📈

Formal Statement - 2

Tags	gen LB
Date	@13/04/2023

Statement

Performing metadynamics simulation by using multi-molecular molecular embedding obtained in latent space of **hierarchical graph auto-encoder** as a collective variable.

Encoding

1) Encoding molecule as a graph

Legends

- d → Number of input features for each node in the input
- n^l → Number of nodes in current step/layer
- n^{l+1} → Number of nodes in the next step/layer
- n^0 → Number of nodes initially → number of atoms
- $A \in R^{n_0 \times n_0}$ → Input adjacency matrix
- $X \in R^{n_0 \times d}$ → Input node feature matrix

Conversion

- A molecule can be converted to a graph by the following method.
    - We can represent each molecule as a graph with,
        - Nodes→ (position(x,y,z), radius, atom type, charge etc)
        - Edges → (bond length, bond type)

    - Now this graph can be represented as an Adjacency matrix
- Doing this we will have an Adjacency matrix $A^{n_0 \times n_0}$ and we will also have a node feature matrix $X^{n_0 \times d}$

2) Encoding whole multiple molecules

There are two primary ways to encode whole configurations which are described and used in the below two architecture.

Ideas for training

Currently, there are two approaches to this problem in my mind

- End to End training auto-encoder
- Coarse Graining encoders + Molecular configuration encoders

Note- Extensive details for each layer and types of layers which we can use are given in the Options for available layers section

1) End-to-End auto-encoder-(Ruled Out)

The idea is to consider the whole configuration of molecules as one big graph and try to regenerate this whole graph using an auto encoder decoder

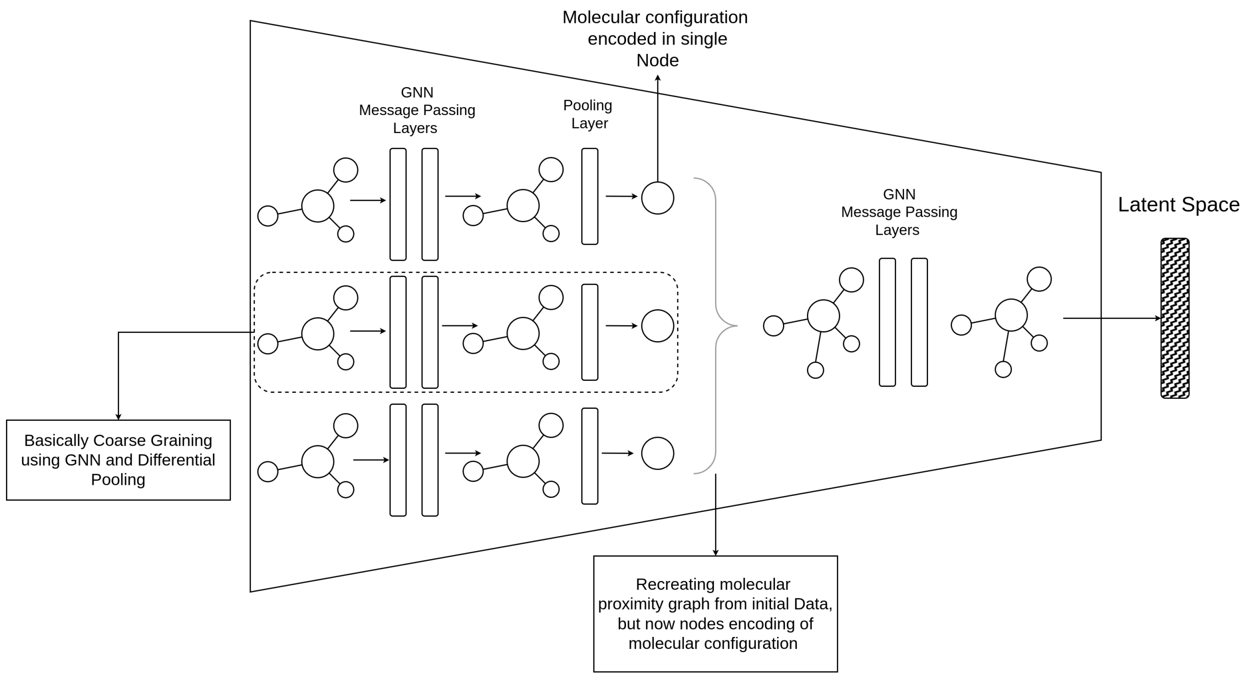
a) Coarse Graining/Representing molecules as a single node

