Stanford CS224W: Deep Generative Models for Graphs

CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu
We want to generate realistic graphs, using graph generative models
Step 1: Properties of real-world graphs
- A successful graph generative model should fit these properties

Step 2: Traditional graph generative models
- Each come with different assumptions on the graph formulation process

Step 3: Deep graph generative models
- Learn the graph formation process from the data
- This lecture!
Deep Graph Encoders

Graph convolutions

Regularization, e.g., dropout

Graph convolutions

Nodes → Activation function → Nodes

... → Nodes

Output: node embeddings
Today: Deep Graph Decoders

Output: Graph Structure!
Graph Generation Tasks

Task 1: Realistic graph generation
- Generate graphs that are similar to a given set of graphs [Focus of this lecture]

Task 2: Goal-directed graph generation
- Generate graphs that optimize given objectives/constraints
  - E.g., Drug molecule generation/optimization
Graph Generative Models

- **Given:** Graphs sampled from $p_{data}(G)$
- **Goal:**
  - Learn the distribution $p_{model}(G)$
  - Sample from $p_{model}(G)$
Setup:
- Assume we want to learn a generative model from a set of data points (i.e., graphs) \( \{x_i\} \)
  - \( p_{\text{data}}(x) \) is the data distribution, which is never known to us, but we have sampled \( x_i \sim p_{\text{data}}(x) \)
  - \( p_{\text{model}}(x; \theta) \) is the model, parametrized by \( \theta \), that we use to approximate \( p_{\text{data}}(x) \)

Goal:
- (1) Make \( p_{\text{model}}(x; \theta) \) close to \( p_{\text{data}}(x) \) (Density estimation)
- (2) Make sure we can sample from \( p_{\text{model}}(x; \theta) \) (Sampling)
  - We need to generate examples (graphs) from \( p_{\text{model}}(x; \theta) \)
(1) Make $p_{model}(x; \theta)$ close to $p_{data}(x)$

- **Key Principle:** Maximum Likelihood
- Fundamental approach to modeling distributions

$$\theta^* = \arg \max_{\theta} \mathbb{E}_{x \sim p_{data}} \log p_{model}(x \mid \theta)$$

- Find parameters $\theta^*$, such that for observed data points $x_i \sim p_{data}$, $\sum_i \log p_{model}(x_i; \theta^*)$ has the highest value, among all possible choices of $\theta$
  - That is, find the model that is most likely to have generated the observed data $x$
(2) Sample from $p_{model}(x; \theta)$

- **Goal:** Sample from a complex distribution
- The most common approach:
  - (1) Sample from a simple noise distribution
    $$z_i \sim N(0,1)$$
  - (2) Transform the noise $z_i$ via $f(\cdot)$
    $$x_i = f(z_i; \theta)$$

Then $x_i$ follows a complex distribution

- **Q:** How to design $f(\cdot)$?
- **A:** Use Deep Neural Networks, and train it using the data we have!
Auto-regressive models:

- $p_{model}(x; \theta)$ is used for both density estimation and sampling (remember our two goals)

- Other models like Variational Auto Encoders (VAEs), Generative Adversarial Nets (GANs) have 2 or more models, each playing one of the roles

- **Idea: Chain rule.** Joint distribution is a product of conditional distributions:

$$p_{model}(x; \theta) = \prod_{t=1}^{n} p_{model}(x_t | x_1, \ldots, x_{t-1}; \theta)$$

- E.g., $x$ is a vector, $x_t$ is the $t$-th dimension;
  $x$ is a sentence, $x_t$ is the $t$-th word.

- **In our case:** $x_t$ will be the $t$-th action (add node, add edge)
Stanford CS224W: GraphRNN: Generating Realistic Graphs
GraphRNN Idea

Generating graphs via sequentially adding nodes and edges

Graph $G$

Generation process $S^\pi$
Graph $G$ with node ordering $\pi$ can be uniquely mapped into a sequence of node and edge additions $S^\pi$.

