**PYTHON CODE**

import torch

import torch.nn as nn

import torch.nn.functional as F

from torch.optim import Adam

from torch\_geometric.nn import GATConv

from torch\_geometric.data import Data

import numpy as np

from transformers import AutoTokenizer, AutoModel

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import roc\_auc\_score, f1\_score

import networkx as nx

from copy import deepcopy

from tqdm import tqdm

# Configuration / Hyperparams

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

BERT\_MODEL = "sentence-transformers/all-MiniLM-L6-v2" # light BERT

HIDDEN\_DIM = 128

GAT\_HEADS = 4

LEARNING\_RATE = 1e-3

EPOCHS = 40

BATCH\_SIZE = 1 # if doing full-graph temporal snapshots, batch by time-step

CAUSAL\_LAMBDA = 0.5

# Utility: Text encoder (BERT)

class TextEncoder:

def \_\_init\_\_(self, model\_name=BERT\_MODEL, device=DEVICE):

self.tokenizer = AutoTokenizer.from\_pretrained(model\_name)

self.model = AutoModel.from\_pretrained(model\_name).to(device)

self.device = device

@torch.no\_grad()

def encode(self, texts):

# returns (n\_texts, emb\_dim)

# Using mean pooling over token embeddings

encoded = self.tokenizer(texts, padding=True, truncation=True, return\_tensors='pt').to(self.device)

out = self.model(\*\*encoded, output\_hidden\_states=True, return\_dict=True)

last\_hidden = out.last\_hidden\_state # (B, T, D)

# mean pooling (ignore attention mask)

mask = encoded['attention\_mask'].unsqueeze(-1)

summed = (last\_hidden \* mask).sum(dim=1)

counts = mask.sum(dim=1).clamp(min=1)

pooled = (summed / counts).cpu()

return pooled # CPU tensor

# Data loader stubs (replace with real loader)

# Expected: time-ordered snapshots. For each snapshot t:

# - node\_features: numpy array (N\_t, feat\_dim)

# - edge\_index: numpy array (2, E\_t)

# - edge\_weight (optional): numpy array (E\_t,)

# - node\_ids mapping, node labels (cascade participation / stance), etc.

def load\_snapshots\_stub():

Replace this stub: return a list of snapshot dicts for t=1..T

Example snapshot dict:

{

"node\_ids": [uid1, uid2, ...],

"node\_features": np.array shape (N, F),

"edge\_index": np.array shape (2, E),

"edge\_weight": np.array shape (E,) optional,

"stance": np.array shape (N,), values in [-1,1],

"cascade\_label": np.array shape (N,) binary label whether node participated in cascade soon after this snapshot

}

snapshots = []

# --- small synthetic example for testing ---

for t in range(5):

N = 50

feat\_dim = 64

node\_ids = np.arange(N)

node\_features = np.random.randn(N, feat\_dim).astype(np.float32)

# random graph

G = nx.erdos\_renyi\_graph(N, p=0.05, seed=t)

edges = np.array(list(G.edges)).T

if edges.size == 0:

edges = np.zeros((2,1), dtype=int)

# stance random

stance = np.random.uniform(-1,1,size=(N,))

cascade\_label = (np.random.rand(N) < 0.1).astype(int)

snapshots.append({

"node\_ids": node\_ids,

"node\_features": node\_features,

"edge\_index": edges,

"edge\_weight": None,

"stance": stance,

"cascade\_label": cascade\_label

})

return snapshots

# Model: Temporal Graph Encoder (stacked GAT per snapshot, with recurrent state)

class TemporalGATEncoder(nn.Module):

def \_\_init\_\_(self, in\_dim, hidden\_dim=HIDDEN\_DIM, heads=GAT\_HEADS):

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

self.gat1 = GATConv(in\_dim, hidden\_dim // heads, heads=heads, concat=True)

self.gat2 = GATConv(hidden\_dim, hidden\_dim // heads, heads=heads, concat=True)

self.gru = nn.GRU(hidden\_dim, hidden\_dim, batch\_first=True)

self.dropout = nn.Dropout(0.2)

def forward(self, x, edge\_index, prev\_state=None):

x: [N, F]

edge\_index: [2, E]

prev\_state: [1, N, H] or None

returns: h: [N, H], new\_state: [1, N, H]

h = F.elu(self.gat1(x, edge\_index))

h = self.dropout(h)

h = F.elu(self.gat2(h, edge\_index))

# GRU expects (batch, seq, feat) - we do single step with nodes as batch

h\_unsq = h.unsqueeze(1) # [N, 1, H]

if prev\_state is None:

out, new\_state = self.gru(h\_unsq) # out [N,1,H]

else:

# prev\_state shape [1, N, H] -> need to permute to match batch dims of GRU

# we will feed prev\_state as initial hidden in GRU (works)

out, new\_state = self.gru(h\_unsq, prev\_state)

return out.squeeze(1), new\_state # [N,H], [1, N, H]

# Causal Estimator (propensity scoring)

class PropensityEstimator:

Fits propensity model (logistic regression) to estimate treatment probability.