Features for each atom

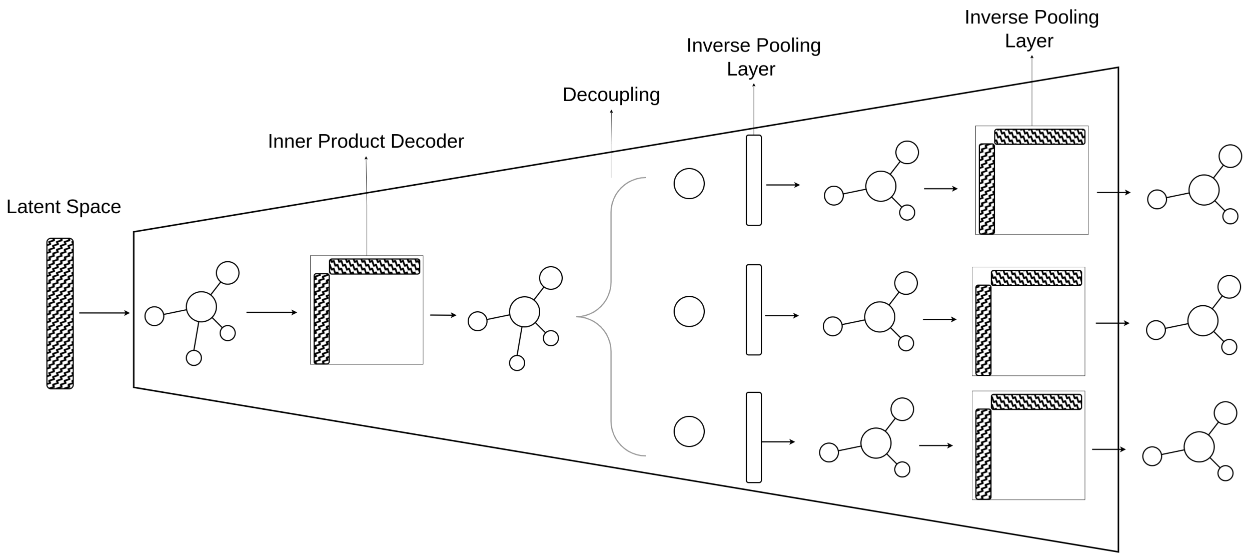
- Atom type
- Atomic radius
- (X, Y, Z) coordinates of atoms in Euclidean space.

b) Architecture

Encoder


Molecular configuration encoded in single Node
GNN Message Passing Layers
Pooling Layer
GNN Message Passing Layers
Latent Space
Basically Coarse Graining using GNN and Differential Pooling
Recreating molecular proximity graph from initial Data, but now nodes encoding of molecular configuration

Decoder



- First, we apply GNN on each molecule individually so that each atom can have information about the other atoms which are part of the same molecule.
- Next, we try to coarse grain/compress the graphs of each molecule individually by using pooling layers.

- After getting a coarse-grained/compressed representation for each molecule, we create a new graph using by following the given rules, this will be done in another layer let be called the **concatenation layer**
    - Use the coarse representation of each molecule as a node.
    - Two nodes will be connected if the distance between the center of mass of the parent molecules of these nodes lies within a threshold distance, let be D_t.
- Now we apply another GNN on this newly obtained graph so that each molecule can have some information about its nearby molecules.
- Another pooling layer can be applied on top of the output of this final GNN to get an even smaller latent space.

2) Coarse Graining encoders + Molecular configuration encoders

a) Coarse Graining encoders

Basically don't regenerate multiple molecular configurations but rather train two encoders, 1st for representing molecules and creating the molecular graphs for each.

The architecture for coarse grain auto-encoder can be similar to those stated above.

Some pre-existing architectures for the same are as follows

- **Compressed graph representation for scalable molecular graph generation** → https://jcheminf.biomedcentral.com/articles/10.1186/s13321-020-00463-2
- **MGCVAE: Multi-Objective Inverse Design via Molecular Graph Conditional Variational Autoencoder** → https://arxiv.org/pdf/2202.07476.pdf

b) Molecular configuration encoders

While creating the molecular graph add information about the position of the molecule, its velocity, momentum, etc so that we can minimize the information loss.

The architecture for Molecular configuration auto-encoders can be similar to those stated above.