Graph $G$ with node ordering $\pi$: $\downarrow \quad \uparrow$

Sequence $S^\pi$: $\begin{align*}
S^\pi &= ( S_1^\pi, S_2^\pi, S_3^\pi, S_4^\pi, S_5^\pi )
\end{align*}$
The sequence $S^\pi$ has two levels ($S$ is a sequence of sequences):

- **Node-level:** add nodes, one at a time
- **Edge-level:** add edges between existing nodes

**Node-level:** At each step, a new node is added

\[
S^\pi = \left( S_1^\pi, S_2^\pi, S_3^\pi, \ldots, S_4^\pi, S_5^\pi \right) \\
\text{“Add node 1”, “Add node 2”, “Add node 3”, “Add node 4”, “Add node 5”}
\]
The sequence $S^\pi$ has two levels:

- Each **Node-level** step is an **edge-level** sequence
- **Edge-level**: At each step, add a new edge

$$S_4^\pi = \left( S_{4,1}^\pi, S_{4,2}^\pi, S_{4,3}^\pi \right)$$

- “Not connect 4, 1”
- “Connect 4, 2”
- “Connect 4, 3”

```
0 1 1
```
Model Graphs as Sequences

- **Summary:** A graph + a node ordering = A sequence of sequences
- Node ordering is randomly selected (we will come back to this)

![Graph](image)

- **Node-level sequence:**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Edge-level sequence:**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Adjacency matrix
Model Graphs as Sequences

- We have transformed graph generation problem into a sequence generation problem

- Need to model two processes:
  1) Generate a state for a new node (Node-level sequence)
  2) Generate edges for the new node based on its state (Edge-level sequence)

- Approach: Use Recurrent Neural Networks (RNNs) to model these processes!
RNNs are designed for **sequential data**

- RNN sequentially takes **input sequence** to **update its hidden states**
- The **hidden states** summarize all the information input to RNN
- The update is conducted via **RNN cells**

![Diagram of RNN cell](attachment:diagram.png)
Background: Recurrent NNs

- **$S_t$: State** of RNN after step $t$
- **$x_t$: Input** to RNN at step $t$
- **$y_t$: Output** of RNN at step $t$
- **RNN cell: $W$, $U$, $V$:** Trainable parameters

In our case, $s_t$, $x_t$, and $y_t$ are scalars.

The RNN cell:
1. **Update hidden state:**
   \[ S_t = \sigma(W \cdot x_t + U \cdot S_{t-1}) \]
2. **Output prediction:**
   \[ y_t = V \cdot S_t \]

- **More expressive cells:** GRU, LSTM, etc.
GraphRNN: Two levels of RNN

- GraphRNN has a node-level RNN and an edge-level RNN

- Relationship between the two RNNs:
  - Node-level RNN generates the initial state for edge-level RNN
  - Edge-level RNN sequentially predict if the new node will connect to each of the previous node
GraphRNN: Two levels of RNN

Node-level RNN generates the initial state for edge-level RNN

Edge-level RNN sequentially predict if the new node will connect to each of the previous node
**GraphRNN: Two levels of RNN**

Node-level RNN generates the initial state for edge-level RNN

Next: How to generate a sequence with RNN?
Q: How to use RNN to generate sequences?
A: Let $x_{t+1} = y_t$! (Use the previous output as input)

Q: How to initialize the input sequence?
A: Use start of sequence token (SOS) as the initial input
   - SOS is usually a vector with all zero/ones

Q: When to stop generation?
A: Use end of sequence token (EOS) as an extra RNN output
   - If output EOS=0, RNN will continue generation
   - If output EOS=1, RNN will stop generation
This is good, but this model is deterministic
**Remember our goal:** Use RNN to model
$$\prod_{k=1}^{n} p_{model}(x_t | x_1, \ldots, x_{t-1}; \theta)$$

Let $$y_t = p_{model}(x_t | x_1, \ldots, x_{t-1}; \theta)$$

Then we need to sample $$x_{t+1}$$ from $$y_t$$: $$x_{t+1} \sim y_t$$

- Each step of RNN outputs a **probability of a single edge**
- We then sample from the distribution, and feed sample to next step:

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s_0</td>
<td>RNN</td>
<td>s_1</td>
</tr>
<tr>
<td></td>
<td>cell</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_1 = SOS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>y_1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s_1</td>
<td>RNN</td>
<td>s_2</td>
</tr>
<tr>
<td></td>
<td>cell</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_2 \sim y_1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>y_2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s_2</td>
<td>RNN</td>
<td>s_3</td>
</tr>
<tr>
<td></td>
<td>cell</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x_3 \sim y_2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>s_3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
```
Suppose we already have trained the model

- $y_t$ is a scalar, following a Bernoulli distribution
- $p$ means value 1 has prob. $p$, value 0 has prob. $1 - p$

How do we use training data $x_1, x_2, \ldots, x_n$?
Training the model:

- We observe a sequence $y^*$ of edges $[1,0,...]$
- Principle: **Teacher Forcing** -- Replace input and output by the real sequence

$$s_0 = SOS$$
$$x_1 = SOS$$

![Diagram](image-url)
**RNN at Training Time**

- **Loss** $L$: **Binary cross entropy**
- **Minimize:**

$$L = -[y_1^* \log(y_1) + (1 - y_1^*) \log(1 - y_1)]$$

Compute loss

$y_1^* = 1$

$y_1 = 0.9$

- If $y_1^* = 1$, we minimize $-\log(y_1)$, making $y_1$ higher
- If $y_1^* = 0$, we minimize $-\log(1 - y_1)$, making $y_1$ lower
- This way, $y_1$ is **fitting** the data samples $y_1^*$
- **Reminder**: $y_1$ is computed by RNN, this loss will **adjust RNN parameters accordingly**, using back propagation!
Our Plan:

1. **Add a new node**: We run Node RNN for a step, and use its output to initialize Edge RNN.
2. **Add new edges for the new node**: We run Edge RNN to predict if the new node will connect to each of the previous node.
3. **Add another new node**: We use the last hidden state of Edge RNN to run Node RNN for another step.
4. **Stop graph generation**: If Edge RNN outputs EOS at step 1, we know no edges are connected to the new node. We stop the graph generation.
Assuming **Node 1** is in the graph
Now adding **Node 2**

```
Observed graph
```

Start the node RNN

\[ \text{SOS} \]
Put Things Together: Training

Edge RNN predicts how Node 2 connects to Node 1

Observed graph

0 1 1
1 0 0
1 0 0

Will node 2 connect to node 1?

Edge RNN

Start the edge RNN

Node RNN

SOS

SOS

0.5
Update Node RNN using Edge RNN’s hidden state
Edge RNN predicts how **Node 3** tries to connects to **Nodes 1, 2**

Teacher forcing: node 3 won’t connect to node 2

**Observed graph**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Update Node RNN using Edge RNN’s hidden state.
Stop generation since we know node 4 won’t connect to any nodes
Put Things Together: Training

For each prediction, we **get supervision from the ground truth**
Backprop through time:
Gradients are accumulated across time steps
Test time: (1) Sample edge connectivity based on predicted distribution
(2) Replace input at each step by GraphRNN’s own predictions
Quick Summary of GraphRNN:

- Generate a graph by generating a two level sequence
- Use RNN to generate the sequences

- **Next**: Making GraphRNN tractable, proper evaluation
Issue: Tractability

- Any node can connect to any prior node
- Too many steps for edge generation
  - Need to generate full adjacency matrix
  - Complex too-long edge dependencies

“Recipe” to generate the left graph:
- Add node 1
- Add node 2
- Add node 3
- Connect 3 with 2 and 1
- Add node 4
- ...

Random node ordering:
Node 5 may connect to any/all previous nodes

How do we limit this complexity?
**Solution: Tractability via BFS**

- **Breadth-First Search node ordering**

  "Recipe" to generate the left graph:
  - Add node 1
  - Add node 2
  - Connect 2 with 1
  - Add node 3
  - Connect 3 with 1
  - Add node 4
  - Connect 4 with 3 and 2

- **BFS node ordering:**
  - Since Node 4 doesn’t connect to Node 1
  - We know all Node 1’s neighbors have already been traversed
  - Therefore, Node 5 and the following nodes will never connect to node 1
  - We only need memory of 2 “steps” rather than $n - 1$ steps
Breadth-First Search node ordering

BFS node ordering: Node 5 will never connect to node 1 (only need memory of 2 “steps” rather than $n - 1$ steps)

Benefits:
- Reduce possible node orderings
  - From $O(n!)$ to number of distinct BFS orderings
- Reduce steps for edge generation
  - Reducing number of previous nodes to look at
Solution: Tractability via BFS

- BFS reduces the number of steps for edge generation

Adjacency matrices

Without BFS ordering

N=10  M=9
Connectivity with All Previous nodes

With BFS ordering

N=10  M=3
Connectivity only with nodes in the BFS frontier
Evaluating Generated Graphs

- **Task:** Compare two sets of graphs

  - **Goal:** Define similarity metrics for graphs

  - **Solution**
    - (1) Visual similarity
    - (2) Graph statistics similarity
(1) Visual Similarity

Grid

Training

GraphRNN

Baselines

(Kronecker) (MMSB) (B-A)
(1) Visual Similarity
Can we do more rigorous comparison?

**Issue:** Direct comparison between two graphs is hard (isomorphism test is NP)!

**Solution:** Compare graph statistics!

Typical Graph Statistics:
- Degree distribution (Deg.)
- Clustering coefficient distribution (Clus.)
- Orbit count statistics (Orbit)

**Note:** Each statistic is a probability distribution.
(2) Graph statistics similarity

- **Issue:** want to compare *sets* of training graph statistics and generated graph statistics

- **Solution:**
  - **Step 1:** How to compare *two* graph statistics
    - Earth Mover Distance (EMD)
  - **Step 2:** How to compare *sets of* graph statistics
    - Maximum Mean Discrepancy (MMD) based on EMD
(2) Graph statistics similarity

- **Step 1: Earth Mover Distance (EMD)**
  - Compare *similarity between 2 distributions*
  - **Intuition:** Measure the minimum effort that *move earth from one pile to the other*

![Diagram showing high and low EMD with degree distributions A, B, and C]
Step 2: Maximum Mean Discrepancy (MMD)

- Compare similarity between 2 sets, based on the similarity between set elements

\[
\{2,3,5\} \quad \text{High MMD} \quad \{6,1,2\} \quad \text{Low MMD} \quad \{2,3,5\} \quad \{5,4,2\}
\]

Similarity between set elements: L2 distance
(Each element is a scalar)

Recall: We compare 2 sets of graph statistics (distributions)

Similarity between set elements: EMD
(Each element is a distribution)
(2) Graph statistics similarity

- Putting things together

How similar?

Compute graph statistics

Report MMD: set distance

where similarity between set elements are computed by EMD distribution distance
(2) Graph statistics similarity

- Example

How similar?

Compute cluster coefficient distribution

Report MMD: set distance

<table>
<thead>
<tr>
<th></th>
<th>Erdos-Renyi</th>
<th>Kronecker</th>
