CAP-TTS-CGAN — Repo Skeleton

This canvas contains a ready-to-run skeleton for adapting **TTS-CGAN** to generate synthetic CAP data from the CAPSLPDB (PhysioNet).

Files included below (you can copy each into your project):

- `README.md` — quick start and overview
- `requirements.txt` — python deps
- `dataloader.py` — CAPDataset & preprocessing
- `models.py` — Generator and Discriminator (multichannel + optional conditioning)
- `train.py` — training script
- `generate.py` — sampling script
- `utils.py` — helper functions: annotation parsing, windowing, saving

---

README.md

````
# CAP-TTS-CGAN (skeleton)

This repository provides a skeleton for training a Time-series GAN (TTS-CGAN style) on the CAP Sleep Database (CAPSLPDB) to produce synthetic CAP EEG segments.

## Quick steps
1. Install dependencies:
   ```bash
   pip install -r requirements.txt
   ```
2. Download CAPSLPDB from PhysioNet and place EDF + annotation files into a folder, e.g. `data/raw`.
3. Preprocess & create .npy dataset with provided dataloader utilities.
4. Train:
   ```bash
   python train.py --data data/processed/cap_windows.npy --epochs 200 --batch_size 64
   ```
5. Generate samples:
   ```bash
   python generate.py --model checkpoints/generator.pth --n 100 --out_dir generated
   ```


## Notes
````

```
- This skeleton is a starting point. Replace the annotation parsing and channel selection with channels you need (e.g., C3-A2, C4-A1).
- Conditioning is optional; there's an example of including label conditioning for CAP phase.
```

```
---

## requirements.txt
```

torch>=1.10 numpy pyedflib tqdm scipy mne

````
---

## dataloader.py

```python
import os
import numpy as np
import pyedflib
from torch.utils.data import Dataset

from utils import parse_cap_annotations, window_signal

class CAPDataset(Dataset):
    """Reads EDF files and corresponding annotation files, returns windows and optional labels.

    Expected input layout:
        data/raw/
            subject1.edf
            subject1_annotations.txt
            subject2.edf
            subject2_annotations.txt

    This dataset returns tensors shaped (channels, seq_len) per sample (PyTorch conv1d expects channels-first).
    """
    def __init__(self, raw_dir, channels=None, window_sec=30, fs=100, use_labels=True, cache_npy=None):
        self.raw_dir = raw_dir
        self.window_sec = window_sec
        self.fs = fs
        self.use_labels = use_labels
        self.channels = channels  # list of channel names or indices to select; None = use all
        self.windows = []  # numpy arrays (channels, seq_len)
````
````

```
        self.labels = []   # ints or None

        if cache_npy and os.path.exists(cache_npy):
            print(f"Loading preprocessed data from {cache_npy}")
            data = np.load(cache_npy, allow_pickle=True)
            self.windows = data["windows"].tolist()
            self.labels = data["labels"].tolist()
        else:
            self._build_dataset()
            if cache_npy:
                np.savez(cache_npy, windows=self.windows, labels=self.labels)

    def _build_dataset(self):
        files = os.listdir(self.raw_dir)
        edf_files = [f for f in files if f.lower().endswith('.edf')]
        for edf in edf_files:
            edf_path = os.path.join(self.raw_dir, edf)
            base = os.path.splitext(edf)[0]
            ann_path = os.path.join(self.raw_dir, base + '_annotations.txt')

            try:
                f = pyedflib.EdfReader(edf_path)
            except Exception as e:
                print(f"Failed to read {edf_path}: {e}")
                continue

            n_signals = f.signals_in_file
            labels = f.getSignalLabels()
            sigs = np.array([f.readSignal(i) for i in range(n_signals)])  #
shape (n_channels, n_samples)
            f._close()
            del f

            # channel selection
            if self.channels is not None:
                # channels may be indices or names
                if all(isinstance(c, int) for c in self.channels):
                    sigs = sigs[self.channels]
                else:
                    idxs = [labels.index(c) for c in self.channels if c in
labels]
                    sigs = sigs[idxs]

            # optionally parse annotations
            anns = None
            if self.use_labels and os.path.exists(ann_path):
                anns = parse_cap_annotations(ann_path)
```

```
            # windowing
            wlen = int(self.window_sec * self.fs)
            windows, win_labels = window_signal(sigs, anns, wlen, fs=self.fs)
            for win, lab in zip(windows, win_labels):
                # normalize per-window (z-score)
                win = (win - win.mean(axis=1, keepdims=True)) / (win.std(axis=1,
keepdims=True) + 1e-8)
                self.windows.append(win.astype(np.float32))
                self.labels.append(lab if lab is not None else -1)

    def __len__(self):
        return len(self.windows)

    def __getitem__(self, idx):
        x = self.windows[idx]  # numpy (channels, seq_len)
        y = self.labels[idx]
        return x, y
```

---

models.py

```
import torch
import torch.nn as nn

class ConditionalVector(nn.Module):
    """Simple embedding for label conditioning (if used).
    Returns a vector that can be concatenated to the latent vector or projected
and added to feature maps.
    """
    def __init__(self, n_classes, latent_dim):
        super().__init__()
        self.embed = nn.Embedding(n_classes, latent_dim)
    def forward(self, labels):
        return self.embed(labels)


class Generator(nn.Module):
    def __init__(self, latent_dim=100, out_channels=3, seq_len=3000, cond=False,
n_classes=3):
        super().__init__()
        self.latent_dim = latent_dim
        self.cond = cond
        input_dim = latent_dim
        if cond:
            self.cond_vec = ConditionalVector(n_classes, latent_dim)
```

```
        # We'll map the latent vector into a set of feature maps and use
ConvTranspose1d to upsample.
        self.project = nn.Sequential(
            nn.Linear(input_dim, 256 * (seq_len // 64)),
            nn.ReLU()
        )
        self.deconv = nn.Sequential(
            nn.ConvTranspose1d(256, 128, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose1d(128, 64, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose1d(64, out_channels, kernel_size=4, stride=2,
padding=1),
            nn.Tanh()
        )

    def forward(self, z, labels=None):
        if self.cond and labels is not None:
            cv = self.cond_vec(labels)
            z = z + cv
        x = self.project(z)  # (batch, feat * time)
        batch = x.shape[0]
        feat_len = x.shape[1]
        # reshape to (batch, feat_maps, time)
        fmap = feat_len // (self.deconv[0].in_channels)
        # safer reshape: assume project set it to 256 * T
        x = x.view(batch, 256, -1)
        x = self.deconv(x)
        return x


class Discriminator(nn.Module):
    def __init__(self, in_channels=3, seq_len=3000, cond=False, n_classes=3):
        super().__init__()
        self.cond = cond
        if cond:
            self.label_proj = nn.Embedding(n_classes, in_channels)

        self.net = nn.Sequential(
            nn.Conv1d(in_channels, 64, kernel_size=7, stride=2, padding=3),
            nn.LeakyReLU(0.2),
            nn.Conv1d(64, 128, kernel_size=7, stride=2, padding=3),
            nn.LeakyReLU(0.2),
            nn.Conv1d(128, 256, kernel_size=7, stride=2, padding=3),
            nn.LeakyReLU(0.2),
            nn.AdaptiveAvgPool1d(1),
            nn.Flatten(),
```

```
            nn.Linear(256, 1)
        )

    def forward(self, x, labels=None):
        # x shape: (batch, channels, seq_len)
        if self.cond and labels is not None:
            # simple conditioning by adding a per-channel bias from embedding
            emb = self.label_proj(labels)  # shape (batch, in_channels)
            emb = emb.unsqueeze(-1)  # (batch, in_channels, 1)
            x = x + emb
        out = self.net(x)
        return out
```

---

utils.py

```
import numpy as np

def parse_cap_annotations(ann_path):
    """Parse CAP annotation file (stub).

    The exact format of annotations in CAPSLPDB varies. This function should be
adapted to the real annotation layout.
    Expected return: list of tuples (onset_sample, duration_samples, label_int)
    """
    anns = []
    with open(ann_path, 'r') as f:
        for line in f:
            line = line.strip()
            if not line: continue
            # Example simple format: onset_sec,duration_sec,label
            parts = line.split(',')
            if len(parts) >= 3:
                onset = float(parts[0])
                dur = float(parts[1])
                lab = parts[2]
                # map lab to int (example)
                lab_int = 1 if 'A' in lab else 0
                anns.append((int(onset), int(dur), lab_int))
    return anns


def window_signal(sigs, anns, wlen, fs=100):
    """Slice multichannel signals into windows.
```

```
    - sigs: np.array (n_channels, n_samples)
    - anns: list of (onset_sec, dur_sec, label_int) OR None
    - wlen: window length in samples

    Returns (windows, labels)
      windows: list of np arrays (n_channels, wlen)
      labels: list of ints (e.g., majority label in window) or -1
    """
    n_channels, n_samples = sigs.shape
    step = wlen  # non-overlapping by default; change to wlen//2 if overlap
desired
    windows = []
    labels = []
    for start in range(0, n_samples - wlen + 1, step):
        win = sigs[:, start:start + wlen]
        label = -1
        if anns:
            # determine label by majority overlap
            start_sec = start / fs
            end_sec = (start + wlen) / fs
            counts = {}
            for a_on, a_dur, a_lab in anns:
                a_end = a_on + a_dur
                # check overlap in seconds
                if (a_on < end_sec) and (a_end > start_sec):
                    counts[a_lab] = counts.get(a_lab, 0) + 1
            if counts:
                # choose label with max overlaps
                label = max(counts.items(), key=lambda x: x[1])[0]
        windows.append(win)
        labels.append(label)
    return windows, labels
```