The main objective of this encoder are as follows

- The latent space should be able to capture the formation of very small nuclei of crystal even though there formation is a rare event.
- The latent space for this auto encoder should be able to capture both crystal and melt phase accurately and both phase should have considerable difference in their latent space representation.

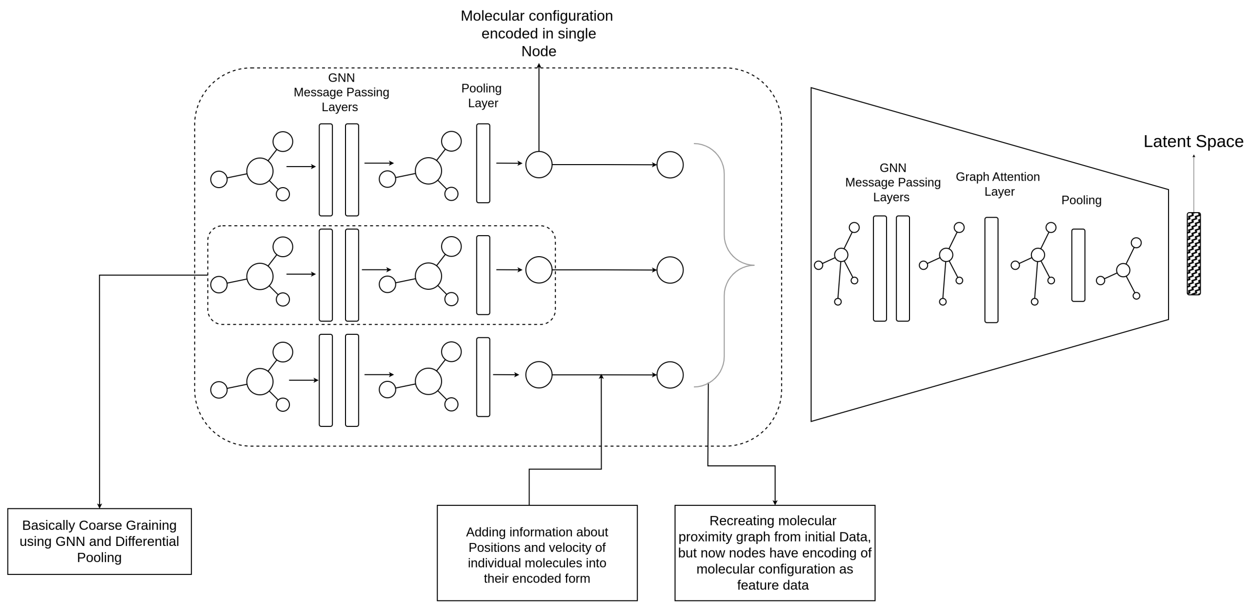
The main problem in above two objectives is to be able to capture the formation of small nuclei of crystal, now since this is a **rare event** and depends on only **small part** of a graph we can add a **self attention mechanism** over the molecular configuration graph to emphasize these events, hence we have added a Graph Attention Layer.

Now there are multiple ways to add attention to a graph based neural network but for us the significant ones are,

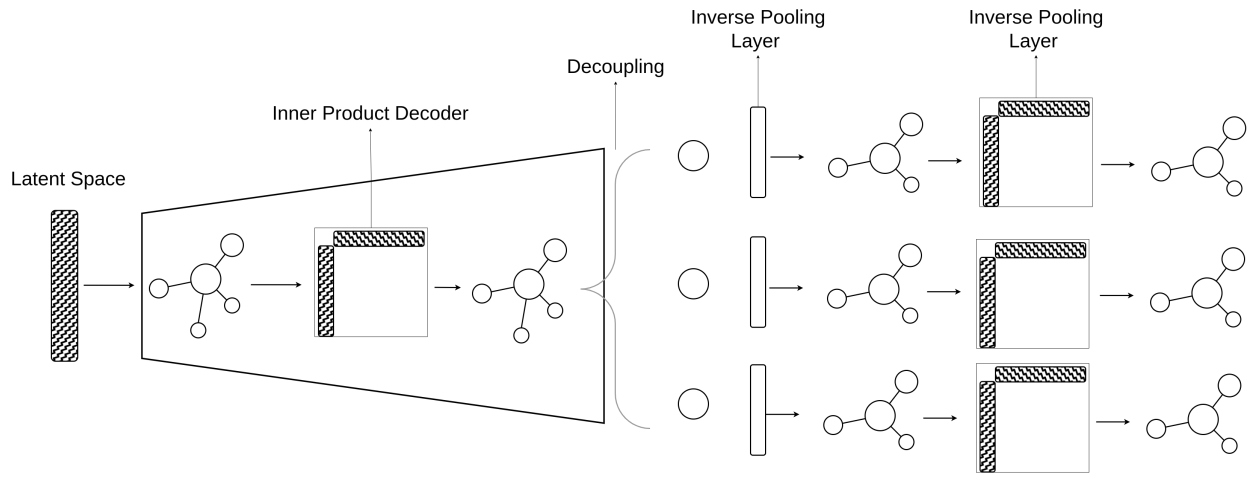
- Attention on Graph layer after message passing. https://arxiv.org/abs/1803.03735
- Attention based Pooling. https://arxiv.org/pdf/1904.08082.pdf

