Open in Colab

In [3]:
```
# basic libraries
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
from torch.utils.data import DataLoader, Dataset
```

Importing and normalizing the data

In [4]:
```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

In [ ]:
```
# upload training data
from google.colab import files

uploaded_1 = files.upload()
```

In [6]:
```
import os

graphs = {}
index = 0

for filename, file_data in uploaded_1.items():
    file_content = file_data.decode('utf-8')

    start_collecting = False
    coordinates = []

    for line in file_content.splitlines():
        if "NODE_COORD_SECTION" in line:
            start_collecting = True
            continue

        if "EOF" in line or line.strip() == '':
            break

        if start_collecting:
            parts = line.split()
            if len(parts) >= 3:
                # Append [x, y] coordinates (ignore the first column)
                coordinates.append([float(parts[1]), float(parts[2])])

    graphs[index] = coordinates
    index += 1
```

In [7]:
```
for graph in graphs:
    print(f"Instance: {graph}\nCoordinates: {graphs[graph]}\n")
    break
```

```
Instance: 0
Coordinates: [[190.90116849279775, 590.4280175750611], [166.92708168528983, 530.2271396316246], [1085.991923849472, 16.922422544709608], [481.1400572117718, 1646.5622569153477], [13.17437595135608, 29.384015131643572], [579.5977925840228, 183.09074463391653], [3152.535909779576, 469.4891656526584], [56.79395127865876, 335.7676041800954], [140.16984710530528, 1415.9907047276663], [298.4056652503395, 432.6438567934434], [1329.4159078188572, 398.07979715224917], [184.34432775505695, 310.6223243278328], [1413.9458436791929, 71.70924057019577], [229.8380192520161, 12.507313429062522], [119.85732963663749, 267.10190426680424], [1068.128373699216, 72.16722331618915], [4.4735957948030896, 260.74107597220853], [315.9927811661698, 453.1032177042161], [74.37417532866615, 462.03510416808166], [391.5463212784245, 568.466736501489], [20.04917685531651, 200.4640408371322], [230.93189213865705, 1251.8991123066855], [1572.7308604790571, 269.1672960446028], [592.4233980658983, 398.928663581323], [270.91848807632886, 111.30947683362179], [937.6511284275634, 502.76960724392404], [379.12301291798457, 291.35789407895794], [423.7620301035877, 835.6244553900403], [531.8204189375267, 36.38573819950644], [17.693326797842897, 25.882043485803397]]
```

In [8]:
```python
print(f"Euclidean coordinates of the 35th graph instance are:")
graphs[35]
```

```
Euclidean coordinates of the 35th graph instance are:
```

Out[8]:
```
[[7207.304443622209, 7324.167128487795],
 [4163.142976744452, 433.4999525698357],
 [2490.932722533464, 4875.366088798042],
 [4551.5205625801955, 2982.122152631126],
 [6763.789886532631, 1030.9891543093809],
 [1192.5835434616927, 6609.37024934381],
 [4233.957584681043, 4034.951770206021],
 [2636.353709072561, 6386.539680951245],
 [5349.1546667631965, 9770.2454151319],
 [5350.938714803364, 7417.2756867157395],
 [3603.702035972396, 966.7842507669433],
 [8336.904519630016, 1307.823731512655],
 [2250.7038620979292, 6451.865150099256],
 [9410.524952419511, 6731.366335870505],
 [8976.457769795643, 8503.88983386449],
 [1713.7074001514418, 1096.4539677469254],
 [2822.1366695079887, 5066.67013828053],
 [3449.0565250305203, 9008.22163939093],
 [7599.90158431922, 6101.292179054742],
 [6282.968640320806, 5723.651901213904],
 [5377.336280017825, 658.9912628221895],
 [5782.466661739941, 3623.846327413507],
 [2612.0291245340845, 6706.035485758186],
 [7905.677924103369, 723.4527207413255],
 [8687.760132503685, 8977.531049384415],
 [3345.3485774394285, 1139.4106084946332],
 [3777.641694394105, 6568.454970385794],
 [8147.04280449171, 1032.2014753951414],
 [2170.319009545205, 1285.1867416777663],
 [2684.280522235982, 8614.963884506185]]
```

In [9]:
```python
print(f"We have {len(graphs)} graphs each having {len(graphs[0])} [x,y] coordi
```

```
We have 85 graphs each having 30 [x,y] coordinate points
```

In [ ]:
```python
# upload training data
from google.colab import files
```

```
uploaded_2 = files.upload()
```

In [11]:
```
import csv
import io

runs = {}
index = 0

for filename, file_data in uploaded_2.items():
    file_content = file_data.decode('utf-8')

    instance_data = []
    csv_reader = csv.reader(io.StringIO(file_content))

    next(csv_reader, None)

    for row in csv_reader:
        relaxation_param = float(row[0])
        p_f = float(row[1])
        e_std = float(row[2])
        e_avg = float(row[3])
        e_min = float(row[4])

        instance_data.append([relaxation_param, p_f, e_std, e_avg, e_min])

    runs[index] = instance_data
    index+=1
```

In [12]:
```
print(f"Now we have all the information. \nThe runs list contains information a
```

```
Now we have all the information.
The runs list contains information about 85 graphs.
Each of which is tested for 100 different relaxation parameters.
For each such parameter we have a vector of length 5 information containing th
e values of the energies as [A, p_f, e_std, e_avg, e_min] extracted from the a
nnealing experiment
```

`data[i][j] = [relaxation_parameter, p_f, e_std, e_avg, e_min]` for the j^{th} run of the i^{th} instance

In [13]:
```
runs[0][0]
```

Out[13]:
```
[1952.0, 0.046875, 19742.68658959116, 1589.5882639783756, 16002.903095621281]
```

Normalization

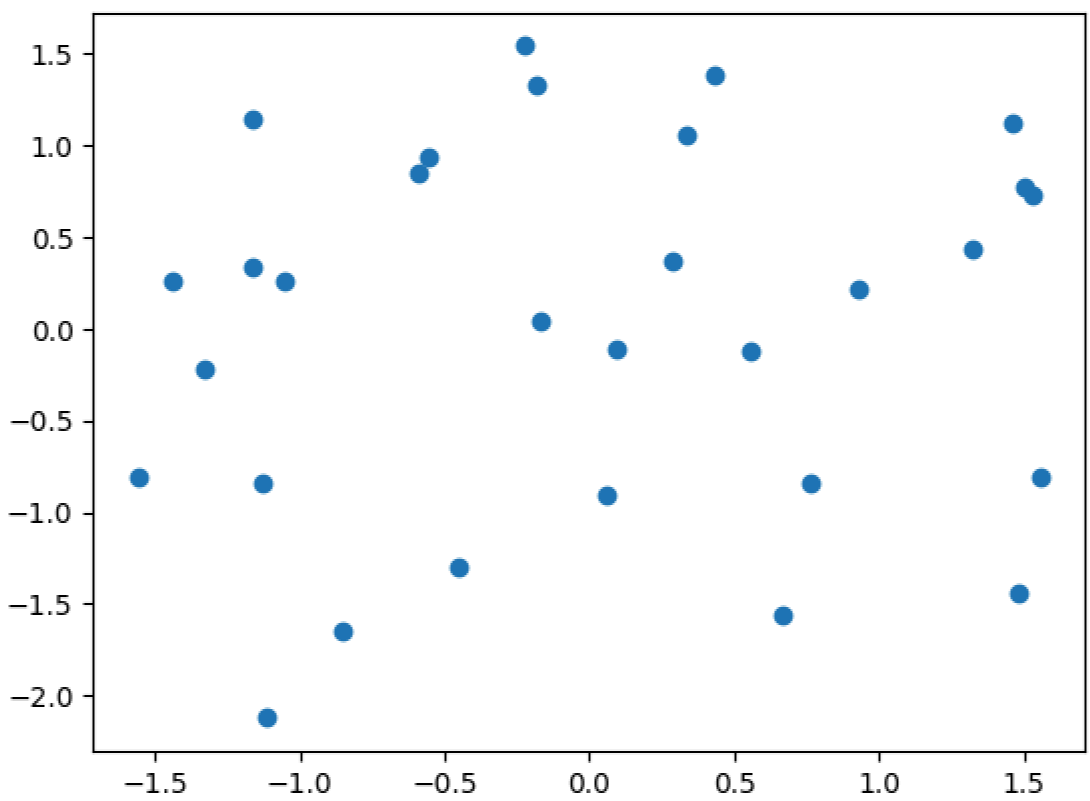
Now the instance coordinates are all normalized as per the formula $x^j_i (norm) = (x^j_i - \mu^j)/ \sigma^j$ where i denotes the i^{th} coordinate and j denotes the j^{th} graph instance indexed from 0 -> 84.

In [14]:
```
import numpy as np