<th>GraphRNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMD score</td>
<td>1.24</td>
<td>1.67</td>
<td>0.002</td>
</tr>
</tbody>
</table>

3/2/21
Stanford CS224W: Application of Deep Graph Generative Models

CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu
Question: Can we learn a model that can generate **valid** and **realistic** molecules with **optimized** property scores?

Model ➔ output ➔ that optimizes ➔ Property

e.g., drug likeness = 0.95

---

Generating graphs that:

- **Optimize a given objective** (High scores)
  - e.g., drug-likeness
- **Obey underlying rules** (Valid)
  - e.g., chemical validity rules
- **Are learned from examples** (Realistic)
  - Imitating a molecule graph dataset
    - We have just covered this part
The Hard Part:

Generating graphs that:

- **Optimize a given objective (High scores)**
  - e.g., drug-likeness
- **Obey underlying rules (Valid)**
  - e.g., chemical validity rules

Including “Black-box” in ML:
Objectives like drug-likeness are governed by physical law, which are assumed to be unknown to us!
Idea: Reinforcement Learning

- A ML agent **observes** the environment, takes an **action** to interact with the environment, and receives positive or negative **reward**
- The agent then **learns from this loop**
- **Key idea**: Agent can directly learn from environment, which is a **blackbox** to the agent
Graph Convolutional Policy Network (GCPN) combines graph representation + RL

Key component of GCPN:

- **Graph Neural Network** captures graph structural information
- **Reinforcement learning** guides the generation towards the desired objectives
- **Supervised training** imitates examples in given datasets
Commonality of GCPN & GraphRNN:
- Generate graphs sequentially
- Imitate a given graph dataset

Main Differences:
- GCPN uses **GNN** to predict the generation action
  - **Pros**: GNN is more expressive than RNN
  - **Cons**: GNN takes longer time to compute than RNN
- GCPN further uses **RL** to direct graph generation to our goals
  - **RL** enables goal-directed graph generation
Sequential graph generation
- **GraphRNN**: predict action based on RNN hidden states

![GraphRNN diagram](image)

- **GCPN**: predict action based on GNN node embeddings

![GCPN diagram](image)

Recall the link prediction head:
\[
\text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}) := \text{Linear}(\text{Concat}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}))
\]
Overview of GCPN

- **(a)** Insert nodes
- **(b,c)** Use GNN to predict which nodes to connect
- **(d)** Take action (check chemical validity)
- **(e, f)** Compute reward
How Do We Set the Reward?

- **Step reward**: Learn to take valid action
  - At each step, assign small positive reward for valid action

- **Final reward**: Optimize desired properties
  - At the end, assign positive reward for high desired property

**Reward** = **Final reward** + **Step reward**
Two parts:

(1) Supervised training: Train policy by imitating the action given by real observed graphs. Use gradient.
   - We have covered this idea in GraphRNN

(2) RL training: Train policy to optimize rewards. Use standard policy gradient algorithm
   - Refer to any RL course, e.g., CS234
Training GCPN

Graph $G_t$ → GCPN → Gradient

- **Query dataset** → Real graph $G_{t+1}^*$ → Cross entropy loss
- **Generated graph $G_{t+1}$** → RL Training
- **Environment** → Final reward
- **Policy gradient** → Supervised Training

- Cross entropy loss: $0.6$
- Step reward: $0.1$
- Final reward: $1$

- Graph $G_t$:
  - C

- Generated graph $G_{t+1}$:
  - C
  - N

- Real graph $G_{t+1}^*$:
  - C

- Environment:
  - C
  - N
  - N
  - C

- Supervised Training:
  - Cross entropy loss

- RL Training:
  - Step reward
  - Final reward
Visualization of GCPN graphs:

- **Property optimization** Generate molecules with high specified property score

(a) Penalized logP optimization

- 7.98
- 7.48
- 7.12
- 23.88*

(b) QED optimization

- 0.948
- 0.945
- 0.944
- 0.941
Visualization of GCPN graphs:

- **Constrained optimization:** Edit a given molecule for a few steps to achieve higher property score

Starting structure

![Starting structure](image1.png)

-8.32

![Starting structure](image2.png)

-5.55

Increase the solubility in octanol

Finished structure

![Finished structure](image3.png)

-0.71

![Finished structure](image4.png)

-1.78

(c) Constrained optimization of penalized logP
Complex graphs can be successfully generated via sequential generation using deep learning.

Each step a decision is made based on hidden state, which can be

- Implicit: vector representation, decode with RNN
- Explicit: intermediate generated graphs, decode with GCN

Possible tasks:

- Imitating a set of given graphs
- Optimizing graphs towards given goals