c) Complete Architecture

Encoder


Molecular configuration encoded in single Node
GNN Message Passing Layers
Pooling Layer
GNN Message Passing Layers
Graph Attention Layer
Pooling
Latent Space
Basically Coarse Graining using GNN and Differential Pooling
Adding information about Positions and velocity of individual molecules into their encoded form
Recreating molecular proximity graph from initial Data, but now nodes have encoding of molecular configuration as feature data

Decoder



Biasing

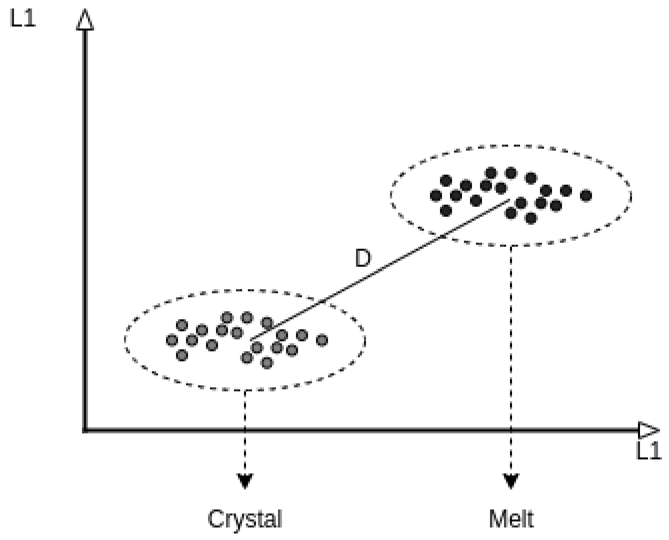
Now once we have our latent space, we need to figure out how we can bias our system in this latent space.

There are currently 2 methods at our disposal

- using **Unsupervised clustering algorithms**.
- Using a **Dense Neural Network** to predict biasing energy.

1) Unsupervised clustering algorithms

Once we have an effective encoding in latent space it can be assumed that the crystalline state and melt state will be part of two different clusters in latent space, for example, if we consider that our latent space is a **2-D** space, we will have



Legend

- **L1** → Latent Dimension 1
- **L2** → Latent Dimension 2
- **D** → Distance between **COM of crystal cluster** and **COM of Melt cluster.**

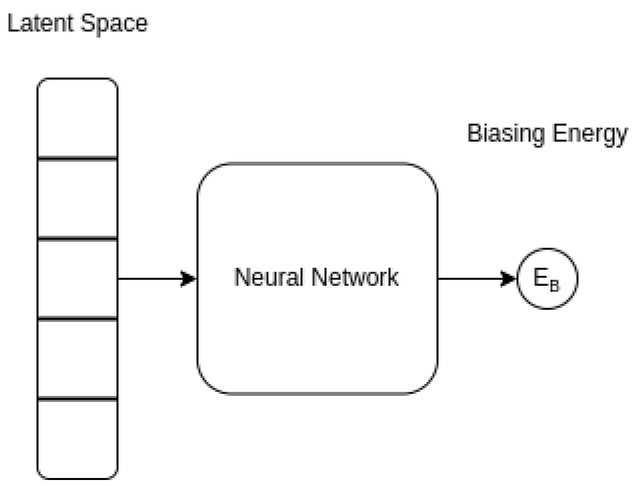
now we can use this distance **D** as a CV to be added to the potential function using metadynamics as follows

$$V(\vec{s},t) = \sum_{k\tau<t} W(k\tau) \exp\left(- \sum_{i=1}^{d} \frac{\left(\mathrm{D_i} - \mathrm{D_i}^{(0)}(k\tau)\right)^2}{2\sigma_i^2} \right)$$

2) Using a Dense Neural Network to predict biasing energy

The above-specified method might not work, because the free energy pathway from melt to crystal is not always a straight line so biasing along a straight line might not always be a good strategy.

We can overcome this issue by learning a mapping from encoded variables to biasing energy using a simple neural network as shown.



Doubt → From where will we get biasing energy to train this NN?

Options for available layers

# GNN Layers

1) Simple Message passing layer

Let h_i^k be the state of the i^{th} node at the k^{th} layer/stage, then we have the relation

$$h_i^{k+1} = Update(h_i^k + Agg(\{h_{j|\forall j \in Neighbour_i}\}))$$

Where $Neighbour_i$ is defined as nodes j such that there exists an edge between node i and node j or formally,

> $j \in Neighbour_i, \;\; if \; A[i][j]! = 0$

Note: By applying n such layers, each node will have some information about its n neighbors.

Now we are left with finding what we should use as our **Update** and **Aggregation** functions for which there are options like,

Update

- Mean
- Max
- Neural Network (Multiplying by a matrix and passing through an activation function and learning the parameters of this matrix)
- RNN (some papers have used an RNN for the same)

Aggregation

- Normalized Sum
- Mean
- Max
- Neural Network (Multiplying by a matrix and passing through an activation function and learning the parameters of this matrix)

It would be best if we use a simple neural network which is usually adopted, the parameters for this NN will be learned during the back propagation.

2) Attention based Message passing layer

We can add attention mechanism to simple message passing algorithm by assigning learnable weights α_i to the inputs of Agg function and $update$ function, making the update rule as,

$$h_i^{k+1} = Update(\alpha_i \cdot h_i^k + Agg(\{\alpha_{i,j} \cdot h_{j|\forall j \in Neighbour_i}\}))$$

Where $Neighbour_i$ is defined as nodes j such that there exists an edge between node i and node j or formally,

> $j \in Neighbour_i, \ \ if \ A[i][j]! = 0$

# Reverse GNN Layers

Most of the papers use the **inner product of the latent space** as the decoder part for the graph-based auto-encoder defined as

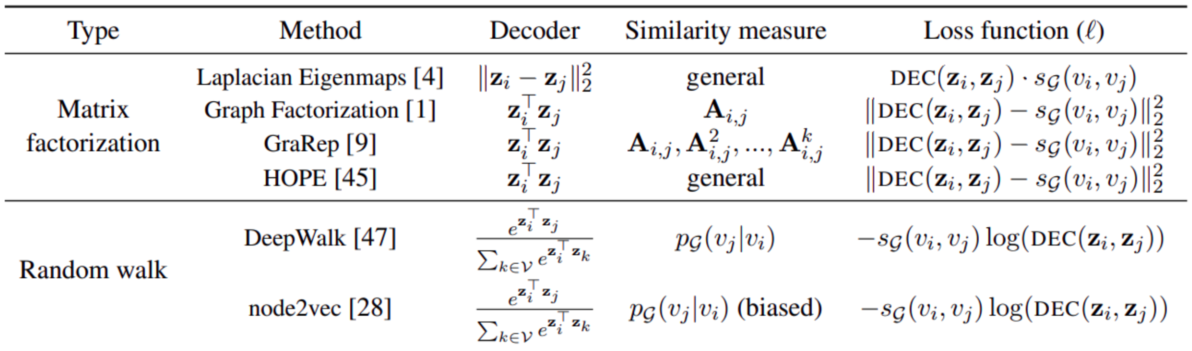
$$\text{decoded output} = Z_i^T \cdot Z_j$$

Later this output can be passed through a softmax function to get probability distributions as follows

$$\text{decoded output} = \sigma(Z_i^T \cdot Z_j)$$

Some other options are

Paper - Representation Learning on Graphs: Methods and Applications (https://arxiv.org/pdf/1709.05584.pdf)