normed_graphs = {}
```

```
for index, coordinates in graphs.items():
    if isinstance(coordinates, list) and all(isinstance(coord, list) and len(c
        x_coords = [coord[0] for coord in coordinates]
        y_coords = [coord[1] for coord in coordinates]

        mean_x = np.mean(x_coords)
        stddev_x = np.std(x_coords)
        mean_y = np.mean(y_coords)
        stddev_y = np.std(y_coords)

        # Normalize formula
        normalized_coords = [
            [(x - mean_x) / stddev_x, (y - mean_y) / stddev_y] for x, y in coo
        ]

        normed_graphs[index] = normalized_coords
    else:
        print(f"Skipping instance {filename}: Invalid data format")
```

In [15]:

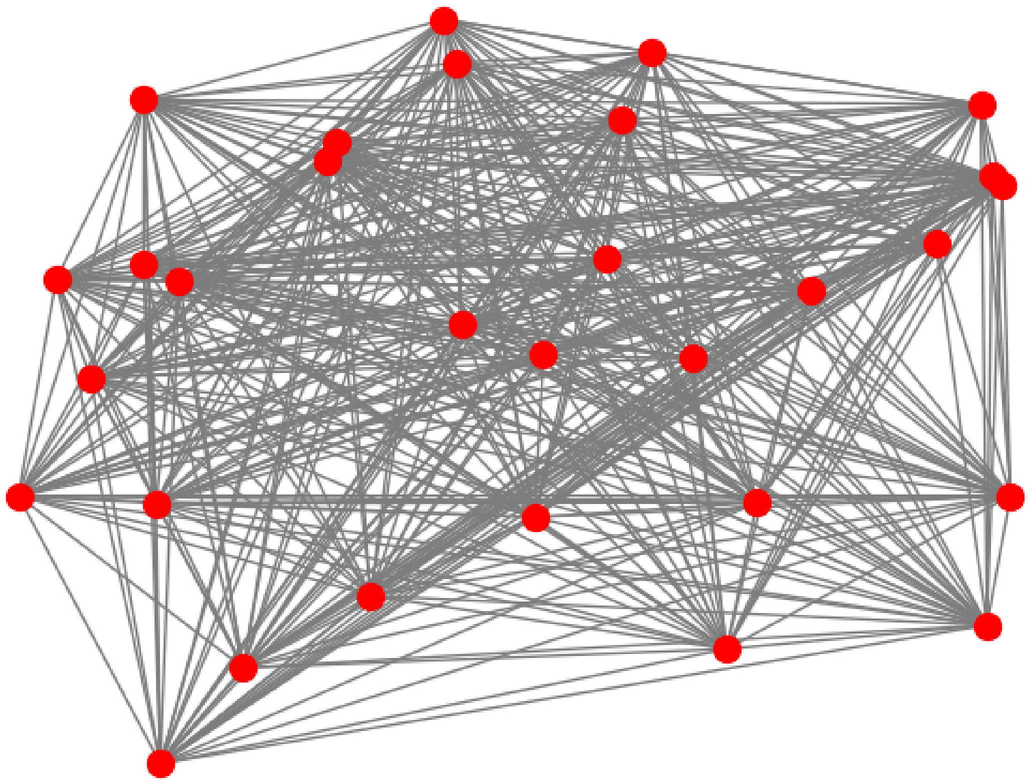
```
ith_run= np.random.randint(100)
ith_run

x = [coord[0] for coord in normed_graphs[ith_run]]
y = [coord[1] for coord in normed_graphs[ith_run]]
plt.scatter(x,y)
plt.show()
```



In [16]:

```
import networkx as nx
import matplotlib.pyplot as plt

G = nx.Graph()
positions = {i: (x[i], y[i]) for i in range(len(x))}
G.add_nodes_from(positions.keys())
```

