**PYTHON CODE**

import os

import json

import math

import random

from copy import deepcopy

from typing import List, Dict, Tuple

import numpy as np

import pandas as pd

import networkx as nx

import matplotlib.pyplot as plt

import seaborn as sns

from tqdm import tqdm

# NLP

from sentence\_transformers import SentenceTransformer

from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer

# Torch & PyG

import torch

import torch.nn as nn

import torch.nn.functional as F

from torch.optim import Adam

USE\_PYG = True

try:

if USE\_PYG:

from torch\_geometric.data import Data

from torch\_geometric.nn import GCNConv

except Exception as e:

print("PyG import failed or disabled. Will use fallback GCN.")

USE\_PYG = False

# sklearn (causal propensities & metrics)

from sklearn.linear\_model import LogisticRegression

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

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

from sklearn.preprocessing import StandardScaler

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

# Config

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

SEED = 42

random.seed(SEED)

np.random.seed(SEED)

torch.manual\_seed(SEED)

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

EMBED\_MODEL = "all-MiniLM-L6-v2" # sentence-transformers model

EMBED\_DIM = 384 # correspondent to the model above

HIDDEN\_DIM = 128

LR = 1e-3

EPOCHS = 40

BATCH\_BY\_SNAPSHOT = True # we process full graph snapshots, not mini-batches

CAUSAL\_LAMBDA = 0.3 # weight for causal regularization

MODEL\_SAVE\_PATH = "cgnn\_model.pth"

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

# Utilities & Data Loading

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

def load\_data\_stub(num\_snapshots=6, nodes\_per\_snapshot=200):

"""

Synthetic data generator for testing and demo.

Real usage: replace with loader that returns a list of snapshot dicts.

Each snapshot dict contains:

- node\_ids: np.array (N,)

- texts: list[str] length N

- edge\_list: list[(u,v)]

- features\_meta: np.array (N, K) (e.g., follower\_count, activity)

- stance: np.array (N,) float in [-1,1] (proxy ideological position)

- label\_cascade: np.array (N,) binary - whether node participates in cascade soon after snapshot

"""

snapshots = []

for t in range(num\_snapshots):

N = nodes\_per\_snapshot

node\_ids = np.arange(N) + t \* 10000 # global unique

# create random texts (short); in real data, these are tweets/comments

texts = [f"Sample post about topic {random.choice(['A','B','C'])} #{random.randint(0,100)}" for \_ in range(N)]

# random graph: mix of communities

p = 0.015 # connectivity

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

# Ensure connected-ish

if not nx.is\_connected(G) and len(G) > 0:

# keep largest component

Gc = max(nx.connected\_components(G), key=len)

G = G.subgraph(Gc).copy()

edge\_list = list(G.edges())

# meta features: followers, activity

followers = np.random.poisson(100, N).astype(float)

activity = np.random.poisson(5, N).astype(float)

features\_meta = np.stack([followers, activity], axis=1)

# stance: mixture of two Gaussians

stance = np.concatenate([np.random.normal(-0.6, 0.18, N//2), np.random.normal(0.6, 0.18, N - N//2)])

stance = np.clip(stance, -1.0, 1.0)

# label cascade: nodes with extreme stance + high degree more likely

deg = np.array([G.degree(i) if i in G else 0 for i in range(N)])

prob = 0.1 + 0.3 \* (np.abs(stance) > 0.7).astype(float) + 0.05 \* (deg > np.percentile(deg,

75)).astype(float)

label = (np.random.rand(N) < prob).astype(int)

snapshots.append({

"node\_ids": node\_ids,

"texts": texts,

"edge\_list": edge\_list,

"features\_meta": features\_meta,

"stance": stance,

"label\_cascade": label

})

return snapshots

# Replace this loader with actual CSV/JSON/Pushshift/Twitter ingestion function

def load\_real\_data\_from\_csv(csv\_path: str, time\_col: str = "timestamp", user\_col: str = "user", text\_col: str = "text"):

"""

Example loader if you have a CSV of posts with timestamps and user IDs.

