AI–Quantum Consciousness Simulation Framework

Hybrid Transformer + Quantum Neural Network (QNN) architecture for evaluating quantum theories of consciousness using synthetic and real EEG/fMRI datasets.

```
├── README.md
├── LICENSE
├── CITATION.cff
├── requirements.txt
├── .gitignore
├── models.py
├── quantum_layer.py
├── data.py
├── metrics.py
├── metrics_pyphi.py
├── train.py
├── visualize.py
├── train_robustness.py
├── benchmark_plot.py
└── eeg_preproc_openneuro.py
```

README.md

AI–Quantum Consciousness Simulation Framework

Hybrid **Transformer + Quantum Neural Network (QNN)** framework to evaluate quantum theories of consciousness.
Includes: synthetic EEG generator, PennyLane-based quantum layer, IIT Φ (PyPhi) integration template, robustness sweep, and plotting utilities.

Quickstart
```bash
python -m venv .venv && source .venv/bin/activate   # Windows: .venv¥Scripts¥activate
pip install -r requirements.txt

# Synthetic EEG demo (hybrid model)
```

```
python train.py --epochs 5 --batch_size 32 --seq_len 128 --n_channels 16 --model hybrid

# Visualize training logs
python visualize.py
```

IIT Φ (PyPhi) — Optional

- True IIT Φ requires a causal/structural model (TPM + connectivity).
- Use metrics_pyphi.py with small systems (3–6 nodes).
- In train.py, enable Φ computation (already wired; runs on validation with down-selected features).

Robustness Sweep

```
python eeg_preproc_openneuro.py --edf path/to/sample.edf --epoch_len 2.0 --out_npz eeg_preprocessed.npz
python train.py --data_npz eeg_preprocessed.npz --model hybrid --epochs 5
```

Notes

- The quantum layer uses **qubits** (RY/RZ/RX + CNOT ladder). Qutrit variants can be built by swapping the device/gates.
- metrics.py provides a **ϕ proxy** (MI-based). Use metrics_pyphi.py for exact Φ via PyPhi.

- Reproducibility: set seeds (numpy, torch) and consider deterministic CUDA flags.

Citation

See CITATION.cff.

License

CC BY 4.0 for text/docs; code under MIT (see LICENSE).

`LICENSE`
```text
MIT License

Copyright (c) 2025

Permission is hereby granted, free of charge, to any person obtaining a copy
...

...
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND...

# CITATION.cff

cff-version: 1.2.0
title: "AI–Quantum Consciousness Simulation Framework"
message: "If you use this code, please cite the associated preprint."
authors:
  - family-names: Shiraishi
    given-names: Kei
```

```
abstract: >
  Hybrid Transformer + QNN framework for evaluating quantum theories of consciousness,
  including PyPhi-based IIT Φ integration template and robustness tools.
license: MIT
version: "1.0.0"
date-released: "2025-10-06"
```

.gitignore

```
# Python
__pycache__/
*.py[cod]
*.ipynb_checkpoints
.venv/
.env
dist/
build/
*.egg-info/

# Logs / artifacts
training_logs.json
*.png
*.npz
robustness_results.csv
```

requirements.txt

```
torch>=2.2
numpy>=1.24
pennylane>=0.36
pennylane-lightning>=0.36
scikit-learn>=1.3
matplotlib>=3.8
mne>=1.6
pandas>=2.2
# Optional for exact IIT Φ:
```

```
pyphi>=1.2.0
```