```
edges = [(i, j) for i in range(len(x)) for j in range(i + 1, len(x))]
G.add_edges_from(edges)
#plt.figure(figsize=(4, 4))
nx.draw(G, pos=positions, node_color='red', node_size=100, edge_color='gray')
plt.show()
```



In [17]:
```
for index in normed_graphs:
  normed_graphs[index] = np.array(normed_graphs[index]).flatten()
```

In [18]:
```
len(normed_graphs[0]), normed_graphs[0]
```

Out[18]:
```
(60,
 array([-0.53957575,  0.4398966 , -0.57636841,  0.28891542,  0.83410605,
        -0.99843048, -0.09415066,  3.08863531, -0.81233028, -0.96717734,
         0.05695089, -0.58168756,  4.00559883,  0.13658722, -0.745388  ,
        -0.19878064, -0.61743232,  2.51037189, -0.37459028,  0.04418079,
         1.20768504, -0.04250437, -0.54963843, -0.2618439 ,  1.33741181,
        -0.86102752, -0.47981997, -1.00950338, -0.64860562, -0.37099122,
         0.80669114, -0.85987892, -0.82568323, -0.3869439 , -0.3475996 ,
         0.09549198, -0.7184079 ,  0.11789276, -0.23164877,  0.38481866,
        -0.80177963, -0.53811608, -0.47814122,  2.09883731,  1.5810967 ,
        -0.36581131,  0.07663415, -0.04037545, -0.41677442, -0.76171185,
         0.60644975,  0.22005312, -0.25071462, -0.31015826, -0.18220782,
         1.05483859, -0.0163723 , -0.94961733, -0.80539512, -0.97596014]))
```

In [19]:
```
from torch.utils.data import DataLoader, Dataset
from tqdm import tqdm
```

EDA

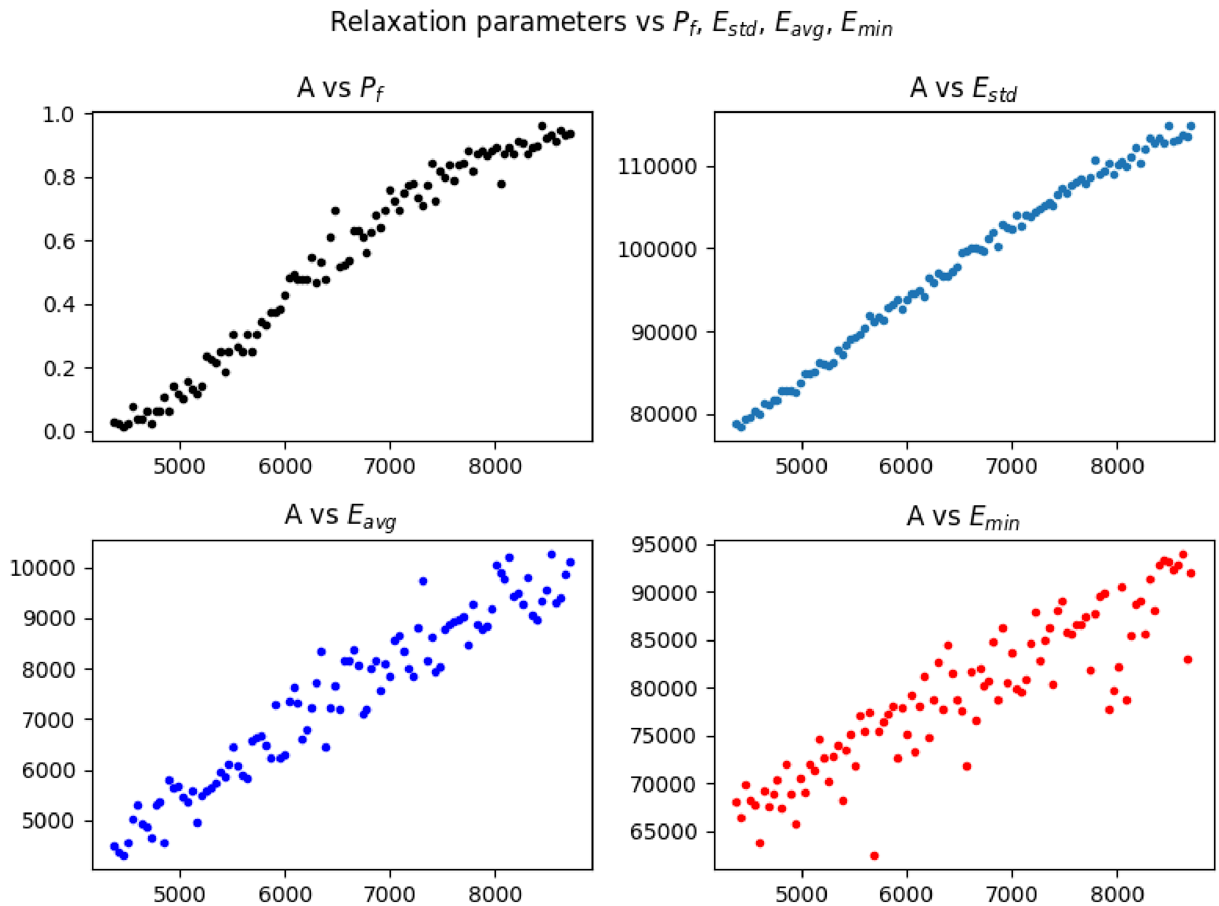
`data tensor = A[graph_index][data for each relax param]`

`data = [A,pf,estd,eavg,emin]`

In [20]:
```
ith_graph = np.random.randint(100)
values = runs[ith_graph]
As = [value[0] for value in values]
pfs = [value[1] for value in values]
estds = [value[2] for value in values]
eavs = [value[3] for value in values]
emins = [value[4] for value in values]
values[1]
```

Out[20]:
```
[4418.75, 0.0234375, 78469.53230931533, 4359.142782179081, 66460.02933320974]
```

In [21]:
```
fig, axs = plt.subplots(2, 2, figsize=(8,6))
fig.suptitle('Relaxation parameters vs $P_f$, $E_{std}$, $E_{avg}$, $E_{min}$'

axs[0, 0].set_title("A vs $P_f$")
axs[0, 0].scatter(As, pfs, marker = ".", c = 'k')

axs[0, 1].set_title("A vs $E_{std}$")
axs[0, 1].scatter(As, estds, marker = ".")

axs[1, 0].set_title("A vs $E_{avg}$")
axs[1, 0].scatter(As, eavs, marker = ".", c ='b')

axs[1, 1].set_title("A vs $E_{min}$")
axs[1, 1].scatter(As, emins, marker = ".", c='r')

fig.tight_layout()
fig.savefig('datavis_qross.png')
```



Next steps for the project

Our QROSS-based project addresses optimizing relaxation parameters for TSP instances using QUBO formulations and surrogate models. The key challenges include creating usable feature vectors for the neural network and understanding how the training process aligns these vectors with relaxation parameters.

My Plan:

1. **Feature Vector Extraction**: Use GCNs or similar methods to process graph inputs and produce fixed-size feature vectors. Still have to figure this out. (currently doing so). Hence we build a sort of **FEATURE EXTRACTOR function/class** for each graph **instance**.
2. **Find the min relaxation parameter**: We then assign relaxation parameters from the **data** where P_f is maximized and energy metrics are minimized for training graphs. For this we make a function called a **Min_relaxation_finder**
3. **Train NN based on these min A and the graph feature vectors as the input**: We then train the model (input → graph feature vector, output → relaxation parameter) using loss functions with `min_relaxation_parameter` as the marker which will use gradient descent etc to backpropagate and thus train our model.
4. **Predict A from test data set for each instance in the test set**: For the test data, we first put them through the FEATURE EXTRACTOR function/class to get the feature vectors of the test graphs annd then retrieve the predicted relaxation parameter for each graph.

5. **Validation**: Now we have the predicted **A**, then we can put the i^{th} graph's data matrix into the min_relaxation_finder function to find the A_min for the particular instance. Then our validation loss shall be (`predicted_A` - `min_A` $)^2$ which should be ideally very very low.

OPTIMAL A VECTOR for both training and testing data `min_relaxation_param` function

In [22]:
```
def min_relaxation_params(data_dict, lambda_1=5.0, lambda_2=1.0):

  optimal_A_dict = {}

  for idx, data_matrix in data_dict.items():
    data_matrix = np.array(data_matrix)
    A_values = data_matrix[:][0]
    P_f = data_matrix[:][1]
    E_avg = data_matrix[:][2]
    E_std = data_matrix[:][3]
    E_min = data_matrix[:][4]

    scores = lambda_1 * P_f - lambda_2 * (E_avg + E_std + E_min)
    best_index = np.argmax(scores)

    optimal_A_dict[idx] = A_values[best_index]

  return optimal_A_dict
```

In [23]:
```
np.asarray(runs[0]).shape, type(runs), len(runs)
```

Out[23]:
```
((100, 5), dict, 85)
```

In [24]:
```
min_relaxations = min_relaxation_params(runs)
type(min_relaxations)
```

Out[24]:
```
dict
```

In [25]:
```
min_relaxations
```

