as a professional in antimalaria drug research. I want to do study on prediction of active or inactive antimalaria drug using gnn. at the end I also want to perform explainability and interpretability for the prediction Note you are to use the test data. I am using jupyer notebook. Execute each cell after each function or class. Use all method in captum package that fits the projects involving molecules. top 10 most important molecular descriptors based.

Make it simple but with rich key explainability information using Captum

Perform explainability one after the other

from collections import defaultdict

from rdkit import Chem

from rdkit.Chem import rdchem

from tqdm import tqdm

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

def analyze\_atom\_features(data, smiles\_col='Smiles'):

"""

Comprehensive analysis of all atom features present in the dataset

Parameters:

-----------

data : pd.DataFrame

DataFrame containing molecular data

smiles\_col : str

Column name containing SMILES strings

Returns:

--------

dict: Comprehensive feature statistics

"""

feature\_stats = defaultdict(set)

feature\_counts = defaultdict(lambda: defaultdict(int))

atom\_types = set()

failed\_molecules = 0

total\_atoms = 0

print("Analyzing atom features across all molecules...")

for idx, row in tqdm(data.iterrows(), total=len(data)):

mol = Chem.MolFromSmiles(row[smiles\_col])

if mol is None:

failed\_molecules += 1

continue

total\_atoms += mol.GetNumAtoms()

for atom in mol.GetAtoms():

# Basic atom properties

symbol = atom.GetSymbol()

atom\_types.add(symbol)

atomic\_num = atom.GetAtomicNum()

degree = atom.GetDegree()

formal\_charge = atom.GetFormalCharge()

hybridization = int(atom.GetHybridization())

is\_aromatic = int(atom.GetIsAromatic())

total\_hs = atom.GetTotalNumHs()

is\_in\_ring = int(atom.IsInRing())

is\_chiral = int(atom.GetChiralTag() != Chem.ChiralType.CHI\_UNSPECIFIED)

implicit\_valence = atom.GetImplicitValence()

num\_radical\_electrons = atom.GetNumRadicalElectrons()

# Collect feature values

feature\_stats['Atomic Numbers'].add(atomic\_num)

feature\_stats['Degrees'].add(degree)

feature\_stats['Formal Charges'].add(formal\_charge)

feature\_stats['Hybridizations'].add(hybridization)

feature\_stats['Is Aromatic'].add(is\_aromatic)

feature\_stats['Total H Count'].add(total\_hs)

feature\_stats['Is In Ring'].add(is\_in\_ring)

feature\_stats['Is Chiral'].add(is\_chiral)

feature\_stats['Implicit Valence'].add(implicit\_valence)

feature\_stats['Num Radical Electrons'].add(num\_radical\_electrons)

# Count occurrences for statistics

feature\_counts['Atomic Numbers'][atomic\_num] += 1

feature\_counts['Degrees'][degree] += 1

feature\_counts['Formal Charges'][formal\_charge] += 1

feature\_counts['Hybridizations'][hybridization] += 1

feature\_counts['Is Aromatic'][is\_aromatic] += 1

feature\_counts['Total H Count'][total\_hs] += 1

feature\_counts['Is In Ring'][is\_in\_ring] += 1

feature\_counts['Atom Types'][symbol] += 1

# Ring size analysis

if atom.IsInRing():

ring\_info = atom.GetRingInfo() if hasattr(atom, 'GetRingInfo') else None

for ring\_size in range(3, 9): # Check rings of size 3-8

if atom.IsInRingSize(ring\_size):

feature\_stats[f'In Ring Size {ring\_size}'].add(1)

feature\_counts[f'In Ring Size {ring\_size}'][1] += 1

# Display comprehensive results

print("\n" + "="\*60)

print("COMPREHENSIVE ATOM FEATURE ANALYSIS")

print("="\*60)

print(f"\nDataset Overview:")

print(f" Total molecules processed: {len(data) - failed\_molecules}")

print(f" Failed molecules: {failed\_molecules}")

print(f" Total atoms analyzed: {total\_atoms:,}")

print(f" Average atoms per molecule: {total\_atoms/(len(data) - failed\_molecules):.2f}")

print(f"\nUnique atom types found: {sorted(atom\_types)}")

print(f"Total unique elements: {len(atom\_types)}")

# Detailed feature analysis

print(f"\n{'Feature':<20} {'Unique Values':<15} {'Range':<15} {'Most Common':<15}")

print("-" \* 65)

for feature, values in feature\_stats.items():

if feature.startswith('In Ring Size'):

continue # Handle ring sizes separately

sorted\_values = sorted(values)

value\_range = f"{min(values)} to {max(values)}" if len(values) > 1 else str(list(values)[0])

# Find most common value

if feature in feature\_counts:

most\_common = max(feature\_counts[feature].items(), key=lambda x: x[1])

most\_common\_str = f"{most\_common[0]} ({most\_common[1]})"

else:

most\_common\_str = "N/A"

print(f"{feature:<20} {len(values):<15} {value\_range:<15} {most\_common\_str:<15}")

# Ring size analysis

ring\_features = {k: v for k, v in feature\_stats.items() if k.startswith('In Ring Size')}

if ring\_features:

print(f"\nRing Size Distribution:")

for ring\_feature in sorted(ring\_features.keys()):

ring\_size = ring\_feature.split()[-1]

count = feature\_counts[ring\_feature].get(1, 0)

print(f" Ring size {ring\_size}: {count} atoms")

# Hybridization mapping for clarity

hybridization\_map = {

0: 'UNSPECIFIED',

1: 'S',

2: 'SP',

3: 'SP2',

4: 'SP3',

5: 'SP3D',

6: 'SP3D2',

7: 'OTHER'

}

print(f"\nHybridization Details:")

for hyb\_code in sorted(feature\_stats['Hybridizations']):

hyb\_name = hybridization\_map.get(hyb\_code, f'Unknown({hyb\_code})')

count = feature\_counts['Hybridizations'][hyb\_code]

print(f" {hyb\_name} (code {hyb\_code}): {count} atoms")

return {

'feature\_stats': feature\_stats,

'feature\_counts': feature\_counts,

'atom\_types': atom\_types,

'total\_atoms': total\_atoms,

'failed\_molecules': failed\_molecules

}

def plot\_atom\_feature\_distributions(analysis\_results, figsize=(15, 12)):

"""

Create visualization of atom feature distributions

"""

feature\_counts = analysis\_results['feature\_counts']

# Create subplot grid

fig, axes = plt.subplots(3, 3, figsize=figsize)

axes = axes.flatten()

# Define features to plot

features\_to\_plot = [

'Atom Types', 'Degrees', 'Formal Charges', 'Hybridizations',

'Is Aromatic', 'Total H Count', 'Is In Ring', 'Atomic Numbers'

]

for i, feature in enumerate(features\_to\_plot):

if i >= len(axes):

break

if feature in feature\_counts:

data = feature\_counts[feature]

keys = list(data.keys())

values = list(data.values())

if feature == 'Atom Types':

# Bar plot for atom types

axes[i].bar(range(len(keys)), values)

axes[i].set\_xticks(range(len(keys)))

axes[i].set\_xticklabels(keys, rotation=45)

else:

# Bar plot for other features

axes[i].bar(keys, values)

if len(keys) > 10:

axes[i].tick\_params(axis='x', rotation=45)

axes[i].set\_title(f'{feature} Distribution')

axes[i].set\_ylabel('Count')

axes[i].grid(True, alpha=0.3)

# Remove empty subplots

for i in range(len(features\_to\_plot), len(axes)):

fig.delaxes(axes[i])

plt.tight\_layout()

plt.show()

def create\_feature\_summary\_table(analysis\_results):

"""

Create a summary table of all features for easy reference

"""

feature\_stats = analysis\_results['feature\_stats']

feature\_counts = analysis\_results['feature\_counts']

summary\_data = []

for feature, values in feature\_stats.items():

if feature.startswith('In Ring Size'):

continue

summary\_data.append({

'Feature': feature,

'Unique\_Values': len(values),

'Min\_Value': min(values),

'Max\_Value': max(values),

'Most\_Common': max(feature\_counts[feature].items(), key=lambda x: x[1])[0] if feature in feature\_counts else 'N/A',

'Most\_Common\_Count': max(feature\_counts[feature].items(), key=lambda x: x[1])[1] if feature in feature\_counts else 0

})

return pd.DataFrame(summary\_data)

# Example usage:

analysis\_results = analyze\_atom\_features(non\_redundant\_df)

summary\_table = create\_feature\_summary\_table(analysis\_results)

plot\_atom\_feature\_distributions(analysis\_results)

import torch

from torch\_geometric.data import Data

from rdkit import Chem

from rdkit.Chem import rdchem

from tqdm import tqdm

import numpy as np

import pandas as pd

def get\_atom\_features(atom, mol):

"""

Get comprehensive atom features based on dataset analysis

Features are extracted in the order that makes most sense for GNN learning:

1. Identity features (atomic number, symbol)

2. Bonding features (degree, valence, hybridization)

3. Chemical properties (charge, aromaticity, hydrogens)

4. Structural features (rings, chirality)

5. Advanced features (radical electrons)

"""

# Get ring info from molecule

ring\_info = mol.GetRingInfo()

atom\_idx = atom.GetIdx()

# Basic identity and bonding

features = [

atom.GetAtomicNum(), # Atomic number (element identity)

atom.GetDegree(), # Number of bonds

atom.GetTotalDegree(), # Including implicit hydrogens

atom.GetImplicitValence(), # Valence electrons

int(atom.GetHybridization()), # SP, SP2, SP3, etc.

# Chemical properties

atom.GetFormalCharge(), # Formal charge

int(atom.GetIsAromatic()), # Aromatic atom

atom.GetTotalNumHs(), # Total hydrogen count

atom.GetNumImplicitHs(), # Implicit hydrogens

atom.GetNumExplicitHs(), # Explicit hydrogens

# Ring properties - fixed to use molecule's ring info

int(atom.IsInRing()), # Is in any ring

len([ring for ring in ring\_info.AtomRings() if atom\_idx in ring]), # Number of rings containing this atom

# Specific ring sizes (most common in organic chemistry)

int(atom.IsInRingSize(3)), # 3-member ring

int(atom.IsInRingSize(4)), # 4-member ring

int(atom.IsInRingSize(5)), # 5-member ring

int(atom.IsInRingSize(6)), # 6-member ring

int(atom.IsInRingSize(7)), # 7-member ring

int(atom.IsInRingSize(8)), # 8-member ring

# Stereochemistry and advanced features

int(atom.GetChiralTag() != Chem.ChiralType.CHI\_UNSPECIFIED), # Is chiral

atom.GetNumRadicalElectrons(), # Radical electrons

# Connectivity features

len([neighbor for neighbor in atom.GetNeighbors()]), # Number of neighbors

sum([bond.GetBondTypeAsDouble() for bond in atom.GetBonds()]), # Sum of bond orders

# Electronegativity-related (approximated by period and group)

\_get\_period(atom.GetAtomicNum()), # Periodic table period

\_get\_group(atom.GetAtomicNum()), # Periodic table group

# Additional chemical properties

int(\_is\_halogen(atom.GetAtomicNum())), # Is halogen

int(\_is\_metal(atom.GetAtomicNum())), # Is metal

int(\_is\_heteroatom(atom.GetAtomicNum())), # Is heteroatom (not C or H)

]

return features

def \_get\_period(atomic\_num):

"""Get periodic table period for an atomic number"""

if atomic\_num <= 2:

return 1

elif atomic\_num <= 10:

return 2

elif atomic\_num <= 18:

return 3

elif atomic\_num <= 36:

return 4

elif atomic\_num <= 54:

return 5

elif atomic\_num <= 86:

return 6

else:

return 7

def \_get\_group(atomic\_num):

"""Get periodic table group for common elements"""

group\_map = {

1: 1, 3: 1, 11: 1, 19: 1, 37: 1, 55: 1, 87: 1, # Group 1

4: 2, 12: 2, 20: 2, 38: 2, 56: 2, 88: 2, # Group 2

5: 13, 13: 13, 31: 13, 49: 13, 81: 13, # Group 13

6: 14, 14: 14, 32: 14, 50: 14, 82: 14, # Group 14

7: 15, 15: 15, 33: 15, 51: 15, 83: 15, # Group 15

8: 16, 16: 16, 34: 16, 52: 16, 84: 16, # Group 16

9: 17, 17: 17, 35: 17, 53: 17, 85: 17, # Group 17

2: 18, 10: 18, 18: 18, 36: 18, 54: 18, 86: 18, # Group 18

}

return group\_map.get(atomic\_num, 0) # 0 for transition metals and others

def \_is\_halogen(atomic\_num):

"""Check if atom is a halogen"""

return atomic\_num in [9, 17, 35, 53, 85] # F, Cl, Br, I, At

def \_is\_metal(atomic\_num):

"""Check if atom is a metal (simplified)"""

metals = {3, 4, 11, 12, 13, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,

37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 55, 56, 57, 58,

59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76,

77, 78, 79, 80, 81, 82, 83, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,

98, 99, 100, 101, 102, 103}

return atomic\_num in metals

def \_is\_heteroatom(atomic\_num):

"""Check if atom is a heteroatom (not C or H)"""

return atomic\_num not in [1, 6] # Not H or C

# Comprehensive feature names for reference

ATOM\_FEATURE\_NAMES = [

# Basic identity and bonding (5 features)

'Atomic Number',

'Degree',

'Total Degree',

'Implicit Valence',

'Hybridization',

# Chemical properties (5 features)

'Formal Charge',

'Is Aromatic',

'Total H Count',

'Num Implicit Hs',

'Num Explicit Hs',

# Ring properties (8 features)

'Is In Ring',

'Num Rings',

'In 3-member Ring',

'In 4-member Ring',

'In 5-member Ring',

'In 6-member Ring',

'In 7-member Ring',

'In 8-member Ring',

# Stereochemistry and advanced (2 features)

'Is Chiral',

'Num Radical Electrons',

# Connectivity (2 features)

'Num Neighbors',

'Sum Bond Orders',

# Periodic properties (2 features)

'Period',

'Group',

# Chemical classification (3 features)

'Is Halogen',

'Is Metal',

'Is Heteroatom'

