'$ is the output node representation.
  - Ω_i is the set of neighboring nodes of the node i.
  - $\mathbf{x}_i$ is the i-th row of X.
  - f_s and f_y are linear transformation functions that are to be learned.

Adjacency Matrix Based GAN - Model 4

After stacking as many GCN layers as your GPU can handle, we can readout the graph representation using the following attention-weighted aggregation:



$$\mathbf{h}_{\mathcal{G}} = \tanh\left(\sum_{v \in \mathcal{V}} \text{sigmoid}(\text{MLP}_{att}(\mathbf{h}_v, \mathbf{x}_v)) \odot \tanh(\text{MLP}(\mathbf{h}_v, \mathbf{x}_v))\right)$$

- $\mathbf{h}_v$ is the node representation returned by the final layer.
- MLP_{att} and MLP are two different MLPs
- $\odot$ means element-wise product.
- $\mathbf{h}_{\mathcal{G}}$ is used to compute the discriminator score $\mathcal{D}_\phi(A, X)$.

Adj. Mtx GAN - Discriminator Objective

$$\max_{\phi} \sum_{i=1}^{B} -\mathcal{D}_\phi(A^{(i)}, X^{(i)}) + \mathcal{D}_\phi(\bar{\mathcal{G}}_\theta(Z^{(i)})) + \alpha \left(\|\nabla_{\hat{A}^{(i)}, \hat{X}^{(i)}} \mathcal{D}_\phi(\hat{A}^{(i)}, \hat{X}^{(i)})\| - 1 \right)^2$$

- B is the mini-batch size
- $Z^{(i)}$ is the i-th sample drawn from the prior
- $A^{(i)}$, $X^{(i)}$ are the i-th graph data drawn from the data distribution
- $\hat{A}^{(i)}$, $\hat{X}^{(i)}$ are their linear combinations:
  - $(\hat{A}^{(i)}, \hat{X}^{(i)}) = \epsilon(A^{(i)} X^{(i)}) + (1-\epsilon)\bar{\mathcal{G}}_\theta(Z^{(i)}), \epsilon \sim \mathcal{U}(0, 1)$
- The squared term penalizes the gradient of the discriminator so that the training becomes more stable.
- α is a weighting term to balance the regularization and the objective.

Adj. Mtx GAN - Generator Objective

The generator is trained by adding the additional constraint-dependent reward:

$$\min_{\theta} \sum_{i=1}^{B} \lambda \mathcal{D}_{\phi}(\bar{\mathcal{G}}_{\theta}(Z^{(i)})) + (1 - \lambda)\mathcal{L}_{RL}(\bar{\mathcal{G}}_{\theta}(Z^{(i)}))$$

- $\mathcal{L}_{RL}$ the negative reward returned by the reward network $\mathcal{R}_{\psi}$
- λ is the weighting hyperparameter to regulate the trade-off between two losses.
- $\mathcal{R}_{\psi}$ is pretrained to produce the desired property scores of the molecules (e.g. water solubility).

Adj. Mtx GAN - Positives

- MolGAN
  - has strong empirical performance on QM9
  - MolGAN is likelihood-free so it has more flexible and powerful generators.
  - Although generator still depends on node ordering, the discriminator and the reward networks are permutation invariant.
- GCPN
  - Very similar with additional domain-specific rewards, but do not use a reward network, instead they use Proximal Policy Optimization (PPO).
  - Generates the adjacency matrix in an entry-by-entry autoregressive fashion whereas MolGAN generates all entries of the adjacency matrix in parallel conditioned on the latent variable.
  - Has strong empirical performance on ZINC.

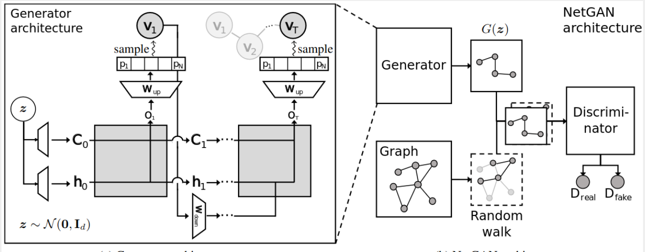
Adj. Mtx GAN - Negatives

- MolGAN
  - Pre-training the reward network is crucial to make training successful.
  - Replacing the GCN discriminators with more powerful discriminators would be promising.
- GCPN
  - Discrete gradient estimators (like PPO) could have large variances slowing down training.
  - With GCPN domain-specific rewards are non-differentiable.
  - Replacing the GCN discriminators with more powerful discriminators (like Lanczos network) would be promising.