```
Out[25]: {0: 19742.68658959116,
 1: 11624.152000070813,
 2: 0.0,
 3: 11506.749468451164,
 4: 18246.745018940368,
 5: 16491.04149047193,
 6: 20442.571485561864,
 7: 17426.207328027966,
 8: 0.0,
 9: 10185.40627015858,
 10: 7717.078702199341,
 11: 15790.34150683544,
 12: 18567.23992090268,
 13: 16704.28466084302,
 14: 18894.239826881414,
 15: 68537.17944943796,
 16: 73729.92762407339,
 17: 71523.61410078392,
 18: 76842.98970748132,
 19: 72229.4909184825,
 20: 77300.68921104146,
 21: 82534.79916936645,
 22: 74228.41565754259,
 23: 76991.24839892563,
 24: 75592.60232978663,
 25: 73988.25920308115,
 26: 78787.26530533658,
 27: 87120.74921811942,
 28: 72684.38739915902,
 29: 69358.95123947253,
 30: 80719.279308477,
 31: 83935.9135087967,
 32: 77628.33783257734,
 33: 82361.63150910153,
 34: 73356.46366435052,
 35: 66520.56738166166,
 36: 76001.08928002197,
 37: 75178.56972050073,
 38: 77326.43231809612,
 39: 76866.22702002636,
 40: 77928.8618546301,
 41: 75801.35170391705,
 42: 73890.14344464957,
 43: 74464.76154530582,
 44: 64942.75760582006,
 45: 84291.6236068402,
 46: 69524.44013403615,
 47: 76913.70438158954,
 48: 84551.5193676568,
 49: 84715.04581958803,
 50: 76404.56512521808,
 51: 72046.4558982431,
 52: 74230.50073867709,
 53: 66984.26387089677,
 54: 4007.384394090696,
 55: 3889.6175895950178,
 56: 3847.5344951641146,
 57: 3907.254766355891,
 58: 3917.7549180167584,
 59: 4270.499817910324,
```

```
 60: 4257.681444827419,
 61: 3727.9756410949367,
 62: 4325.472496767866,
 63: 3600.8935395819794,
 64: 3843.982220546579,
 65: 3914.4725674881533,
 66: 3654.262995589604,
 67: 3695.8561984333214,
 68: 3950.720288209761,
 69: 4414.393605088297,
 70: 3577.969909958158,
 71: 4313.719905639739,
 72: 4094.4185798638796,
 73: 8087.25554273873,
 74: 7121.984567633525,
 75: 7975.501572683426,
 76: 8164.673110970353,
 77: 8495.43136300059,
 78: 8056.635923956374,
 79: 7940.668844952924,
 80: 7209.904152251787,
 81: 8233.05419569656,
 82: 7238.450331941446,
 83: 7463.951649376246,
 84: 8368.710941354633}
```

Graph FEATURE VECTOR EXTRACTOR

In [26]:

```
print(f"We have {len(graphs)} graphs which have {len(graphs[0])} nodes each wi
```

```
We have 85 graphs which have 30 nodes each with 2 coordinates.
```

In [29]:

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.data import Data
from torch_geometric.nn import GCNConv, global_mean_pool

num_graphs = len(graphs)
num_nodes = len(graphs[0])
input_dim = len(graphs[0][0])
```

In [30]:

```
def compute_adjacency(coords):
    num_nodes = coords.shape[0]
    adjacency_matrix = np.zeros((num_nodes, num_nodes))
    for i in range(num_nodes):
        for j in range(num_nodes):
            adjacency_matrix[i, j] = np.sqrt((coords[i, 0] - coords[j, 0])**2 
    return adjacency_matrix
```

In [31]:

```
graph_data_list = []
for i in range(num_graphs):
    coords = graphs[i]
    adjacency_matrix = compute_adjacency(np.array(coords))

    edge_index = np.array(np.nonzero(adjacency_matrix)).astype(np.int64)
```

```
    edge_weight = adjacency_matrix[edge_index[0], edge_index[1]]

    # Convert to PyTorch tensors
    edge_index = torch.tensor(edge_index, dtype=torch.long)
    edge_weight = torch.tensor(edge_weight, dtype=torch.float32)
    node_features = torch.tensor(coords, dtype=torch.float32)

    # Create graph data object
    data = Data(x=node_features, edge_index=edge_index, edge_attr=edge_weight)
    graph_data_list.append(data)
```

In [32]:
```
print(f"We got graph data list for each {len(graph_data_list)} graphs in the s
```

```
We got graph data list for each 85 graphs in the samples we took for experimen
t of the graph pooling
```

In [33]:
```
class GraphFeatureExtractor(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(GraphFeatureExtractor, self).__init__()
        self.conv1 = GCNConv(input_dim, hidden_dim)
        self.conv2 = GCNConv(hidden_dim, hidden_dim)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, data):
        x, edge_index, edge_attr = data.x, data.edge_index, data.edge_attr
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = self.conv2(x, edge_index)
        x = F.relu(x)
        x = global_mean_pool(x, torch.zeros(x.size(0), dtype=torch.long))  # G
        x = self.fc(x)
        return x
```

In [34]:
```
#initialize the Graph convolutional neural network
hidden_dim = 64 # neurons in hidden layer
output_dim = 128 # dimension of the graph feature vector
model = GraphFeatureExtractor(input_dim, hidden_dim, output_dim) # instantiate

graph_tensor = graph_data_list[0]
graph_feature_vector = model(graph_tensor)
```

In [35]:
```
# Batch processing for multiple graphs
from torch_geometric.loader import DataLoader

loader = DataLoader(graph_data_list, batch_size=1, shuffle=True)
batch_graph_features_vector = []
for batch in loader:
    batch_graph_features = model(batch)  # Batch graph-level features
    batch_graph_features_vector.append(batch_graph_features)
```

In [36]:
```
print(f"Number of features per graph for encoding scheme - {len(batch_graph_fe
```

```
Number of features per graph for encoding scheme - 128.
```

In [37]:
```
batch_graph_features_vector[0]
```

Out[37]:
```
tensor([[-16.2039,  35.7280,  -1.6098,  15.3845, -17.4187, -36.8933,  -3.8663,
         -16.7205, -42.9882, -21.1069,  -7.0415,  45.5929,  60.7329,  63.7199,
         -27.6432,  -3.1078,   5.6890,  12.3322,  22.2650,  -9.9119,  19.1045,
           3.9202, -39.7652,  26.8623, -33.6483,  50.7305,  -2.3023,  28.4820,
         -13.6962, -87.9388,  19.4690,   2.5472,  16.0834,  18.4338,   2.0746,
          31.2055,  37.0711, -38.8986, -34.3320, -24.2267, -54.9358, -14.2121,
           8.4565,  36.3858,   9.0439, -18.9822, -29.4031, -15.8358,  -7.7291,
         -15.8303,   9.7275,  -7.2643,  59.4637,   2.5601,  31.5887, -50.7334,
          20.9196,   2.9593,  53.8579, -19.6543,  42.1364,   2.7349,  37.1305,
         -62.3103,  18.6068,  36.9797,   5.4836,  23.3735, -28.7694,  13.6844,
         -56.2224, -19.9935,   7.4272,   6.6961,  17.2861,  10.7661,  39.9079,
         -51.2833,  10.2631,  28.6051,  53.1382, -12.1615,   4.1954, -14.6642,
          -1.5960,  24.8644,  16.5333,  19.0423,  64.8191,  30.9689,  51.4978,
          21.7265, -65.7373, -25.1604,  36.6337,  25.0730, -15.9549, -20.1817,
          22.7462, -31.0154,  18.0740, -25.3012, -34.8885,   3.6314, -49.6184,
         -28.3277, -55.0793,  30.3211, -18.6459, -63.9143,  68.1015,   3.6699,
         -30.1418,  68.7537, -32.4476,  -4.9086, -32.7871,  38.2202, -35.1110,
           7.0136,  24.5631,  38.2890,   8.8671, -54.6320, -47.1095, -14.7614,
          20.6043, -13.3520]], grad_fn=<AddmmBackward0>)
```