]

def get\_bond\_features(bond):

"""

Get bond features for edge attributes

"""

bond\_type\_map = {

Chem.BondType.SINGLE: 1,

Chem.BondType.DOUBLE: 2,

Chem.BondType.TRIPLE: 3,

Chem.BondType.AROMATIC: 4

}

features = [

bond\_type\_map.get(bond.GetBondType(), 0), # Bond type

int(bond.GetIsAromatic()), # Is aromatic bond

int(bond.IsInRing()), # Is in ring

int(bond.GetIsConjugated()), # Is conjugated

bond.GetBondTypeAsDouble(), # Bond order as float

]

return features

BOND\_FEATURE\_NAMES = [

'Bond Type',

'Is Aromatic Bond',

'Bond In Ring',

'Is Conjugated',

'Bond Order'

]

def smiles\_to\_graph(smiles, molecular\_descriptors=None, include\_bond\_features=True,

graph\_idx=None, original\_smiles=None):

"""

Convert SMILES to PyTorch Geometric graph with comprehensive features and explainability support

Parameters:

-----------

smiles : str

SMILES string

molecular\_descriptors : dict or list, optional

Molecular-level descriptors to add as graph-level features

include\_bond\_features : bool

Whether to include edge features

graph\_idx : int, optional

Graph index for explainability tracking

original\_smiles : str, optional

Original SMILES (if different from processed SMILES)

Returns:

--------

torch\_geometric.data.Data or None if invalid SMILES

"""

mol = Chem.MolFromSmiles(smiles)

if mol is None:

return None

# Atom features (node features) - pass mol to get\_atom\_features

atom\_features = []

atom\_symbols = [] # Store atom symbols for explainability

for atom in mol.GetAtoms():

features = get\_atom\_features(atom, mol)

atom\_features.append(features)

atom\_symbols.append(atom.GetSymbol())

# Edge indices and features

edge\_indices = []

edge\_features = []

for bond in mol.GetBonds():

i = bond.GetBeginAtomIdx()

j = bond.GetEndAtomIdx()

# Add both directions for undirected graph

edge\_indices.extend([[i, j], [j, i]])

if include\_bond\_features:

bond\_feat = get\_bond\_features(bond)

edge\_features.extend([bond\_feat, bond\_feat]) # Same features for both directions

# Convert to tensors

x = torch.tensor(atom\_features, dtype=torch.float)

edge\_index = torch.tensor(edge\_indices, dtype=torch.long).t().contiguous() if edge\_indices else torch.empty((2, 0), dtype=torch.long)

# Create data object

data\_dict = {'x': x, 'edge\_index': edge\_index}

# Add edge features if requested

if include\_bond\_features and edge\_features:

edge\_attr = torch.tensor(edge\_features, dtype=torch.float)

data\_dict['edge\_attr'] = edge\_attr

# Add molecular descriptors as graph-level features

if molecular\_descriptors is not None:

if isinstance(molecular\_descriptors, dict):

# Convert dict values to list, handling NaN values

mol\_desc\_values = []

for key, value in molecular\_descriptors.items():

if pd.isna(value):

mol\_desc\_values.append(0.0) # Replace NaN with 0

else:

mol\_desc\_values.append(float(value))

molecular\_descriptors = mol\_desc\_values

# Convert to tensor and add to graph

mol\_desc\_tensor = torch.tensor(molecular\_descriptors, dtype=torch.float)

data\_dict['mol\_descriptors'] = mol\_desc\_tensor

# Add explainability information

data\_dict['smiles'] = original\_smiles if original\_smiles is not None else smiles

data\_dict['atom\_symbols'] = atom\_symbols

if graph\_idx is not None:

data\_dict['graph\_idx'] = graph\_idx

return Data(\*\*data\_dict)

def convert\_dataset\_to\_graphs(data, smiles\_col='Smiles', label\_col='Labels',

molecular\_descriptor\_cols=None, include\_bond\_features=True,

normalize\_descriptors=True, verbose=True):

"""

Convert entire dataset to graphs with both atomic and molecular features, plus explainability support

Parameters:

-----------

data : pd.DataFrame

Dataset containing SMILES, labels, and molecular descriptors

smiles\_col : str

Column name for SMILES

label\_col : str

Column name for labels

molecular\_descriptor\_cols : list, optional

List of column names containing molecular descriptors to include

If None, will automatically detect numeric columns (excluding SMILES and labels)

include\_bond\_features : bool

Whether to include edge features

normalize\_descriptors : bool

Whether to normalize molecular descriptors (recommended)

verbose : bool

Whether to show progress

Returns:

--------

list: List of Data objects with explainability information

list: List of labels

dict: Statistics and metadata

sklearn.preprocessing.StandardScaler: Fitted scaler for descriptors (if normalized)

dict: Mapping from graph index to original dataset index for explainability

"""

from sklearn.preprocessing import StandardScaler

graphs = []

labels = []

failed\_conversions = 0

scaler = None

graph\_to\_dataset\_mapping = {} # For explainability: graph\_idx -> original\_dataset\_idx

# Auto-detect molecular descriptor columns if not provided

if molecular\_descriptor\_cols is None:

# Get all numeric columns except SMILES and labels

numeric\_cols = data.select\_dtypes(include=[np.number]).columns.tolist()

exclude\_cols = [smiles\_col, label\_col]

molecular\_descriptor\_cols = [col for col in numeric\_cols if col not in exclude\_cols]

if verbose:

print(f"Auto-detected {len(molecular\_descriptor\_cols)} molecular descriptor columns:")

for i, col in enumerate(molecular\_descriptor\_cols, 1):

print(f" {i:2d}. {col}")

# Prepare molecular descriptors

mol\_desc\_data = None

if molecular\_descriptor\_cols:

mol\_desc\_data = data[molecular\_descriptor\_cols].copy()

# Handle missing values

mol\_desc\_data = mol\_desc\_data.fillna(0)

# Normalize descriptors if requested

if normalize\_descriptors:

scaler = StandardScaler()

mol\_desc\_data\_scaled = scaler.fit\_transform(mol\_desc\_data)

mol\_desc\_data = pd.DataFrame(mol\_desc\_data\_scaled,

columns=molecular\_descriptor\_cols,

index=mol\_desc\_data.index)

if verbose:

print(f"Normalized {len(molecular\_descriptor\_cols)} molecular descriptors")

# Convert molecules to graphs

iterator = tqdm(data.iterrows(), total=len(data)) if verbose else data.iterrows()

graph\_counter = 0 # Counter for successfully created graphs

for dataset\_idx, row in iterator:

# Get molecular descriptors for this molecule

mol\_descriptors = None

if mol\_desc\_data is not None:

mol\_descriptors = mol\_desc\_data.loc[dataset\_idx].values.tolist()

