vignette

December 13, 2024

1 Introduction

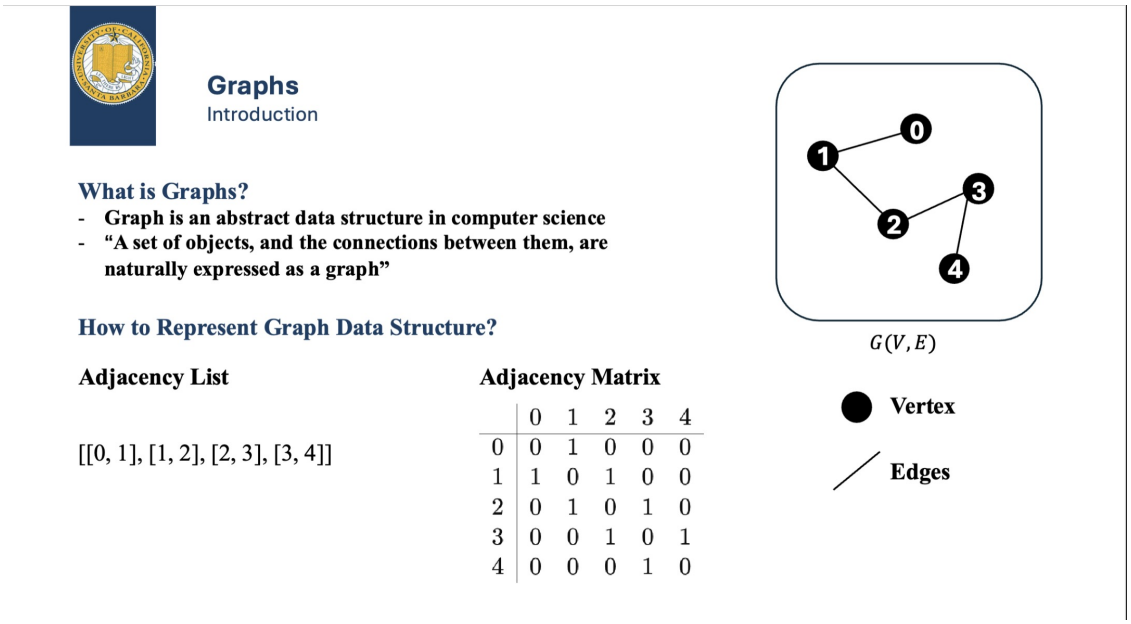
In this vignette, we touch on the topic of cheminformatics and bioinformatics, aiming to predict the solubility of molecules using deep learning. The neural network we will be working on is called **Graph Neural Network**. It serves as a powerful tool for learning from graph-structured data, which is perfectly suitable for the molecular data which atoms and bonds can be represented as a graph.

The vignette is divided into three sections. In the first section, we will introduce the graph data structure. The second part will be about the data. In the last section, we will build a graph neural network to predict the solubility of molecules.

1.1 Graph Data Structure

A graph data structure consists of a finite (and possibly mutable) set of vertices (also called nodes or points), together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. [1]

1.1.1 Implement a Graph Data Structure



Here is a short demo to implement an undirected graph data structure in Python using adjacency list.

```
class vertex:
    def __init__(self, value):
        self.value = value # info the vertex stores
        self.index = None # index of the vertex in the graph
        self.edges = [] # list of edges connected to the vertex

class graph:
    def __init__(self):
        self.vertices = [] # list of vertices in the graph
        self.adjacency_list = {} # dictionary to store the vertices and edges

    def add_vertex(self, value):
        v = vertex(value)
        v.index = len(self.vertices) # assign index to the vertex
        self.vertices.append(v)
        self.adjacency_list[v.index] = [] # record the edges connected to the vertex
        return v

    def add_edge(self, v1, v2):
        self.adjacency_list[v1.index].append(v2.index) # v1 connects to v2
        self.adjacency_list[v2.index].append(v1.index) # v2 connects to v1
        v1.edges.append(v2)
        v2.edges.append(v1)
```

Usage:

```
g = graph()
v1 = g.add_vertex(1)
v2 = g.add_vertex(2)
v3 = g.add_vertex(3)
g.add_edge(v1, v2)
g.add_edge(v2, v3)
```

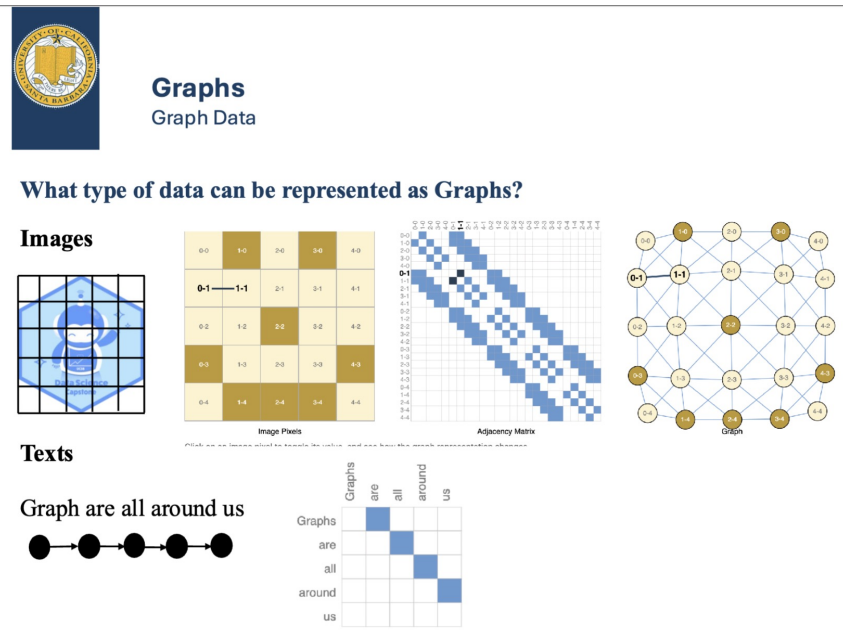
1.1.2 Your Task:

Implement a **directed** graph data structure in Python using adjacency matrix.

