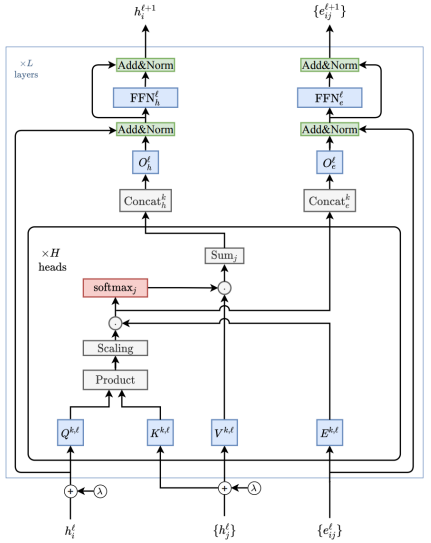
Outline

- Review of graph network architectures
- Graph positional encoding
- Transformers for sequences
- **Graph Transformers**
- ViT/MLP-Mixer for images
- Graph ViT/MLP-Mixer
- Conclusion

Graph Transformers[1]

- Contributions
  - Generalizing Transformers[2,3] to arbitrary graphs with O(E+N) complexity.
  - Leveraging graph as topological inductive bias.
  - Generalizing cos/sin positional encoding (PE) to graphs (node ordering) with Laplacian eigenvectors.
  - Introducing edge feature through bilinear product (for e.g. molecular bound or relationship between entities in knowledge graph).
  - Graph Transformer (GT) and its improvements[4] have gained popularity as a backbone architecture in biology (used by winner and runner-up at NeurIPS'22 OGB Large-Scale Challenge competition[5])
- Limitations
  - Over-squashing and poor long-range dependency
  - Small graphs



[1] Dwivedi, Bresson, A generalization of transformer networks to graphs, AAAI 2021
[2] Bahdanau, Cho, Bengio, Neural machine translation by jointly learning to align and translate, 2014
[3] Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, Polosukhin, Attention is all you need, 2017
[4] Ying, Cai, Luo, Zheng, Ke, He, Shen, Liu, Do Transformers Really Perform Bad for Graph Representation, 2021
[5] https://ogb.stanford.edu/neurips2022/workshop

Transformers[1]

- **Standard Transformer layer (for sequences[1] w/PE, grids/images[2], sets wo/PE) :**

layer normalization — seq length / #pixels / #set elements

$$\bar{h}^\ell = \mathrm{LN}\left(h^\ell + \mathrm{MHA}(h^\ell)\right) \in \mathbb{R}^{N\times d}$$

$$h^{\ell+1} = \mathrm{LN}\left(\bar{h}^\ell + \mathrm{MLP}(\bar{h}^\ell)\right) \in \mathbb{R}^{N\times d}$$

Multi-head attention

$$\text{with } \mathrm{MHA}(h) = \mathrm{Concat}_{k=1}^{H}\left(\mathrm{HA}(h_k)\right) W_O \in \mathbb{R}^{N\times d},\ h_k \in \mathbb{R}^{N\times d'=d/H}, W_O \in \mathbb{R}^{d\times d}$$

Head attention

$$\mathrm{HA}(h) = \mathrm{Softmax}\left(\frac{QK^T}{\sqrt{d'}}\right) V \in \mathbb{R}^{N\times d'=d/H}$$

Q: query, K: key, V: value

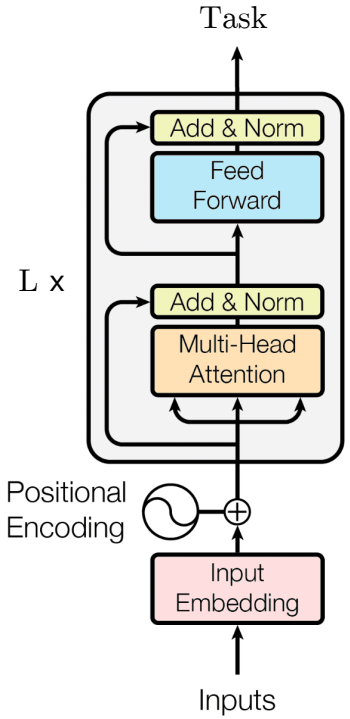
$$\mathrm{HA}(h)_i = \sum_{j\in N} \underbrace{\frac{\exp(q_i^T k_j/\sqrt{d'})}{\sum_{j'\in N}\exp(q_i^T k_{j'}/\sqrt{d'})}}_{\text{attention score}_{ij}} v_j \quad \text{(point-wise equation)}$$

$$Q = h_k W_Q, K = h_k W_K, V = h_k W_V \in \mathbb{R}^{N\times d'=d/H}, W_Q, W_K, W_V \in \mathbb{R}^{d'\times d'}$$

$$h^{\ell=0} = \mathrm{LL}(h_0) + p_0 \in \mathbb{R}^{N\times d} \text{ (input word/patch and positional encoding)}$$

cos/sin PE

$$\text{with } (p_0)_{ik} = \begin{cases} \sin(2\pi f_k i) \text{ if } i \text{ even} \\ \cos(2\pi f_k i) \text{ if } i \text{ odd} \end{cases} \quad \text{with } f_k = \frac{10,000^{\frac{d}{\lfloor 2k \rfloor}}}{2\pi}$$



