import numpy as np

import torch

from torch.optim import Adam

from tqdm import tqdm

# Optimizer

optim = Adam(embedder.parameters())

# Hyperparameters

n\_per\_example = 2

epochs = 3

losses = []

distances\_captured = []

# Function to normalize adjacency matrix safely

def normalize\_adjacency\_matrix(adj):

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

# Debugging check for zero columns

if np.any(col\_sums == 0):

print("Warning: Some columns in the adjacency matrix sum to zero.")

# Apply epsilon to avoid division by zero

adj = np.divide(adj, col\_sums + 1e-8)

# Replace NaN or Inf with 0

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

return adj

# Training Loop

for ep in range(epochs):

print(f"Epoch = {ep}")

for ib, batch in tqdm(enumerate(dataloader), total=len(dataset)//batch\_size):

optim.zero\_grad()

# Normalize adjacency matrix

adj = normalize\_adjacency\_matrix(batch)

# 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 the Markov distances

distances = []

for i, (l, r) in enumerate(index\_pairs):

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

distances.append(d)

distances\_captured.append(d)

distances = torch.tensor(distances)

# Get the neural network output vectors

left\_vecs = embedder(batch[left].to(torch.float32))

right\_vecs = embedder(batch[right].to(torch.float32))

diff = (left\_vecs - right\_vecs)

# Calculate loss

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

# Backpropagation

loss.backward()

optim.step()

# Store the loss

losses.append(loss.item())

print("Training complete. Losses:", losses)