models.py

```python
import math
import torch
import torch.nn as nn
from quantum_layer import QuantumTorchLayer

class PositionalEncoding(nn.Module):
    def __init__(self, d_model: int, max_len: int = 10000):
        super().__init__()
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        # x: [B, T, D]
        T = x.size(1)
        return x + self.pe[:, :T, :]

class TransformerBaseline(nn.Module):
    """Simple Transformer encoder for multi-channel time series.
    Input: [B, C, T] -> project to D -> [B, T, D] -> encoder -> pool -> [B, D]
    """
    def __init__(self, n_channels=16, d_model=128, nhead=4, num_layers=2, dim_feedforward=256, dropout=0.1):
        super().__init__()
        self.input_proj = nn.Conv1d(n_channels, d_model, kernel_size=1)
        enc_layer = nn.TransformerEncoderLayer(d_model, nhead, dim_feedforward, dropout, batch_first=True)
```

```
        self.encoder = nn.TransformerEncoder(enc_layer, num_layers=num_layers)
        self.posenc = PositionalEncoding(d_model)
        self.pool = nn.AdaptiveAvgPool1d(1)

    def forward(self, x):
        # x: [B, C, T]
        z = self.input_proj(x)          # [B, D, T]
        z = z.permute(0, 2, 1)          # [B, T, D]
        z = self.posenc(z)              # [B, T, D]
        z = self.encoder(z)             # [B, T, D]
        z = z.permute(0, 2, 1)          # [B, D, T]
        z = self.pool(z).squeeze(-1)    # [B, D]
        return z
class HybridModel(nn.Module):
    """Transformer encoder -> QuantumTorchLayer (PennyLane) -> MLP head"""
    def __init__(self, n_channels=16, d_model=128, n_qubits=4, out_dim=2):
        super().__init__()
        self.backbone = TransformerBaseline(n_channels=n_channels, d_model=d_model)
        self.to_q = nn.Linear(d_model, n_qubits)
        self.q_layer = QuantumTorchLayer(n_qubits=n_qubits, n_layers=2)
        self.head = nn.Sequential(
            nn.Linear(n_qubits, 64),
            nn.ReLU(),
            nn.Linear(64, out_dim)
        )

    def forward(self, x):
        f = self.backbone(x)          # [B, D]
        q_in = self.to_q(f)           # [B, n_qubits]
        q_out = self.q_layer(q_in)    # [B, n_qubits]
        return self.head(q_out)
```

quantum_layer.py

```
import torch
import pennylane as qml
```

```python
class QuantumTorchLayer(torch.nn.Module):
    """PennyLane variational circuit (angle embedding + HEA with CNOT ladder).
      Returns expval Z per qubit.
    """
    def __init__(self, n_qubits=4, n_layers=2):
        super().__init__()
        self.n_qubits = n_qubits
        self.n_layers = n_layers
        self.dev = qml.device("default.qubit", wires=n_qubits, shots=None)

        # (layers, qubits, 3) for RY, RZ, RX
        self.theta = torch.nn.Parameter(torch.randn(n_layers, n_qubits, 3) * 0.01)

        @qml.qnode(self.dev, interface="torch", diff_method="backprop")
        def circuit(x, theta):
            # Embed inputs
            for i in range(n_qubits):
                qml.RY(x[i], wires=i)
            # Variational blocks
            for l in range(n_layers):
                for q in range(n_qubits):
                    qml.RY(theta[l, q, 0], wires=q)
                    qml.RZ(theta[l, q, 1], wires=q)
                    qml.RX(theta[l, q, 2], wires=q)
                for q in range(n_qubits - 1):
                    qml.CNOT(wires=[q, q + 1])
            return [qml.expval(qml.PauliZ(i)) for i in range(n_qubits)]
        self.circuit = circuit

    def forward(self, x):
        # x: [B, n_qubits]
        outs = []
        for row in x:
            outs.append(self.circuit(row, self.theta))
        return torch.stack(outs, dim=0)
```

data.py

```python
import numpy as np

def generate_synthetic_eeg(n_samples=1024, n_channels=16, seq_len=128,
noise_sigma=0.2, seed=42):
    """Synthetic EEG-like data with multi-frequency components + noise.
    Returns X: [N, C, T], y: [N] (binary labels)
    """
    rng = np.random.default_rng(seed)
    X = np.zeros((n_samples, n_channels, seq_len), dtype=np.float32)
    y = rng.integers(0, 2, size=(n_samples,), dtype=np.int64)

    t = np.linspace(0, 1, seq_len)
    freqs = [6, 10, 40]  # theta/alpha/gamma-ish
    for i in range(n_samples):
        for c in range(n_channels):
            sig = sum(np.sin(2*np.pi*f*t + rng.uniform(0, 2*np.pi)) for f in freqs)
            if y[i] == 1:
                sig += 0.5 * np.sin(2*np.pi*20*t + rng.uniform(0, 2*np.pi))
            sig += rng.normal(0, noise_sigma, size=seq_len)
            X[i, c] = sig.astype(np.float32)
    return X, y

def load_npz(path):
    d = np.load(path)
    X = d["X"].astype(np.float32)
    y = d["y"].astype(np.int64)
    return X, y
```