TRAINING THE MODEL (correct approach)

The GCN structure remains the same

In [58]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch_geometric.data import Data, DataLoader
from torch_geometric.nn import GCNConv
import random

class GraphFeatureExtractor(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(GraphFeatureExtractor, self).__init__()
        self.conv1 = GCNConv(input_dim, hidden_dim)
        self.conv2 = GCNConv(hidden_dim, hidden_dim)
        self.lin = torch.nn.Linear(hidden_dim, output_dim)

    def forward(self, x, edge_index, batch):
        x = self.conv1(x, edge_index).relu()
        x = self.conv2(x, edge_index).relu()
        x = global_mean_pool(x, batch)  # Aggregate node features for each gra
        return self.lin(x)  # Predict relaxation parameter
```

Convert our graph in pytorch.geometric representation. Then subsequently prepare out dataset.

In [59]:
```
def generate_edge_index(num_nodes):
    edge_index = []
    for i in range(num_nodes):
        for j in range(num_nodes):
            if i != j:
                edge_index.append([i, j])
    return torch.tensor(edge_index, dtype=torch.long).t().contiguous() # just 

def prepare_dataset(graphs_dict, optimal_A):
    dataset = []
```

```
    for idx, coords in graphs_dict.items():
        x = torch.tensor(coords, dtype=torch.float32) # input as graph
        edge_index = generate_edge_index(x.shape[0]) # generates the edge indi
        y = torch.tensor([optimal_A[idx]], dtype=torch.float32) # output as th
        data = Data(x=x, edge_index=edge_index, y=y) # get the data in pytorch
        dataset.append(data)
    return dataset
```

In [60]:

```
dataset = prepare_dataset(graphs,min_relaxations)
len(dataset), type(dataset[0]).x, type(dataset[0]).y, type(dataset[0]).edge_in
```

Out[60]:

```
(85,
 <property at 0x782ddd145580>,
 <property at 0x782ddd1456c0>,
 <property at 0x782ddd1455d0>)
```

Split data into test and train

In [61]:

```
def split_data(graphs, optimal_A, split_ratio=0.8):
    indices = list(graphs.keys()) # get keys
    random.shuffle(indices) # shuffle the indices list
    split = int(len(indices) * split_ratio) # split wrt split ratio
    train_indices = indices[:split] # get training graph indices
    test_indices = indices[split:] # get testing graph indices

    train_graphs = {i: graphs[i] for i in train_indices}
    train_A = {i: optimal_A[i] for i in train_indices}

    test_graphs = {i: graphs[i] for i in test_indices}
    test_A = {i: optimal_A[i] for i in test_indices}

    return train_graphs, train_A, test_graphs, test_A
```

In [62]:

```
train_graphs, train_A, test_graphs, test_A = split_data(graphs,min_relaxations

list(test_graphs.keys()) == list(test_A.keys()) # just checking
```

Out[62]:

```
True
```

Build Training Routine

In [63]:

```
def train_model(model, train_loader, optimizer, lossfunc):
    model.train()
    total_loss = 0
    for data in train_loader:
        optimizer.zero_grad()
        out = model(data.x, data.edge_index, data.batch)  # Include batch info
        out = out.view_as(data.y)  # Ensure output shape matches target shape
        loss = lossfunc(out, data.y)  # Compute loss
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    return total_loss / len(train_loader)
```

Validation Routine

In [69]:
```
def validate_model(model, test_loader, lossfunc, batch_size):
    model.eval()
    total_loss = 0
    validation_vector = {}  # Dictionary to store squared differences by graph

    with torch.no_grad():
        for idx, data in enumerate(test_loader):
            out = model(data.x, data.edge_index, data.batch)  # Include batch 
            out = out.view_as(data.y)  # Match output shape to target
            loss = lossfunc(out, data.y)
            total_loss += loss.item()

            # Compute squared difference ||actual[i] - predicted[i]||^2
            squared_diff = ((data.y - out)**2).cpu().tolist()

            # Store squared differences in the validation vector
            for i, val in enumerate(squared_diff):
                graph_index = idx * batch_size + i  # Derive graph index from 
                validation_vector[graph_index] = val

    avg_loss = total_loss / len(test_loader)
    return avg_loss, validation_vector
```

In [70]:
```
train_dataset = prepare_dataset(train_graphs, train_A)
test_dataset = prepare_dataset(test_graphs, test_A)

from torch_geometric.loader import DataLoader

train_loader = DataLoader(train_dataset, batch_size=8, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=8)
```

In [71]:
```
input_dim = 2 # 2 coordinates
hidden_dim = 16 # 16 hidden neurons (can be tested for other values)
output_dim = 1 # relaxation parameter for each graph input
model = GraphFeatureExtractor(input_dim, hidden_dim, output_dim) # our beautif
lossfunc = nn.MSELoss() # standard mse loss
optimizer = optim.Adam(model.parameters(), lr=0.01) # adam optimizer for optim


######################## CHECK MODEL CONSISTENCY WITH THE DEVICE -> SINCE WE A
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = model.to(device)
model, device
```

Out[71]:
```
(GraphFeatureExtractor(
   (conv1): GCNConv(2, 16)
   (conv2): GCNConv(16, 16)
   (lin): Linear(in_features=16, out_features=1, bias=True)
 ),
 device(type='cpu'))
```

In [72]:
```
train_loader = [data.to(device) for data in train_loader]
test_loader = [data.to(device) for data in test_loader]
```

In [74]:
```
train_losses =[]
val_losses =[]
epochs = 200
```

```
for epoch in range(epochs):

    train_loss = train_model(model, train_loader, optimizer, lossfunc)
    train_losses.append(train_loss)

    batch_size = 8
    val_loss, validation_vector = validate_model(model, test_loader, lossfunc,
    val_losses.append(val_loss)

    print(f"Epoch {epoch+1}/{epochs}, Train Loss: {train_loss:.4f}, Val Loss:
```

```
Epoch 1/200, Train Loss: 2554008987.5556, Val Loss: 1713991722.6667
Epoch 2/200, Train Loss: 2358546648.8889, Val Loss: 1519219008.0000
Epoch 3/200, Train Loss: 1975586240.8889, Val Loss: 1170307050.6667
Epoch 4/200, Train Loss: 1351085608.4444, Val Loss: 676586037.3333
Epoch 5/200, Train Loss: 590794758.6667, Val Loss: 212793877.3333
Epoch 6/200, Train Loss: 84642076.7778, Val Loss: 78427917.3333
Epoch 7/200, Train Loss: 76903453.5556, Val Loss: 90547573.3333
Epoch 8/200, Train Loss: 67113618.6667, Val Loss: 77532090.6667
Epoch 9/200, Train Loss: 37772269.7778, Val Loss: 89915475.3333
Epoch 10/200, Train Loss: 38902403.1111, Val Loss: 84128998.0000
Epoch 11/200, Train Loss: 33872681.0000, Val Loss: 78022205.3333
Epoch 12/200, Train Loss: 34867471.8889, Val Loss: 77662005.3333
Epoch 13/200, Train Loss: 35138617.5556, Val Loss: 79371110.6667
Epoch 14/200, Train Loss: 34600277.7778, Val Loss: 80104576.6667
Epoch 15/200, Train Loss: 34321401.1111, Val Loss: 79401636.6667
Epoch 16/200, Train Loss: 34364586.8889, Val Loss: 79001194.6667
Epoch 17/200, Train Loss: 34522878.0000, Val Loss: 79198512.0000
Epoch 18/200, Train Loss: 34511691.3333, Val Loss: 79386146.6667
Epoch 19/200, Train Loss: 34461094.0000, Val Loss: 79320304.0000
Epoch 20/200, Train Loss: 34469321.3333, Val Loss: 79239922.0000
Epoch 21/200, Train Loss: 34503449.5556, Val Loss: 79261752.6667
Epoch 22/200, Train Loss: 34512595.5556, Val Loss: 79292866.0000
Epoch 23/200, Train Loss: 34510900.2222, Val Loss: 79282299.3333
Epoch 24/200, Train Loss: 34519284.8889, Val Loss: 79268813.3333
Epoch 25/200, Train Loss: 34531743.1111, Val Loss: 79272279.3333
Epoch 26/200, Train Loss: 34539540.6667, Val Loss: 79275632.6667
Epoch 27/200, Train Loss: 34545814.2222, Val Loss: 79272022.6667
Epoch 28/200, Train Loss: 34553998.8889, Val Loss: 79269265.3333
Epoch 29/200, Train Loss: 34562361.7778, Val Loss: 79269058.6667
Epoch 30/200, Train Loss: 34569737.5556, Val Loss: 79267973.3333
Epoch 31/200, Train Loss: 34577018.8889, Val Loss: 79266020.0000
Epoch 32/200, Train Loss: 34584546.8889, Val Loss: 79264424.0000
Epoch 33/200, Train Loss: 34591892.2222, Val Loss: 79263041.3333
Epoch 34/200, Train Loss: 34599008.6667, Val Loss: 79261421.3333
Epoch 35/200, Train Loss: 34606095.3333, Val Loss: 79259642.0000
Epoch 36/200, Train Loss: 34613079.7778, Val Loss: 79257974.6667
Epoch 37/200, Train Loss: 34620009.5556, Val Loss: 79256264.6667
Epoch 38/200, Train Loss: 34626838.0000, Val Loss: 79254457.3333
Epoch 39/200, Train Loss: 34633572.4444, Val Loss: 79252520.6667
Epoch 40/200, Train Loss: 34640226.4444, Val Loss: 79250769.3333
Epoch 41/200, Train Loss: 34646780.0000, Val Loss: 79248818.6667
Epoch 42/200, Train Loss: 34653318.0000, Val Loss: 79246816.0000
Epoch 43/200, Train Loss: 34659750.6667, Val Loss: 79244817.3333
Epoch 44/200, Train Loss: 34666085.5556, Val Loss: 79242832.6667
Epoch 45/200, Train Loss: 34672368.6667, Val Loss: 79240697.3333
Epoch 46/200, Train Loss: 34678637.7778, Val Loss: 79238604.6667
Epoch 47/200, Train Loss: 34684776.8889, Val Loss: 79236459.3333
Epoch 48/200, Train Loss: 34690862.6667, Val Loss: 79234287.3333
Epoch 49/200, Train Loss: 34696910.6667, Val Loss: 79232089.3333
Epoch 50/200, Train Loss: 34702851.3333, Val Loss: 79229896.6667
Epoch 51/200, Train Loss: 34708784.8889, Val Loss: 79227627.3333
Epoch 52/200, Train Loss: 34714663.1111, Val Loss: 79225302.6667
Epoch 53/200, Train Loss: 34720474.6667, Val Loss: 79222999.3333
Epoch 54/200, Train Loss: 34726182.8889, Val Loss: 79220578.0000
Epoch 55/200, Train Loss: 34731846.4444, Val Loss: 79218210.6667
Epoch 56/200, Train Loss: 34737503.7778, Val Loss: 79215882.0000
Epoch 57/200, Train Loss: 34743058.8889, Val Loss: 79213352.6667
Epoch 58/200, Train Loss: 34748604.4444, Val Loss: 79210885.3333
Epoch 59/200, Train Loss: 34754079.1111, Val Loss: 79208416.0000
Epoch 60/200, Train Loss: 34759483.3333, Val Loss: 79205903.3333
```

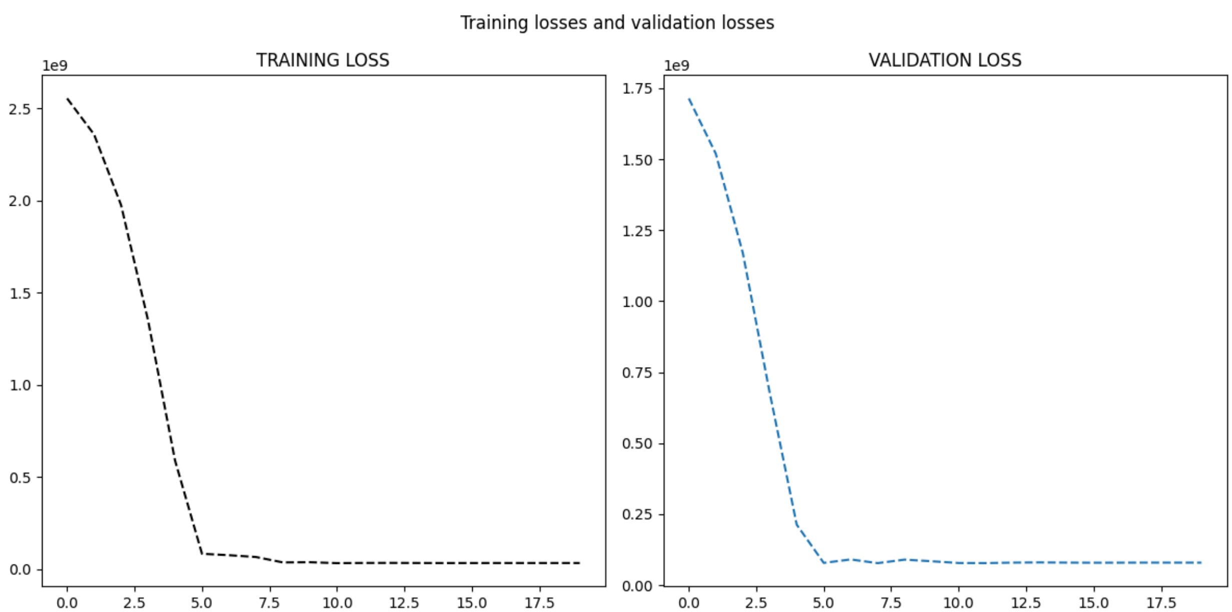
```
Epoch 61/200, Train Loss: 34764882.0000, Val Loss: 79203353.3333
Epoch 62/200, Train Loss: 34770192.6667, Val Loss: 79200752.6667
Epoch 63/200, Train Loss: 34775459.1111, Val Loss: 79198102.0000
Epoch 64/200, Train Loss: 34780725.5556, Val Loss: 79195523.3333
Epoch 65/200, Train Loss: 34785859.1111, Val Loss: 79192858.0000
Epoch 66/200, Train Loss: 34791038.0000, Val Loss: 79190205.3333
Epoch 67/200, Train Loss: 34796112.0000, Val Loss: 79187505.3333
Epoch 68/200, Train Loss: 34801174.2222, Val Loss: 79184844.6667
Epoch 69/200, Train Loss: 34806147.1111, Val Loss: 79182026.6667
Epoch 70/200, Train Loss: 34811151.7778, Val Loss: 79179291.3333
Epoch 71/200, Train Loss: 34816039.3333, Val Loss: 79176516.0000
Epoch 72/200, Train Loss: 34820908.2222, Val Loss: 79173661.3333
Epoch 73/200, Train Loss: 34825735.1111, Val Loss: 79170844.0000
Epoch 74/200, Train Loss: 34830520.2222, Val Loss: 79167981.3333
Epoch 75/200, Train Loss: 34835304.6667, Val Loss: 79165162.6667
Epoch 76/200, Train Loss: 34839985.7778, Val Loss: 79162262.0000
Epoch 77/200, Train Loss: 34844643.5556, Val Loss: 79159386.0000
Epoch 78/200, Train Loss: 34849274.4444, Val Loss: 79156443.3333
Epoch 79/200, Train Loss: 34853868.4444, Val Loss: 79153488.0000
Epoch 80/200, Train Loss: 34858412.4444, Val Loss: 79150482.6667
Epoch 81/200, Train Loss: 34862949.3333, Val Loss: 79147535.3333
Epoch 82/200, Train Loss: 34867371.5556, Val Loss: 79144496.6667
Epoch 83/200, Train Loss: 34871812.4444, Val Loss: 79141498.0000
Epoch 84/200, Train Loss: 34876227.1111, Val Loss: 79138474.6667
Epoch 85/200, Train Loss: 34880588.2222, Val Loss: 79135417.3333
Epoch 86/200, Train Loss: 34884893.3333, Val Loss: 79132396.6667
Epoch 87/200, Train Loss: 34889199.3333, Val Loss: 79129300.0000
Epoch 88/200, Train Loss: 34893432.6667, Val Loss: 79126096.6667
Epoch 89/200, Train Loss: 34897668.2222, Val Loss: 79123030.6667
Epoch 90/200, Train Loss: 34901846.8889, Val Loss: 79119899.3333
Epoch 91/200, Train Loss: 34905980.2222, Val Loss: 79116770.0000
Epoch 92/200, Train Loss: 34910061.7778, Val Loss: 79113518.6667
Epoch 93/200, Train Loss: 34914163.7778, Val Loss: 79110393.3333
Epoch 94/200, Train Loss: 34918176.0000, Val Loss: 79107138.6667
Epoch 95/200, Train Loss: 34922205.3333, Val Loss: 79103932.0000
Epoch 96/200, Train Loss: 34926156.8889, Val Loss: 79100753.3333
Epoch 97/200, Train Loss: 34930081.1111, Val Loss: 79097526.0000
Epoch 98/200, Train Loss: 34933987.1111, Val Loss: 79094266.6667
Epoch 99/200, Train Loss: 34937870.6667, Val Loss: 79091019.3333
Epoch 100/200, Train Loss: 34941712.8889, Val Loss: 79087700.0000
Epoch 101/200, Train Loss: 34945472.2222, Val Loss: 79084387.3333
Epoch 102/200, Train Loss: 34949239.1111, Val Loss: 79081064.0000
Epoch 103/200, Train Loss: 34953003.1111, Val Loss: 79077727.3333
Epoch 104/200, Train Loss: 34956703.7778, Val Loss: 79074385.3333
Epoch 105/200, Train Loss: 34960370.6667, Val Loss: 79071082.0000
Epoch 106/200, Train Loss: 34963961.3333, Val Loss: 79067698.6667
Epoch 107/200, Train Loss: 34967593.7778, Val Loss: 79064282.0000
Epoch 108/200, Train Loss: 34971146.8889, Val Loss: 79060851.3333
Epoch 109/200, Train Loss: 34974729.7778, Val Loss: 79057532.0000
Epoch 110/200, Train Loss: 34978217.1111, Val Loss: 79054052.0000
Epoch 111/200, Train Loss: 34981717.3333, Val Loss: 79050662.6667
Epoch 112/200, Train Loss: 34985137.5556, Val Loss: 79047231.3333
Epoch 113/200, Train Loss: 34988567.5556, Val Loss: 79043742.6667
Epoch 114/200, Train Loss: 34991944.6667, Val Loss: 79040313.3333
Epoch 115/200, Train Loss: 34995312.6667, Val Loss: 79036854.6667
Epoch 116/200, Train Loss: 34998636.8889, Val Loss: 79033354.6667
Epoch 117/200, Train Loss: 35001916.8889, Val Loss: 79029811.3333
Epoch 118/200, Train Loss: 35005182.8889, Val Loss: 79026335.3333
Epoch 119/200, Train Loss: 35008418.8889, Val Loss: 79022813.3333
Epoch 120/200, Train Loss: 35011637.7778, Val Loss: 79019354.0000
```

```
Epoch 121/200, Train Loss: 35014762.0000, Val Loss: 79015694.0000
Epoch 122/200, Train Loss: 35017935.3333, Val Loss: 79012226.6667
Epoch 123/200, Train Loss: 35021038.8889, Val Loss: 79008638.6667
Epoch 124/200, Train Loss: 35024139.1111, Val Loss: 79005108.0000
Epoch 125/200, Train Loss: 35027241.5556, Val Loss: 79001503.3333
Epoch 126/200, Train Loss: 35030221.1111, Val Loss: 78997896.6667
Epoch 127/200, Train Loss: 35033184.4444, Val Loss: 78994306.6667
Epoch 128/200, Train Loss: 35036165.3333, Val Loss: 78990722.0000
Epoch 129/200, Train Loss: 35039147.1111, Val Loss: 78987062.6667
Epoch 130/200, Train Loss: 35042018.8889, Val Loss: 78983416.6667
Epoch 131/200, Train Loss: 35044930.0000, Val Loss: 78979840.0000
Epoch 132/200, Train Loss: 35047772.2222, Val Loss: 78976203.3333
Epoch 133/200, Train Loss: 35050604.0000, Val Loss: 78972511.3333
Epoch 134/200, Train Loss: 35053389.5556, Val Loss: 78968849.3333
Epoch 135/200, Train Loss: 35056161.1111, Val Loss: 78965159.3333
Epoch 136/200, Train Loss: 35058933.1111, Val Loss: 78961493.3333
Epoch 137/200, Train Loss: 35061633.7778, Val Loss: 78957850.6667
Epoch 138/200, Train Loss: 35064308.6667, Val Loss: 78954117.3333
Epoch 139/200, Train Loss: 35066998.2222, Val Loss: 78950425.3333
Epoch 140/200, Train Loss: 35069614.2222, Val Loss: 78946706.0000
Epoch 141/200, Train Loss: 35072221.5556, Val Loss: 78943012.0000
Epoch 142/200, Train Loss: 35074793.1111, Val Loss: 78939259.3333
Epoch 143/200, Train Loss: 35077380.4444, Val Loss: 78935466.6667
Epoch 144/200, Train Loss: 35079908.0000, Val Loss: 78931776.6667
Epoch 145/200, Train Loss: 35082375.5556, Val Loss: 78927994.0000
Epoch 146/200, Train Loss: 35084866.4444, Val Loss: 78924224.6667
Epoch 147/200, Train Loss: 35087326.2222, Val Loss: 78920438.6667
Epoch 148/200, Train Loss: 35089730.8889, Val Loss: 78916704.0000
Epoch 149/200, Train Loss: 35092148.0000, Val Loss: 78912864.6667
Epoch 150/200, Train Loss: 35094547.5556, Val Loss: 78909118.6667
Epoch 151/200, Train Loss: 35096855.7778, Val Loss: 78905285.3333
Epoch 152/200, Train Loss: 35099198.2222, Val Loss: 78901513.3333
Epoch 153/200, Train Loss: 35101495.5556, Val Loss: 78897745.3333
Epoch 154/200, Train Loss: 35103735.5556, Val Loss: 78893816.0000
Epoch 155/200, Train Loss: 35106004.0000, Val Loss: 78890070.6667
Epoch 156/200, Train Loss: 35108218.6667, Val Loss: 78886200.6667
Epoch 157/200, Train Loss: 35110397.1111, Val Loss: 78882344.0000
Epoch 158/200, Train Loss: 35112572.0000, Val Loss: 78878528.6667
Epoch 159/200, Train Loss: 35114724.2222, Val Loss: 78874621.3333
Epoch 160/200, Train Loss: 35116857.3333, Val Loss: 78870752.0000
Epoch 161/200, Train Loss: 35118938.2222, Val Loss: 78866872.6667
Epoch 162/200, Train Loss: 35121037.1111, Val Loss: 78863018.0000
Epoch 163/200, Train Loss: 35123046.6667, Val Loss: 78859106.6667
Epoch 164/200, Train Loss: 35125107.5556, Val Loss: 78855212.6667
Epoch 165/200, Train Loss: 35127118.2222, Val Loss: 78851382.6667
Epoch 166/200, Train Loss: 35129084.0000, Val Loss: 78847464.0000
Epoch 167/200, Train Loss: 35131033.3333, Val Loss: 78843578.6667
Epoch 168/200, Train Loss: 35132943.3333, Val Loss: 78839604.0000
Epoch 169/200, Train Loss: 35134862.8889, Val Loss: 78835711.3333
Epoch 170/200, Train Loss: 35136737.1111, Val Loss: 78831784.0000
Epoch 171/200, Train Loss: 35138595.5556, Val Loss: 78827840.0000
Epoch 172/200, Train Loss: 35140451.5556, Val Loss: 78823898.0000
Epoch 173/200, Train Loss: 35142253.3333, Val Loss: 78819955.3333
Epoch 174/200, Train Loss: 35144043.5556, Val Loss: 78816052.0000
Epoch 175/200, Train Loss: 35145806.2222, Val Loss: 78812066.0000
Epoch 176/200, Train Loss: 35147554.6667, Val Loss: 78808084.6667
Epoch 177/200, Train Loss: 35149278.4444, Val Loss: 78804145.3333
Epoch 178/200, Train Loss: 35150969.3333, Val Loss: 78800176.6667
Epoch 179/200, Train Loss: 35152671.5556, Val Loss: 78796207.3333
Epoch 180/200, Train Loss: 35154334.2222, Val Loss: 78792161.3333
```