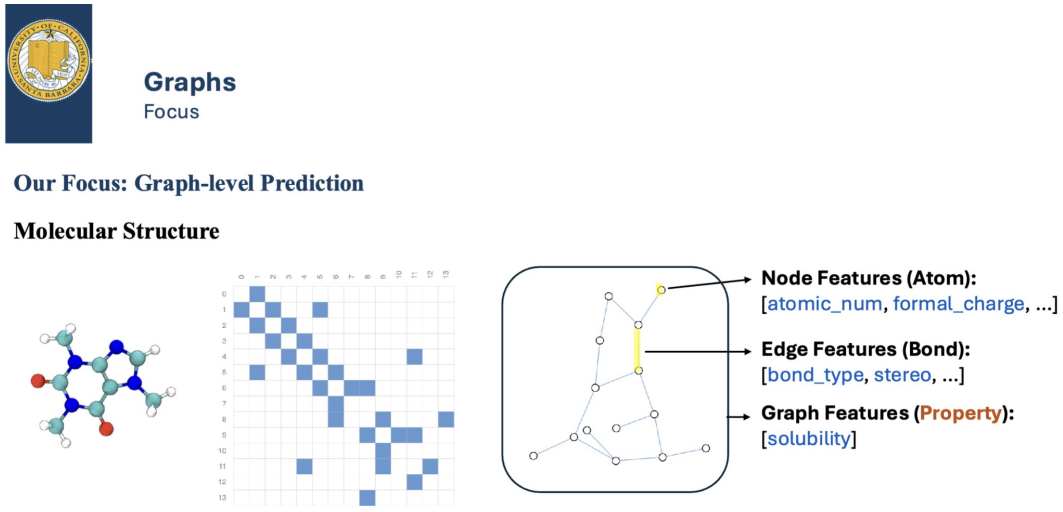
1.1.3 What type of data can be represented as a graph?

Graph data structure can be used to represent most of data, like images and texts. One of the key features of graph data structure is that it can represent the relationship between data points.

- Images: Each pixel can be a node, and each node connects to all its neighbors.
- Texts: Each word can be a node, and each node connects to the words that appear in the same sentence.



Besides of texts and images. Molecules can be naturally represented as a graph, with atoms as nodes and bonds as edges.



1.2 Data

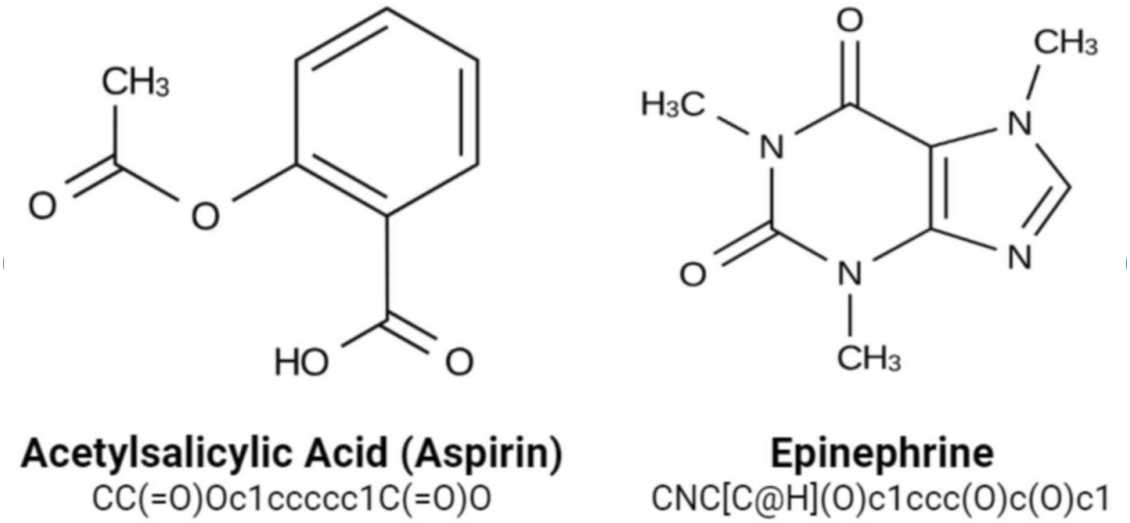
1.3 Data preview and preparation

To predict the solubility of molecules, we will use the **ESOL dataset** that is built in the **torch_geometric** packages and `datasets` modules. The dataset contains 1,128 molecules and their solubility in water. We will use **RDKit** to process the molecules.

RDKit is an open-source toolkit for cheminformatics. It provides a wide range of functions for processing molecules, such as reading and writing molecules, calculating molecular descriptors, and generating molecular fingerprints. Specifically, the `MoleculeNet` class in `torch_geometric.datasets` module utilized `from_smiles` to convert the SMILES representation of molecules to graph data structure.