[1] Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, Polosukhin, Attention is all you need, 2017
[2] Dosovitskiy et-al, An image is worth 16x16 words: Transformers for image recognition at scale, 2020

Graph Transformers[1]

- **Graph Transformer layer (for arbitrary graph) :**

batch normalization, # nodes

$$\bar{h}^{\ell} = \mathrm{BN/LN}\left(h^{\ell} + \mathrm{gMHA}(h^{\ell})\right) \in \mathbb{R}^{N\times d}$$

$$h^{\ell+1} = \mathrm{BN/LN}\left(\bar{h}^{\ell} + \mathrm{MLP}(\bar{h}^{\ell})\right) \in \mathbb{R}^{N\times d}$$

$$\text{with } \mathrm{gMHA}(h) = \mathrm{Concat}_{k=1}^{H}\left(\mathrm{gHA}(h_k)\right) W_O \in \mathbb{R}^{N\times d},\ h_k \in \mathbb{R}^{N\times d'=d/H}, W_O \in \mathbb{R}^{d\times d}$$

Graph multi-head attention

Graph head attention

$$\mathrm{gHA}(h) = \mathrm{Softmax}\left(A_G \odot \frac{QK^T}{\sqrt{d'}}\right) V \in \mathbb{R}^{N\times d'=d/H},$$

$$A_G \in \mathbb{R}^{N\times N} \text{ (graph adjacency matrix)}$$

$$\mathrm{gHA}(h)_i = \sum_{j\in\mathcal{N}_i} \underbrace{\frac{\exp(q_i^T k_j/\sqrt{d'})}{\sum_{j'\in\mathcal{N}_i}\exp(q_i^T k_{j'}/\sqrt{d'})}}_{\text{graph attention score}_{ij}} v_j \quad \text{(point-wise equation)}$$

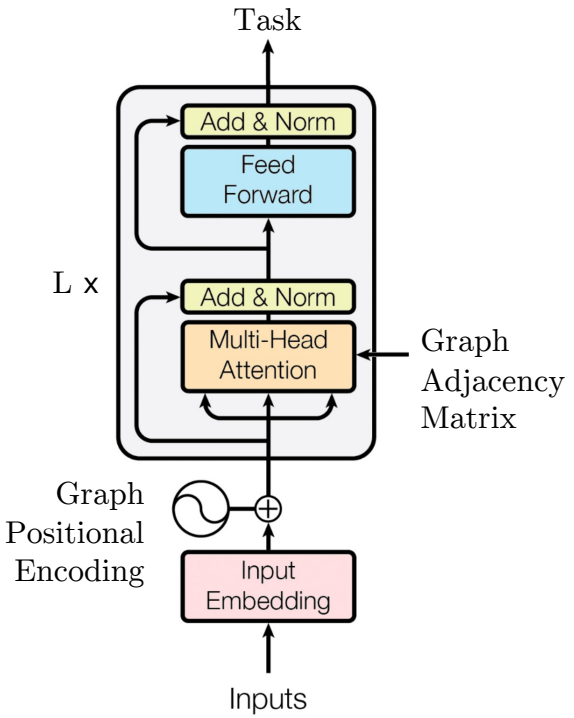
neighbors of node i

$$Q = h_k W_Q, K = h_k W_K, V = h_k W_V \in \mathbb{R}^{N\times d'=d/H}, W_Q, W_K, W_V \in \mathbb{R}^{d'\times d'}$$

$$h^{\ell=0} = \mathrm{LL}_1(h_0) + \mathrm{LL}_2(p_0) \in \mathbb{R}^{N\times d} \text{ (input node feature and positional encoding)}$$

$$\text{with } p_0 = \Phi_{\{2,..,K+1\}} \in \mathbb{R}^{N\times K},\ \Delta = \Phi\Lambda\Phi^T \in \mathbb{R}^{N\times N}$$

First non-trivial K eigenvectors; Laplacian; Eigenvectors / eigenvalues



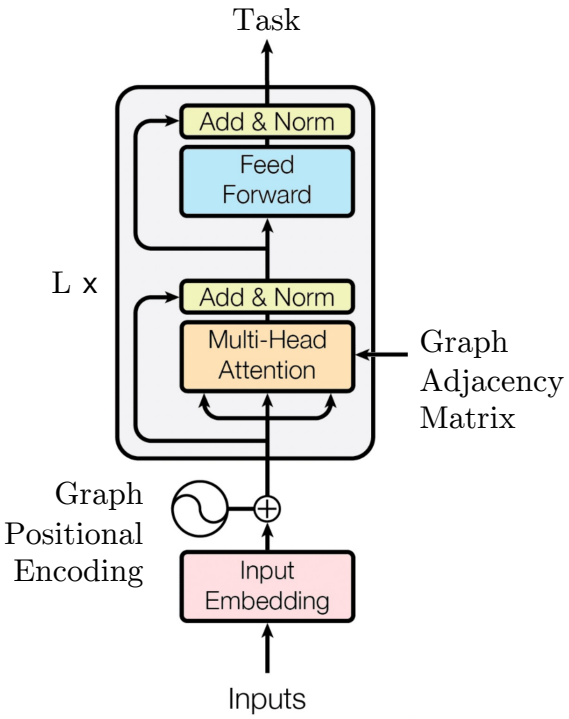
[1] Dwivedi, Bresson, A generalization of transformer networks to graphs, 2020

Graph adjacency matrix

- **Definition of adjacency matrix A_G in the graph head attention function :**

$$\text{gHA}(h) = \text{Softmax}\left(A_G \odot \frac{QK^T}{\sqrt{d'}}\right) V \in \mathbb{R}^{N \times d'}$$

$$\text{with } A_{G\,ij} = \begin{cases} A_{ij}^{\text{original}} & \text{if } ij \text{ connected} \\ 1 & \text{for } j = i \text{ (enforcing self-loop)} \\ -\infty & \text{if } ij \text{ not connected} \end{cases}$$



- **Alternative formulation :**

$$\text{gHA}(h) = \text{Normalize}_{\text{dim=rows}}\left(A'_G \odot \exp\left(\frac{QK^T}{\sqrt{d'}}\right)\right) V \in \mathbb{R}^{N \times d'}$$

$$= A'_G \odot \exp\left(\frac{QK^T}{\sqrt{d'}}\right) \left[\left(A'_G \odot \exp\left(\frac{QK^T}{\sqrt{d'}}\right)\right) \mathbf{1}_{N \times N}\right]^{-1} V \in \mathbb{R}^{N \times d'}$$

$$\text{with } A'_{G\,ij} = \begin{cases} A_{ij}^{\text{original}} & \text{if } ij \text{ connected} \\ 1 & \text{for } j = i \text{ (enforcing self-loop)} \end{cases}$$

Lab 1 : Vanilla Graph Transformers

- GT without graph PE : Run “01_vanilla_graph_transformers.ipynb”

jupyter 01_vanilla_graph_transformers Last Checkpoint: an hour ago (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 (ipykernel)

Run Code

Lab 01 : Vanilla Graph Transformers - demo

Dwivedi, Bresson, A generalization of transformer networks to graphs, 2020
https://arxiv.org/pdf/2012.09699.pdf

Xavier Bresson

Notebook goals :
- Implement vanilla graph transformers (i.e. no positional encoding) with DGL
- Compare performance of GT with GCN and GatedGCN

In [1]:
```
# For Google Colaboratory
import sys, os
if 'google.colab' in sys.modules:
    # mount google drive
    from google.colab import drive
    drive.mount('/content/gdrive')
    path_to_file = '/content/gdrive/My Drive/GML_May23_codes/codes/labs_lecture07'
    print(path_to_file)
    # change current path to the folder containing "path_to_file"
    os.chdir(path_to_file)
    !pwd
    !pip install dgl # Install DGL
```

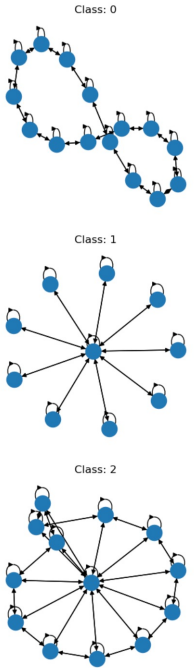
In [2]:
```
# Libraries
import dgl
from dgl.data import MiniGCDataset
import dgl.function as fn
import matplotlib.pyplot as plt
import networkx as nx
import torch
import torch.nn.functional as F
from torch.utils.data import DataLoader
import torch.nn as nn
import time
```

Visualize the artifical graph dataset used in this notebook

Train the network

In [7]:
```
def run_one_epoch(net, data_loader, train=True):
    if train:
        net.train() # during training
    else:
        net.eval()  # during inference/test
    epoch_loss = 0
    epoch_acc = 0
    nb_data = 0
    gpu_mem = 0
    for iter, (batch_graphs, batch_labels) in enumerate(data_loader):
        batch_x = batch_graphs.ndata['feat']
        batch_labels = batch_labels
        batch_scores = net.forward(batch_graphs, batch_x)
        loss = net.loss(batch_scores, batch_labels)
        if train: # during training, run backpropagation
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        epoch_loss += loss.detach().item()
        epoch_acc += net.accuracy(batch_scores, batch_labels)
        nb_data += batch_labels.size(0)
    epoch_loss /= (iter + 1)
    epoch_acc /= nb_data
    return epoch_loss, epoch_acc


# dataset loaders
train_loader = DataLoader(trainset, batch_size=50, shuffle=True, collate_fn=collate)
test_loader = DataLoader(testset, batch_size=50, shuffle=False, collate_fn=collate)
val_loader = DataLoader(valset, batch_size=50, shuffle=False, drop_last=False, collate_fn=collate)

# Instantiate one network
net_parameters = {}
net_parameters['input_dim'] = 1
net_parameters['hidden_dim'] = 128
net_parameters['output_dim'] = 8 # nb of classes
net_parameters['num_heads'] = 8
net_parameters['norm'] = 'BN'
#net_parameters['norm'] = 'LN'
net_parameters['L'] = 4
net = GraphTransformer_net(net_parameters)

# optimizer
optimizer = torch.optim.Adam(net.parameters(), lr=0.0001)

# training loop
for epoch in range(50):
    start = time.time()
    epoch_train_loss, epoch_train_acc = run_one_epoch(net, train_loader, True)
    with torch.no_grad():
        epoch_test_loss, epoch_test_acc = run_one_epoch(net, test_loader, False)
        epoch_val_loss, epoch_val_acc = run_one_epoch(net, val_loader, False)
    print('Epoch {}, time {:.4f}, train_loss: {:.4f}, test_loss: {:.4f}, val_loss: {:.4f}'.format(epoch, time.time()-st
    print('                  train_acc: {:.4f}, test_acc: {:.4f}, val_acc: {:.4f}'.format(epoch_train_acc, epoch_te
```

```
Epoch 0, time 2.1793, train_loss: 2.0309, test_loss: 2.0326, val_loss: 2.0326
                      train_acc: 0.2600, test_acc: 0.2800, val_acc: 0.2800
Epoch 1, time 1.5107, train_loss: 1.9360, test_loss: 1.9336, val_loss: 1.9336
                      train_acc: 0.4514, test_acc: 0.4900, val_acc: 0.4900
Epoch 2, time 1.6516, train_loss: 1.8630, test_loss: 1.8500, val_loss: 1.8500
                      train_acc: 0.4971, test_acc: 0.5200, val_acc: 0.5200
```



