Chem 277B Spring 2024 Tutorial 11

Outline

1. Variational Auto-Encoder
2. Graph Neural Network

In [ ]:
```
import itertools

from tqdm import tqdm
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

import torch
import torch.nn as nn
```

1. Variational Auto-Encoder (VAE)

In [ ]:
```
class VAE(nn.Module):
    def __init__(self, in_channels=1, z_dim=8):
        super().__init__()
        self.encoder = nn.Sequential(
            # conv1, input_channel -> 4
            # relu
            # conv2, channel 4 -> 8
            # relu
            # flatten
            nn.Conv2d(in_channels, 4, kernel_size=4, padding=1, stride=2),
            nn.ReLU(),
            nn.Conv2d(4, 8, kernel_size=4, padding=1, stride=2),
            nn.ReLU(),
            nn.Flatten()
        )

        # manually calculate the dimension after all convolutions
        dim_after_conv = 8
        hidden_dim = 8 * dim_after_conv * dim_after_conv

        self.readout_mu = nn.Linear(hidden_dim, z_dim)
        self.readout_sigma = nn.Linear(hidden_dim, z_dim)

        # You can use nn.ConvTranspose2d to decode
        self.decoder = nn.Sequential(
            nn.Linear(z_dim, hidden_dim),
            nn.Unflatten(1, (8, dim_after_conv, dim_after_conv)),
            # transpose-conv, channel 8 -> 4
```

```
            # relu
            # transpose-conv, channel 4 -> input_channel, which is 1
            # use a sigmoid activation to squeeze the outputs between 0 and
            nn.ConvTranspose2d(8, 4, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(4, in_channels, kernel_size=4, stride=2, padd
            nn.Sigmoid(),
        )

    def reparameterize(self, mu, sigma):
        """
        Reparameterize, i.e. generate a z ~ N(\mu, \sigma)
        """
        # generate epsilon ~ N(0, I)
        # hint: use torch.randn or torch.randn_like
        epsilon = torch.rand_like(sigma)
        # z = \mu + \sigma * \epsilon
        z = mu + sigma * epsilon
        return z

    def encode(self, x):
        # call the encoder to map input to a hidden state vector
        h = self.encoder(x)
        # use the "readout" layer to get \mu and \sigma
        mu = self.readout_mu(h)
        sigma = self.readout_sigma(h)
        return mu, sigma

    def decode(self, z):
        # call the decoder to map z back to x
        return self.decoder(z)

    def forward(self, x):
        mu, sigma = self.encode(x)
        z = self.reparameterize(mu, sigma)
        x_recon = self.decode(z)
        return x_recon, mu, sigma
```

In [ ]:
```
vae = VAE(in_channels=3)
x_recon, mu, sigma = vae(torch.rand(10, 3, 32, 32))
x_recon.shape
```

Out[ ]:
```
torch.Size([10, 3, 32, 32])
```

2. Graph Neural Network (GNN)

In [ ]:
```
from torch_geometric.datasets import QM9
from torch_geometric.loader import DataLoader as GraphDataLoader
from torch_geometric.utils import scatter
```

In [ ]:
```
def load_qm9(path="./QM9"):
    def transform(data):
        edge_index = torch.tensor(
```

```
            list(itertools.permutations(range(data.x.shape[0]), 2)),
            dtype=torch.long
        ).T
        edge_feature = 1 / torch.sqrt(
            torch.sum(
                (data.pos[edge_index[0]] - data.pos[edge_index[1]]) ** 2,
                axis=1, keepdim=True
            )
        )
        data.edge_index = edge_index
        data.edge_attr = edge_feature
        data.y = data.y[:, [-7]]
        return data

    qm9 = QM9(path, transform=transform)
    return qm9

qm9 = load_qm9("./QM9")
```

In [ ]:
```
class Layer(nn.Module):
    """
    Basic layer, a linear layer with a ReLU activation
    """
    def __init__(self, in_dim, out_dim):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(in_dim, out_dim), # linear layer
            nn.ReLU() # relu
        )

    def forward(self, x):
        return self.layers(x)


class MessagePassingLayer(nn.Module):
    """
    A message passing layer that updates nodes/edge features
    """
    def __init__(self, node_hidden_dim, edge_hidden_dim):
        super().__init__()
        # figure out the input/output dimension
        self.edge_net = Layer(2*node_hidden_dim + edge_hidden_dim, edge_hidc
        # figure out the input/output dimension
        self.node_net = Layer(node_hidden_dim + edge_hidden_dim, node_hidder

    def forward(self, node_features, edge_features, edge_index):
        """
        Update node and edge features

        Parameters
        ----------
        node_features: torch.Tensor
            Node features from the previous layer
        edge_features: torch.Tensor
            Edge features from the previous layer
        edge_index: torch.Tensor
```

```
            A sparse matrix (n_edge, 2) in which each column denotes node in
        """
        # concatnate previous edge features with node features forming the e
        # hint: use edge_features[edge_index[0(or 1)]] to get node features 
        concate_edge_features = torch.cat([
            node_features[edge_index[0]], # features of one node
            node_features[edge_index[1]], # features of the other node
            edge_features # previous edge features
        ], dim=1)

        # pass through the "edge_net" to map it back to the original dimensi
        updated_edge_features = self.edge_net(concate_edge_features)


        # use scatter to aggrate the edge features to nodes
        aggr_edge_features = scatter(updated_edge_features, edge_index[0])
        # concatenate it with previous node features
        concate_node_features = torch.cat([aggr_edge_features, node_features
        # pass through the "node_net" to map it back to the original dimensi
        updated_node_features = self.node_net(concate_node_features)

        return updated_node_features, updated_edge_features


class GraphNet(nn.Module):
    def __init__(self, node_input_dim, edge_input_dim, node_hidden_dim, edge
        super().__init__()
        # embed the input node features
        self.node_embed = Layer(node_input_dim, node_hidden_dim)
        # embed the input edge features
        self.edge_embed = Layer(edge_input_dim, edge_hidden_dim)
        # use a linear layer as readout to get the "atomic" energy contribut
        self.readout = Layer(node_hidden_dim, 1)
        # message passing layer
        self.message_passing = MessagePassingLayer(node_hidden_dim, edge_hic

    def forward(self, node_features, edge_features, edge_index, batch):
        """
        Update node and edge features

        Parameters
        ----------
        node_features: torch.Tensor
            Node features from the previous layer
        edge_features: torch.Tensor
            Edge features from the previous layer
        edge_index: torch.Tensor
            A sparse matrix (n_edges, 2) in which each column denotes node i
        batch: torch.Tensor
            A 1-D tensor (n_nodes,) that tells you each node belongs to whic
        """
        node_hidden = self.node_embed(node_features) # call the node embeddi
        edge_hidden = self.edge_embed(edge_features) # call the edge embeddi
        updated_node_hidden, updated_edge_hidden = self.message_passing(node
        readout = self.readout(updated_node_hidden) # use the readout layer 
```

```
        out = scatter(readout, batch) # use the scatter function to aggregat
        return out
```

In [ ]:
```
qm9[0].x.shape
```

Out[ ]:
```
torch.Size([5, 11])
```

In [ ]:
```
node_input_dim = 11
edge_input_dim = 1
node_hidden_dim = 64
edge_hidden_dim = 64

net = GraphNet(node_input_dim, edge_input_dim, node_hidden_dim, edge_hidden_
```

In [ ]:
```
batch_data = next(iter(GraphDataLoader(qm9[:10], batch_size=2)))
batch_pred = net(
    batch_data.x, batch_data.edge_attr,
    batch_data.edge_index, batch_data.batch
)
batch_pred
```

Out[ ]:
```
tensor([[1.6505],
        [1.1540]], grad_fn=<ScatterAddBackward0>)
```

In [ ]: