import numpy as np

import torch

import torch.nn as nn

import torch.nn.functional as F

from torch.utils.data import Dataset, DataLoader

from torch.optim import AdamW

from torch.optim.lr\_scheduler import ReduceLROnPlateau

import pandas as pd

import gc

# ------------------------- #

# ⚡ Set up Device

# ------------------------- #

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

print(f"Running on: {device}")

# ------------------------- #

# 📊 Dataset Class

# ------------------------- #

class MatrixDataset(Dataset):

def \_\_init\_\_(self, df, n\_nodes):

self.df = df

self.n\_nodes = n\_nodes

def get\_adj(self, path):

adj = np.zeros((self.n\_nodes, self.n\_nodes), dtype=np.float16) # Use float16 to reduce memory

adj[path[0], 0] = 1 # From source

adj[self.n\_nodes - 1, path[-1]] = 1 # To sink

adj[self.n\_nodes - 1, self.n\_nodes - 1] = 1 # Sink absorbs

for i in range(len(path) - 1):

adj[path[i + 1], path[i]] = 1

adj\_sum = adj.sum(axis=0)

adj\_sum[adj\_sum == 0] = 1 # Prevent division by zero

adj = np.divide(adj, adj\_sum)

adj = np.nan\_to\_num(adj)

return adj

def \_\_getitem\_\_(self, index):

path = self.df.iloc[index]['path']

adj = self.get\_adj(path)

return torch.tensor(adj, dtype=torch.float16) # Reduced precision

def \_\_len\_\_(self):

return len(self.df)

# ------------------------- #

# 🧠 Model Definition

# ------------------------- #

class MatrixEmbedder(nn.Module):

def \_\_init\_\_(self, n\_nodes, embed\_dim):

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

self.layer1 = nn.Sequential(nn.Linear(n\_nodes\*\*2, 500), nn.BatchNorm1d(500), nn.LeakyReLU(), nn.Dropout(0.3))

self.layer2 = nn.Sequential(nn.Linear(500, 250), nn.BatchNorm1d(250), nn.LeakyReLU(), nn.Dropout(0.3))

self.layer3 = nn.Sequential(nn.Linear(250, 100), nn.BatchNorm1d(100), nn.LeakyReLU(), nn.Dropout(0.3))

self.layer4 = nn.Sequential(nn.Linear(100, 50), nn.BatchNorm1d(50), nn.LeakyReLU(), nn.Dropout(0.3))

self.layer5 = nn.Linear(50, embed\_dim) # Output layer

def forward(self, x):

x = x.view(x.size(0), -1) # Flatten the adjacency matrix

x = self.layer1(x)

x = self.layer2(x)

x = self.layer3(x)

x = self.layer4(x)

x = self.layer5(x)

return x

# ------------------------- #

# ⚡ Contrastive Loss Function

# ------------------------- #

class ContrastiveLoss(nn.Module):

def \_\_init\_\_(self, margin=1.0):

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

self.margin = margin

def forward(self, output1, output2, label):

euclidean\_distance = F.pairwise\_distance(output1, output2)

loss = torch.mean((1 - label) \* torch.pow(euclidean\_distance, 2) +

(label) \* torch.pow(torch.clamp(self.margin - euclidean\_distance, min=0.0), 2))

return loss

# ------------------------- #

# 🔄 Early Stopping Mechanism

# ------------------------- #

class EarlyStopping:

def \_\_init\_\_(self, patience=5, verbose=False):

self.patience = patience

self.verbose = verbose

self.counter = 0

self.best\_loss = None

def \_\_call\_\_(self, val\_loss):

if self.best\_loss is None or val\_loss < self.best\_loss:

self.best\_loss = val\_loss

self.counter = 0

else:

self.counter += 1

if self.counter >= self.patience:

if self.verbose:

print("Early stopping triggered")

return True

return False

# ------------------------- #

# 🚀 Training Loop (Optimized)

# ------------------------- #

def train\_model(df, n\_nodes, embed\_dim=20, batch\_size=32, epochs=20):

# Dataset and DataLoader

dataset = MatrixDataset(df, n\_nodes)

dataloader = DataLoader(dataset, batch\_size=batch\_size, shuffle=True, num\_workers=0)

# Model, Optimizer, Loss

model = MatrixEmbedder(n\_nodes=n\_nodes, embed\_dim=embed\_dim).to(device).half() # Use half precision

optimizer = AdamW(model.parameters(), lr=1e-3, weight\_decay=1e-4)

scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=5, verbose=True)

criterion = ContrastiveLoss(margin=1.0)

early\_stopping = EarlyStopping(patience=7, verbose=True)

for epoch in range(epochs):

model.train()

total\_loss = 0

for batch in dataloader:

batch = batch.to(device).half() # Convert batch to half precision

optimizer.zero\_grad()

# Generate random pairs for contrastive loss

idx = torch.randint(0, batch.size(0), (batch.size(0),))

batch\_pairs = batch[idx].to(device)

with torch.no\_grad(): # Reduce memory for non-gradient steps

embeddings1 = model(batch)

embeddings2 = model(batch\_pairs)

labels = torch.randint(0, 2, (batch.size(0),), dtype=torch.float16).to(device) # Random 0/1 labels

loss = criterion(embeddings1, embeddings2, labels)

loss.backward()

optimizer.step()

total\_loss += loss.item()

# Explicitly clear memory

del batch, batch\_pairs, embeddings1, embeddings2

torch.cuda.empty\_cache()

gc.collect()

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

print(f"Epoch [{epoch+1}/{epochs}], Loss: {avg\_loss:.4f}")

scheduler.step(avg\_loss)

if early\_stopping(avg\_loss):

print("Stopping Training Early")

break

return model

# ------------------------- #

# 📊 Example Usage

# ------------------------- #

if \_\_name\_\_ == "\_\_main\_\_":

# Example DataFrame creation

data = {

'channel\_visit\_id': [111, 222, 333, 444, 555],

'path': [[1, 2, 3], [2, 3, 4, 5], [1, 1, 1], [4, 5, 6], [2, 2, 2, 3]]

}

df = pd.DataFrame(data)

n\_nodes = df['path'].apply(max).max() + 2 # +2 for source and sink nodes

trained\_model = train\_model(df, n\_nodes=n\_nodes, embed\_dim=20, batch\_size=16, epochs=50)