Random Walk Based GAN - Overview

NetGAN

- Essentially maps a graph to a set of random walks and learns a generator/discriminator in the space of random walks.
  - **Generator**: generates random walks similar to those sampled from observed graphs.
  - **Discriminator**: should correctly distinguish whether a random walk comes from a data distribution or the distribution corresponding to the generator.

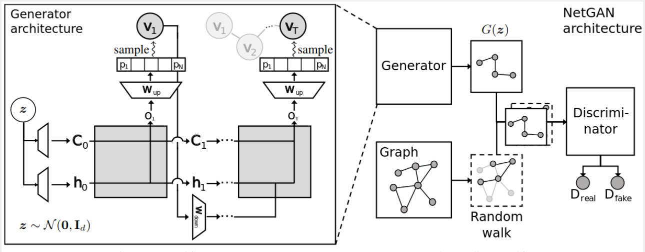


(a) Generator architecture (b) NetGAN architecture

Figure 2: The NetGAN architecture proposed in this work (b) and the generator architecture (a).

NetGAN - Model Overview

1. Draw a random vector from a fixed prior $\mathcal{N}(0, I)$ and initialized the memory c_0 and the hidden state $\mathbf{h}_0$ of an LSTM.
2. The **LSTM generator** generates which node to visit per step and is unrolled for a fixed number of steps T.
3. The one-hot-encoding of node index is fed to the LSTM as the input for the next step.
4. The **LSTM discriminator** performs a binary classification to predict if a given random walk is sampled from a data distribution. (Bojchevski et al, 2018)

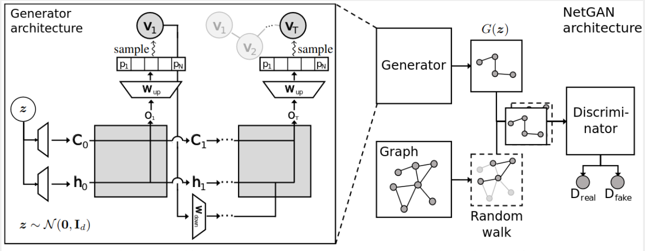


(a) Generator architecture (b) NetGAN architecture

Figure 2: The NetGAN architecture proposed in this work (b) and the generator architecture (a).

NetGAN - Model

1. Given a graph $\mathcal{G}$ with N nodes and adjacency matrix $A \in \{0,1\}^{N \times N}$ sample a set of random walks of length T.
   - Random walk: $(v_1, \cdots .v_T)$ where v_i is one node in $\mathcal{G}$
   - Assume the maximum number of nodes for any graph is N.
   - Random walks are invariant under node reordering.
   - Random walks only include nonzero entries of A

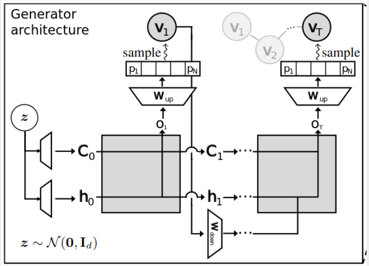


(a) Generator architecture (b) NetGAN architecture

Figure 2: The NetGAN architecture proposed in this work (b) and the generator architecture (a).

NetGAN - Generator Architecture 1

- The generator is modelled as a sequential process based on a neural network f_θ parameterized by θ. At each step t, f_θ produces two values:
    1. the probability distribution over the next node to be sampled, parameterized by the logits $\mathbf{p}_t$.
    2. the current memory state of the model, denoted as $\mathbf{m}_t$
- The next node v_t represented as a one-hot vector, is sampled from a categorical distribution $v_t \sim \text{Cat}(\sigma(\mathbf{p}_t))$, where $\sigma(\cdot)$ denotes the softmax function

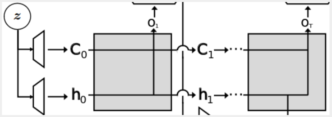


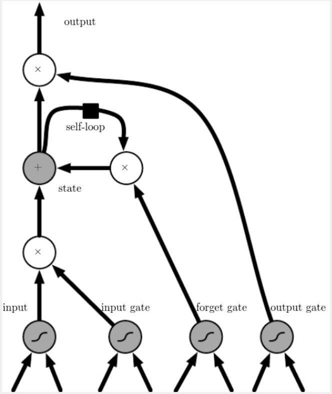
Generative Process

$$
\begin{array}{lr}
 & \boldsymbol{z} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}_d) \\
 & \boldsymbol{m}_0 = g_{\theta'}(\boldsymbol{z}) \\
\boldsymbol{v}_1 \sim \text{Cat}(\sigma(\boldsymbol{p}_1)), & (\boldsymbol{p}_1, \boldsymbol{m}_1) = f_\theta(\boldsymbol{m}_0, \mathbf{0}) \\
\boldsymbol{v}_2 \sim \text{Cat}(\sigma(\boldsymbol{p}_2)), & (\boldsymbol{p}_2, \boldsymbol{m}_2) = f_\theta(\boldsymbol{m}_1, \boldsymbol{v}_1) \\
\vdots & \vdots \\
\boldsymbol{v}_T \sim \text{Cat}(\sigma(\boldsymbol{p}_T)), & (\boldsymbol{p}_T, \boldsymbol{m}_T) = f_\theta(\boldsymbol{m}_{T-1}, \boldsymbol{v}_{T-1})
\end{array}
$$

NetGAN - Generator Architecture 2

- f_θ uses a Long short-term memory (LSTM) architecture.
- The memory state $\mathbf{m}_t$ of the LSTM is represented by:
  - the cell state C_t
  - the hidden state $\mathbf{h}_t$
  - $(\mathbf{C}_0, \mathbf{h}_0)$ are initialized by pass z through two separate steams of two fill connected layers with a `tanh` activation.
- To save on computational overhead the LSTM output $\mathbf{o}_t \in \mathbb{R}^H$ instead of $\mathbf{p}_t$ with $H << N$





Papers Discussed I

- (De Cao and Kipf, 2018; Bojchevski et al, 2018; You et al, 2018a) that apply the idea of generative adversarial networks (GAN) (Goodfellow et al, 2014b)
- Adjacency Matrix Based GAN
  - MolGAN (De Cao and Kipf, 2018) and graph convolutional policy network (GCPN) (You et al, 2018a)
  - This sampling procedure can be implemented using the Gumbel softmax (Jang et al, 2017; Maddison et al, 2017).
  - e.g., a graph convolutional network (GCN) (Kipf and Welling, 2017b). In particular, we use a variant of GCN (Schlichtkrull et al, 2018)
  - To address certain issues in training GANs such as the mode collapse and the instability, Wasserstein GAN (WGAN) (Arjovsky et al, 2017) and its improved version (Gulrajani et al, 2017) have been proposed.

Papers Discussed II

- To learn the model with the non-differentiable reward, the deep deterministic policy gradient (DDPG) (Lillicrap et al, 2015) is used.
- MolGAN demonstrates strong empirical performances on a large chemical database called QM9 (Ramakrishnan et al, 2014)
- To deal with the learning of non-differentiable reward, GCPN leverages the proximal policy optimization (PPO) (Schulman et al, 2017) method,
- GCPN also achieves impressive empirical results on another large chemical database called ZINC (Irwin et al,2012).
- GCNs as the discriminator, which is shown to be insufficient in distinguishing certain graphs 4 (Xu et al, 2019d).
- Lanczos network (Liao et al, 2019b) that exploits the spectrum of the graph Laplacian as the input feature would be promising to further improve the performance of the above methods.

Papers Discussed III

- Graph Recurrent Neural Networks Random Walk Based GAN
    - NetGAN (Bojchevski et al, 2018)
    - biased second order random walk sampling strategy described in (Grover and Leskovec, 2016).
    - The model is trained with the same objective as the improved WGAN (Gulrajani et al, 2017).

Questions

Questions?

GraphRNN Extras I

Full Version of Graph RNN uses another autoregressive construction to model the dependencies among entries within one column of the adjacency matrix instead of using edge-independent Bernoulli distribution.

- $p(A_t|A_{<t}) = \prod_{i=1}^{n} p(A_{i,t}|A_{<i,<t})$
- $p(A_{i,t}|A_{<i,<t}) = sigmoid(g_{out}(\tilde{\mathbf{h}}_{i,t}))$
- $\tilde{\mathbf{h}}_{i,t} = g_{trans}(\tilde{\mathbf{h}}_{i-1,t}, A_{<i,t})$
- $\tilde{\mathbf{h}}_{0,t} = h_t$
- $\mathbf{h}_t = f_{trans}(\mathbf{h}_{t-1}, A_{t-1})$

GraphRNN Extras II

The bottom RNN cell function f_{trans} still recurrently updates the hidden state to get $\mathbf{h}_t$, thus implementing the conditioning on all previous $t-1$ columns of the adjacency matrix A. To generate individual entries of the t-th column, the top RNN cell function g_{trans} takes its own hidden state $\tilde{\mathbf{h}}_{i-1,t}$ and the already generated t-th column A has input and updates teh hidden state as $\tilde{\mathbf{h}}_{i,t}$. The output distribution is a Bernoulli parameterized by the output of an MLP g_{out} which takes $\tilde{\mathbf{h}}_{i,t}$ as input. Note that the initial hidden state $\tilde{\mathbf{h}}_{0,t}$ of the top RNN is set to the hidden state $\mathbf{h}_t$ returned by the bottom RNN.