Lab 2 : Graph Transformers

- GT with Laplacian PE : Run “02_graph_transformers_classification_exercise.ipynb”

jupyter 02_graph_transformers_classification_solution Last Checkpoint: an hour ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted Python 3 (ipykernel)

Run Code

Lab 02 : Graph Transformers with positional encoding for classification - solution

Dwivedi, Bresson, A generalization of transformer networks to graphs, 2020
https://arxiv.org/pdf/2012.09699.pdf

Xavier Bresson

Notebook goals :
• Implement graph transformers with positional encoding (PE)
• Compare performance with and without PE (i.e. vanilla GT)

In [1]:
```
# For Google Colaboratory
import sys, os
if 'google.colab' in sys.modules:
    # mount google drive
    from google.colab import drive
    drive.mount('/content/gdrive')
    path_to_file = '/content/gdrive/My Drive/GML_May23_codes/codes/labs_lecture07'
    print(path_to_file)
    # change current path to the folder containing "path_to_file"
    os.chdir(path_to_file)
    !pwd
    !pip install dgl # Install DGL
```

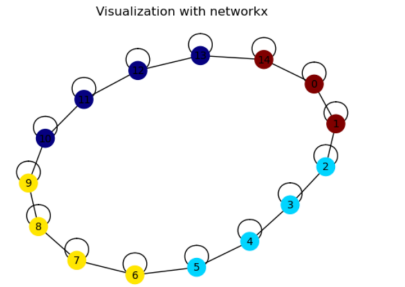
In [2]:
```
# Libraries
import dgl
from dgl.data import MiniGCDataset
import dgl.function as fn
import matplotlib.pyplot as plt
import networkx as nx
import torch
import torch.nn.functional as F
from torch.utils.data import DataLoader
import torch.nn as nn
import time
from utils import compute_ncut
```

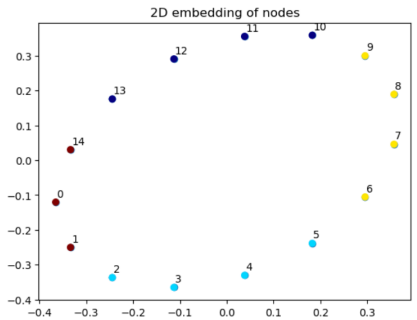
Visualize the artifical graph dataset used in this notebook

Train the network

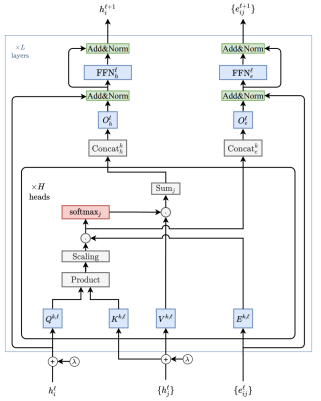
In [8]:
```
def run_one_epoch(net, data_loader, train=True):
    if train:
        net.train() # during training
    else:
        net.eval()  # during inference/test
    epoch_loss = 0
    epoch_acc = 0
    nb_data = 0
    gpu_mem = 0
    for iter, (batch_graphs, batch_labels) in enumerate(data_loader):
        batch_x = batch_graphs.ndata['feat']
        batch_pe = batch_graphs.ndata['pos_enc']
        batch_pe = batch_pe * ( 2 * torch.randint(low=0, high=2, size=(1,pos_enc_dim)).float() - 1.0 ) # randomly flip
        batch_labels = batch_labels
        batch_scores = net.forward(batch_graphs, batch_x, batch_pe)
        loss = net.loss(batch_scores, batch_labels)
        if train: # during training, run backpropagation
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        epoch_loss += loss.detach().item()
        epoch_acc += net.accuracy(batch_scores,batch_labels)
        nb_data += batch_labels.size(0)
    epoch_loss /= (iter + 1)
    epoch_acc /= nb_data
    return epoch_loss, epoch_acc


# dataset loaders
train_loader = DataLoader(trainset, batch_size=50, shuffle=True, collate_fn=collate)
test_loader = DataLoader(testset, batch_size=50, shuffle=False, collate_fn=collate)
val_loader = DataLoader(valset, batch_size=50, shuffle=False, drop_last=False, collate_fn=collate)

# Instantiate one network
net_parameters = {}
net_parameters['input_dim'] = 1
net_parameters['pos_enc_dim'] = pos_enc_dim
net_parameters['hidden_dim'] = 128
net_parameters['output_dim'] = 8 # nb of classes
net_parameters['num_heads'] = 8
net_parameters['norm'] = 'BN'
net_parameters['L'] = 4
del net
net = GraphTransformer_net(net_parameters)

# optimizer
optimizer = torch.optim.Adam(net.parameters(), lr=0.0003)

# training loop
start = time.time()
for epoch in range(50):
    epoch_train_loss, epoch_train_acc = run_one_epoch(net, train_loader, True)
    with torch.no_grad():
        epoch_test_loss, epoch_test_acc = run_one_epoch(net, test_loader, False)
        epoch_val_loss, epoch_val_acc = run_one_epoch(net, val_loader, False)
    print('Epoch {}, time {:.4f}, train_loss: {:.4f}, test_loss: {:.4f}, val_loss: {:.4f}'.format(epoch, time.time()-st
    print('                    train_acc: {:.4f}, test_acc: {:.4f}, val_acc: {:.4f}'.format(epoch_train_acc, epoch_te
```

```
Epoch 0, time 1.4386, train_loss: 2.0601, test_loss: 2.0812, val_loss: 2.0811
                      train_acc: 0.1457, test_acc: 0.1200, val_acc: 0.1200
Epoch 1, time 3.0223, train_loss: 2.0042, test_loss: 2.0674, val_loss: 2.0680
                      train_acc: 0.3971, test_acc: 0.1300, val_acc: 0.1300
```