Type	Method	Decoder	Similarity measure	Loss function (ℓ)
Matrix factorization	Laplacian Eigenmaps [4]	$\lVert \mathbf{z}_i - \mathbf{z}_j \rVert_2^2$	general	$\text{DEC}(\mathbf{z}_i, \mathbf{z}_j) \cdot s_{\mathcal{G}}(v_i, v_j)$
	Graph Factorization [1]	$\mathbf{z}_i^\top \mathbf{z}_j$	$\mathbf{A}_{i,j}$	$\lVert \text{DEC}(\mathbf{z}_i, \mathbf{z}_j) - s_{\mathcal{G}}(v_i, v_j) \rVert_2^2$
	GraRep [9]	$\mathbf{z}_i^\top \mathbf{z}_j$	$\mathbf{A}_{i,j}, \mathbf{A}_{i,j}^2, ..., \mathbf{A}_{i,j}^k$	$\lVert \text{DEC}(\mathbf{z}_i, \mathbf{z}_j) - s_{\mathcal{G}}(v_i, v_j) \rVert_2^2$
	HOPE [45]	$\mathbf{z}_i^\top \mathbf{z}_j$	general	$\lVert \text{DEC}(\mathbf{z}_i, \mathbf{z}_j) - s_{\mathcal{G}}(v_i, v_j) \rVert_2^2$
Random walk	DeepWalk [47]	$\frac{e^{\mathbf{z}_i^\top \mathbf{z}_j}}{\sum_{k \in \mathcal{V}} e^{\mathbf{z}_i^\top \mathbf{z}_k}}$	$p_{\mathcal{G}}(v_j \| v_i)$	$-s_{\mathcal{G}}(v_i, v_j) \log(\text{DEC}(\mathbf{z}_i, \mathbf{z}_j))$
	node2vec [28]	$\frac{e^{\mathbf{z}_i^\top \mathbf{z}_j}}{\sum_{k \in \mathcal{V}} e^{\mathbf{z}_i^\top \mathbf{z}_k}}$	$p_{\mathcal{G}}(v_j \| v_i)$ (biased)	$-s_{\mathcal{G}}(v_i, v_j) \log(\text{DEC}(\mathbf{z}_i, \mathbf{z}_j))$

# Graph Pooling Layers

1) Simple Graph Pooling Layer

For every graph pooling layer l, we try to learn a cluster assignment matrix $S^l \in R^{n_l \times n_{l+1}}$, the output for this layer is given as

$\rightarrow A^{l+1} = (S^l)^T \cdot Z^l \cdot (S^l) \qquad \textit{which belongs to } R^{n_{l+1} \times n_{l+1}}$

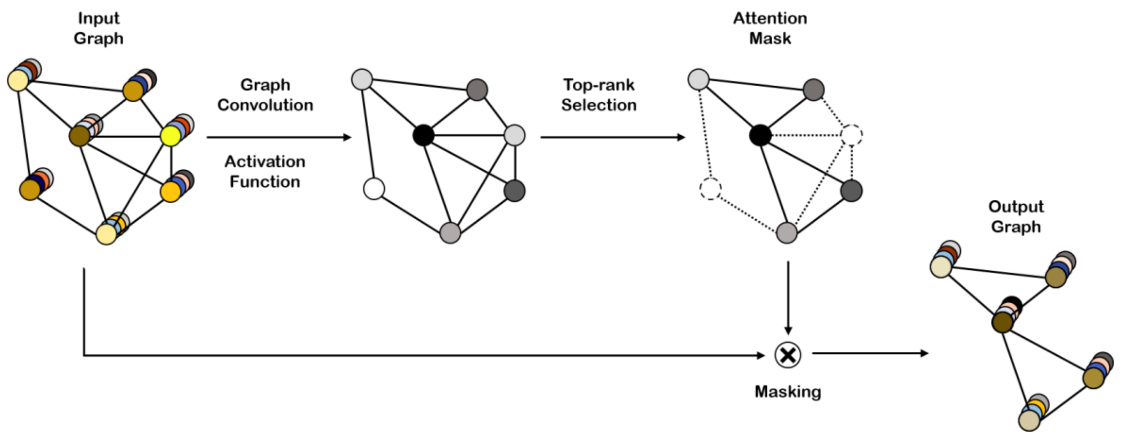
$\rightarrow X^{l+1} = (S^l)^T \cdot Z^l \qquad \textit{which belongs to } R^{n_{l+1} \times d}$

This method is followed by the paper **Hierarchical Graph Representation Learning with Differentiable Pooling** (DiffPool[2018]) (https://arxiv.org/abs/1806.08804)

2) Self-Attention based Graph Pooling

Rather than applying a simple graph pooling layer to reduce the graph size, we can use a method called self-attention-based graph pooling, in this method, pooling is done by creating a mask layer based on nodes features.