---

train.py

```
import argparse
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
import numpy as np
from tqdm import tqdm

from dataloader import CAPDataset
from models import Generator, Discriminator
```

```
def train(args):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    # load dataset
    ds = CAPDataset(args.data_dir, window_sec=args.window_sec, fs=args.fs,
cache_npy=args.cache)
    # convert to DataLoader with collate to convert to torch tensors
    loader = DataLoader(ds, batch_size=args.batch_size, shuffle=True,
drop_last=True)

    # infer channels and seq_len from first sample
    sample_x, _ = ds[0]
    in_channels, seq_len = sample_x.shape

    G = Generator(latent_dim=args.latent_dim, out_channels=in_channels,
seq_len=seq_len).to(device)
    D = Discriminator(in_channels=in_channels, seq_len=seq_len).to(device)

    g_opt = torch.optim.Adam(G.parameters(), lr=args.lr, betas=(0.5, 0.999))
    d_opt = torch.optim.Adam(D.parameters(), lr=args.lr, betas=(0.5, 0.999))

    criterion = nn.BCEWithLogitsLoss()

    for epoch in range(args.epochs):
        pbar = tqdm(loader)
        for real, _ in pbar:
            real = torch.tensor(real).to(device)  # numpy -> tensor
            # ensure shape (batch, channels, seq_len)
            if real.ndim == 3:
                pass
            else:
                real = real.unsqueeze(1)

            batch_size = real.size(0)

            # Train D
            z = torch.randn(batch_size, args.latent_dim).to(device)
            z = z
            fake = G(z)
            # make sure fake shape matches
            if fake.dim() == 2:
                fake = fake.unsqueeze(1)

            d_real = D(real)
            d_fake = D(fake.detach())
            real_labels = torch.ones_like(d_real)
```

```
            fake_labels = torch.zeros_like(d_fake)

            d_loss = criterion(d_real, real_labels) + criterion(d_fake,
fake_labels)
            d_opt.zero_grad(); d_loss.backward(); d_opt.step()

            # Train G
            d_fake_for_g = D(fake)
            g_loss = criterion(d_fake_for_g, real_labels)
            g_opt.zero_grad(); g_loss.backward(); g_opt.step()

            pbar.set_description(f"E{epoch} D:{d_loss.item():.4f} G:
{g_loss.item():.4f}")

        # checkpoint
        torch.save(G.state_dict(), f"checkpoints/generator_epoch{epoch}.pth")
        torch.save(D.state_dict(), f"checkpoints/
discriminator_epoch{epoch}.pth")


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--data_dir', type=str, default='data/raw')
    parser.add_argument('--cache', type=str, default='data/processed/
cap_windows.npz')
    parser.add_argument('--window_sec', type=int, default=30)
    parser.add_argument('--fs', type=int, default=100)
    parser.add_argument('--epochs', type=int, default=50)
    parser.add_argument('--batch_size', type=int, default=32)
    parser.add_argument('--latent_dim', type=int, default=128)
    parser.add_argument('--lr', type=float, default=2e-4)
    args = parser.parse_args()
    train(args)
```

generate.py

```
import argparse
import os
import torch
import numpy as np
from models import Generator


def generate(args):
```

```
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    # load generator
    G = Generator(latent_dim=args.latent_dim, out_channels=args.channels,
seq_len=args.seq_len).to(device)
    G.load_state_dict(torch.load(args.model, map_location=device))
    G.eval()

    os.makedirs(args.out_dir, exist_ok=True)
    n = args.n
    batch = 64
    generated = []
    for i in range(0, n, batch):
        b = min(batch, n - i)
        z = torch.randn(b, args.latent_dim).to(device)
        with torch.no_grad():
            fake = G(z)
        fake = fake.cpu().numpy()  # (batch, channels, seq_len)
        for j in range(fake.shape[0]):
            out = fake[j]
            np.save(os.path.join(args.out_dir, f"sample_{i+j}.npy"), out)
            generated.append(out)
    print(f"Saved {len(generated)} samples to {args.out_dir}")

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--model', type=str, required=True)
    parser.add_argument('--n', type=int, default=100)
    parser.add_argument('--out_dir', type=str, default='generated')
    parser.add_argument('--latent_dim', type=int, default=128)
    parser.add_argument('--channels', type=int, default=3)
    parser.add_argument('--seq_len', type=int, default=3000)
    args = parser.parse_args()
    generate(args)
```

Final notes

- This skeleton aims to be a practical starting point. Replace the annotation parsing logic in `utils.parse_cap_annotations` with the real format used by CAPSLPDB.
- Tune model depth, kernel sizes, strides, and training hyperparameters for your dataset and sequence length.

<!-- End of canvas -->