Graph Transformers with edge feature[1]



- **Node feature update :**

Edge feature

$$\bar{h}^{\ell} = \text{BN/LN}\left(h^{\ell} + \text{gMHA}(h^{\ell}, e^{\ell})\right) \in \mathbb{R}^{N\times d}$$

$$h^{\ell+1} = \text{BN/LN}\left(\bar{h}^{\ell} + \text{MLP}(\bar{h}^{\ell})\right) \in \mathbb{R}^{N\times d}$$

$$\text{with gMHA}(h,e) = \text{Concat}_{k=1}^{H}\left(\text{gHA}(h_k, e_k)\right) W_O \in \mathbb{R}^{N\times d},\ h_k \in \mathbb{R}^{N\times d'=d/H}, e_k \in \mathbb{R}^{E\times d'}, W_O \in \mathbb{R}^{d\times}$$

$$\text{gHA}(h,e)_i = \sum_{j\in\mathcal{N}_i} \underbrace{\frac{\exp(q_i^T \text{diag}(e_{ij}) k_j/\sqrt{d'})}{\sum_{j'\in\mathcal{N}_i}\exp(q_i^T \text{diag}(e_{ij'}) k_{j'}/\sqrt{d'})}}_{\text{graph attention score w/ edge feature}_{ij}} v_j \quad \text{(point-wise equation)}$$

Bilinear product w/ node and edge features

$$Q = h_k W_Q, K = h_k W_K, V = h_k W_V \in \mathbb{R}^{N\times d'=d/H}, E = e_k W_E \in \mathbb{R}^{E\times d'=d/H}, W_Q, W_K, W_V, W_I$$

$$h^{\ell=0} = \text{LL}_1(h_0) + \text{LL}_2(p_0) \in \mathbb{R}^{N\times d} \text{ (input node feature and positional encoding)}$$

$$\text{with } p_0 = \Phi_{\{2,..,K+1\}} \in \mathbb{R}^{N\times K},\ \Delta = \Phi\Lambda\Phi^T \in \mathbb{R}^{N\times N}$$

- **Edge feature update :**

$$\bar{e}^{\ell} = \text{BN/LN}\left(e^{\ell} + \text{gMHE}(e^{\ell}, h^{\ell})\right) \in \mathbb{R}^{E\times d}$$

$$e^{\ell+1} = \text{BN/LN}\left(\bar{e}^{\ell} + \text{MLP}(\bar{e}^{\ell})\right) \in \mathbb{R}^{E\times d}$$

$$\text{with gMHE}(e,h) = \text{Concat}_{k=1}^{H}\left(\text{gHE}(e_k, h_k)\right) W_O^e \in \mathbb{R}^{E\times d},\ h_k \in \mathbb{R}^{N\times d'=d/H}, e_k \in \mathbb{R}^{E\times d'}, W_O^e \in \mathbb{R}^{d\times}$$

$$\text{with gHE}(e,h)_{ij} = q_i \odot e_{ij} \odot k_j/\sqrt{d'} \in \mathbb{R}^{d'} \text{ (point-wise equation)}$$