paper- Self-Attention Graph Pooling - https://arxiv.org/pdf/1904.08082.pdf



The attention mask is obtained by first calculating Z using the formula

$$Z = \sigma(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} X \Theta_{att})$$

And then using the top indices of Z for calculating the mask

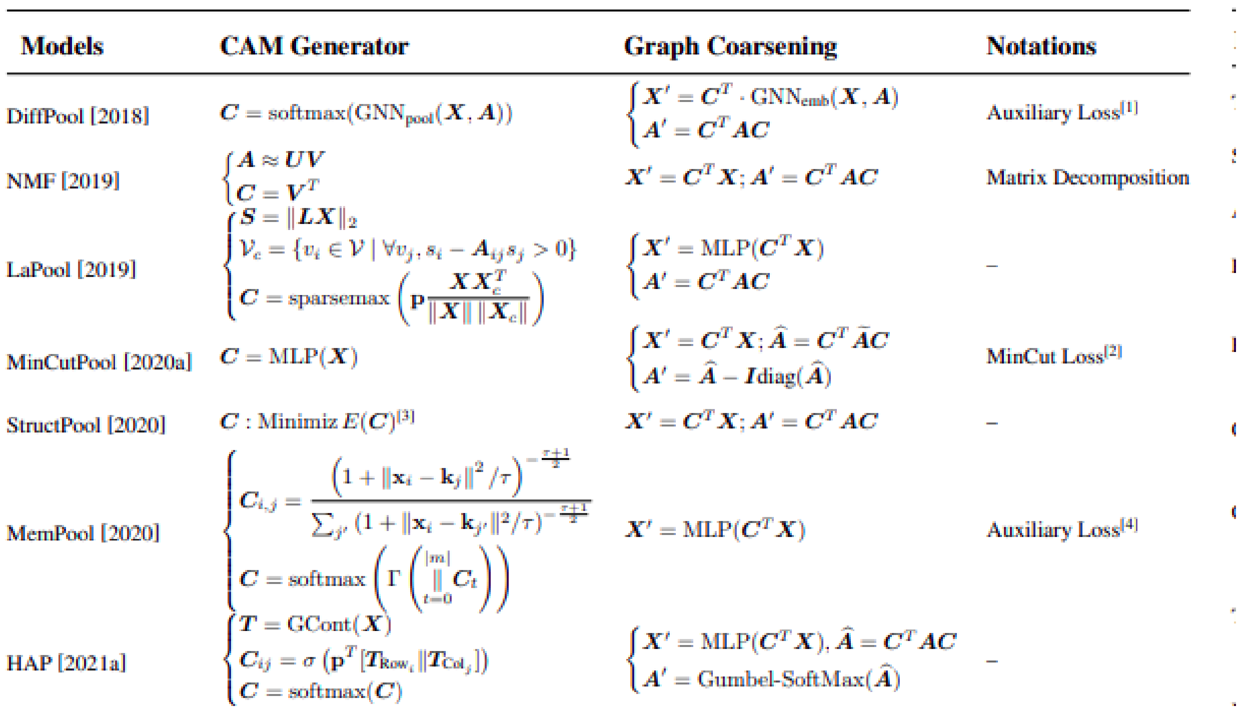
$$\mathrm{idx} = \text{top-rank}(Z, \lceil kN \rceil), \quad Z_{mask} = Z_{\mathrm{idx}}$$

The inspiration behind using attention-based pooling layers is that we need to be able to capture the formation of small crystal nuclei and this formation should be reflected in the latent space for effective biasing.

Note - We don't really need to use multi-headed attention because the formation of a crystal nucleus is already a rare event and the probability of more than 2 nuclei getting formed is very rare.

3) Some other methods are described in Graph Coarsening column here,

Paper- Graph Pooling for Graph Neural Networks: Progress, Challenges, and Opportunities (https://arxiv.org/pdf/2204.07321.pdf)



Models	CAM Generator	Graph Coarsening	Notations
DiffPool [2018]	$C = \mathrm{softmax}(\mathrm{GNN}_{\mathrm{pool}}(X, A))$	$\begin{cases} X' = C^T \cdot \mathrm{GNN}_{\mathrm{emb}}(X, A) \\ A' = C^T AC \end{cases}$	Auxiliary Loss[1]
NMF [2019]	$\begin{cases} A \approx UV \\ C = V^T \end{cases}$	$X' = C^T X; A' = C^T AC$	Matrix Decomposition
LaPool [2019]	$\begin{cases} S = \Vert LX \Vert_2 \\ \mathcal{V}_c = \{v_i \in \mathcal{V} \mid \forall v_j, s_i - A_{ij} s_j > 0\} \\ C = \mathrm{sparsemax}\left(\mathbf{p} \frac{X X_c^T}{\Vert X \Vert \, \Vert X_c \Vert}\right) \end{cases}$	$\begin{cases} X' = \mathrm{MLP}(C^T X) \\ A' = C^T AC \end{cases}$	–
MinCutPool [2020a]	$C = \mathrm{MLP}(X)$	$\begin{cases} X' = C^T X; \widehat{A} = C^T \tilde{A} C \\ A' = \widehat{A} - I\mathrm{diag}(\widehat{A}) \end{cases}$	MinCut Loss[2]
StructPool [2020]	C : Minimiz $E(C)$[3]	$X' = C^T X; A' = C^T AC$	–
MemPool [2020]	$\begin{cases} C_{i,j} = \frac{\left(1 + \Vert \mathbf{x}_i - \mathbf{k}_j \Vert^2 / \tau\right)^{-\frac{\tau+1}{2}}}{\sum_{j'} (1 + \Vert \mathbf{x}_i - \mathbf{k}_{j'} \Vert^2 / \tau)^{-\frac{\tau+1}{2}}} \\ C = \mathrm{softmax}\left(\Gamma\left(\Vert_{t=0}^{\lvert m \rvert} C_t\right)\right) \end{cases}$	$X' = \mathrm{MLP}(C^T X)$	Auxiliary Loss[4]
HAP [2021a]	$\begin{cases} T = \mathrm{GCont}(X) \\ C_{ij} = \sigma\left(\mathbf{p}^T [T_{\mathrm{Row}_i} \Vert T_{\mathrm{Col}_j}]\right) \\ C = \mathrm{softmax}(C) \end{cases}$	$\begin{cases} X' = \mathrm{MLP}(C^T X), \widehat{A} = C^T AC \\ A' = \mathrm{Gumbel\text{-}SoftMax}(\widehat{A}) \end{cases}$	–

# Reverse Graph Pooling Layers

Simple reverse graph pooling layer

A simple graph pooling layer is just an inner product with a cluster assignment matrix $S^l \in R^{n_l \times n_{l+1}}$ so we can simply just create another matrix $S_{inv} \in R^{n_{l+1} \times n_l}$ and take the product with it to get our original output

More formally,

$\rightarrow A^{l+1} = (S^l_{inv})^T \cdot Z^l \cdot (S^l_{inv})$

$\rightarrow X^{l+1} = (S^l_{inv})^T \cdot Z^l$

# Concatenation layer

1) Simple concatenation Layer

The concatenation layer basically generates a new graph

- Use the coarse representation of each molecule as a node.
- Two nodes will be connected if the distance between the center of mass of the parent molecules of these nodes lies within a threshold distance, let be D_t.

Note- Another important objective of concatenation layer is to add the position and velocity data of each molecule into its compressed node representation.

Formally,

we construct a graph $G = \{V, E\}$ such that,

$V = \{\text{coarse grained representation} + \text{Position} + \text{Velocity}\}$

$E = \{\text{1 if two dist(n1,n2)} < D_t \text{ ; else 0}\}$

2) Attention based concatenation Layer

Again the objective of this layer is to construct a new graph using the coarse representation of each molecule as a node but this time rather than having binary value 0 or 1 for representing an edge we add a weighted learnable coefficient α_i which signifies how likely molecule i is to be a part of a crystal.

Apart for that we will still add the position and velocity data of each molecule into its compressed node representation.

Formally,

we construct a graph $G = \{V, E\}$ such that,

$V = \{\text{coarse grained representation} + \text{Position} + \text{Velocity}\}$

$E = \{\alpha_{i,j}\}$

This E matrix can be treated as a learnable parameter.

# Loss functions

1) Reconstruction Loss

For the loss function, we can use the reconstruction loss given as

$$L = E_{q(Z|X,A)}[logp(A|Z)]$$

2) Reconstruction Loss + KL Divergence

we can also ad KL divergence for regularization into the reconstruction loss

$$L = E_{q(Z|X,A)}[logp(A|Z)] - KL[q(Z|X,A)||p(Z)]$$