metrics.py

```python
import numpy as np
from sklearn.metrics import mutual_info_score

def accuracy(pred_logits, y_true):
    pred = pred_logits.argmax(axis=1)
```

```
    return (pred == y_true).mean()


def phi_proxy(features):
    """MI-based proxy for integration: average pairwise MI across feature dims (discretized)."""
    N, C = features.shape
    disc = np.floor((features - features.min())/(features.ptp()+1e-8)*20).astype(int)
    mis = []
    for i in range(C):
        for j in range(i+1, C):
            mis.append(mutual_info_score(disc[:, i], disc[:, j]))
    return float(np.mean(mis)) if mis else 0.0
```

metrics_pyphi.py

```
from typing import Tuple
import numpy as np

"""
PyPhi integration template for computing IIT Phi.

- build_binary_states(X): discretize continuous features to binary.
- estimate_tpm(states, k): estimate a first-order TPM (binary states).
- compute_phi_from_tpm(tpm, connectivity, state): compute Phi via pyphi.

Note: Keep the system small (3–6 nodes). Provide meaningful connectivity and a system state.
"""


def build_binary_states(X: np.ndarray, thresh: float=None) -> np.ndarray:
    if thresh is None:
        thresh = np.median(X, axis=0, keepdims=True)
    return (X > thresh).astype(int)


def estimate_tpm(states: np.ndarray, k: int = 1) -> np.ndarray:
    C = states.shape[1]
    n_states = 2**C
    def to_index(s):
        return int("".join(str(x) for x in s[::-1]), 2)  # little-endian
```

```
    counts = np.zeros((n_states, n_states), dtype=np.float64)
    for t in range(len(states)-k):
        i = to_index(states[t])
        j = to_index(states[t+k])
        counts[i, j] += 1.0
    with np.errstate(divide='ignore', invalid='ignore'):
        tpm = counts / counts.sum(axis=1, keepdims=True)
    tpm[np.isnan(tpm)] = 0.0
    return tpm

def compute_phi_from_tpm(tpm: np.ndarray, connectivity: np.ndarray, state: np.ndarray) -> float:
    import pyphi
    C = connectivity.shape[0]
    net = pyphi.Network(tpm, connectivity_matrix=connectivity)
    state_index = int("".join(str(int(x)) for x in state), 2)  # big-endian vs little-endian: be consistent
    sub = pyphi.Subsystem(net, nodes=tuple(range(C)), state=state_index)
    phi = pyphi.compute.big_phi(sub)
    return float(phi)
```