# Convert to graph with explainability information

graph = smiles\_to\_graph(

row[smiles\_col],

molecular\_descriptors=mol\_descriptors,

include\_bond\_features=include\_bond\_features,

graph\_idx=graph\_counter, # Assign graph index

original\_smiles=row[smiles\_col] # Store original SMILES

)

if graph is not None:

# Add label to graph

if label\_col in row:

graph.y = torch.tensor([row[label\_col]], dtype=torch.long)

labels.append(row[label\_col])

# Store original dataset index for explainability

graph.dataset\_idx = dataset\_idx

# Add to mapping

graph\_to\_dataset\_mapping[graph\_counter] = dataset\_idx

graphs.append(graph)

graph\_counter += 1

else:

failed\_conversions += 1

# Calculate statistics

if graphs:

num\_node\_features = graphs[0].x.shape[1]

num\_edge\_features = graphs[0].edge\_attr.shape[1] if hasattr(graphs[0], 'edge\_attr') else 0

num\_mol\_features = graphs[0].mol\_descriptors.shape[0] if hasattr(graphs[0], 'mol\_descriptors') else 0

avg\_nodes = np.mean([g.x.shape[0] for g in graphs])

avg\_edges = np.mean([g.edge\_index.shape[1] for g in graphs])

else:

num\_node\_features = num\_edge\_features = num\_mol\_features = avg\_nodes = avg\_edges = 0

stats = {

'total\_molecules': len(data),

'successful\_conversions': len(graphs),

'failed\_conversions': failed\_conversions,

'success\_rate': len(graphs) / len(data) \* 100,

'num\_node\_features': num\_node\_features,

'num\_edge\_features': num\_edge\_features,

'num\_molecular\_features': num\_mol\_features,

'molecular\_descriptor\_cols': molecular\_descriptor\_cols,

'avg\_nodes\_per\_graph': avg\_nodes,

'avg\_edges\_per\_graph': avg\_edges,

'descriptors\_normalized': normalize\_descriptors,

'explainability\_enabled': True

}

if verbose:

print(f"\n=== GRAPH CONVERSION SUMMARY ===")

print(f"Total molecules: {stats['total\_molecules']}")

print(f"Successful conversions: {stats['successful\_conversions']}")

print(f"Failed conversions: {stats['failed\_conversions']}")

print(f"Success rate: {stats['success\_rate']:.2f}%")

print(f"Node features per atom: {stats['num\_node\_features']}")

print(f"Edge features per bond: {stats['num\_edge\_features']}")

print(f"Molecular features per graph: {stats['num\_molecular\_features']}")

print(f"Average nodes per graph: {stats['avg\_nodes\_per\_graph']:.2f}")

print(f"Average edges per graph: {stats['avg\_edges\_per\_graph']:.2f}")

print(f"Explainability support: ENABLED")

return graphs, labels, stats, scaler, graph\_to\_dataset\_mapping

def get\_graph\_info\_for\_explainability(graph):

"""

Extract explainability information from a graph

Parameters:

-----------

graph : torch\_geometric.data.Data

Graph object with explainability information

Returns:

--------

dict: Information needed for explainability

"""

info = {

'smiles': getattr(graph, 'smiles', None),

'graph\_idx': getattr(graph, 'graph\_idx', None),

'dataset\_idx': getattr(graph, 'dataset\_idx', None),

'atom\_symbols': getattr(graph, 'atom\_symbols', None),

'num\_atoms': graph.x.shape[0],

'num\_bonds': graph.edge\_index.shape[1] // 2 if graph.edge\_index.numel() > 0 else 0,

'label': graph.y.item() if hasattr(graph, 'y') and graph.y is not None else None

}

return info

def create\_explainability\_dataset\_mapping(graphs, original\_data, smiles\_col='Smiles'):

"""

Create comprehensive mapping for explainability between graphs and original dataset

Parameters:

-----------

graphs : list

List of graph objects

original\_data : pd.DataFrame

Original dataset

smiles\_col : str

Column name for SMILES in original dataset

Returns:

--------

pd.DataFrame: Explainability mapping with graph info and original data

"""

mapping\_data = []

for graph in graphs:

info = get\_graph\_info\_for\_explainability(graph)

# Get original row data if dataset\_idx is available

if info['dataset\_idx'] is not None:

original\_row = original\_data.loc[info['dataset\_idx']].to\_dict()

else:

original\_row = {}

mapping\_row = {

'graph\_idx': info['graph\_idx'],

'dataset\_idx': info['dataset\_idx'],

'smiles': info['smiles'],

'num\_atoms': info['num\_atoms'],

'num\_bonds': info['num\_bonds'],

'label': info['label'],

'atom\_symbols': info['atom\_symbols'],

\*\*original\_row # Include all original data columns

}

mapping\_data.append(mapping\_row)

return pd.DataFrame(mapping\_data)

# Print feature information

print(f"\nUsing {len(ATOM\_FEATURE\_NAMES)} atom features:")

for i, name in enumerate(ATOM\_FEATURE\_NAMES, 1):

print(f"{i:2d}. {name}")

print(f"\nUsing {len(BOND\_FEATURE\_NAMES)} bond features:")

for i, name in enumerate(BOND\_FEATURE\_NAMES, 1):

print(f"{i:2d}. {name}")

# Descriptor columns definition

descriptor\_cols = [

'MolWt', 'LogP', 'NumHDonors', 'RingCount', 'SlogP\_VSA3', 'PEOE\_VSA10', 'PEOE\_VSA14',

'NumSaturatedCarbocycles', 'NumSaturatedHeterocycles', 'NumAromaticHeterocycles', 'Ipc',

'MaxEStateIndex', 'MinEStateIndex', 'MinPartialCharge', 'EState\_VSA3', 'EState\_VSA4',

'EState\_VSA5', 'EState\_VSA6', 'EState\_VSA7', 'EState\_VSA8', 'EState\_VSA9', 'EState\_VSA11',

'VSA\_EState4', 'VSA\_EState5', 'VSA\_EState7', 'VSA\_EState8', 'fr\_Al\_COO', 'fr\_Al\_OH',

'fr\_Ar\_NH', 'fr\_Ar\_OH', 'fr\_N\_O', 'fr\_furan', 'fr\_imidazole', 'fr\_piperdine', 'fr\_thiazole', 'fr\_thiophene'

]

# Example usage with explainability:

graphs, labels, stats, scaler, graph\_mapping = convert\_dataset\_to\_graphs(

non\_redundant\_df,

molecular\_descriptor\_cols=descriptor\_cols

)

# Create explainability mapping

explainability\_df = create\_explainability\_dataset\_mapping(graphs, non\_redundant\_df)

# Example: Get information for a specific graph

graph\_info = get\_graph\_info\_for\_explainability(graphs[0])

print("Graph 0 info:", graph\_info)

# Example: Find original data for a graph

graph\_idx = 5

original\_dataset\_idx = graph\_mapping[graph\_idx]

original\_row = non\_redundant\_df.loc[original\_dataset\_idx]

print(f"Graph {graph\_idx} corresponds to dataset row {original\_dataset\_idx}")

print(f"Original SMILES: {original\_row['Smiles']}")

print(graphs\_m

)

Data(x=[49, 27], edge\_index=[2, 104], edge\_attr=[104, 5], mol\_descriptors=[36], smiles='CCCCCCCCCCCCNC(=O)C(CCC=C(C)C)=C1C(OC(C)=O)CC2(C)C1CC(O)C1C3(C)CCC(O)C(C)C3CCC12C', atom\_symbols=[49], graph\_idx=0, y=[1], dataset\_idx=0)

# Split data into 70% train, 30% temp (test + val)

train\_graphs, temp\_graphs = train\_test\_split(graphs\_m, test\_size=0.3, random\_state=42, stratify=labels)

# Split the temp 30% into 15% test and 15% val (50-50 split of the temp data)

val\_graphs, test\_graphs = train\_test\_split(temp\_graphs, test\_size=0.5, random\_state=42)

print(f"Train: {len(train\_graphs)}, Val: {len(val\_graphs)}, Test: {len(test\_graphs)}")

train\_loader = DataLoader(train\_graphs, batch\_size=32, shuffle=True)

val\_loader = DataLoader(val\_graphs, batch\_size=32, shuffle=False)

test\_loader = DataLoader(test\_graphs, batch\_size=32, shuffle=False)

import torch

import torch.nn as nn

import torch.nn.functional as F

from sklearn.metrics import accuracy\_score, roc\_auc\_score

import numpy as np

import os

from datetime import datetime

# Define loss and optimizer with dynamic configuration - CORRECTED

def create\_training\_components(model, config=None):

"""Create training components based on configuration"""

if config is None:

config = {}

# Loss function configuration - CORRECTED for binary classification

loss\_config = config.get('loss', {})

if loss\_config.get('type') == 'bce':

criterion = nn.BCELoss() # For binary classification with sigmoid

else:

criterion = nn.BCELoss() # Default to BCE for binary classification

# Optimizer configuration

optimizer\_config = config.get('optimizer', {})

lr = optimizer\_config.get('lr', 0.001)

weight\_decay = optimizer\_config.get('weight\_decay', 1e-4)

if optimizer\_config.get('type') == 'adam':

optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight\_decay=weight\_decay)

else:

optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight\_decay=weight\_decay)

# Scheduler configuration

scheduler\_config = config.get('scheduler', {})

scheduler\_type = scheduler\_config.get('type', 'plateau')

if scheduler\_type == 'plateau':

scheduler = torch.optim.lr\_scheduler.ReduceLROnPlateau(

optimizer,

mode=scheduler\_config.get('mode', 'min'),

factor=scheduler\_config.get('factor', 0.5),

patience=scheduler\_config.get('patience', 10),

min\_lr=scheduler\_config.get('min\_lr', 1e-6)

)

elif scheduler\_type == 'step':

scheduler = torch.optim.lr\_scheduler.StepLR(

optimizer,

step\_size=scheduler\_config.get('step\_size', 30),

gamma=scheduler\_config.get('gamma', 0.1)

)

else:

scheduler = torch.optim.lr\_scheduler.ReduceLROnPlateau(

optimizer,

mode='min',

factor=0.5,

patience=10,

min\_lr=1e-6

)

return criterion, optimizer, scheduler

# Training functions - CORRECTED

def train(model, train\_loader, criterion, optimizer, device):

model.train()

total\_loss = 0

predictions = []

true\_labels = []

all\_probs = []

for batch in train\_loader:

batch = batch.to(device)

optimizer.zero\_grad()

# CORRECTED: Pass the entire batch object to model

out = model(batch)

# CORRECTED: BCE loss for binary classification

loss = criterion(out.squeeze(), batch.y.float())

loss.backward()

optimizer.step()

total\_loss += loss.item()

# CORRECTED: Binary classification predictions

probs = out.squeeze()

pred = (probs > 0.5).float()

predictions.extend(pred.cpu().numpy())

true\_labels.extend(batch.y.cpu().numpy())

all\_probs.extend(probs.detach().cpu().numpy())

avg\_loss = total\_loss / len(train\_loader)

accuracy = accuracy\_score(true\_labels, predictions)

auc = roc\_auc\_score(true\_labels, all\_probs)

return avg\_loss, accuracy, auc

def evaluate(model, loader, criterion, device):

model.eval()

total\_loss = 0

predictions = []

true\_labels = []

all\_probs = []

with torch.no\_grad():

for batch in loader:

batch = batch.to(device)

# CORRECTED: Pass the entire batch object to model

out = model(batch)

# CORRECTED: BCE loss for binary classification

loss = criterion(out.squeeze(), batch.y.float())

total\_loss += loss.item()

# CORRECTED: Binary classification predictions

probs = out.squeeze()

pred = (probs > 0.5).float()

predictions.extend(pred.cpu().numpy())

true\_labels.extend(batch.y.cpu().numpy())

all\_probs.extend(probs.cpu().numpy())

avg\_loss = total\_loss / len(loader)

accuracy = accuracy\_score(true\_labels, predictions)

auc = roc\_auc\_score(true\_labels, all\_probs)

return avg\_loss, accuracy, auc, np.array(predictions), np.array(true\_labels), np.array(all\_probs)

# training function

def train\_model(model, train\_loader, val\_loader, config=None):

"""

Train the model with comprehensive training enhancements

Args:

model: The GNN model to train

train\_loader: Training data loader

val\_loader: Validation data loader

config: Configuration dictionary with all training parameters

"""

# Default configuration

if config is None:

config = {}

# Extract parameters from config

epochs = config.get('epochs', 200)

device = config.get('device', 'cpu')

initial\_lr = config.get('initial\_lr', 0.001)

weight\_decay = config.get('weight\_decay', 1e-4)

scheduler\_patience = config.get('scheduler\_patience', 10)

scheduler\_factor = config.get('scheduler\_factor', 0.5)

gradient\_clip\_value = config.get('gradient\_clip\_value', None)

early\_stop\_patience = config.get('early\_stop\_patience', 25)

warmup\_epochs = config.get('warmup\_epochs', 0)

cosine\_annealing = config.get('cosine\_annealing', False)

dropout\_schedule = config.get('dropout\_schedule', False)

model\_name = config.get('model\_name', None)

save\_dir = config.get('save\_dir', './models')

# Create save directory if it doesn't exist

os.makedirs(save\_dir, exist\_ok=True)

# Generate dynamic model name if not provided

if model\_name is None:

timestamp = datetime.now().strftime("%Y%m%d\_%H%M%S")

scheduler\_name = "cosine" if cosine\_annealing else "plateau"

model\_name = f"gnn\_lr{initial\_lr}\_wd{weight\_decay}\_sched{scheduler\_name}\_{timestamp}"

# Define model save path

model\_path = os.path.join(save\_dir, f"{model\_name}\_best.pth")

# CORRECTED: Initialize loss function for binary classification

criterion = nn.BCELoss() # Binary Cross Entropy for binary classification

optimizer = torch.optim.Adam(model.parameters(), lr=initial\_lr, weight\_decay=weight\_decay)