For simplicity we treat 'treatment' as binary feature on nodes (e.g., high sentiment)

"""

def \_\_init\_\_(self):

self.model = LogisticRegression(max\_iter=200)

def fit(self, X, treat):

# X: (N, D) numpy, treat: (N,) binary

self.model.fit(X, treat)

def propensity(self, X):

return self.model.predict\_proba(X)[:,1]

# C-GNN full model wrapper (temporal encoder + causal layer + prediction head)

class CausalGNN(nn.Module):

def \_\_init\_\_(self, in\_dim, hidden\_dim=HIDDEN\_DIM):

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

self.encoder = TemporalGATEncoder(in\_dim, hidden\_dim)

self.pred\_head = nn.Sequential(

nn.Linear(hidden\_dim, hidden\_dim//2),

nn.ReLU(),

nn.Linear(hidden\_dim//2, 1)

)

def forward(self, x, edge\_index, prev\_state=None):

# x: torch.tensor (N, F)

h, new\_state = self.encoder(x, edge\_index, prev\_state)

logits = self.pred\_head(h).squeeze(-1) # (N,)

prob = torch.sigmoid(logits)

return prob, logits, new\_state, h

# Loss functions

def weighted\_bce\_loss(logits, labels, weights=None):

bce = F.binary\_cross\_entropy\_with\_logits(logits, labels.float(), reduction='none')

if weights is not None:

bce = bce \* weights

return bce.mean()

# causal regularization term (encourages small correlation between confounders & predictions)

def causal\_regularizer(preds, confounders):

# preds: (N,) tensor probabilities

# confounders: (N, K) tensor numeric

# simple penalty: covariance between preds and confounders

preds\_centered = preds - preds.mean()

conf\_centered = confounders - confounders.mean(dim=0, keepdim=True)

cov = torch.abs((preds\_centered.unsqueeze(1) \* conf\_centered).mean(dim=0)).sum()

return cov

# Training & Evaluation pipeline

def train\_cgnn(snapshots, feature\_dim, num\_epochs=EPOCHS):

model = CausalGNN(in\_dim=feature\_dim).to(DEVICE)

optimizer = Adam(model.parameters(), lr=LEARNING\_RATE)

# We'll use propensity estimator externally (scikit)

propensity\_est = PropensityEstimator()

# Precompute a "treatment" column for simple demo: treat if stance magnitude > 0.6

# and assemble one big training set for propensity fitting (could be per-snapshot in practice)

X\_for\_prop = []

T\_for\_prop = []

for snap in snapshots:

meta = snap["node\_features"] # (N,F)

# build simple treatment indicator from stance

treat = (np.abs(snap["stance"]) > 0.6).astype(int)

X\_for\_prop.append(meta)

T\_for\_prop.append(treat)

X\_for\_prop = np.vstack(X\_for\_prop)

T\_for\_prop = np.concatenate(T\_for\_prop)

propensity\_est.fit(X\_for\_prop, T\_for\_prop)

prev\_state = None

for epoch in range(num\_epochs):

model.train()

total\_loss = 0.0

total\_auc = []

for snap in snapshots:

node\_features = torch.tensor(snap["node\_features"], dtype=torch.float32).to(DEVICE)

edge\_index = torch.tensor(snap["edge\_index"], dtype=torch.long).to(DEVICE)

labels = torch.tensor(snap["cascade\_label"], dtype=torch.float32).to(DEVICE)

# get propensity weights for causal regularization / weighting

prop = propensity\_est.propensity(snap["node\_features"])

weights = torch.tensor((T\_for\_prop.mean() / (prop + 1e-6)), dtype=torch.float32).to(DEVICE)

# forward

prob, logits, new\_state, hidden = model(node\_features, edge\_index, prev\_state)

# compute losses

loss\_pred = weighted\_bce\_loss(logits, labels, weights=None) # main predictive loss

# causal reg using confounders (stance + degree)

confounders = torch.tensor(np.stack([snap["stance"],

np.clip(np.array([np.sum(edge\_index.cpu().numpy()[1]==i) for i in range(node\_features.shape[0])]), 0, 100)], axis=1), dtype=torch.float32).to(DEVICE)

loss\_causal = causal\_regularizer(prob, confounders)

loss = loss\_pred + CAUSAL\_LAMBDA \* loss\_causal

optimizer.zero\_grad()

loss.backward()

optimizer.step()

prev\_state = new\_state.detach()

total\_loss += loss.item()

try:

auc = roc\_auc\_score(labels.cpu().numpy(), prob.detach().cpu().numpy())

total\_auc.append(auc)

except Exception:

pass

avg\_auc = np.mean(total\_auc) if total\_auc else 0.0

print(f"Epoch {epoch+1}/{num\_epochs} - Loss: {total\_loss/len(snapshots):.4f} - AUC: {avg\_auc:.4f}")

return model, propensity\_est

# Intervention simulation utilities

def rewire\_cross\_group(snap, fraction=0.05, stance\_threshold=0.0):

"""

Rewire `fraction` of edges to connect nodes across stance groups (promote cross exposure).

snap: snapshot dict; returns a new snapshot dict with modified edge\_index

"""

new\_snap = deepcopy(snap)

edge\_index = snap["edge\_index"].copy()

N = len(snap["node\_ids"])

# identify group members

left = np.where(snap["stance"] <= stance\_threshold)[0]

right = np.where(snap["stance"] > stance\_threshold)[0]

# edges to rewire

E = edge\_index.shape[1]

k = max(1, int(E \* fraction))

idxs = np.random.choice(np.arange(E), size=k, replace=False)

for idx in idxs:

# create cross edge

if np.random.rand() < 0.5 and left.size and right.size:

u = np.random.choice(left)

v = np.random.choice(right)

else:

u = np.random.choice(right)

v = np.random.choice(left)

edge\_index[:, idx] = [u, v]

new\_snap["edge\_index"] = edge\_index

return new\_snap

def rerank\_content\_scores(snap, moderation\_mask=None, boost\_neutral=True):

"""

Simulate re-ranking by altering node features or visibility score.

moderation\_mask: boolean array of nodes to reduce amplification

"""

new\_snap = deepcopy(snap)

# Example: adjust node\_features first column as "visibility" proxy

vis = new\_snap["node\_features"][:, 0]

if moderation\_mask is None:

moderation\_mask = np.abs(new\_snap["stance"]) > 0.7

# reduce visibility for extreme nodes

vis[moderation\_mask] \*= 0.7

# optionally boost neutral

if boost\_neutral:

neutral\_mask = np.abs(new\_snap["stance"]) <= 0.3

vis[neutral\_mask] \*= 1.2

new\_snap["node\_features"][:, 0] = vis

return new\_snap

# Example: Running pipeline on stub snapshots

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

snapshots = load\_snapshots\_stub()

feature\_dim = snapshots[0]["node\_features"].shape[1]

print("Training C-GNN on snapshots ...")

model, propensity\_est = train\_cgnn(snapshots, feature\_dim, num\_epochs=10)

# Evaluate baseline PI (a simple modularity-based proxy)

def polarization\_index(snapshot):

# simplistic: compute modularity between two groups from stance sign

node\_stance = snapshot["stance"]

G = nx.Graph()

N = len(node\_stance)

G.add\_nodes\_from(range(N))

for u,v in snapshot["edge\_index"].T:

G.add\_edge(int(u), int(v))

# assign communities

comm = {i: 0 if node\_stance[i] <= 0 else 1 for i in range(N)}

# modularity (networkx)

try:

from networkx.algorithms.community.quality import modularity

communities = [ [i for i in range(N) if comm[i]==0], [i for i in range(N) if comm[i]==1] ]

mod = modularity(G, communities)

return mod

except Exception:

return 0.0

before\_PI = np.mean([polarization\_index(s) for s in snapshots])

print("Baseline avg polarization index:", before\_PI)

# simulate intervention: rewire 5% edges in each snapshot

sim\_snaps = [rewire\_cross\_group(s, fraction=0.05) for s in snapshots]

after\_PI = np.mean([polarization\_index(s) for s in sim\_snaps])

print("Post-intervention avg polarization index:", after\_PI)

print("Delta PI:", before\_PI - after\_PI)