train.py

```
import argparse, json, numpy as np, torch
import torch.nn as nn, torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
from models import HybridModel, TransformerBaseline
from data import generate_synthetic_eeg, load_npz
from metrics import accuracy, phi_proxy

# Optional IIT Φ
try:
    from metrics_pyphi import build_binary_states, estimate_tpm, compute_phi_from_tpm
    HAS_PYPHI = True
except Exception:
    HAS_PYPHI = False
```

```
def select_top_features(feats, k=4):
    var = feats.var(axis=0)
    idx = np.argsort(var)[::-1][:k]
    return feats[:, idx], idx

def make_connectivity(C):
    return np.eye(C, dtype=int)  # simple diagonal; replace with better adjacency if needed

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--epochs", type=int, default=5)
    ap.add_argument("--batch_size", type=int, default=32)
    ap.add_argument("--seq_len", type=int, default=128)
    ap.add_argument("--n_channels", type=int, default=16)
    ap.add_argument("--lr", type=float, default=1e-3)
    ap.add_argument("--model", type=str, default="hybrid", choices=["hybrid","transformer"])
    ap.add_argument("--data_npz", type=str, default="")
    ap.add_argument("--noise_sigma", type=float, default=0.2)
    ap.add_argument("--seed", type=int, default=7)
    args = ap.parse_args()

    np.random.seed(args.seed); torch.manual_seed(args.seed)

    if args.data_npz:
        X, y = load_npz(args.data_npz)
        args.n_channels, args.seq_len = X.shape[1], X.shape[2]
    else:
        X, y = generate_synthetic_eeg(n_samples=2048, n_channels=args.n_channels, seq_len=args.seq_len, noise_sigma=args.noise_sigma, seed=args.seed)

    # Train/val split
    n = len(X)
    idx = np.arange(n); np.random.shuffle(idx)
    split = int(0.8*n)
    tr_idx, va_idx = idx[:split], idx[split:]
```

```
    Xtr, ytr = X[tr_idx], y[tr_idx]
    Xva, yva = X[va_idx], y[va_idx]

    train_loader = DataLoader(TensorDataset(torch.tensor(Xtr), torch.tensor(ytr)),
batch_size=args.batch_size, shuffle=True)
    val_loader = DataLoader(TensorDataset(torch.tensor(Xva), torch.tensor(yva)),
batch_size=args.batch_size, shuffle=False)

    if args.model == "hybrid":
        model = HybridModel(n_channels=args.n_channels, d_model=128, n_qubits=4,
out_dim=2)
    else:
        base = TransformerBaseline(n_channels=args.n_channels, d_model=128)
        model = nn.Sequential(base, nn.Linear(128, 2))

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.AdamW(model.parameters(), lr=args.lr)
    logs = {"train_loss": [], "val_loss": [], "val_acc": [], "phi_proxy": [], "phi_pyphi": []}

    for epoch in range(1, args.epochs+1):
        model.train()
        total_loss = 0.0
        for xb, yb in train_loader:
            xb, yb = xb.to(device), yb.to(device)
            optimizer.zero_grad()
            logits = model(xb)
            loss = criterion(logits, yb)
            loss.backward(); optimizer.step()
            total_loss += loss.item() * xb.size(0)
        tr_loss = total_loss / len(train_loader.dataset)

        # Validation
        model.eval()
```

```
total_vloss = 0.0
all_logits, all_y, feats_for_phi = [], [], []
with torch.no_grad():
    for xb, yb in val_loader:
        xb = xb.to(device)
        logits = model(xb)
        vloss = criterion(logits, yb.to(device))
        total_vloss += vloss.item() * xb.size(0)
        all_logits.append(logits.cpu().numpy())
        all_y.append(yb.numpy())
        # Penultimate features
        if isinstance(model, nn.Sequential):
            feats = model[0](xb).cpu().numpy()
        else:
            f = model.backbone(xb); q_in = model.to_q(f); feats = q_in.cpu().numpy()
        feats_for_phi.append(feats)

v_loss = total_vloss / len(val_loader.dataset)
logits_np = np.concatenate(all_logits, axis=0)
y_np = np.concatenate(all_y, axis=0)
val_acc = accuracy(logits_np, y_np)

feats_np = np.concatenate(feats_for_phi, axis=0)
phi_val = phi_proxy(feats_np)

# Exact IIT Φ (optional, small C)
phi_pyphi = None
if HAS_PYPHI:
    subN = min(256, feats_np.shape[0])
    sub = feats_np[np.random.choice(feats_np.shape[0], size=subN, replace=False)]
    sub4, _ = select_top_features(sub, k=4)
    S = build_binary_states(sub4)
    tpm = estimate_tpm(S, k=1)
    conn = make_connectivity(C=4)
    state = S[-1]
    try:
```

```
            phi_pyphi = compute_phi_from_tpm(tpm, conn, state)
        except Exception:
            phi_pyphi = None

        logs["train_loss"].append(tr_loss)
        logs["val_loss"].append(v_loss)
        logs["val_acc"].append(float(val_acc))
        logs["phi_proxy"].append(float(phi_val))
        logs["phi_pyphi"].append(float(phi_pyphi) if phi_pyphi is not None else None)

        print(f"Epoch {epoch:02d} | train_loss={tr_loss:.4f} val_loss={v_loss:.4f}
val_acc={val_acc:.3f} "
              f"phi_proxy={phi_val:.4f} phi_pyphi={phi_pyphi if phi_pyphi is not None else
'NA'}")

    with open("training_logs.json", "w") as f:
        json.dump(logs, f, indent=2)

if __name__ == "__main__":
    main()
```

