import pandas as pd

# Assuming mapped\_journeys\_df is your existing DataFrame with 'channel\_visit\_id' and 'path'

# Step 1: Function to remove consecutive duplicates from the 'path'

def remove\_consecutive\_duplicates(path):

"""

Removes consecutive duplicate entries from a list.

"""

return [v for i, v in enumerate(path) if i == 0 or v != path[i - 1]]

# Step 2: Apply the function to clean the 'path' column

mapped\_journeys\_df['cleaned\_path'] = mapped\_journeys\_df['path'].apply(remove\_consecutive\_duplicates)

# Step 3: Create a new DataFrame with 'channel\_visit\_id' and 'cleaned\_path'

cleaned\_journeys\_df = mapped\_journeys\_df[['channel\_visit\_id', 'cleaned\_path']]

# Step 4: Rename 'cleaned\_path' to 'path'

cleaned\_journeys\_df.rename(columns={'cleaned\_path': 'path'}, inplace=True)

# Step 5: Display the cleaned DataFrame

print("Cleaned DataFrame:")

print(cleaned\_journeys\_df.head())

# Step 6: (Optional) Check for any empty paths after cleaning

empty\_paths = cleaned\_journeys\_df[cleaned\_journeys\_df['path'].apply(len) == 0]

if not empty\_paths.empty:

print("\nWarning: Some journeys have empty paths after cleaning:")

print(empty\_paths)

else:

print("\nAll journeys have valid paths after cleaning.")

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

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

# 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))

# 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 = 128

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}')

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

distances = []

for (l, r) in index\_pairs:

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

distances.append(d)

distances = torch.tensor(distances)

# 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()

losses.append(loss.item())

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

print("Training completed successfully!")