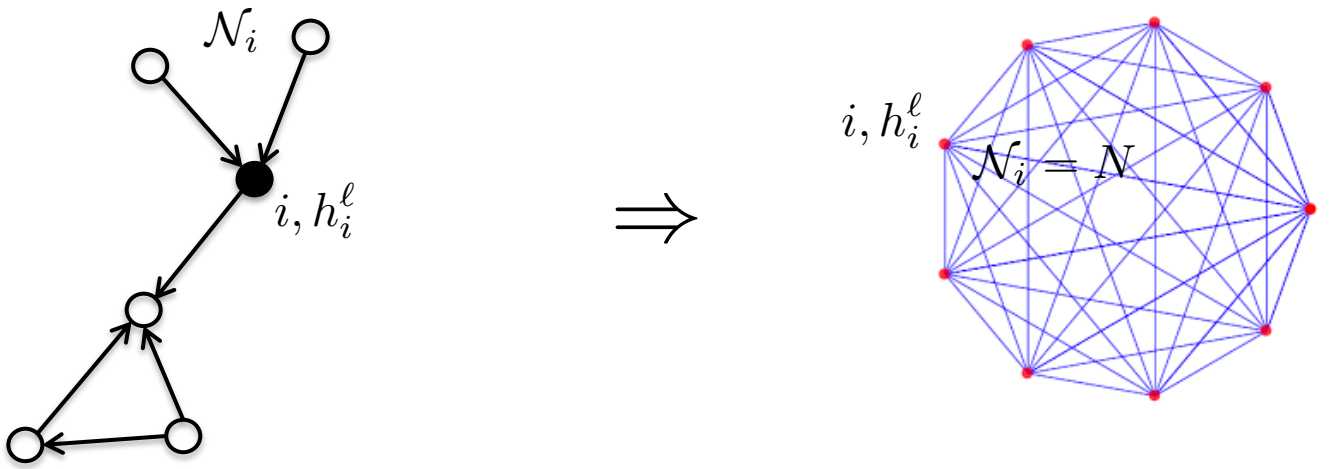
$$e^{\ell=0} = \text{LL}(e_0) \in \mathbb{R}^{E\times d} \text{ (input edge feature)}$$

Hadamard product w/ node and edge features

[1] Dwivedi, Bresson, A generalization of transformer networks to graphs, 2020

Consistency

- Graph Transformers[1] reduces to (original) Transformers[2] when
  - The graph is fully connected, i.e. $\mathcal{N}_i = N$ (set of all nodes)
  - The value of edge feature is 1, i.e. $e_{ij} = 1_d$ (d-dim vector of ones)
  - The LapPE are cos/sin functions for sequences



Sparsely connected graph/
Graph Transformers[1]

Fully connected graph/
Transformers[2]

[1] Dwivedi, Bresson, A generalization of transformer networks to graphs, 2020
[2] Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, Polosukhin, Attention is all you need, 2017

Lab 3 : (Full model) Graph Transformers

- GT with edge feature : Run "03_graph_transformers_regression_exercise.ipynb"

jupyter 03_graph_transformers_regression_solution Last Checkpoint: a few seconds ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 (ipykernel)

Run Code

Lab 03 : Graph Transformers with edge feature for regression - solution

Dwivedi, Bresson, A generalization of transformer networks to graphs, 2020
https://arxiv.org/pdf/2012.09699.pdf

Xavier Bresson

Notebook goals :
- Implement graph transformers with edge feature update
- Compare performance with and without edge feature

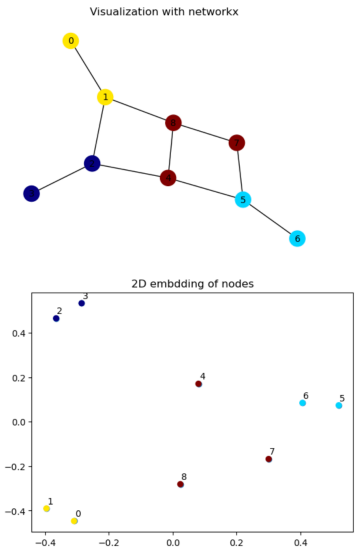
In [1]:
```
# For Google Colaboratory
import sys, os
if 'google.colab' in sys.modules:
    # mount google drive
    from google.colab import drive
    drive.mount('/content/gdrive')
    path_to_file = '/content/gdrive/My Drive/GML_May23_codes/codes/labs_lecture07'
    print(path_to_file)
    # change current path to the folder containing "path_to_file"
    os.chdir(path_to_file)
    !pwd
    !pip install dgl # Install DGL
```

In [2]:
```
# Libraries
import pickle
from utils import Dictionary, MoleculeDataset, MoleculeDGL, Molecule
import dgl
from dgl.data import MiniGCDataset
import dgl.function as fn
import matplotlib.pyplot as plt
import torch
import torch.nn.functional as F
from torch.utils.data import DataLoader
import torch.nn as nn
import time
import networkx as nx
from utils import compute_ncut
import os, datetime
```

Load molecular datasets

```
# load initial net for comparison
checkpoint_file = "checkpoint/checkpoint_QM9_23-05-07--12-43-01.pkl"
checkpoint = torch.load(checkpoint_file)
net.load_state_dict(checkpoint['net_init'])
del checkpoint

# optimizer
optimizer = torch.optim.Adam(net.parameters(), lr=0.0003)

# training loop
start = time.time()
for epoch in range(15):
    epoch_train_loss = run_one_epoch(net, train_loader, True)
    with torch.no_grad():
        epoch_test_loss = run_one_epoch(net, test_loader, False)
        epoch_val_loss = run_one_epoch(net, val_loader, False)
    print('Epoch {}, time {:.4f}, train_loss: {:.4f}, test_loss: {:.4f}, val_loss: {:.4f}'.format(epoch, time.time()-st
```

```
Epoch 0, time 3.5131, train_loss: 1.3346, test_loss: 1.2535, val_loss: 1.1561
Epoch 1, time 6.7691, train_loss: 1.2669, test_loss: 1.2039, val_loss: 1.1113
Epoch 2, time 10.3584, train_loss: 1.2324, test_loss: 1.1567, val_loss: 1.0669
Epoch 3, time 13.7182, train_loss: 1.2093, test_loss: 1.2073, val_loss: 1.0984
Epoch 4, time 17.0723, train_loss: 1.1978, test_loss: 1.1398, val_loss: 1.0696
Epoch 5, time 20.3562, train_loss: 1.1787, test_loss: 1.1375, val_loss: 1.0430
Epoch 6, time 23.7448, train_loss: 1.1680, test_loss: 1.0924, val_loss: 1.0480
Epoch 7, time 26.9114, train_loss: 1.1676, test_loss: 1.0864, val_loss: 1.0423
Epoch 8, time 30.1197, train_loss: 1.1419, test_loss: 1.0865, val_loss: 1.0132
Epoch 9, time 33.2233, train_loss: 1.1378, test_loss: 1.1378, val_loss: 1.0924
Epoch 10, time 36.2970, train_loss: 1.1382, test_loss: 1.0667, val_loss: 1.0160
Epoch 11, time 39.4832, train_loss: 1.1227, test_loss: 1.0736, val_loss: 0.9958
Epoch 12, time 42.5499, train_loss: 1.1206, test_loss: 1.0553, val_loss: 0.9891
Epoch 13, time 45.6317, train_loss: 1.1078, test_loss: 1.0671, val_loss: 1.0056
Epoch 14, time 48.8477, train_loss: 1.0953, test_loss: 1.0440, val_loss: 0.9644
```



Different graph attention functions

- **Graph-free attention**[1] :
$$\mathrm{HA}(h) = \mathrm{Softmax}\left(\frac{QK^T}{\sqrt{d'}}\right) V \in \mathbb{R}^{N \times d'}$$

- **Standard graph attention**[2] :
$$\mathrm{gHA}(h) = \mathrm{Softmax}\left(A_G \odot \frac{QK^T}{\sqrt{d'}}\right) V \in \mathbb{R}^{N \times d'}, \; A_{G\,ij} = \begin{cases} A_{ij}^{\text{original}} & \text{if } i \sim j \\ 1 & \text{for } j = i \\ -\infty & \text{if } i \not\sim j \end{cases}$$

- **Kernel graph attention**[3] :
$$\mathrm{gHA}(h) = \mathrm{Softmax}\left(A_G \odot \frac{QK^T}{\sqrt{d'}}\right) V \in \mathbb{R}^{N \times d'}, \; A_{G\,ij} = A_{ij}^{\text{kernel}}$$

- **Additive graph attention**[4] :
$$\mathrm{gHA}(h) = \mathrm{Softmax}\left(\frac{QK^T}{\sqrt{d'}} + D\right) V \in \mathbb{R}^{N \times d'}, \; D_{ij} = \text{ learnable shortest path } (i,j)$$

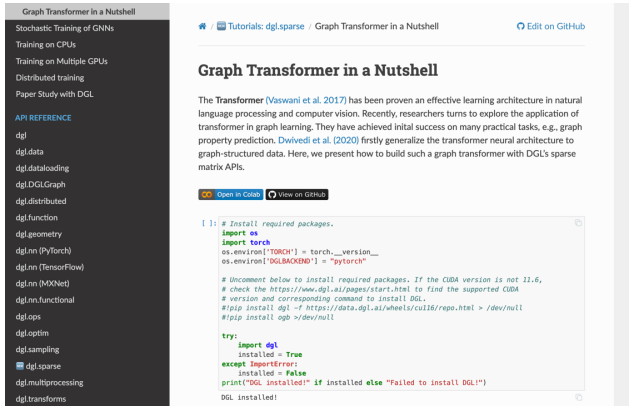
- **Hadamard graph attention**[5] :
$$\mathrm{gHA}(h) = A_G \odot \mathrm{Softmax}\left(\frac{QK^T}{\sqrt{d'}}\right) V \in \mathbb{R}^{N \times d'}, \; A_{G\,ij} = A_{ij}^{\text{original}}$$

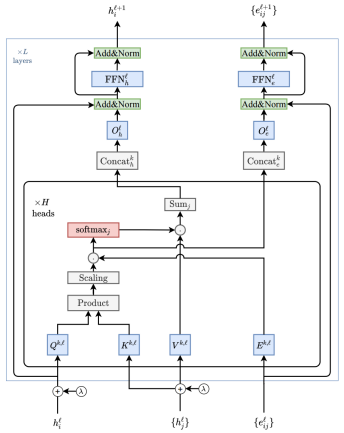
[1] Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, Polosukhin, Attention is all you need, 2017
[2] Dwivedi, Bresson, A generalization of transformer networks to graphs, 2020
[3] Mialon, Chen, Selosse, Mairal, GraphiT: Encoding Graph Structure in Transformers, 2021
[4] Ying, Cai, Luo, Zheng, Ke, He, Shen, Liu, Do Transformers Really Perform Bad for Graph Representation?, 2021
[5] He, Hooi, Laurent, Perold, LeCun, Bresson, A Generalization of ViT/MLP-Mixer to Graphs, 2022

Graph Transformers

- Material
  - GitHub repo (600*+) : https://github.com/graphdeeplearning/graphtransformer
  - Blogpost (Vijay Dwivedi) : https://towardsdatascience.com/graph-transformer-generalization-of-transformers-to-graphs-ead2448cff8b
  - Andrew Ng's article : https://www.deeplearning.ai/the-batch/issue-151
  - DGL tutorial : https://docs.dgl.ai/en/latest/notebooks/sparse/graph_transformer.html

DeepLearning.AI
THE BATCH





Hybrid models

- Graph Transformer[1] is a type of MP-GNN s.a. GCN[2], GAT[3], etc.
  - In practice, GT compares favorably to GCN, GAT on the benchmark[4].
- However, all MP-GNNs are limited by over-squashing and poor long-range interactions.
  - As the number of layers increases, the local reception field grows exponentially, and aggregation functions struggle to distinguish critical from irrelevant information.
- A well-known solution from NLP is to use (self-)attention to all nodes simultaneously.
  - This idea was a breakthrough in NLP for standard/fully-connected Transformers[5] (T).
  - However, standard T do not perform well experimentally for graph learning tasks and have a higher complexity $O(N^2)$.
  - It is actually critical to use the graph topology as it offers a strong inductive bias for generalization.

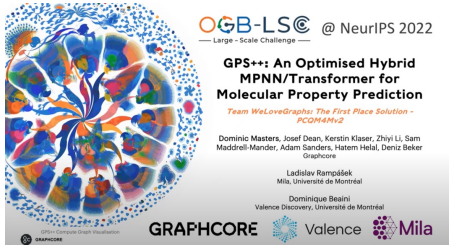
[1] Dwivedi, Bresson, A generalization of transformer networks to graphs, AAAI 2021
[2] Kipf, Welling, Semi-supervised classification with graph convolutional networks, 2017
[3] Velickovic, Cucurull, Casanova, Romero, Lio, Bengio, Graph Attention Networks, 2018
[4] Dwivedi, Joshi, Laurent, Bengio, Bresson, Benchmarking graph neural networks, 2020
[5] Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, Polosukhin, Attention is all you need, 2017

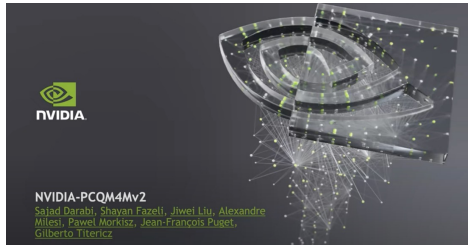
Hybrid models

- This led to design hybrid models mixing GT+T, potentially solving over-squashing/long-range issues while retaining the inductive bias of graph topology.
- Hybrid GT+T architectures have been showed successful in practice.
  - Winner[1] and runner-up[2] at the NeurIPS'22 OGB Large-Scale Challenge competition[3]

OGB-LSC — Large-Scale Challenge — @ NeurIPS 2022

- 10:50PM–11:40PM, UTC: Graph-Level Track (PCQM4Mv2)
  - Intro to the task (5min) [Video]
  - Live presentation by 1st place winner: **WeLoveGraphs** (10min) [Video]
  - Live presentation by 2nd place winner: **NVIDIA-PCQM4Mv2** (10min) [Video]
  - Live presentation by 2nd place winner: **ViSNet** (10min) [Video]
  - Live Q&A and discussion (15min) [Video]





[1] Masters et.al, GPS++: An Optimised Hybrid MPNN/Transformer for Molecular Property Prediction, 2022
[2] Darabi et-al, Heterogenous Ensemble of Models for Molecular Property Prediction, 2022
[3] https://ogb.stanford.edu/neurips2022/workshop

Hybrid models

- Hybrid GT+T models can be appealing but
  - Their complexity is higher $O(N^2+E)$[1,2]. Linear approximation of the attention mechanism[3] can be used to reduce the complexity of GT+T to $O(N+E)$[4].
  - Hybrid architectures are more complex models, often less stable to train and require more hyper-parameter tuning (i.e. industry resources). A similar example is LSTM+T[5] in NLP, which was SOTA until 2016 and LSTM was later discarded for more stable/scalable deep T.
  - Hybrid architectures can be any MP-GNN+T combination.
- In my opinion
  - Hybridization MP-GNN+T is temporary and new models, simpler and more stable, will be developed, similar to what has happened in NLP. See next sections.
- Survey[6] on the development of GT-based models
  - Study properties of isomorphism expressivity, graph topology as good inductive bias, reduce over-squashing and attention interpretability in GT architectures.

[1] Kreuzer, Beaini, Hamilton, Letourneau, Tossou, Rethinking Graph Transformers with Spectral Attention, 2021
[2] Dwivedi, Luu, Laurent, Bengio, Bresson, Graph neural networks with learnable structural and positional representations, 2021
[3] Choromanski et-al, Rethinking attention with performers, 2021
[4] Rampasek et-al, Recipe for a General, Powerful, Scalable Graph Transformer, 2022
[5] Wu et-al, Google's neural machine translation system: Bridging the gap between human and machine translation, 2016
[6] Muller, Galkin, Morris, Rampasek, Attending to Graph Transformers, 2023