# Learning rate scheduling options

if cosine\_annealing:

scheduler = torch.optim.lr\_scheduler.CosineAnnealingLR(optimizer, T\_max=epochs, eta\_min=1e-6)

scheduler\_type = "CosineAnnealing"

else:

scheduler = torch.optim.lr\_scheduler.ReduceLROnPlateau(

optimizer, mode='min', factor=scheduler\_factor, patience=scheduler\_patience,

min\_lr=1e-6

)

scheduler\_type = "ReduceLROnPlateau"

# Warmup scheduler (if enabled)

if warmup\_epochs > 0:

warmup\_scheduler = torch.optim.lr\_scheduler.LinearLR(

optimizer, start\_factor=0.1, total\_iters=warmup\_epochs

)

# Initialize history with comprehensive tracking

history = {

'train\_loss': [],

'train\_acc': [],

'train\_auc': [],

'val\_loss': [],

'val\_acc': [],

'val\_auc': [],

'learning\_rate': [],

'gradient\_norm': [],

'epochs': []

}

best\_val\_auc = 0

patience\_counter = 0

print("Starting enhanced training...")

print(f"Starting enhanced training for model: {model\_name}")

print(f"Model will be saved to: {model\_path}")

print(f"Initial LR: {initial\_lr}, Weight decay: {weight\_decay}")

print(f"Scheduler: {scheduler\_type}, Gradient clipping: {gradient\_clip\_value}")

print(f"Warmup epochs: {warmup\_epochs}")

print(f"Early stopping patience: {early\_stop\_patience} epochs")

print("-" \* 110)

print(f"{'Epoch':^8} | {'Train Loss':^11} | {'Train Acc':^10} | {'Train AUC':^10} | {'Val Loss':^11} | {'Val Acc':^10} | {'Val AUC':^10} | {'LR':^10} | {'Grad Norm':^10}")

print("-" \* 110)

for epoch in range(epochs):

# Dynamic dropout scheduling (optional)

if dropout\_schedule and hasattr(model, 'dropout'):

# Reduce dropout over time

current\_dropout = max(0.1, 0.5 \* (1 - epoch / epochs))

for module in model.modules():

if isinstance(module, nn.Dropout):

module.p = current\_dropout

# Training phase

model.train()

total\_loss = 0

predictions = []

true\_labels = []

all\_probs = []

total\_grad\_norm = 0

for batch in train\_loader:

batch = batch.to(device)

optimizer.zero\_grad()

# CORRECTED: Pass the entire batch object to model

out = model(batch)

# CORRECTED: BCE loss for binary classification

loss = criterion(out.squeeze(), batch.y.float())

loss.backward()

# Gradient clipping

if gradient\_clip\_value is not None:

grad\_norm = torch.nn.utils.clip\_grad\_norm\_(model.parameters(), gradient\_clip\_value)

total\_grad\_norm += grad\_norm.item()

else:

# Calculate gradient norm for monitoring

grad\_norm = 0

for param in model.parameters():

if param.grad is not None:

grad\_norm += param.grad.data.norm(2).item() \*\* 2

total\_grad\_norm += grad\_norm \*\* 0.5

optimizer.step()

total\_loss += loss.item()

# CORRECTED: Binary classification predictions

probs = out.squeeze()

pred = (probs > 0.5).float()

predictions.extend(pred.cpu().numpy())

true\_labels.extend(batch.y.cpu().numpy())

all\_probs.extend(probs.detach().cpu().numpy())

avg\_grad\_norm = total\_grad\_norm / len(train\_loader)

train\_loss = total\_loss / len(train\_loader)

train\_acc = accuracy\_score(true\_labels, predictions)

train\_auc = roc\_auc\_score(true\_labels, all\_probs)

# Validation phase

model.eval()

total\_val\_loss = 0

val\_predictions = []

val\_true\_labels = []

val\_probs = []

with torch.no\_grad():

for batch in val\_loader:

batch = batch.to(device)

# CORRECTED: Pass the entire batch object to model

out = model(batch)

# CORRECTED: BCE loss for binary classification

loss = criterion(out.squeeze(), batch.y.float())

total\_val\_loss += loss.item()

# CORRECTED: Binary classification predictions

probs = out.squeeze()

pred = (probs > 0.5).float()

val\_predictions.extend(pred.cpu().numpy())

val\_true\_labels.extend(batch.y.cpu().numpy())

val\_probs.extend(probs.cpu().numpy())

val\_loss = total\_val\_loss / len(val\_loader)

val\_acc = accuracy\_score(val\_true\_labels, val\_predictions)

val\_auc = roc\_auc\_score(val\_true\_labels, val\_probs)

# Learning rate scheduling

if warmup\_epochs > 0 and epoch < warmup\_epochs:

warmup\_scheduler.step()

elif cosine\_annealing:

scheduler.step()

else:

scheduler.step(val\_loss)

current\_lr = optimizer.param\_groups[0]['lr']

# Save to history

history['train\_loss'].append(train\_loss)

history['train\_acc'].append(train\_acc)

history['train\_auc'].append(train\_auc)

history['val\_loss'].append(val\_loss)

history['val\_acc'].append(val\_acc)

history['val\_auc'].append(val\_auc)

history['learning\_rate'].append(current\_lr)

history['gradient\_norm'].append(avg\_grad\_norm)

history['epochs'].append(epoch)

# Early stopping and best model saving

if val\_auc > best\_val\_auc:

best\_val\_auc = val\_auc

patience\_counter = 0

history['best\_epoch'] = epoch

history['best\_val\_auc'] = val\_auc

torch.save(model.state\_dict(), model\_path)

else:

patience\_counter += 1

# Print metrics every 5 epochs

if epoch % 5 == 0 or epoch == epochs - 1:

print(f"{epoch:^8d} | {train\_loss:^11.4f} | {train\_acc:^10.4f} | {train\_auc:^10.4f} | {val\_loss:^11.4f} | {val\_acc:^10.4f} | {val\_auc:^10.4f} | {current\_lr:^10.2e} | {avg\_grad\_norm:^10.4f}")

# Early stopping

if patience\_counter >= early\_stop\_patience:

print(f"\nEarly stopping at epoch {epoch}! No improvement for {early\_stop\_patience} epochs.")

break

# Stop if learning rate becomes too small

if current\_lr < 1e-6:

print(f"\nStopping training at epoch {epoch}. Learning rate too small: {current\_lr:.2e}")

break

# Print final results

print("-" \* 110)

print(f"{'Final':^8} | {train\_loss:^11.4f} | {train\_acc:^10.4f} | {train\_auc:^10.4f} | {val\_loss:^11.4f} | {val\_acc:^10.4f} | {val\_auc:^10.4f} | {current\_lr:^10.2e} | {avg\_grad\_norm:^10.4f}")

print("-" \* 110)

print(f"\nBest model saved at epoch {history['best\_epoch']} with validation AUC: {history['best\_val\_auc']:.4f}")

# Convert to numpy arrays

for key in ['train\_loss', 'train\_acc', 'train\_auc', 'val\_loss', 'val\_acc', 'val\_auc', 'learning\_rate', 'gradient\_norm']:

history[key] = np.array(history[key])

return history

import os

import torch

import torch.nn as nn

from sklearn.metrics import precision\_score, recall\_score, f1\_score, roc\_curve, auc, classification\_report

def print\_classification\_results(y\_true, y\_pred, y\_scores):

"""

Computes and prints classification metrics.

Args:

y\_true: Ground truth labels.

y\_pred: Predicted labels.

y\_scores: Predicted probabilities or scores.

Returns:

A dictionary containing all the classification metrics.

"""

acc = (y\_true == y\_pred).mean()

precision = precision\_score(y\_true, y\_pred)

recall = recall\_score(y\_true, y\_pred)

f1 = f1\_score(y\_true, y\_pred)

fpr, tpr, thresholds = roc\_curve(y\_true, y\_scores)

roc\_auc = auc(fpr, tpr)

print(classification\_report(y\_true, y\_pred,

target\_names=['Inactive', 'Active']))

print("\n=== Test Results ===")

print(f"Accuracy : {acc:.4f}")

print(f"Precision: {precision:.4f}")

print(f"Recall : {recall:.4f}")

print(f"F1-Score : {f1:.4f}")

print(f"AUC-ROC : {roc\_auc:.4f}")

return {

'accuracy': acc,

'precision': precision,

'recall': recall,

'f1\_score': f1,

'roc\_auc': roc\_auc,

'fpr': fpr,

'tpr': tpr,

'thresholds': thresholds

}

def test\_model(model, test\_loader, model\_path=None, device='cpu', config=None):

"""

Test the trained model and return comprehensive metrics.

Args:

model: The GNN model to test.

test\_loader: DataLoader for the test dataset.

model\_path: Optional path to saved model weights.

device: Device to run the test on ('cpu' or 'cuda').

config: Optional configuration dictionary for additional settings.

Returns:

Dictionary containing test metrics and predictions.

"""

if config is None:

config = {}

# Device check

if device == 'cuda':

if torch.cuda.is\_available():

device = f'cuda:{torch.cuda.current\_device()}'

else:

print("Warning: CUDA requested but not available. Falling back to CPU.")

device = 'cpu'

# Load model weights

if model\_path is not None and os.path.exists(model\_path):

model.load\_state\_dict(torch.load(model\_path, map\_location=device))

print(f"Loaded model weights from: {model\_path}")

model = model.to(device)

# Define loss

label\_smoothing = config.get('label\_smoothing', 0.0)

criterion = nn.CrossEntropyLoss(label\_smoothing=label\_smoothing) if label\_smoothing > 0 else nn.CrossEntropyLoss()

print("Starting model testing...")

print("-" \* 80)

# Evaluate

test\_loss, test\_acc, test\_auc, test\_predictions, test\_true\_labels, test\_probs = evaluate(

model, test\_loader, criterion, device

)

# Compute classification metrics

metrics = print\_classification\_results(test\_true\_labels, test\_predictions, test\_probs)

return {

'loss': test\_loss,

'accuracy': test\_acc,

'auc': test\_auc,

'predictions': test\_predictions,

'true\_labels': test\_true\_labels,

'probabilities': test\_probs,

'classification\_metrics': metrics

}

# Define Transformer-based GNN model - CORRECTED for Binary Classification

import torch

import torch.nn as nn

import torch.nn.functional as F

from torch\_geometric.nn import TransformerConv, global\_add\_pool, global\_mean\_pool, global\_max\_pool

from torch\_geometric.nn import BatchNorm

class GraphTransformer(nn.Module):

def \_\_init\_\_(self, num\_node\_features, num\_edge\_features,

hidden\_dim=128, num\_heads=8, num\_layers=4,

dropout=0.2, pooling='mean'):

"""

Graph Transformer model combining GNN with multi-head attention - CORRECTED for binary classification

Args:

num\_node\_features: Number of input node features

num\_edge\_features: Number of edge features (for future use)

hidden\_dim: Hidden dimension (must be divisible by num\_heads)

num\_heads: Number of attention heads

num\_layers: Number of transformer layers

dropout: Dropout rate

pooling: Pooling method ('mean', 'max', 'add', or 'all')

"""

super(GraphTransformer, self).\_\_init\_\_()

# Ensure hidden\_dim is divisible by num\_heads

assert hidden\_dim % num\_heads == 0, f"hidden\_dim ({hidden\_dim}) must be divisible by num\_heads ({num\_heads})"

self.num\_layers = num\_layers

self.pooling = pooling

self.mol\_desc\_encoder = None # Will be created dynamically

print(f"Initializing GraphTransformer with:")

print(f" Node features: {num\_node\_features}")

print(f" Edge features: {num\_edge\_features}")

print(f" Hidden dim: {hidden\_dim}, Heads: {num\_heads}, Layers: {num\_layers}")

# Input projection for node features

self.input\_proj = nn.Linear(num\_node\_features, hidden\_dim)

# Edge feature processing (for future use with edge-aware layers)

self.edge\_encoder = nn.Sequential(

nn.Linear(num\_edge\_features, 16),

nn.ReLU(),

nn.Linear(16, 8)

)

# Transformer layers

self.transformer\_convs = nn.ModuleList()

self.batch\_norms = nn.ModuleList()

self.dropout\_layers = nn.ModuleList()

for \_ in range(num\_layers):

self.transformer\_convs.append(

TransformerConv(hidden\_dim, hidden\_dim // num\_heads,

heads=num\_heads, dropout=dropout, concat=True)

)

self.batch\_norms.append(BatchNorm(hidden\_dim))

self.dropout\_layers.append(nn.Dropout(dropout))

# Output dimensions based on pooling

if pooling == 'all':

graph\_out\_dim = hidden\_dim \* 3 # mean + max + add

else:

graph\_out\_dim = hidden\_dim

# Graph feature processing after pooling

self.graph\_encoder = nn.Sequential(

nn.Linear(graph\_out\_dim, 64),

nn.ReLU(),

nn.Dropout(dropout)

)

# Combined features processing (graph + molecular descriptors)

# 64 (from graph) + 32 (from mol descriptors) = 96

self.classifier = nn.Sequential(

nn.Linear(96, 64),

nn.ReLU(),

nn.Dropout(dropout),

nn.Linear(64, 32),

nn.ReLU(),

nn.Dropout(dropout \* 0.5),

nn.Linear(32, 1) # Binary classification - single output with sigmoid

)

self.dropout = nn.Dropout(dropout)

self.relu = nn.ReLU()

def \_create\_mol\_descriptor\_encoder(self, mol\_dim, device):

"""Create molecular descriptor encoder dynamically"""

self.mol\_desc\_encoder = nn.Sequential(

nn.Linear(mol\_dim, 64),

nn.ReLU(),

nn.Dropout(0.2),

nn.Linear(64, 32)

).to(device)

print(f"Created mol\_desc\_encoder with input dim: {mol\_dim}")

def forward(self, data):

# CORRECTED: Accept data object like DrugActivityGNN

x, edge\_index, edge\_attr, mol\_descriptors, batch = data.x, data.edge\_index, data.edge\_attr, data.mol\_descriptors, data.batch