visualize.py

```
import json
import matplotlib.pyplot as plt

with open("training_logs.json", "r") as f:
    logs = json.load(f)

plt.figure()
plt.plot(logs["train_loss"], label="train_loss")
plt.plot(logs["val_loss"], label="val_loss")
plt.legend(); plt.title("Loss over epochs"); plt.xlabel("epoch"); plt.ylabel("loss")
plt.savefig("loss.png", dpi=160); print("Saved loss.png")

plt.figure()
plt.plot(logs["val_acc"], label="val_acc")
```

```
plt.plot(logs["phi_proxy"], label="phi_proxy")
if any(x is not None for x in logs.get("phi_pyphi", [])):
    plt.plot([x if x is not None else None for x in logs["phi_pyphi"]], label="phi_pyphi")
plt.legend(); plt.title("Val Acc & Phi Scores"); plt.xlabel("epoch"); plt.ylabel("score")
plt.savefig("val_acc_phi.png", dpi=160); print("Saved val_acc_phi.png")
```

train_robustness.py

```
import argparse, csv, numpy as np, torch
import torch.nn as nn, torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
from models import HybridModel, TransformerBaseline
from data import generate_synthetic_eeg
from metrics import accuracy, phi_proxy

def run_once(model_name, n_channels, seq_len, noise_sigma, epochs, batch_size, seed=7):
    np.random.seed(seed); torch.manual_seed(seed)
    X, y = generate_synthetic_eeg(n_samples=1024, n_channels=n_channels, seq_len=seq_len, noise_sigma=noise_sigma, seed=seed)
    idx = np.arange(len(X)); np.random.shuffle(idx)
    split = int(0.8*len(X)); tr, va = idx[:split], idx[split:]
    Xtr, ytr, Xva, yva = X[tr], y[tr], X[va], y[va]

    train_loader = DataLoader(TensorDataset(torch.tensor(Xtr), torch.tensor(ytr)), batch_size=batch_size, shuffle=True)
    val_loader = DataLoader(TensorDataset(torch.tensor(Xva), torch.tensor(yva)), batch_size=batch_size)

    if model_name == "hybrid":
        model = HybridModel(n_channels=n_channels, d_model=128, n_qubits=4, out_dim=2)
    else:
        base = TransformerBaseline(n_channels=n_channels, d_model=128)
        model = nn.Sequential(base, nn.Linear(128, 2))

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)
```

```
    opt = optim.AdamW(model.parameters(), lr=1e-3)
    loss_fn = nn.CrossEntropyLoss()

    for ep in range(epochs):
        model.train()
        for xb, yb in train_loader:
            xb, yb = xb.to(device), yb.to(device)
            opt.zero_grad(); loss = loss_fn(model(xb), yb); loss.backward(); opt.step()

    model.eval(); all_logits, feats, ys = [], [], []
    with torch.no_grad():
        for xb, yb in val_loader:
            xb = xb.to(device); logits = model(xb)
            all_logits.append(logits.cpu().numpy()); ys.append(yb.numpy())
            if isinstance(model, nn.Sequential):
                feats.append(model[0](xb).cpu().numpy())
            else:
                f = model.backbone(xb); q_in = model.to_q(f); feats.append(q_in.cpu().numpy())

    logits_np = np.concatenate(all_logits, 0); y_np = np.concatenate(ys, 0)
    acc = accuracy(logits_np, y_np)
    feat_np = np.concatenate(feats, 0); phi = phi_proxy(feat_np)
    return float(acc), float(phi)

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--seq_len", type=int, default=128)
    ap.add_argument("--n_channels", type=int, default=8)
    ap.add_argument("--epochs", type=int, default=5)
    ap.add_argument("--batch_size", type=int, default=32)
    ap.add_argument("--noise_sigmas", type=float, nargs="+", default=[0.1, 0.3, 0.5])
    ap.add_argument("--repeats", type=int, default=3)
    ap.add_argument("--out_csv", type=str, default="robustness_results.csv")
    args = ap.parse_args()

    rows = [["model","noise_sigma","repeat","val_acc","phi_proxy"]]
```

```
    for sigma in args.noise_sigmas:
        for r in range(args.repeats):
            for model in ["transformer","hybrid"]:
                acc, phi = run_once(model, args.n_channels, args.seq_len, sigma, args.epochs, args.batch_size, seed=7+r)
                rows.append([model, sigma, r, acc, phi])
                print(f"{model} sigma={sigma} rep={r} -> acc={acc:.3f} phi={phi:.4f}")

    with open(args.out_csv, "w", newline="") as f:
        csv.writer(f).writerows(rows)
    print(f"Saved {args.out_csv}")

if __name__ == "__main__":
    main()
```

benchmark_plot.py

```
import argparse, csv, numpy as np
import matplotlib.pyplot as plt

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--csv", type=str, default="robustness_results.csv")
    args = ap.parse_args()

    with open(args.csv) as f:
        r = csv.DictReader(f)
        rows = list(r)

    def agg(metric):
        out = {}
        for row in rows:
            key = (row["model"], float(row["noise_sigma"]))
            out.setdefault(key, []).append(float(row[metric]))
        xs = sorted(set(float(r["noise_sigma"]) for r in rows))
        return xs, {m: [np.mean(out[(m, x)]) for x in xs] form in ["transformer","hybrid"]}
```

```
    xs, accs = agg("val_acc")
    _, phis = agg("phi_proxy")

    plt.figure()
    plt.plot(xs, accs["transformer"], marker="o", label="Transformer (val_acc)")
    plt.plot(xs, accs["hybrid"], marker="o", label="Hybrid (val_acc)")
    plt.xlabel("noise_sigma"); plt.ylabel("val_acc"); plt.title("Validation Accuracy vs Noise")
    plt.legend(); plt.grid(True); plt.savefig("benchmark_acc.png", dpi=160); print("Saved benchmark_acc.png")

    plt.figure()
    plt.plot(xs, phis["transformer"], marker="o", label="Transformer (phi_proxy)")
    plt.plot(xs, phis["hybrid"], marker="o", label="Hybrid (phi_proxy)")
    plt.xlabel("noise_sigma"); plt.ylabel("phi_proxy"); plt.title("Phi Proxy vs Noise")
    plt.legend(); plt.grid(True); plt.savefig("benchmark_phi.png", dpi=160); print("Saved benchmark_phi.png")

if __name__ == "__main__":
    main()
```

eeg_preproc_openneuro.py

```
import argparse, numpy as np, mne

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--edf", type=str, required=True, help="Path to EEG EDF/FIF file")
    ap.add_argument("--l_freq", type=float, default=1.0)
    ap.add_argument("--h_freq", type=float, default=50.0)
    ap.add_argument("--epoch_len", type=float, default=2.0, help="seconds")
    ap.add_argument("--sfreq", type=float, default=None, help="resample Hz (optional)")
    ap.add_argument("--out_npz", type=str, default="eeg_preprocessed.npz")
    args = ap.parse_args()

    raw = mne.io.read_raw(args.edf, preload=True)
    raw.filter(args.l_freq, args.h_freq, fir_design="firwin")
    if args.sfreq:
```

```
        raw.resample(args.sfreq)

    data, times = raw.get_data(return_times=True)  # [C, T]
    C, T = data.shape
    step = int(args.epoch_len * raw.info["sfreq"])
    X = []
    for start in range(0, T - step, step):
        X.append(data[:, start:start+step])
    X = np.stack(X, axis=0)  # [N, C, T_window]

    # Placeholder labels; adapt to your events/logs
    y = np.zeros((X.shape[0],), dtype=np.int64)
    np.savez_compressed(args.out_npz, X=X.astype(np.float32), y=y)
    print(f"Saved {args.out_npz} with shape {X.shape} (N, C, T), sfreq={raw.info['sfreq']}
Hz")

if __name__ == "__main__":
    main()
```