The output format should match load\_data\_stub's snapshot dicts.

This function is a guideline and will require adaptation.

"""

df = pd.read\_csv(csv\_path, parse\_dates=[time\_col])

# Decide snapshot window (e.g., daily/weekly). Here: weekly.

df['week'] = df[time\_col].dt.isocalendar().week

snapshots = []

for wk, g in df.groupby('week'):

user\_map = {u: i for i, u in enumerate(g[user\_col].unique())}

N = len(user\_map)

node\_ids = np.array(list(user\_map.values()))

texts = [''] \* N

for u, grp in g.groupby(user\_col):

idx = user\_map[u]

texts[idx] = ' '.join(grp[text\_col].astype(str).tolist())[:512]

# build edge\_list using replies/mentions/retweets columns if available

# placeholder empty edges

edge\_list = []

features\_meta = np.zeros((N, 2))

stance = np.zeros(N)

label = np.zeros(N, dtype=int)

snapshots.append({"node\_ids": node\_ids, "texts": texts, "edge\_list": edge\_list,

"features\_meta": features\_meta, "stance": stance, "label\_cascade": label})

return snapshots

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

# Text Embeddings & Sentiment

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

class NLPEncoder:

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

print("Loading sentence-transformer model:", model\_name)

self.model = SentenceTransformer(model\_name)

self.analyzer = SentimentIntensityAnalyzer()

def embed\_texts(self, texts: List[str]) -> np.ndarray:

# returns numpy array (N, EMBED\_DIM)

embeddings = self.model.encode(texts, show\_progress\_bar=False, convert\_to\_numpy=True)

return embeddings

def sentiment\_scores(self, texts: List[str]) -> np.ndarray:

scores = []

for t in texts:

v = self.analyzer.polarity\_scores(t)

scores.append(v['compound']) # -1..1

return np.array(scores, dtype=float)

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

# Graph & Snapshot Processing

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

def build\_graph\_from\_snapshot(snap: dict) -> Tuple[nx.Graph, np.ndarray]:

"""

Builds an undirected NetworkX graph for the snapshot and returns adjacency and node order mapping.

"""

texts = snap["texts"]

N = len(texts)

G = nx.Graph()

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

for u, v in snap["edge\_list"]:

if u < N and v < N:

G.add\_edge(u, v)

return G

def snapshot\_to\_pyg\_data(snap: dict, text\_embeds: np.ndarray, sentiment: np.ndarray) -> 'torch\_geometric.data.Data':

"""

Convert snapshot to PyG Data object (if PyG available).

Node features will be: [text\_embed\_mean\_reduce (maybe PCA), meta features, sentiment, stance]

"""

N = len(snap["texts"])

meta = snap["features\_meta"] if snap.get("features\_meta") is not None else np.zeros((N,2))

stance = snap["stance"].reshape(-1,1)

# combine

node\_feat = np.concatenate([text\_embeds, meta, stance, sentiment.reshape(-1,1)], axis=1)

# edge\_index

if len(snap["edge\_list"]) == 0:

edge\_index = np.zeros((2,0), dtype=int)

else:

edges = np.array(snap["edge\_list"]).T

edge\_index = edges.astype(int)

# label

label = snap["label\_cascade"]

# build PyG Data

if USE\_PYG:

x = torch.tensor(node\_feat, dtype=torch.float)

y = torch.tensor(label, dtype=torch.long)

if edge\_index.size == 0:

edge\_index\_t = torch.empty((2,0), dtype=torch.long)

else:

edge\_index\_t = torch.tensor(edge\_index, dtype=torch.long)

data = Data(x=x, edge\_index=edge\_index\_t, y=y)

return data

else:

return {"x": node\_feat, "edge\_index": edge\_index, "y": label}

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

# Model: Temporal GCN Encoder + Prediction

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

class TemporalGCNEncoder(nn.Module):

"""

Basic temporal encoder: run a GCN per snapshot, then feed node embeddings to a GRU to maintain temporal state.

This is not an event-based TGNN but is sufficient for snapshot-based temporal modeling.

"""

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

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

if USE\_PYG:

self.conv1 = GCNConv(in\_dim, hidden\_dim)

self.conv2 = GCNConv(hidden\_dim, hidden\_dim)

else:

# fallback: simple linear layers if PyG not installed

self.lin1 = nn.Linear(in\_dim, hidden\_dim)

self.lin2 = nn.Linear(hidden\_dim, hidden\_dim)

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

self.hidden\_dim = hidden\_dim

def forward\_one(self, x, edge\_index):

# x: (N, F) tensor

if USE\_PYG:

h = F.relu(self.conv1(x, edge\_index))

h = F.relu(self.conv2(h, edge\_index))

else:

h = F.relu(self.lin1(x))

h = F.relu(self.lin2(h))

return h

def forward(self, sequence\_of\_data: List['Data']):

"""

sequence\_of\_data: list of PyG Data objects (snapshots in temporal order)

We'll process each snapshot, collect h\_t (N\_t, H) and feed into GRU across time

For simplicity, we assume node correspondence across snapshots (same ordering / indexed users).

"""

# This implementation handles same node order across snapshots; for general case you'd need alignment/inductive approach.

embeddings = []

for data in sequence\_of\_data:

x = data.x.to(DEVICE)

if USE\_PYG:

edge\_index = data.edge\_index.to(DEVICE)

else:

edge\_index = None

h = self.forward\_one(x, edge\_index) # (N, H)

embeddings.append(h.unsqueeze(1)) # (N, 1, H)

# stack: (N, T, H)

H\_stack = torch.cat(embeddings, dim=1)

# GRU expects (batch, seq, feat) -> we treat nodes as batch, time as seq

out, h\_n = self.gru(H\_stack) # out: (N, T, H), h\_n: (1, N, H)

# return last timestep embeddings: out[:, -1, :]

return out[:, -1, :], h\_n # node embeddings at last time

class CausalGNNModel(nn.Module):

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

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

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

self.pred\_head = nn.Sequential(

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

nn.ReLU(),

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

)

# causal weight vector (learnable) for direct effect scoring (small explainability module)

self.causal\_weights = nn.Parameter(torch.randn(in\_dim) \* 0.01)

def forward(self, seq\_data: List['Data']):

# seq\_data: list of Data objects (temporal sequence with same N)

node\_emb, \_ = self.encoder(seq\_data) # (N, H)

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

probs = torch.sigmoid(logits)

return probs, logits, node\_emb

def causal\_score(self, node\_features: torch.Tensor) -> torch.Tensor:

"""

Produces a per-node causal "score" as linear combination of features via causal\_weights.

This is simplistic: for rigorous causal ATE, use DoWhy/EconML.

"""

return node\_features.to(DEVICE) @ self.causal\_weights

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

# Causal Tools (Propensity / Weighted Loss / ATE)

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

class PropensityModel:

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

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

def fit(self, X: np.ndarray, treat: np.ndarray):

# X: (M, D), treat: (M,) binary

scaler = StandardScaler()

Xs = scaler.fit\_transform(X)

self.scaler = scaler

self.model.fit(Xs, treat)

return self

def predict\_proba(self, X: np.ndarray) -> np.ndarray:

Xs = self.scaler.transform(X)

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

def weighted\_bce\_with\_logits(logits, targets, weights=None):

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

if weights is not None:

loss = loss \* weights

return loss.mean()

def compute\_ATE\_effects(propen\_scores: np.ndarray, outcomes: np.ndarray, treatment: np.ndarray):

"""

Very small ATE estimate (IPW-style): E[Y|T=1] - E[Y|T=0] weighted by propensity.

treatment: binary vector indicating 'treated' nodes (e.g., extreme stance)

"""

eps = 1e-6

w\_t = treatment / (propen\_scores + eps)

w\_c = (1 - treatment) / (1 - propen\_scores + eps)

ate = np.mean(w\_t \* outcomes) - np.mean(w\_c \* outcomes)

return ate

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

# Training loop & evaluation

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

def train\_pipeline(snapshots: List[dict], epochs=EPOCHS):

"""

snapshots: list of snapshot dicts in chronological order

We'll create rolling windows of length T\_window and train the model to predict cascade at t+1

For simplicity we use windows of length = 3 snapshots (configurable).

"""

n\_snap = len(snapshots)

window = 3

# NLP encoder

nlp = NLPEncoder()

# Precompute embeddings & sentiment for each snapshot

for snap in snapshots:

snap['text\_embeds'] = nlp.embed\_texts(snap['texts'])

snap['sentiment'] = nlp.sentiment\_scores(snap['texts'])

# update features\_meta shape if needed

if snap.get('features\_meta') is None:

N = len(snap['texts'])

snap['features\_meta'] = np.zeros((N,2))

# Construct train examples (each example uses snapshots t..t+window-1 to predict label at t+window)

examples = []

for t in range(n\_snap - window):

seq = snapshots[t:t+window] # list of snapshot dicts

target\_snap = snapshots[t+window]

# For simplicity we require same N across snapshots (or align users), here using stub data this holds

examples.append((seq, target\_snap))

print(f"Created {len(examples)} temporal examples (window={window})")

if len(examples) == 0:

raise ValueError("Not enough snapshots to create training examples. Increase snapshots or reduce window.")

# Build full dataset with PyG Data sequences

seq\_data\_list = []

y\_list = []

node\_feat\_for\_propensity = []

treat\_labels = []

for seq, target in examples:

# convert each snapshot into PyG Data

pyg\_seq = []

for s in seq:

data = snapshot\_to\_pyg\_data(s, s['text\_embeds'], s['sentiment'])

pyg\_seq.append(data)

# target labels (next snapshot's cascade label)

y = target['label\_cascade'] # numpy array (N,)

y\_list.append(y)

seq\_data\_list.append(pyg\_seq)

# For propensity model we will use features from last snapshot in seq

last = seq[-1]

# define 'treatment' as extreme stance abs > 0.65

treat = (np.abs(last['stance']) > 0.65).astype(int)

treat\_labels.append(treat)

# features for propensity: stance + meta features + sentiment

X\_prop = np.concatenate([last['features\_meta'], last['stance'].reshape(-1,1), last['sentiment'].reshape(-1,1)], axis=1)

node\_feat\_for\_propensity.append(X\_prop)

# Flatten for propensity training (many nodes across many examples)

X\_prop\_all = np.vstack(node\_feat\_for\_propensity)

T\_prop\_all = np.concatenate(treat\_labels)

# Fit propensity model

propmod = PropensityModel()

propmod.fit(X\_prop\_all, T\_prop\_all)

print("Fitted propensity model (logistic) for treatment balancing.")

# Create model

# infer node feature dim from data

sample\_data = seq\_data\_list[0][0] # snapshot 0 of first example

in\_dim = sample\_data.x.shape[1] if USE\_PYG else sample\_data['x'].shape[1]

model = CausalGNNModel(in\_dim=in\_dim).to(DEVICE)

optimizer = Adam(model.parameters(), lr=LR)

# training loop (model sees each example, computing per-node loss)

for epoch in range(epochs):

model.train()

total\_loss = 0.0

all\_preds = []

all\_targets = []

for i, (seq\_pygs, target) in enumerate(zip(seq\_data\_list, y\_list)):

# Move seq data to device

seq\_dev = []

for d in seq\_pygs:

if USE\_PYG:

d = deepcopy(d)

d.x = d.x.to(DEVICE)

d.edge\_index = d.edge\_index.to(DEVICE)

else:

d = {'x': torch.tensor(d['x'], dtype=torch.float).to(DEVICE),

'edge\_index': torch.tensor(d['edge\_index'], dtype=torch.long).to(DEVICE)}

seq\_dev.append(d)

# forward

probs, logits, node\_emb = model(seq\_dev)

# target labels for nodes

labels = torch.tensor(target, dtype=torch.float32).to(DEVICE)

# compute propensity weights per node using last snapshot features

last = seq\_dev[-1]

# get features for propensity X (numpy)

if USE\_PYG:

# get original last snapshot's stacked features from snapshots list

last\_snap\_index = i # mapping aligned earlier

# For simplicity, recompute propensities using node features from snapshots

# (we have X\_prop\_all and propmod; compute per-node propensity using last snapshot in this example)

X\_local = node\_feat\_for\_propensity[i] # note: this indexing matches earlier assembly

p\_scores = propmod.predict\_proba(X\_local)

# create weights for IPW

eps = 1e-6

treat\_local = treat\_labels[i]

weights = (treat\_local / (p\_scores + eps)) + ((1 - treat\_local) / (1 - p\_scores + eps))

weights\_t = torch.tensor(weights, dtype=torch.float32).to(DEVICE)

# losses: predictive + causal regularization (encourage lower covariance with confounders)

loss\_pred = weighted\_bce\_with\_logits(logits, labels, weights=None)

# causal reg: covariance between model probs and confounders (stance, degree)

# compute confounders: stance + degree

# degree from last snapshot's edge\_index

last\_snap = snapshots[i + (len(seq\_pygs) - 1)]

deg = np.array([0]\*last\_snap['node\_ids'].shape[0])

try:

Gtmp = build\_graph\_from\_snapshot(last\_snap)

deg = np.array([Gtmp.degree(u) if u in Gtmp.nodes else 0 for u in range(len(last\_snap['node\_ids']))])

except Exception:

deg = np.zeros(len(last\_snap['node\_ids']))

conf = np.stack([last\_snap['stance'], deg], axis=1)

conf\_t = torch.tensor(conf, dtype=torch.float32).to(DEVICE)

# causal regularizer: absolute covariance sum

probs\_centered = probs - probs.mean()

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

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

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

optimizer.zero\_grad()

loss.backward()

optimizer.step()

total\_loss += loss.item()

preds\_np = (probs.detach().cpu().numpy() > 0.5).astype(int)

all\_preds.extend(preds\_np.tolist())

all\_targets.extend(target.tolist())

# epoch metrics

if len(all\_targets) > 0:

f1 = f1\_score(all\_targets, all\_preds)

auc = None

try:

auc = roc\_auc\_score(all\_targets, all\_preds)

except Exception:

auc = None

else:

f1 = 0.0

auc = None

print(f"Epoch [{epoch+1}/{epochs}] Loss: {total\_loss/len(seq\_data\_list):.4f} F1: {f1:.4f} AUC(binPred): {auc}")

# save model

torch.save({'model\_state': model.state\_dict(), 'propensity\_model': propmod.model, 'propensity\_scaler': propmod.scaler}, MODEL\_SAVE\_PATH)

print("Saved model to", MODEL\_SAVE\_PATH)

return model, propmod, seq\_data\_list, y\_list

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

# Intervention simulation

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

def simulate\_rewire(snapshot: dict, fraction=0.05):

"""Rewire a fraction of edges to connect across stance groups."""

N = len(snapshot['texts'])

G = nx.Graph()

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

G.add\_edges\_from(snapshot['edge\_list'])

edges = list(G.edges())

E = len(edges)

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

left = [i for i,s in enumerate(snapshot['stance']) if s <= 0]

right = [i for i,s in enumerate(snapshot['stance']) if s > 0]

if not left or not right:

return snapshot

edges\_to\_replace = random.sample(edges, min(k, len(edges)))

for e in edges\_to\_replace:

G.remove\_edge(\*e)

u = random.choice(left)

v = random.choice(right)

G.add\_edge(u, v)

new\_edges = list(G.edges())

new\_snap = deepcopy(snapshot)

new\_snap['edge\_list'] = new\_edges

return new\_snap

def simulate\_rerank(snapshot: dict, moderation\_threshold=0.75, boost\_neutral=True):

"""Simulate content re-ranking by lowering meta 'visibility' of extreme nodes and boosting neutral ones."""

new\_snap = deepcopy(snapshot)

N = len(snapshot['texts'])

# if features\_meta first column represents 'visibility', otherwise create one

if new\_snap['features\_meta'] is None or new\_snap['features\_meta'].shape[1] < 1:

new\_snap['features\_meta'] = np.ones((N,2))

vis = new\_snap['features\_meta'][:, 0]

extreme\_mask = (np.abs(new\_snap['stance']) > moderation\_threshold)

vis[extreme\_mask] \*= 0.6 # reduce visibility for extreme nodes

if boost\_neutral:

neutral\_mask = (np.abs(new\_snap['stance']) <= 0.2)

vis[neutral\_mask] \*= 1.15

new\_snap['features\_meta'][:, 0] = vis

return new\_snap

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

# Visualizations

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

def visualize\_snapshot(snapshot: dict, preds: np.ndarray = None, title="Snapshot Visualization"):

"""Plot network with matplotlib using stance or prediction colors."""

N = len(snapshot['texts'])

G = nx.Graph()

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

G.add\_edges\_from(snapshot['edge\_list'])

pos = nx.spring\_layout(G, seed=SEED)

# node colors: if preds provided use them otherwise use stance sign

if preds is not None:

colors = ['red' if p==1 else 'blue' for p in preds]

sizes = [80 if p==1 else 40 for p in preds]

else:

colors = ['red' if s > 0 else 'blue' for s in snapshot['stance']]

sizes = [80 if abs(s) > 0.6 else 40 for s in snapshot['stance']]

plt.figure(figsize=(8,6))

nx.draw(G, pos=pos, node\_color=colors, node\_size=sizes, with\_labels=False, edge\_color='gray', alpha=0.7)

plt.title(title)

plt.show()

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

# Main runnable demonstration

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

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

print("Starting C-GNN pipeline demo...")

# 1) Load data (synthetic demo)

snaps = load\_data\_stub(num\_snapshots=6, nodes\_per\_snapshot=150)

# 2) Train pipeline

model, propmod, seq\_data\_list, y\_list = train\_pipeline(snaps, epochs=12)

# 3) Pick last example to demonstrate interventions and predictions

# Build sequence for final window

window = 3

final\_seq = snaps[-window:]

nlp = NLPEncoder()

for s in final\_seq:

s['text\_embeds'] = nlp.embed\_texts(s['texts'])

s['sentiment'] = nlp.sentiment\_scores(s['texts'])

seq\_pyg = [snapshot\_to\_pyg\_data(s, s['text\_embeds'], s['sentiment']) for s in final\_seq]

model.eval()

with torch.no\_grad():

probs, logits, emb = model(seq\_pyg)

preds = (probs.cpu().numpy() > 0.5).astype(int)

print("Predicted polarized nodes (count):", int(preds.sum()))

visualize\_snapshot(final\_seq[-1], preds, title="Model Predictions on Final Snapshot")

# 4) Simulate intervention (rewire edges)

new\_snap = simulate\_rewire(final\_seq[-1], fraction=0.08)

# Recompute embeddings/sentiment on new snapshot if needed

new\_snap['text\_embeds'] = final\_seq[-1]['text\_embeds']

new\_snap['sentiment'] = final\_seq[-1]['sentiment']

# Build new seq with rewired final snapshot

new\_seq = final\_seq[:-1] + [new\_snap]

seq\_pyg2 = [snapshot\_to\_pyg\_data(s, s['text\_embeds'], s['sentiment']) for s in new\_seq]

with torch.no\_grad():

probs2, logits2, emb2 = model(seq\_pyg2)

preds2 = (probs2.cpu().numpy() > 0.5).astype(int)

print("After rewire predicted polarized nodes (count):", int(preds2.sum()))

visualize\_snapshot(new\_snap, preds2, title="After Rewiring Intervention")

# 5) Report delta in predicted polarized nodes

print("Delta polarized node count:", int(preds.sum()) - int(preds2.sum()))

print("Demo complete.")