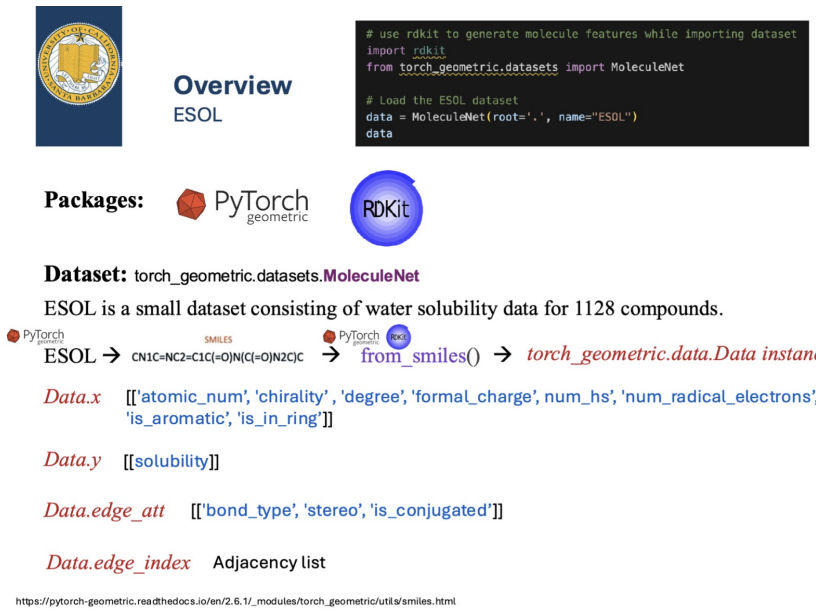
> SMILES is a specification in the form of a line notation for describing the structure of chemical species using short ASCII strings 2

For example, acetylsalicylic acid (aspirin) can be represented as `CC(=O)Oc1ccccc1C(=O)O`, and epinephrine can be represented as `CNC[C@H](O)c1ccc(O)c(O)c1`. 3



ESOL provides the SMILES representation of a molecule and uses `from_smiles` to return a `Data` instance, which contains the following essentiall attribute:

- `x`: Node feature matrix with shape `[num_nodes, 9]`, where the `num_nodes` is the number of atoms in the molecule and 9 is the number of features per atom. These features that are stored in nodes are `atomic_num`, `chirality`, `degree`, `formal_charge`, `hybridization`, `is_aromatic`, `is_in_ring`, `num_radical_electrons`, and `num_hs`.
- `y`: Target to predict, which is `solubility` here.
- `edge_att`: Edge feature matrix with shape `[num_edges, 3]`, where the `num_edges` is the number of bonds in the molecule and 3 is the number of features that are stored in each bond. These features are `bond_type`, `is_conjugated`, and `stereo`.
- `edge_index`: Graph connectivity in COO format with shape `[num_edges, 2]`, where the first element in the first row is the source node and the first element in the second row is the target node.



1.3.1 Code Demo

[2]:
```
# use rdkit to generate molecule features while importing dataset
import rdkit
from torch_geometric.datasets import MoleculeNet

# Load and connect the ESOL dataset
data = MoleculeNet(root='.', name="ESOL")
data
```

[2]:
```
ESOL(1128)
```

[3]:
```
# Investigating the dataset
print("Dataset type: ", type(data))
print("Dataset features: ", data.num_features)
print("Dataset target: ", data[0].y.shape[0])
print("Dataset length: ", data.len())
print("Dataset sample: ", data[0])
print("Sample nodes: ", data[0].num_nodes)
print("Sample edges: ", data[0].num_edges)
```

```
Dataset type:  <class 'torch_geometric.datasets.molecule_net.MoleculeNet'>
Dataset features:  9
Dataset target:  1
Dataset length:  1128
Dataset sample:  Data(x=[32, 9], edge_index=[2, 68], edge_attr=[68, 3],
smiles='OCC3OC(OCC2OC(OC(C#N)c1ccccc1)C(O)C(O)C2O)C(O)C(O)C3O ', y=[1, 1])
Sample nodes:  32
Sample edges:  68
```

Each node represents an atom with descriptive features such as atomic number, hybridization state, and charge. The node feature matrix for a graph has a shape [num_nodes, num_features], where each row is a node's feature vector.

[44]:
```
# Investigating the features
# Shape: [num_nodes, num_node_features = 9]
data[0].x
```

[44]:
```
tensor([[8, 0, 2, 5, 1, 0, 4, 0, 0],
        [6, 0, 4, 5, 2, 0, 4, 0, 0],
        [6, 0, 4, 5, 1, 0, 4, 0, 1],
        [8, 0, 2, 5, 0, 0, 4, 0, 1],
        [6, 0, 4, 5, 1, 0, 4, 0, 1],
        [8, 0, 2, 5, 0, 0, 4, 0, 0],
        [6, 0, 4, 5, 2, 0, 4, 0, 0],
        [6, 0, 4, 5, 1, 0, 4, 0, 1],
        [8, 0, 2, 5, 0, 0, 4, 0, 1],
        [6, 0, 4, 5, 1, 0, 4, 0, 1],
        [8, 0, 2, 5, 0, 0, 4, 0, 0],
        [6, 0, 4, 5, 1, 0, 4, 0, 0],
        [6, 0, 2, 5, 0, 0, 2, 0, 0],
        [7, 0, 1, 5, 0, 0, 2, 0, 0],
        [6, 0, 3, 5, 0, 0, 3, 1, 1],
        [6, 0, 3, 5, 1, 0, 3, 1, 1],
        [6, 0, 3, 5, 1, 0, 3, 1, 1],
        [6, 0, 3, 5, 1, 0, 3, 1, 1],
        [6, 0, 3, 5, 1, 0, 3, 1, 1],
        [6, 0, 3, 5, 1, 0, 3, 1, 1],
        [6, 0, 4, 5, 1, 0, 4, 0, 1],
        [8, 0, 2, 5, 1, 0, 4, 0, 0],
        [6, 0, 4, 5, 1, 0, 4, 0, 1],
        [8, 0, 2, 5, 1, 0, 4, 0, 0],
        [6, 0, 4, 5, 1, 0, 4, 0, 1],
        [8, 0, 2, 5, 1, 0, 4, 0, 0],
        [6, 0, 4, 5, 1, 0, 4, 0, 1],
        [8, 0, 2, 5, 1, 0, 4, 0, 0],
        [6, 0, 4, 5, 1, 0, 4, 0, 1],
        [8, 0, 2, 5, 1, 0, 4, 0, 0],
        [6, 0, 4, 5, 1, 0, 4, 0, 1],
        [8, 0, 2, 5, 1, 0, 4, 0, 0]])
```

Edges represent bonds information between atoms, stored efficiently in sparse COO format.

[68]:
```
# Shape: [num_edges, edge_features = 3]
data[0].edge_attr
```

[68]:
```
tensor([[ 1,  0,  0],
        [ 1,  0,  0],
```

```
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 3,  0,  0],
[ 3,  0,  0],
[ 1,  0,  0],
[12,  0,  1],
[12,  0,  1],
[12,  0,  1],
[12,  0,  1],
[12,  0,  1],
[12,  0,  1],
[12,  0,  1],
[12,  0,  1],
[12,  0,  1],
[12,  0,  1],
[12,  0,  1],
[12,  0,  1],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
[ 1,  0,  0],
```

```
        [ 1,  0,  0],
        [ 1,  0,  0],
        [ 1,  0,  0],
        [ 1,  0,  0],
        [ 1,  0,  0],
        [ 1,  0,  0],
        [ 1,  0,  0],
        [ 1,  0,  0],
        [ 1,  0,  0],
        [ 1,  0,  0],
        [ 1,  0,  0],
        [ 1,  0,  0],
        [ 1,  0,  0],
        [ 1,  0,  0],
        [ 1,  0,  0],
        [ 1,  0,  0],
        [ 1,  0,  0],
        [ 1,  0,  0],
        [ 1,  0,  0]])
```

Each edge is a pair of connected nodes, and the edge matrix has a shape [2, num_edges]. This format is computationally efficient for large, sparse graphs. Edges allow GNNs to propagate information between connected nodes, capturing molecular relationships and structure.

[5]:
```
# Investigating the edges in sparse COO format
# Shape: [2, num_edges]
# i.e. node 0 is connected to node 1
# i.e. node 1 is connected to node 0
# i.e. node 1 is connected to node 2
# etc.
# NOTE: more efficient representation than adjacency matrix
data[0].edge_index.t()
```

[5]:
```
tensor([[ 0,  1],
        [ 1,  0],
        [ 1,  2],
        [ 2,  1],
        [ 2,  3],
        [ 2, 30],
        [ 3,  2],
        [ 3,  4],
        [ 4,  3],
        [ 4,  5],
        [ 4, 26],
        [ 5,  4],
        [ 5,  6],
        [ 6,  5],
        [ 6,  7],
```

```
[ 7,  6],
[ 7,  8],
[ 7, 24],
[ 8,  7],
[ 8,  9],
[ 9,  8],
[ 9, 10],
[ 9, 20],
[10,  9],
[10, 11],
[11, 10],
[11, 12],
[11, 14],
[12, 11],
[12, 13],
[13, 12],
[14, 11],
[14, 15],
[14, 19],
[15, 14],
[15, 16],
[16, 15],
[16, 17],
[17, 16],
[17, 18],
[18, 17],
[18, 19],
[19, 14],
[19, 18],
[20,  9],
[20, 21],
[20, 22],
[21, 20],
[22, 20],
[22, 23],
[22, 24],
[23, 22],
[24,  7],
[24, 22],
[24, 25],
[25, 24],
[26,  4],
[26, 27],
[26, 28],
[27, 26],
[28, 26],
[28, 29],
```

```
[28, 30],
[29, 28],
[30,  2],
[30, 28],
[30, 31],
[31, 30]])
```

Each sample in the dataset includes a target value y, the solubility of the corresponding molecule.

[7]:
```
data[0].y
```

[7]:
```
tensor([[-0.7700]])
```

1.3.2 The SMILES Feature from ESOL

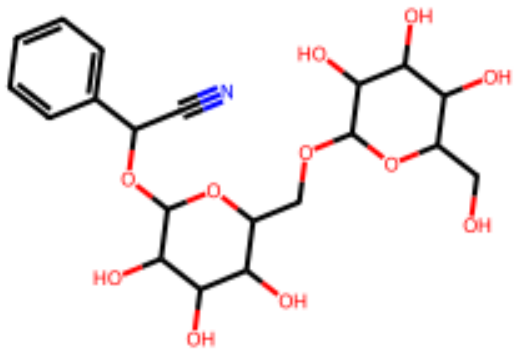
The ESOL dataset includes molecules in the form of SMILES (Simplified Molecular Input Line Entry System) strings, a standard representation for chemical structures. Using RDKit, these SMILES strings are converted into molecular graph representations for visualization and further processing.

[47]:
```
data[0]["smiles"]
```

[47]:
```
'OCC3OC(OCC2OC(OC(C#N)c1ccccc1)C(O)C(O)C2O)C(O)C(O)C3O '
```

[48]:
```
from rdkit import Chem
from rdkit.Chem.Draw import IPythonConsole
molecule = Chem.MolFromSmiles(data[0]["smiles"])
molecule
```

[48]:



1.4 Graph Neural Network

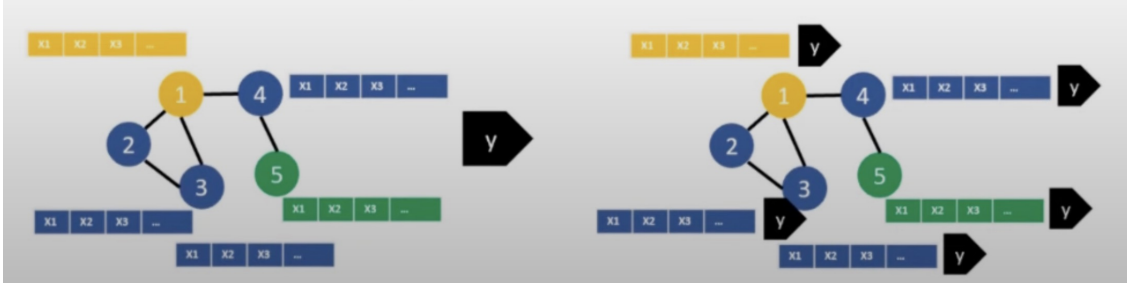
1.4.1 What type of problem can be solved by Graph Neural Network?

Graph neural network utilizes the advantange of graph data structure to learn from graph-structured data, specially the connectivity and how information can be stored in all three components of graph, namely graph, nodes and edges. It is a powerful tool for learning from molecular data.

Typically, there are three types of problems that can be solved by graph neural network:

- Graph-level prediction: Predict the properties of the whole graph, such as the solubility of an entire molecule.
- Node-level prediction: Predict the properties of each node in the graph, such as the atom property of each atom in a molecule. For example, will the atom be in a ring? Will the atom be aromatic? Will the atom's hybridization be sp3? etc.
- Edge-level prediction: Predict the properties of each edge in the graph, such as the bond type of each bond in a molecule. For example, will the bond between the carbon and oxygen be a single bond or double bond? Will the bond be conjugated? Will the bond be stereo? etc.

Left: Graph-level prediction. Right: Node-level prediction.



Clearly, our task is a **graph-level prediction problem**. We will build a graph neural network to predict the solubility of molecules.

1.4.2 Graph Convolutional Network (GCN) Implementation

To deeply understand what everything is happening in the code, we will take a look at the source code of the Graph Convolutional Network (GCN) in PyTorch. Along the way, we will explain both the mathematical and implementation details of the architecture.

Understand Linear Layer 4

Overlook

```
class Linear(Module):
  __constants__ = ["in_features", "out_features"]
    in_features: int
    out_features: int
    weight: Tensor

    def __init__(
        self,
        in_features: int,
        out_features: int,
        bias: bool = True,
        device=None,
        dtype=None,
    ) -> None:
        factory_kwargs = {"device": device, "dtype": dtype}
```

```
        super().__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = Parameter(
            torch.empty((out_features, in_features), **factory_kwargs)
        )
        if bias:
            self.bias = Parameter(torch.empty(out_features, **factory_kwargs))
        else:
            self.register_parameter("bias", None)
        self.reset_parameters()

    def reset_parameters(self) -> None:
        # Setting a=sqrt(5) in kaiming_uniform is the same as initializing with
        # uniform(-1/sqrt(in_features), 1/sqrt(in_features)). For details, see
        # https://github.com/pytorch/pytorch/issues/57109
        init.kaiming_uniform_(self.weight, a=math.sqrt(5))
        if self.bias is not None:
            fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
            bound = 1 / math.sqrt(fan_in) if fan_in > 0 else 0
            init.uniform_(self.bias, -bound, bound)

    def forward(self, input: Tensor) -> Tensor:
        return F.linear(input, self.weight, self.bias)

    def extra_repr(self) -> str:
        return f"in_features={self.in_features}, out_features={self.out_features}, bias={self.bi
```

Define Data Type The first snippet defines the constants that will stay unchanged duirng the lifecycle of the `Linear` class, and the data types of the input parameters.

- `in_features`: The number of input features.
- `out_features`: The number of output features.
- `weight`: The learnable weights of the layer, which is a tensor object (like numpy array).

```
__constants__ = ["in_features", "out_features"]
  in_features: int
  out_features: int
  weight: Tensor
```

Initialization Similarly, the `__init__()` function in the `Linear` class initializes the attributes of the class instance.

```
    def __init__(
        self,
        in_features: int,
        out_features: int,
        bias: bool = True,
```

```
        device=None,
        dtype=None,
    ) -> None:
```

The arrow `-> None` thing is a convention in Python to indicate that the function returns `None`. For example:

```
def some_function() -> None:
    print("This function doesn't return anything explicit.")
```

```
factory_kwargs = {"device": device, "dtype": dtype}
        super().__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = Parameter(
            torch.empty((out_features, in_features), **factory_kwargs)
        )
```

The factory_kwargs is a dictionary that contains the device and data type of the tensor. They will be passed into the some pytorch built-in functions to create tensors that having the specified device and data type, like the `torch.empty()` function. Because gpu is faster than cpu, which is efficient at dealing floating point operations, many deep learning models are trained on gpu. The `device` parameter is used to specify the device that the tensor will be stored on.

For example:

```
torch.device('cuda:0') # tensor is assigned to gpu:0 (if multiple gpus are available)

torch.device('cpu') # tensor is assgined to cpu
```

Here, the `device` and `dtype` params are passed to the `torch.empty()` function to create a tensor with the specified device and data type. `torch.empty()` initializes the tensor with random values of specified shape.

```
torch.empty((2,3), dtype=torch.int64)
```

```
tensor([[ 9.4064e+13,  2.8000e+01,  9.3493e+13],
        [ 7.5751e+18,  7.1428e+18,  7.5955e+18]])
```

`super().__init__()` calls the parent class's `__init__()` method, which is the `Module` class in this case. The `Module` class is the base class for all neural network modules in PyTorch. It provides a lot of useful methods and attributes for building neural networks, for example the `register_parameter()` method.

```
if bias:
    self.bias = Parameter(torch.empty(out_features, **factory_kwargs))
else:
    self.register_parameter("bias", None)
self.reset_parameters()
```

Here, the `bias` parameter is used to determine whether the layer will have a bias term. If `bias` is `True`, a bias tensor is created and stored in the `bias` attribute. `Parameter()` function from `torch.nn.parameter` acts similar like `reset_parameters()`, which will treat bias as learnable parameters for the layer. Otherwise, the `bias` attribute is set to `None`.

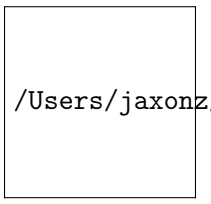
The GCN model is designed to process molecular graph data using PyTorch Geometric. It includes an initial convolutional layer and three additional GCN layers, each with an embedding size of 64, to progressively learn hierarchical graph representations. A global pooling layer combines max and mean pooling to aggregate node features into a graph-level representation, which is passed through a linear layer to predict the solubility value.

[49]:
```
import torch
from torch.nn import Linear
import torch.nn.functional as F
from torch_geometric.nn import GCNConv, TopKPooling, global_mean_pool
from torch_geometric.nn import global_mean_pool as gap, global_max_pool as gmp
embedding_size = 64

class GCN(torch.nn.Module):
  def __init__(self):
    # Init parent
    super(GCN, self).__init__()
    torch.manual_seed(197)

    # GCN layers
    self.initial_conv = GCNConv(data.num_features, embedding_size)
    self.conv1 = GCNConv(embedding_size, embedding_size)
    self.conv2 = GCNConv(embedding_size, embedding_size)
    self.conv3 = GCNConv(embedding_size, embedding_size)

    # Output layer
    self.out = Linear(embedding_size*2, data.y.shape[1]) # output shape 1 =␣
 ↪regression
    # twice the size for accomodating the global pooling layer

  def forward(self, x, edge_index, batch_index):
    # First Conv layer
    hidden = self.initial_conv(x, edge_index)
    hidden = F.tanh(hidden) # activation function

    # Other Conv layers
    hidden = self.conv1(hidden, edge_index)
    hidden = F.tanh(hidden)
    hidden = self.conv2(hidden, edge_index)
    hidden = F.tanh(hidden)
    hidden = self.conv3(hidden, edge_index)
    hidden = F.tanh(hidden)

    # Global Pooling Layer (stack different aggregations)
    hidden = torch.cat([gmp(hidden, batch_index),
                        gap(hidden, batch_index)], dim=1)
    # gmp = global max pooling
```

```
    # gap = gloabl average pooling
    # batch_index is used to select the relavent nodes for the pooling operation

    # Apply a final (linear) classifier.
    out = self.out(hidden)

    return out, hidden
```

1.4.3 Graph Pooling

Aggregate all information of the final graph



1.5 Model Summary and Parameter Count

The GCN model comprises an initial convolutional layer, three additional GCN layers, and a linear output layer, resulting in 107,806 parameters. This design balances complexity and computational efficiency, ensuring the model is expressive enough to capture molecular graph features while remaining practical for training.

[50]:
```
print(data.y.shape[1])
```

```
1
```

[51]:
```
model = GCN()
print(model)
print("Model parameters: ", sum(p.numel() for p in model.parameters()))
```

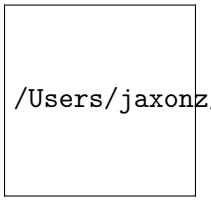
```
GCN(
  (initial_conv): GCNConv(9, 64)
  (conv1): GCNConv(64, 64)
  (conv2): GCNConv(64, 64)
  (conv3): GCNConv(64, 64)
  (out): Linear(in_features=128, out_features=1, bias=True)
)
Model parameters:  13249
```

1.6 Training the GNN

1.6.1 Batching with Graphs

Concatenate all node features in a large matrix. Combine each individual adjacency information into a large adjacency matrix or list.

NOTE: each individual graph is disconnected in the adjacency matrix, thus no information is shared between graphs.



The GNN is trained using the root MSE as the loss function and the Adam optimizer with a learning rate of 0.0007. The model is moved to a GPU if available, and the dataset is split 80:20 into training and testing sets. A DataLoader batches 64 graphs at a time, shuffling the data to ensure effective training.

[52]:
```
from torch_geometric.data import DataLoader
import warnings
warnings.filterwarnings("ignore")

# Root mean squared error
loss_fn = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0007)

# Use GPU for training
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = model.to(device)

# Wrap data in a data loader
data_size = len(data)
NUM_GRAPHS_PER_BATCH = 64
loader = DataLoader(data[:int(data_size * 0.8)],
                    batch_size=NUM_GRAPHS_PER_BATCH,
                    shuffle=True) # 20% training

test_loader = DataLoader(data[int(data_size * 0.8):],
                         batch_size=NUM_GRAPHS_PER_BATCH,
                         shuffle=True) # 80% testing
```

The GNN is trained over 2,000 epochs using batches of graphs. For each batch, predictions are made, loss is computed using root mean squared error, and gradients are updated to minimize the loss. The training progress is logged every 100 epochs, showing the model's improvement in solubility prediction.

[53]:
```
def train(data):
  # Enumerate over the data
  for batch in loader:
    # Use GPU
    batch.to(device)
```

```
    # Reset gradients
    optimizer.zero_grad()
    # Passing the node features and the connection info
    pred, embedding = model(batch.x.float(), batch.edge_index, batch.batch)
    # Calculating the loss and gradients
    loss = torch.sqrt(loss_fn(pred, batch.y))
    loss.backward()
    # Update using the gradients
    optimizer.step()
  return loss, embedding

print("Start training...")
losses = []
for epoch in range(2000):
  loss, h = train(data)
  losses.append(loss)
  if epoch % 100 == 0:
    print(f"Epoch {epoch} | Train Loss {loss}")
```

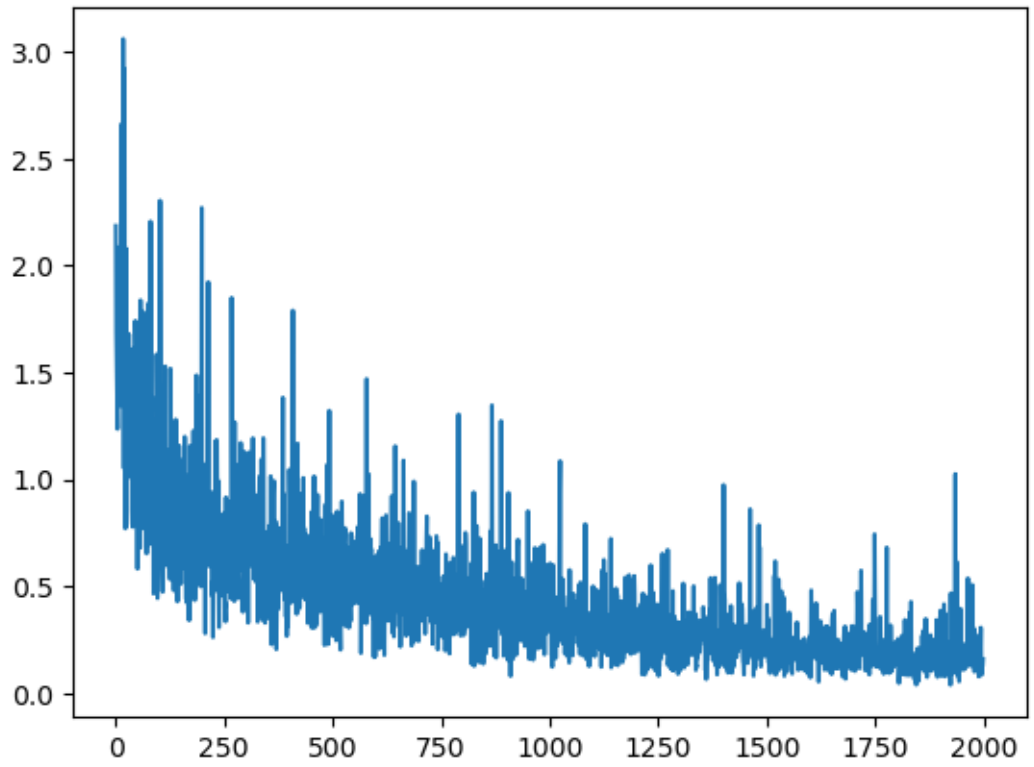
```
Start training...

Epoch 0 | Train Loss 2.18645977973938
Epoch 100 | Train Loss 0.7082846164703369
Epoch 200 | Train Loss 0.7985488772392273
Epoch 300 | Train Loss 0.5039166808128357
Epoch 400 | Train Loss 1.046067237854004
Epoch 500 | Train Loss 0.26778772473335266
Epoch 600 | Train Loss 0.6440434455871582
Epoch 700 | Train Loss 0.4965696334838867
Epoch 800 | Train Loss 0.4896949231624603
Epoch 900 | Train Loss 0.3031683564186096
Epoch 1000 | Train Loss 0.5666918754577637
Epoch 1100 | Train Loss 0.5001586079597473
Epoch 1200 | Train Loss 0.20951056480407715
Epoch 1300 | Train Loss 0.19144876301288605
Epoch 1400 | Train Loss 0.19382454454898834
Epoch 1500 | Train Loss 0.20428800582885742
Epoch 1600 | Train Loss 0.2662540078163147
Epoch 1700 | Train Loss 0.18329177796840668
Epoch 1800 | Train Loss 0.153456449508667
Epoch 1900 | Train Loss 0.12538152933120728
```

1.7 Visualizing the Training Loss

[54]:
```
import seaborn as sns
losses_float = [float(loss.cpu().detach().numpy()) for loss in losses]
loss_indices = [i for i, l in enumerate(losses_float)]
plt = sns.lineplot(x=loss_indices, y=losses_float)
```

```
plt
```

[54]: <Axes: >



1.8 Getting a test prediction

The model is tested on a batch of data in inference mode, and the real (y_real) and predicted (y_pred) solubility values are extracted and stored in a DataFrame for easy analysis. A scatter plot shows their alignment, with points near the diagonal indicating strong prediction accuracy.

[55]:
```
import pandas as pd

# Analyze the results for one batch
test_batch = next(iter(test_loader))
with torch.no_grad():
  test_batch.to(device)
  pred, embed = model(test_batch.x.float(), test_batch.edge_index, test_batch.
 ↪batch)
  df = pd.DataFrame()
  df["y_real"] = test_batch.y.tolist()
  df["y_pred"] = pred.tolist()
```

```
df["y_real"] = df["y_real"].apply(lambda row: row[0])
df["y_pred"] = df["y_pred"].apply(lambda row: row[0])
df
```

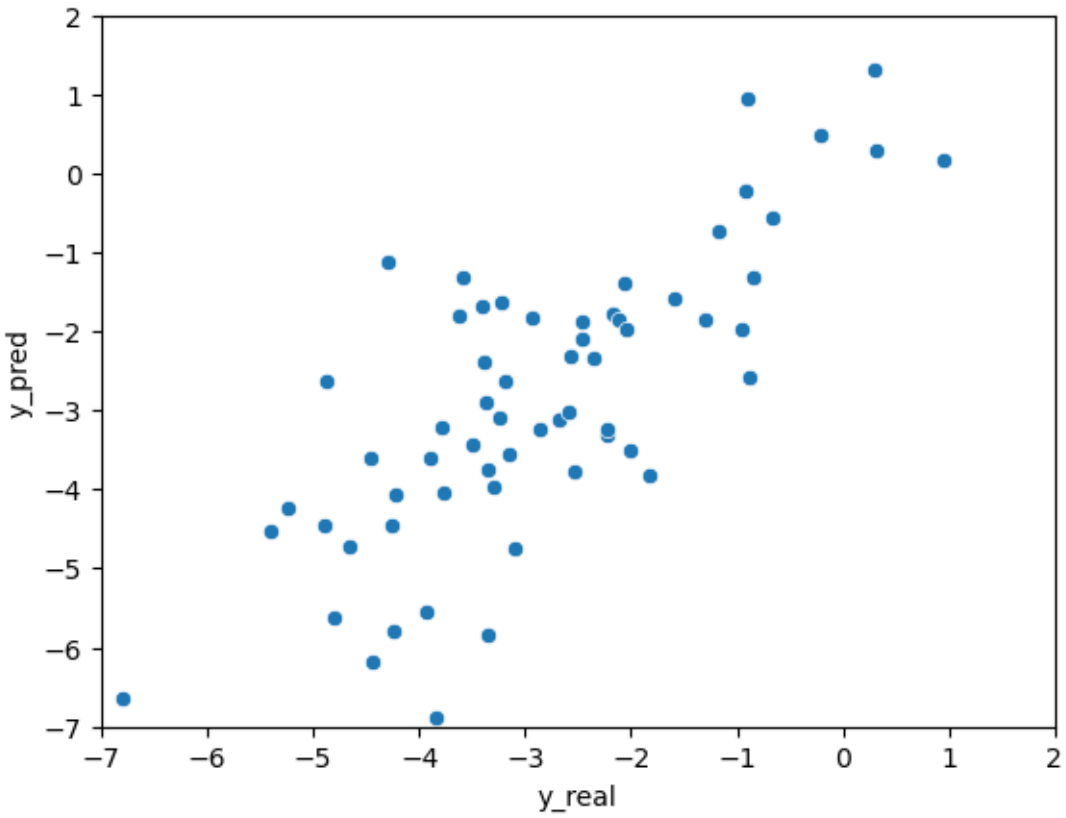
[55]:

	y_real	y_pred
0	-3.790	-3.201251
1	-4.880	-2.633608
2	-2.218	-3.296799
3	-4.900	-4.461855
4	-0.900	0.959851
..	...	...
59	-3.360	-3.739055
60	-2.220	-3.236182
61	-2.943	-1.826369
62	-3.350	-5.849453
63	-2.000	-3.515806

[64 rows x 2 columns]

[56]:
```
plt = sns.scatterplot(data=df, x="y_real", y="y_pred")
plt.set(xlim=(-7, 2))
plt.set(ylim=(-7, 2))
plt
```

[56]:
```
<Axes: xlabel='y_real', ylabel='y_pred'>
```



1.9 Improving the model:

- Dropouts
- Other (more intelligent) Pooling Layers
- Global Pooling Layers
- Batch Normalization
- More MP layers
- Other hidden state sizes
- Test metrics (test error) and Hyperparameter optimization
- ...