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)
```

```python
        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)
```