```
Epoch 181/200, Train Loss: 35156002.4444, Val Loss: 78788256.6667
Epoch 182/200, Train Loss: 35157581.3333, Val Loss: 78784157.3333
Epoch 183/200, Train Loss: 35159180.4444, Val Loss: 78780166.6667
Epoch 184/200, Train Loss: 35160720.6667, Val Loss: 78776195.3333
Epoch 185/200, Train Loss: 35162289.7778, Val Loss: 78772159.3333
Epoch 186/200, Train Loss: 35163839.1111, Val Loss: 78768167.3333
Epoch 187/200, Train Loss: 35165334.2222, Val Loss: 78764159.3333
Epoch 188/200, Train Loss: 35166818.6667, Val Loss: 78760105.3333
Epoch 189/200, Train Loss: 35168294.2222, Val Loss: 78756108.0000
Epoch 190/200, Train Loss: 35169730.4444, Val Loss: 78752067.3333
Epoch 191/200, Train Loss: 35171186.4444, Val Loss: 78747986.6667
Epoch 192/200, Train Loss: 35172546.0000, Val Loss: 78743958.6667
Epoch 193/200, Train Loss: 35173945.1111, Val Loss: 78739900.6667
Epoch 194/200, Train Loss: 35175310.2222, Val Loss: 78735838.0000
Epoch 195/200, Train Loss: 35176666.6667, Val Loss: 78731850.0000
Epoch 196/200, Train Loss: 35177964.4444, Val Loss: 78727742.6667
Epoch 197/200, Train Loss: 35179278.6667, Val Loss: 78723668.0000
Epoch 198/200, Train Loss: 35180580.4444, Val Loss: 78719649.3333
Epoch 199/200, Train Loss: 35181793.3333, Val Loss: 78715453.3333
Epoch 200/200, Train Loss: 35183080.2222, Val Loss: 78711516.0000
```

In [80]:

```python
fig, axs = plt.subplots(1, 2, figsize=(12,6))
fig.suptitle('Training losses and validation losses')

axs[0].set_title("TRAINING LOSS")
axs[0].plot(train_losses[:20], c = "k", ls="--")

axs[1].set_title("VALIDATION LOSS")
axs[1].plot(val_losses[:20], ls="--")

fig.tight_layout()
```



In [79]:

```python
validation_vector
```

Out[79]:
```
[5915717728.400635,
 51982719.97009659,
 15323096.17652893,
 16059128.74630189,
 7105078045.140625,
 6783438559.541077,
 7590025080.5625,
 4217561792.29303,
 15266640.719262123,
 5474262294.128967,
 18127852.95291519,
 66661890.746355295,
 18608177.87933445,
 7148958794.484619,
 4486891841.320557,
 6207433224.664307,
 4833647409.691406]
```

In [85]:
```
val_losses[-5:]
```

Out[85]:
```
[78727742.66666667,
 78723668.0,
 78719649.33333333,
 78715453.33333333,
 78711516.0]
```

- Training is not happening. There is no optimization, the curve is flat so straightaway something is wrong in the code even if it makes logical sense.
- Awfully disgusting training !! Need to tune our model or something is wrong with the implementation. The validation vector should be ideally of the order E-03 (or 10^{-3}) at least.
- Also, I do not know why the validation vector is of size 17 when the number of graphs that I put in were 85. Maybe it is pytorch batch size stuff. I must have made some mistake somewhere. Validation vector must be 85 in total and of the order at least E-02 (or 10^{-2}).
- Further experimentation remains. For that we shall use a Graph Attention Transformer instead of a Graph Convolutional Neural Network.

Now training is happening but the validation vector is giving horrific results !! So something needs to change to verify whether the model has been correctly trained. Also not sure whether the validation loss and training loss is correct or not.

In [ ]: