neurips-gnn-4

July 29, 2025

```
[ ]: print("Hello")
```

```
Hello
```

```
[1]: !pip install rdkit
```

```
Collecting rdkit
  Downloading rdkit-2025.3.3-cp311-cp311-manylinux_2_28_x86_64.whl.metadata (4.0
kB)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages
(from rdkit) (2.0.2)
Requirement already satisfied: Pillow in /usr/local/lib/python3.11/dist-packages
(from rdkit) (11.3.0)
Downloading rdkit-2025.3.3-cp311-cp311-manylinux_2_28_x86_64.whl (34.9 MB)
                         34.9/34.9 MB
24.0 MB/s eta 0:00:00
Installing collected packages: rdkit
Successfully installed rdkit-2025.3.3
```

```
[2]: !pip install torch_geometric
```

```
Collecting torch_geometric
  Downloading torch_geometric-2.6.1-py3-none-any.whl.metadata (63 kB)
                           63.1/63.1 kB
4.1 MB/s eta 0:00:00
Requirement already satisfied: aiohttp in /usr/local/lib/python3.11/dist-
packages (from torch_geometric) (3.12.14)
Requirement already satisfied: fsspec in /usr/local/lib/python3.11/dist-packages
(from torch_geometric) (2025.3.0)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages
(from torch_geometric) (3.1.6)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages
(from torch_geometric) (2.0.2)
Requirement already satisfied: psutil>=5.8.0 in /usr/local/lib/python3.11/dist-
packages (from torch_geometric) (5.9.5)
Requirement already satisfied: pyparsing in /usr/local/lib/python3.11/dist-
packages (from torch_geometric) (3.2.3)
Requirement already satisfied: requests in /usr/local/lib/python3.11/dist-
```

```
packages (from torch_geometric) (2.32.3)
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages
(from torch_geometric) (4.67.1)
Requirement already satisfied: aiohappyeyeballs>=2.5.0 in
/usr/local/lib/python3.11/dist-packages (from aiohttp->torch_geometric) (2.6.1)
Requirement already satisfied: aiosignal>=1.4.0 in
/usr/local/lib/python3.11/dist-packages (from aiohttp->torch_geometric) (1.4.0)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.11/dist-
packages (from aiohttp->torch_geometric) (25.3.0)
Requirement already satisfied: frozenlist>=1.1.1 in
/usr/local/lib/python3.11/dist-packages (from aiohttp->torch_geometric) (1.7.0)
Requirement already satisfied: multidict<7.0,>=4.5 in
/usr/local/lib/python3.11/dist-packages (from aiohttp->torch_geometric) (6.6.3)
Requirement already satisfied: propcache>=0.2.0 in
/usr/local/lib/python3.11/dist-packages (from aiohttp->torch_geometric) (0.3.2)
Requirement already satisfied: yarl<2.0,>=1.17.0 in
/usr/local/lib/python3.11/dist-packages (from aiohttp->torch_geometric) (1.20.1)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.11/dist-packages (from jinja2->torch_geometric) (3.0.2)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.11/dist-packages (from requests->torch_geometric) (3.4.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-
packages (from requests->torch_geometric) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.11/dist-packages (from requests->torch_geometric) (2.5.0)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.11/dist-packages (from requests->torch_geometric)
(2025.7.14)
Requirement already satisfied: typing-extensions>=4.2 in
/usr/local/lib/python3.11/dist-packages (from
aiosignal>=1.4.0->aiohttp->torch_geometric) (4.14.1)
Downloading torch_geometric-2.6.1-py3-none-any.whl (1.1 MB)
                          1.1/1.1 MB
40.3 MB/s eta 0:00:00
Installing collected packages: torch_geometric
Successfully installed torch_geometric-2.6.1
```

[5]:

```python
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.data import Data, DataLoader
from torch_geometric.nn import GCNConv, global_mean_pool, global_add_pool
from rdkit import Chem
from rdkit.Chem import Descriptors
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.impute import KNNImputer
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
import warnings
warnings.filterwarnings('ignore')
```

[6]:

```
train_df = pd.read_csv('/content/train (1) (2).csv')
test_df = pd.read_csv('/content/test (1) (2).csv')

print(f"Train shape: {train_df.shape}")
print(f"Test shape: {test_df.shape}")
print(f"Train columns: {train_df.columns.tolist()}")
print(f"Missing values in train:")
print(train_df.isnull().sum())

property_columns = ['Tg', 'FFV', 'Tc', 'Density', 'Rg']
train_df_clean = train_df.dropna(subset=property_columns, how='all').copy()
print(f"Clean train shape after removing all-NaN rows: {train_df_clean.shape}")
```

```
Train shape: (7973, 7)
Test shape: (3, 2)
Train columns: ['id', 'SMILES', 'Tg', 'FFV', 'Tc', 'Density', 'Rg']
Missing values in train:
id            0
SMILES        0
Tg         7462
FFV         943
Tc         7236
Density    7360
Rg         7359
dtype: int64
Clean train shape after removing all-NaN rows: (7973, 7)
```

[7]:

```
knn_imputer = KNNImputer(n_neighbors=5, weights='uniform')
train_df_clean[property_columns] = knn_imputer.
 ↪fit_transform(train_df_clean[property_columns])

print(f"Missing values after KNN imputation:")
print(train_df_clean[property_columns].isnull().sum())
```

```
Missing values after KNN imputation:
Tg         0
FFV        0
Tc         0
Density    0
Rg         0
```

```
dtype: int64
```

[8]:
```
iterative_imputer = IterativeImputer(random_state=42, max_iter=10)
train_df_clean[property_columns] = iterative_imputer.
 ↪fit_transform(train_df_clean[property_columns])

print(f"Missing values after Iterative imputation:")
print(train_df_clean[property_columns].isnull().sum())
```

```
Missing values after Iterative imputation:
Tg         0
FFV        0
Tc         0
Density    0
Rg         0
dtype: int64
```

[9]:
```
def atom_features(atom):
    features = []
    atomic_numbers = [1, 6, 7, 8, 9, 15, 16, 17, 35, 53]
    features.extend([atom.GetAtomicNum() == x for x in atomic_numbers])
    features.extend([atom.GetDegree() == x for x in range(6)])
    features.extend([atom.GetFormalCharge() == x for x in range(-2, 3)])
    features.extend([atom.GetHybridization() == x for x in [
        Chem.rdchem.HybridizationType.SP,
        Chem.rdchem.HybridizationType.SP2,
        Chem.rdchem.HybridizationType.SP3,
        Chem.rdchem.HybridizationType.SP3D,
        Chem.rdchem.HybridizationType.SP3D2
    ]])
    features.append(atom.GetIsAromatic())
    features.append(atom.IsInRing())
    return np.array(features, dtype=np.float32)
```

[10]:
```
def bond_features(bond):
    features = []
    bond_types = [
        Chem.rdchem.BondType.SINGLE,
        Chem.rdchem.BondType.DOUBLE,
        Chem.rdchem.BondType.TRIPLE,
        Chem.rdchem.BondType.AROMATIC
    ]
    features.extend([bond.GetBondType() == x for x in bond_types])
    features.append(bond.GetIsConjugated())
    features.append(bond.IsInRing())
    return np.array(features, dtype=np.float32)
```

[11]:

```
def smiles_to_graph(smiles):
    mol = Chem.MolFromSmiles(smiles)
    if mol is None:
        return None
    mol = Chem.AddHs(mol)
    atom_feat = []
    for atom in mol.GetAtoms():
        atom_feat.append(atom_features(atom))
    if len(atom_feat) == 0:
        return None
    x = torch.tensor(np.array(atom_feat), dtype=torch.float)
    edge_indices = []
    edge_attrs = []
    for bond in mol.GetBonds():
        i = bond.GetBeginAtomIdx()
        j = bond.GetEndAtomIdx()
        edge_indices.extend([[i, j], [j, i]])
        bond_feat = bond_features(bond)
        edge_attrs.extend([bond_feat, bond_feat])
    if len(edge_indices) == 0:
        edge_index = torch.empty((2, 0), dtype=torch.long)
        edge_attr = torch.empty((0, 6), dtype=torch.float)
    else:
        edge_index = torch.tensor(edge_indices, dtype=torch.long).t().
 ↪contiguous()
        edge_attr = torch.tensor(np.array(edge_attrs), dtype=torch.float)
    return Data(x=x, edge_index=edge_index, edge_attr=edge_attr)
print("Testing featurization...")
test_smiles = "*CC(*)c1ccccc1C(=O)OCCCCCC"
test_graph = smiles_to_graph(test_smiles)
if test_graph is not None:
    print(f"Node features shape: {test_graph.x.shape}")
    print(f"Edge index shape: {test_graph.edge_index.shape}")
    print(f"Edge attributes shape: {test_graph.edge_attr.shape}")
else:
    print("Failed to create graph from test SMILES")
```

```
Testing featurization…
Node features shape: torch.Size([39, 28])
Edge index shape: torch.Size([2, 78])
Edge attributes shape: torch.Size([78, 6])
```

[12]:

```
train_graphs = []
train_targets = []
valid_indices = []

print("Creating training graphs...")
```

```
for idx, row in train_df_clean.iterrows():
    graph = smiles_to_graph(row['SMILES'])
    if graph is not None:
        train_graphs.append(graph)
        targets = [row[col] for col in property_columns]
        train_targets.append(targets)
        valid_indices.append(idx)

print(f"Successfully created {len(train_graphs)} training graphs")
train_targets = np.array(train_targets, dtype=np.float32)
print(f"Training targets shape: {train_targets.shape}")
```

```
Creating training graphs…
Successfully created 7973 training graphs
Training targets shape: (7973, 5)
```

[13]:

```
scaler = StandardScaler()
train_targets_scaled = scaler.fit_transform(train_targets)
print(f"Scaled targets shape: {train_targets_scaled.shape}")

train_idx, val_idx = train_test_split(
    range(len(train_graphs)),
    test_size=0.2,
    random_state=42,
    stratify=None
)

train_graphs_split = [train_graphs[i] for i in train_idx]
val_graphs_split = [train_graphs[i] for i in val_idx]
train_targets_split = train_targets_scaled[train_idx]
val_targets_split = train_targets_scaled[val_idx]

print(f"Train split: {len(train_graphs_split)} graphs")
print(f"Validation split: {len(val_graphs_split)} graphs")
print(f"Train targets split shape: {train_targets_split.shape}")
print(f"Val targets split shape: {val_targets_split.shape}")
```

```
Scaled targets shape: (7973, 5)
Train split: 6378 graphs
Validation split: 1595 graphs
Train targets split shape: (6378, 5)
Val targets split shape: (1595, 5)
```

[14]:

```
for i, (graph, target) in enumerate(zip(train_graphs_split,␣
 ↪train_targets_split)):
    graph.y = torch.tensor(target, dtype=torch.float).reshape(1, -1)
```

```python
for i, (graph, target) in enumerate(zip(val_graphs_split, val_targets_split)):
    graph.y = torch.tensor(target, dtype=torch.float).reshape(1, -1)

print("Creating test graphs...")
test_graphs = []
test_valid_indices = []

for idx, row in test_df.iterrows():
    graph = smiles_to_graph(row['SMILES'])
    if graph is not None:
        test_graphs.append(graph)
        test_valid_indices.append(idx)

print(f"Successfully created {len(test_graphs)} test graphs")
```

```
Creating test graphs…
Successfully created 3 test graphs
```

[15]:

```python
class MultiTaskGNN(nn.Module):
    def __init__(self, node_features, edge_features, hidden_dim=256,␣
 ↪num_layers=4, dropout=0.2):
        super(MultiTaskGNN, self).__init__()
        self.node_features = node_features
        self.edge_features = edge_features
        self.hidden_dim = hidden_dim
        self.num_layers = num_layers
        self.dropout = dropout
        self.node_embedding = nn.Linear(node_features, hidden_dim)
        self.edge_embedding = nn.Linear(edge_features, hidden_dim)
        self.conv_layers = nn.ModuleList()
        self.batch_norms = nn.ModuleList()
        for i in range(num_layers):
            self.conv_layers.append(GCNConv(hidden_dim, hidden_dim))
            self.batch_norms.append(nn.BatchNorm1d(hidden_dim))
        self.global_pool = global_mean_pool
        self.task_heads = nn.ModuleList()
        for i in range(5):
            head = nn.Sequential(
                nn.Linear(hidden_dim, hidden_dim // 2),
                nn.ReLU(),
                nn.Dropout(dropout),
                nn.Linear(hidden_dim // 2, hidden_dim // 4),
                nn.ReLU(),
                nn.Dropout(dropout),
                nn.Linear(hidden_dim // 4, 1)
            )
            self.task_heads.append(head)
    def forward(self, data):
        x, edge_index, edge_attr, batch = data.x, data.edge_index, data.
 ↪edge_attr, data.batch
        if x.size(1) != self.node_features:
            raise ValueError(f"Expected node features: {self.node_features},␣
 ↪got: {x.size(1)}")
        x = self.node_embedding(x)
        x = F.relu(x)
        x = F.dropout(x, p=self.dropout, training=self.training)
        for i, (conv, bn) in enumerate(zip(self.conv_layers, self.batch_norms)):
            x_residual = x
            x = conv(x, edge_index)
            x = bn(x)
            x = F.relu(x)
            x = F.dropout(x, p=self.dropout, training=self.training)
            if i > 0:
                x = x + x_residual
        x = self.global_pool(x, batch)
        outputs = []
        for head in self.task_heads:
            out = head(x)
            outputs.append(out)
        return torch.cat(outputs, dim=1)
```

[16]:

```
if len(train_graphs_split) > 0:
    sample_graph = train_graphs_split[0]
    node_features = sample_graph.x.shape[1]
    edge_features = sample_graph.edge_attr.shape[1] if sample_graph.edge_attr.
 ↪size(0) > 0 else 6
    print(f"Node features dimension: {node_features}")
    print(f"Edge features dimension: {edge_features}")
    model = MultiTaskGNN(
        node_features=node_features,
        edge_features=edge_features,
        hidden_dim=256,
        num_layers=4,
        dropout=0.2
    )
    print(f"Model created with {sum(p.numel() for p in model.parameters())}␣
 ↪parameters")
    print(f"Model: {model}")
else:
    print("No training graphs available for model creation")
```

```
Node features dimension: 28
Edge features dimension: 6
```

```
Model created with 480517 parameters
Model: MultiTaskGNN(
  (node_embedding): Linear(in_features=28, out_features=256, bias=True)
  (edge_embedding): Linear(in_features=6, out_features=256, bias=True)
  (conv_layers): ModuleList(
    (0-3): 4 x GCNConv(256, 256)
  )
  (batch_norms): ModuleList(
    (0-3): 4 x BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
  (task_heads): ModuleList(
    (0-4): 5 x Sequential(
      (0): Linear(in_features=256, out_features=128, bias=True)
      (1): ReLU()
      (2): Dropout(p=0.2, inplace=False)
      (3): Linear(in_features=128, out_features=64, bias=True)
      (4): ReLU()
      (5): Dropout(p=0.2, inplace=False)
      (6): Linear(in_features=64, out_features=1, bias=True)
    )
  )
)
```

[17]:

```python
class WeightedMAELoss(nn.Module):
    def __init__(self, train_targets, K=5):
        super(WeightedMAELoss, self).__init__()
        self.K = K
        self.weights = self.calculate_weights(train_targets)
        print(f"Calculated weights: {self.weights}")

    def calculate_weights(self, targets):
        targets = targets.copy()
        n_samples, n_properties = targets.shape
        weights = []
        for i in range(n_properties):
            property_values = targets[:, i]
            valid_mask = ~np.isnan(property_values)
            valid_values = property_values[valid_mask]
            if len(valid_values) == 0:
                weights.append(1.0)
                continue
            n_i = len(valid_values)
            r_i = np.max(valid_values) - np.min(valid_values)
            if r_i == 0:
                r_i = 1.0
            scale_factor = 1.0 / r_i
            inverse_sqrt_factor = np.sqrt(1.0 / n_i)
            weight = scale_factor * inverse_sqrt_factor
            weights.append(weight)
        weights = np.array(weights)
        sum_weights = np.sum(weights)
        if sum_weights > 0:
            weights = (weights / sum_weights) * self.K
        return torch.tensor(weights, dtype=torch.float32)

    def forward(self, predictions, targets):
        if predictions.size() != targets.size():
            raise ValueError(f"Size mismatch: predictions {predictions.size()},␣
 ↪targets {targets.size()}")
        weights = self.weights.to(predictions.device)
        abs_errors = torch.abs(predictions - targets)
        weighted_errors = abs_errors * weights.unsqueeze(0)
        mae_per_property = torch.mean(weighted_errors, dim=0)
        wmae = torch.mean(mae_per_property)
        return wmae
```

[18]:

```
train_targets_for_weights = train_targets.copy()
for i in range(train_targets_for_weights.shape[1]):
    col_mean = np.nanmean(train_targets_for_weights[:, i])
    train_targets_for_weights[:, i] = np.where(
        np.isnan(train_targets_for_weights[:, i]),
        col_mean,
        train_targets_for_weights[:, i]
    )
criterion = WeightedMAELoss(train_targets_for_weights)
print("Weighted MAE Loss function created successfully")
```

```
Calculated weights: tensor([1.6555e-03, 1.8666e+00, 2.1505e+00, 9.4007e-01,
4.1165e-02])
Weighted MAE Loss function created successfully
```

[19]:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")

model = model.to(device)
criterion = criterion.to(device)

optimizer = torch.optim.AdamW(model.parameters(), lr=1e-4, weight_decay=1e-5)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode='min', factor=0.5, patience=10, verbose=True
)

train_loader = DataLoader(train_graphs_split, batch_size=32, shuffle=True)
```

```
val_loader = DataLoader(val_graphs_split, batch_size=32, shuffle=False)

print(f"Train loader: {len(train_loader)} batches")
print(f"Val loader: {len(val_loader)} batches")
```

```
Using device: cuda
Train loader: 200 batches
Val loader: 50 batches
```

[20]:

```
def train_epoch(model, loader, optimizer, criterion, device):
    model.train()
    total_loss = 0
    num_batches = 0
    for batch in loader:
        batch = batch.to(device)
        optimizer.zero_grad()
        if batch.x.size(0) == 0:
            continue
        outputs = model(batch)
        targets = batch.y
        if targets.dim() == 3 and targets.size(1) == 1:
            targets = targets.squeeze(1)
        elif targets.dim() == 2 and targets.size(1) == 5:
            targets = targets
        else:
            print(f"Unexpected target shape: {targets.shape}")
            continue
        if outputs.size() != targets.size():
            print(f"Size mismatch: outputs {outputs.size()}, targets {targets.
 ↪size()}")
            continue
        loss = criterion(outputs, targets)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
        total_loss += loss.item()
        num_batches += 1
    return total_loss / max(num_batches, 1)

def validate_epoch(model, loader, criterion, device):
    model.eval()
    total_loss = 0
    num_batches = 0
    with torch.no_grad():
        for batch in loader:
            batch = batch.to(device)
            if batch.x.size(0) == 0:
                continue
            outputs = model(batch)
            targets = batch.y
            if targets.dim() == 3 and targets.size(1) == 1:
                targets = targets.squeeze(1)
            elif targets.dim() == 2 and targets.size(1) == 5:
                targets = targets
            else:
                print(f"Unexpected target shape: {targets.shape}")
                continue
            if outputs.size() != targets.size():
                print(f"Size mismatch: outputs {outputs.size()}, targets␣
 ↪{targets.size()}")
                continue
            loss = criterion(outputs, targets)
            total_loss += loss.item()
            num_batches += 1
    return total_loss / max(num_batches, 1)

print("Training setup complete")
```

```
Training setup complete
```

[21]:

```
print("=== DEBUGGING TENSOR SHAPES ===")
for i in range(min(3, len(train_graphs_split))):
    print(f"Train graph {i}: x.shape = {train_graphs_split[i].x.shape}, y.shape␣
 ↪= {train_graphs_split[i].y.shape}")

train_batch = next(iter(train_loader))
print(f"Train batch: x.shape = {train_batch.x.shape}, y.shape = {train_batch.y.
 ↪shape}")
print(f"Train batch: batch.shape = {train_batch.batch.shape}")
print(f"Train batch size (num graphs): {train_batch.batch.max().item() + 1}")

model.eval()
with torch.no_grad():
    train_batch = train_batch.to(device)
    outputs = model(train_batch)
    print(f"Model outputs shape: {outputs.shape}")
    print(f"Expected targets shape: {train_batch.y.shape}")

print("=== END DEBUG ===")

print("Fixing target tensor assignment...")

for i, (graph, target) in enumerate(zip(train_graphs_split,␣
 ↪train_targets_split)):
    graph.y = torch.tensor(target, dtype=torch.float).reshape(1, -1)

for i, (graph, target) in enumerate(zip(val_graphs_split, val_targets_split)):
    graph.y = torch.tensor(target, dtype=torch.float).reshape(1, -1)

print("Target tensors fixed")

train_loader = DataLoader(train_graphs_split, batch_size=32, shuffle=True)
val_loader = DataLoader(val_graphs_split, batch_size=32, shuffle=False)

train_batch = next(iter(train_loader))
print(f"Fixed train batch: x.shape = {train_batch.x.shape}, y.shape =␣
 ↪{train_batch.y.shape}")
print(f"Fixed batch size (num graphs): {train_batch.batch.max().item() + 1}")

model.eval()
with torch.no_grad():
    train_batch = train_batch.to(device)
    outputs = model(train_batch)
    print(f"Model outputs shape: {outputs.shape}")
    print(f"Targets shape: {train_batch.y.shape}")
    if outputs.shape == train_batch.y.shape:
        print("  Shapes match! Ready for training.")
    else:
        print("  Shapes still don't match. Need further debugging.")
```

```
=== DEBUGGING TENSOR SHAPES ===
Train graph 0: x.shape = torch.Size([103, 28]), y.shape = torch.Size([1, 5])
Train graph 1: x.shape = torch.Size([61, 28]), y.shape = torch.Size([1, 5])
Train graph 2: x.shape = torch.Size([95, 28]), y.shape = torch.Size([1, 5])
Train batch: x.shape = torch.Size([1689, 28]), y.shape = torch.Size([32, 5])
Train batch: batch.shape = torch.Size([1689])
Train batch size (num graphs): 32
Model outputs shape: torch.Size([32, 5])
Expected targets shape: torch.Size([32, 5])
=== END DEBUG ===
Fixing target tensor assignment…
Target tensors fixed
Fixed train batch: x.shape = torch.Size([2088, 28]), y.shape = torch.Size([32,
5])
Fixed batch size (num graphs): 32
Model outputs shape: torch.Size([32, 5])
Targets shape: torch.Size([32, 5])
  Shapes match! Ready for training.
```

[22]:

```
num_epochs = 100
best_val_loss = float('inf')
```

```python
patience = 15
patience_counter = 0

print("Starting training...")

for epoch in range(num_epochs):
    train_loss = train_epoch(model, train_loader, optimizer, criterion, device)
    val_loss = validate_epoch(model, val_loader, criterion, device)
    scheduler.step(val_loss)
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        patience_counter = 0
        torch.save(model.state_dict(), 'best_model.pth')
    else:
        patience_counter += 1
    if epoch % 10 == 0:
        print(f'Epoch {epoch:03d}, Train Loss: {train_loss:.4f}, Val Loss:␣
 ↪{val_loss:.4f}')
    if patience_counter >= patience:
        print(f'Early stopping at epoch {epoch}')
        break

print("Training completed")
print(f"Best validation loss: {best_val_loss:.4f}")

model.load_state_dict(torch.load('best_model.pth'))
model.eval()

print("Model loaded with best weights")
```

```
Starting training…
Epoch 000, Train Loss: 0.7320, Val Loss: 0.6502
Epoch 010, Train Loss: 0.5843, Val Loss: 0.5650
Epoch 020, Train Loss: 0.5634, Val Loss: 0.5456
Epoch 030, Train Loss: 0.5529, Val Loss: 0.5255
Epoch 040, Train Loss: 0.5451, Val Loss: 0.5223
Epoch 050, Train Loss: 0.5377, Val Loss: 0.5172
Epoch 060, Train Loss: 0.5312, Val Loss: 0.4958
Epoch 070, Train Loss: 0.5280, Val Loss: 0.4998
Epoch 080, Train Loss: 0.5214, Val Loss: 0.4980
Epoch 090, Train Loss: 0.5182, Val Loss: 0.4889
Training completed
Best validation loss: 0.4884
Model loaded with best weights
```

[23]:

```python
test_loader = DataLoader(test_graphs, batch_size=32, shuffle=False)
```

```
test_predictions = []
with torch.no_grad():
    for batch in test_loader:
        batch = batch.to(device)
        if batch.x.size(0) == 0:
            continue
        outputs = model(batch)
        test_predictions.append(outputs.cpu())

test_predictions = torch.cat(test_predictions, dim=0).numpy()

print("\nTest predictions generated.")
print(f"Shape of test predictions: {test_predictions.shape}")

final_val_loss = validate_epoch(model, val_loader, criterion, device)
print(f"Final Validation Weighted MAE Loss (using best model): {final_val_loss:.
 ↪4f}")
```

```
Test predictions generated.
Shape of test predictions: (3, 5)
Final Validation Weighted MAE Loss (using best model): 0.4884
```

[24]:

```
test_loader = DataLoader(test_graphs, batch_size=32, shuffle=False)

all_predictions = []

print("Making predictions on test set...")
model.eval()
with torch.no_grad():
    for batch in test_loader:
        batch = batch.to(device)
        if batch.x.size(0) == 0:
            continue
        outputs = model(batch)
        predictions = outputs.cpu().numpy()
        all_predictions.append(predictions)

if len(all_predictions) > 0:
    all_predictions = np.vstack(all_predictions)
    print(f"Predictions shape: {all_predictions.shape}")
    predictions_original_scale = scaler.inverse_transform(all_predictions)
    print(f"Predictions original scale shape: {predictions_original_scale.
 ↪shape}")
    submission_df = pd.DataFrame({
        'id': [test_df.iloc[i]['id'] for i in test_valid_indices],
        'Tg': predictions_original_scale[:, 0],
        'FFV': predictions_original_scale[:, 1],
        'Tc': predictions_original_scale[:, 2],
        'Density': predictions_original_scale[:, 3],
        'Rg': predictions_original_scale[:, 4]
    })
    missing_ids = set(test_df['id']) - set(submission_df['id'])
    if missing_ids:
        print(f"Missing {len(missing_ids)} predictions for invalid SMILES")
        mean_predictions = np.mean(predictions_original_scale, axis=0)
        for missing_id in missing_ids:
            new_row = {'id': missing_id}
            for i, col in enumerate(property_columns):
                new_row[col] = mean_predictions[i]
            submission_df = pd.concat([submission_df, pd.DataFrame([new_row])],␣
 ↪ignore_index=True)
    submission_df = submission_df.sort_values('id').reset_index(drop=True)
    print(f"Final submission shape: {submission_df.shape}")
    print("Submission preview:")
    print(submission_df.head())
    submission_df.to_csv('submission.csv', index=False)
    print("Submission saved as 'submission.csv'")
    print("\nSummary statistics:")
    print(submission_df[property_columns].describe())
else:
    print("No predictions made - check test data processing")
```

```
Making predictions on test set…
Predictions shape: (3, 5)
Predictions original scale shape: (3, 5)
Final submission shape: (3, 6)
Submission preview:
           id        Tg       FFV        Tc   Density         Rg
0  1109053969 -2.526850  0.372904  0.238266  0.987924  16.185196
1  1422188626 -2.526854  0.381491  0.247944  0.962958  16.175173
2  2032016830 -2.526848  0.353418  0.230338  1.053573  15.666262
Submission saved as 'submission.csv'

Summary statistics:
             Tg       FFV        Tc   Density         Rg
count  3.000000  3.000000  3.000000  3.000000   3.000000
mean  -2.526851  0.369271  0.238849  1.001485  16.008875
std    0.000003  0.014385  0.008817  0.046805   0.296756
min   -2.526854  0.353418  0.230338  0.962958  15.666262
25%   -2.526852  0.363161  0.234302  0.975441  15.920717
50%   -2.526850  0.372904  0.238266  0.987924  16.175173
75%   -2.526849  0.377197  0.243105  1.020748  16.180184
max   -2.526848  0.381491  0.247944  1.053573  16.185196
```