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.cuda.amp import GradScaler, autocast

from torch.optim.lr\_scheduler import ReduceLROnPlateau

import pandas as pd

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

# 📊 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))

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.float32)

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

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

# 🧮 Markov Distance Calculation

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

def dme(mat1, mat2, vector):

mat = np.matmul(mat1.T, mat2)

v = np.matmul(mat, vector)

out = np.matmul(vector.T, v)

return out[0, 0]

def markov\_distance(mata, matb):

v1 = np.ones((mata.shape[0], 1))

out = dme(mata, matb, v1) - 0.5 \* dme(mata, mata, v1) - 0.5 \* dme(matb, matb, v1)

return out

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

# 🔄 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 with AMP

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

def train\_model(df, n\_nodes, embed\_dim=20, batch\_size=128, epochs=20):

# Dataset and DataLoader

dataset = MatrixDataset(df, n\_nodes)

dataloader = DataLoader(dataset, batch\_size=batch\_size, shuffle=True, num\_workers=4, pin\_memory=True)

# Model, Optimizer, Loss

model = MatrixEmbedder(n\_nodes=n\_nodes, embed\_dim=embed\_dim).cuda()

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)

scaler = GradScaler() # For Mixed Precision

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

for epoch in range(epochs):

model.train()

total\_loss = 0

for batch in dataloader:

batch = batch.cuda()

optimizer.zero\_grad()

# Generate random pairs for contrastive loss

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

batch\_pairs = batch[idx]

with autocast():

embeddings1 = model(batch)

embeddings2 = model(batch\_pairs)

labels = torch.randint(0, 2, (batch.size(0),)).float().cuda() # Random 0/1 labels

loss = criterion(embeddings1, embeddings2, labels)

scaler.scale(loss).backward()

scaler.step(optimizer)

scaler.update()

total\_loss += loss.item()

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=32, epochs=50)

import numpy as np

import pandas as pd

import torch

import torch.nn as nn

import torch.nn.functional as F

from torch.utils.data import Dataset, DataLoader

from torch.optim import Adam

import gc

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

# MatrixDataset Definition

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

class MatrixDataset(Dataset):

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

super().\_\_init\_\_()

self.df = df

self.n\_nodes = n\_nodes

def get\_adj(self, path):

# Initialize the adjacency matrix

adj = np.zeros((self.n\_nodes, self.n\_nodes), dtype=np.float32)

# Populate adjacency matrix based on path

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

# Fill adjacency based on the path

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

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

# Normalize adjacency matrix safely

col\_sums = adj.sum(axis=0)

col\_sums[col\_sums == 0] = 1e-6 # Avoid division by zero

adj = adj / col\_sums

# Replace NaNs or infinite values

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

return adj

def get\_path(self, index):

return self.df.at[index, 'path']

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

path = self.get\_path(index)

adj = self.get\_adj(path)

return torch.tensor(adj, dtype=torch.float32)

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

return len(self.df)

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

# MatrixEmbedder Definition

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

class MatrixEmbedder(nn.Module):

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

super().\_\_init\_\_()

self.layer1 = nn.Linear(n\_nodes \*\* 2, 500, bias=True)

self.layer2 = nn.Linear(500, 250, bias=True)

self.layer3 = nn.Linear(250, 100, bias=True)

self.layer4 = nn.Linear(100, 50, bias=True)

self.layer5 = nn.Linear(50, embed\_dim, bias=True)

def forward(self, x):

x = x.flatten(start\_dim=1, end\_dim=-1)

x = F.tanh(self.layer1(x))

x = F.tanh(self.layer2(x))

x = F.tanh(self.layer3(x))

x = F.tanh(self.layer4(x))

x = F.tanh(self.layer5(x)) # Activation function isn't necessary here but helps

return x

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

# Distance Functions

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

def dme(mat1, mat2, vector):

"""

Double matrix expectation relative to a vector.

i.e., vector^T mat1^T mat2 vector

"""

mat = np.matmul(mat1.T, mat2)

v = np.matmul(mat, vector)

out = np.matmul(vector.T, v)

return out[0, 0]

def markov\_distance(mata, matb):

v1 = np.ones((mata.shape[0], 1))

out = dme(mata, matb, v1) - 0.5 \* dme(mata, mata, v1) - 0.5 \* dme(matb, matb, v1)

return out

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

# Data Preparation

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

# Example DataFrame

mapped\_journeys\_df = pd.DataFrame({

'channel\_visit\_id': [1, 2, 3],

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

})

# Hyperparameters

n\_nodes = mapped\_journeys\_df['path'].apply(max).max() + 2

batch\_size = 16 # Reduced batch size to minimize memory usage

embed\_dim = 20

# Dataset and Dataloader

dataset = MatrixDataset(df=mapped\_journeys\_df, n\_nodes=n\_nodes)

dataloader = DataLoader(dataset, batch\_size=batch\_size)

# Embedder Model

embedder = MatrixEmbedder(n\_nodes=n\_nodes, embed\_dim=embed\_dim)

# Optimizer

optim = Adam(embedder.parameters())

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

# Training Loop

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

n\_per\_example = 2

epochs = 3

losses = []

for ep in range(epochs):

print(f'Epoch = {ep+1}/{epochs}')

for ib, batch in enumerate(dataloader):

optim.zero\_grad()

# Pair up the paths into n\_per\_example \* batch\_size pairs

right = np.random.randint(0, batch.shape[0], n\_per\_example \* batch.shape[0])

left = np.repeat(np.arange(0, batch.shape[0]), n\_per\_example)

index\_pairs = np.stack([left, right]).T

# Calculate Markov distances with no\_grad to save memory

distances = []

with torch.no\_grad():

for (l, r) in index\_pairs:

d = markov\_distance(batch[l].cpu().numpy(), batch[r].cpu().numpy())

distances.append(d)

distances = torch.tensor(distances, dtype=torch.float32)

# Get neural network output vectors

left\_vecs = embedder(batch[left])

right\_vecs = embedder(batch[right])

diff = left\_vecs - right\_vecs

# Calculate loss

loss = torch.pow(((diff \*\* 2).sum(axis=1) - distances \*\* 2), 2).sum()

loss.backward()

optim.step()

# Clear memory after each batch

del batch, left\_vecs, right\_vecs, diff, distances

gc.collect()

torch.cuda.empty\_cache()

losses.append(loss.item())

print(f"Epoch {ep + 1}/{epochs}, Loss: {loss.item()}")

print("Training completed successfully!")