# Handle molecular descriptors automatically

batch\_size = batch.max().item() + 1

# Reshape mol\_descriptors to [batch\_size, mol\_descriptor\_dim]

if mol\_descriptors.dim() == 1:

# Auto-detect the correct mol\_descriptor\_dim

total\_elements = mol\_descriptors.shape[0]

mol\_descriptor\_dim = total\_elements // batch\_size

if total\_elements % batch\_size != 0:

raise ValueError(f"Cannot evenly divide mol\_descriptors {total\_elements} by batch\_size {batch\_size}")

mol\_descriptors = mol\_descriptors.view(batch\_size, mol\_descriptor\_dim)

elif mol\_descriptors.dim() == 2:

mol\_descriptor\_dim = mol\_descriptors.shape[1]

if mol\_descriptors.shape[0] != batch\_size:

if mol\_descriptors.shape[0] == 1:

mol\_descriptors = mol\_descriptors.repeat(batch\_size, 1)

else:

raise ValueError(f"mol\_descriptors batch size mismatch: {mol\_descriptors.shape[0]} vs {batch\_size}")

else:

mol\_descriptors = mol\_descriptors.view(batch\_size, -1)

mol\_descriptor\_dim = mol\_descriptors.shape[1]

# Create molecular descriptor encoder if it doesn't exist or has wrong input size

if self.mol\_desc\_encoder is None:

self.\_create\_mol\_descriptor\_encoder(mol\_descriptor\_dim, mol\_descriptors.device)

elif self.mol\_desc\_encoder[0].in\_features != mol\_descriptor\_dim:

self.\_create\_mol\_descriptor\_encoder(mol\_descriptor\_dim, mol\_descriptors.device)

# Input projection for node features

x = self.input\_proj(x)

x = self.relu(x)

x = self.dropout(x)

# Transformer layers with residual connections

for i in range(self.num\_layers):

x\_residual = x

x = self.transformer\_convs[i](x, edge\_index)

x = self.batch\_norms[i](x)

x = self.relu(x)

x = self.dropout\_layers[i](x)

x = x + x\_residual # Residual connection

# Graph-level pooling

if self.pooling == 'mean':

graph\_features = global\_mean\_pool(x, batch)

elif self.pooling == 'max':

graph\_features = global\_max\_pool(x, batch)

elif self.pooling == 'add':

graph\_features = global\_add\_pool(x, batch)

elif self.pooling == 'all':

x\_mean = global\_mean\_pool(x, batch)

x\_max = global\_max\_pool(x, batch)

x\_add = global\_add\_pool(x, batch)

graph\_features = torch.cat([x\_mean, x\_max, x\_add], dim=1)

# Process graph features

graph\_features = self.graph\_encoder(graph\_features) # Shape: [batch\_size, 64]

# Process molecular descriptors

mol\_features = self.mol\_desc\_encoder(mol\_descriptors) # Shape: [batch\_size, 32]

# Combine features

combined\_features = torch.cat([graph\_features, mol\_features], dim=1) # Shape: [batch\_size, 96]

# Final classification

output = self.classifier(combined\_features)

return torch.sigmoid(output) # Apply sigmoid for binary classification

# Initialize the model

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

transformer\_model = GraphTransformer(

num\_node\_features=27,

num\_edge\_features=5,

hidden\_dim=128,

num\_heads=8,

num\_layers=4,

dropout=0.2,

pooling='all'

).to(device)

# Print model information

print("=" \* 60)

print("GraphTransformer Model Initialized")

print("=" \* 60)

# Count parameters

def count\_parameters(model):

total\_params = sum(p.numel() for p in model.parameters())

trainable\_params = sum(p.numel() for p in model.parameters() if p.requires\_grad)

return total\_params, trainable\_params

total\_params, trainable\_params = count\_parameters(transformer\_model)

print(f"Device: {device}")

print(f"Model: GraphTransformer")

print(f"Total parameters: {total\_params:,}")

print(f"Trainable parameters: {trainable\_params:,}")

print(f"Model size: {total\_params \* 4 / 1024 / 1024:.2f} MB (float32)")

print("\nModel Architecture:")

print(f" Node features: 27")

print(f" Edge features: 5")

print(f" Hidden dimensions: 128")

print(f" Transformer layers: 4")

print(f" Attention heads: 8")

print(f" Dropout: 0.2")

print(f" Pooling: all (mean + max + add)")

print(f" Output: Binary classification (drug activity)")

print("\nModel Components:")

print(f" Transformer convolution layers: {len(transformer\_model.transformer\_convs)}")

print(f" Batch normalization layers: {len(transformer\_model.batch\_norms)}")

print(f" Molecular descriptor encoder: Dynamic (auto-created)")

print(f" Final classifier: 3-layer MLP")

print("\nFeatures:")

print("✓ Transformer layers with multi-head attention")

print("✓ Auto-detects molecular descriptor dimensions")

print("✓ Automatically handles flattening")

print("✓ Binary classification for drug activity")

print("✓ Multiple graph pooling (mean + max + add)")

print("✓ Residual connections")

print("=" \* 60)

print("GraphTransformer ready for training!")

print("=" \* 60)

# Execute training

transformer\_model\_config = {

'epochs': 300,

'device': 'cpu', # Use 'cuda' if GPU is available

'initial\_lr': 0.0005, # Initial learning rate

'weight\_decay': 1e-4, # L2 regularization

'scheduler\_factor': 0.5, # Factor by which to reduce LR

'scheduler\_patience': 10, # Reduce LR after 10 epochs without improvement

'early\_stop\_patience': 25, # Early stopping patience

'warmup\_epochs': 5, # Number of warmup epochs

'cosine\_annealing': True, # Use plateau scheduler instead of cosine

'dropout\_schedule': False, # Dynamic dropout scheduling

'gradient\_clip\_value': 1.0, # Gradient clipping value

'label\_smoothing': 0.1, # Label smoothing for regularization

'model\_name': 'transformer\_model',

'save\_dir': './trained\_models'

}

# history = train\_model(model, train\_loader, val\_loader, config=config)

transformer\_model\_history = train\_model(

model=transformer\_model,

train\_loader=train\_loader,

val\_loader=val\_loader,

config=transformer\_model\_config,

)

Task

This how my graph is print(graphs\_m

)

Data(x=[49, 27], edge\_index=[2, 104], edge\_attr=[104, 5], mol\_descriptors=[36], smiles='CCCCCCCCCCCCNC(=O)C(CCC=C(C)C)=C1C(OC(C)=O)CC2(C)C1CC(O)C1C3(C)CCC(O)C(C)C3CCC12C', atom\_symbols=[49], graph\_idx=0, y=[1], dataset\_idx=0)

Don’t forget any aspect of this graph during explainability

1. Load the model

trained model path

trained\_models/transformer\_model\_best.pth

1. Perform explainability
2. Interpret the explainability

I am using jupyter notebook. Excute code after each cell. Focus on explainability and interpretability.

Don’t use the standardise form of the data during explainability

Remember to add the name of the smile’s molecule

Focus only on explain the model I want you to explain and not create a new model

Also consider the edge\_atrr mole features atoms for the explainability

Let it be for aggregated instead of individual test data