In [24]:
```
# %% [markdown]
# # Neuromorphic KWS: PyTorch SNN + Loihi Emulator
#
# This notebook:
# 1. Loads the SpeechCommands sample data (local folder).
# 2. Uses the same MFCC preprocessing as the baseline CNN.
# 3. Rebuilds the CNN + SNN architecture in PyTorch.
# 4. Loads your trained weights from `saved_models/`.
# 5. Runs PyTorch SNN inference for a test example.
# 6. Builds a small Nengo+Loihi network for a toy demo (later cells).

# %%
import os
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

from pathlib import Path

import torchaudio
import torchaudio.transforms as T

import nengo
import nengo_loihi

print("PyTorch :", torch.__version__)
print("Torchaudio :", torchaudio.__version__)
print("Nengo :", nengo.__version__)
print("Nengo Loihi :", nengo_loihi.__version__)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Device:", device)
```

```
PyTorch : 2.9.1
Torchaudio : 2.9.1
Nengo : 4.1.0
Nengo Loihi : 1.1.0
Device: cpu
```

In [25]:
```
from pathlib import Path

# Project root = the folder ABOVE this notebook
NOTEBOOK_DIR = Path.cwd()
PROJECT_ROOT = NOTEBOOK_DIR.parent

print("Notebook is running from:", NOTEBOOK_DIR)
print("Project root resolved as:", PROJECT_ROOT)

DATA_DIR = PROJECT_ROOT / "sample_data" / "speech_commands_v0.02"
MODEL_DIR = PROJECT_ROOT / "saved_models"
```

```
print("DATA_DIR exists:", DATA_DIR.exists())
print("MODEL_DIR exists:", MODEL_DIR.exists())
```

```
Notebook is running from: /Users/maddy/Desktop/PLEP/Project/CS-576-Fina
l-Project/loihi_emulator
Project root resolved as: /Users/maddy/Desktop/PLEP/Project/CS-576-Fina
l-Project
DATA_DIR exists: True
MODEL_DIR exists: True
```

In [26]:
```
# %%
SAMPLE_RATE = 16000
N_MFCC = 40

import soundfile as sf
import librosa

def wav_to_mfcc(path: Path) -> torch.Tensor:
    """
    Load WAV using soundfile + compute MFCC using librosa.
    Avoids all Torchaudio/TorchCodec backends.
    """

    # ---- Load WAV using SoundFile ----
    waveform, sr = sf.read(str(path))      # numpy array
    if waveform.ndim > 1:                  # stereo → mono
        waveform = waveform.mean(axis=1)

    waveform = waveform.astype("float32")

    # ---- Resample if needed ----
    if sr != SAMPLE_RATE:
        waveform = librosa.resample(waveform, orig_sr=sr, target_sr=SA

    # ---- Compute MFCC using librosa ----
    mfcc = librosa.feature.mfcc(
        y=waveform,
        sr=SAMPLE_RATE,
        n_mfcc=N_MFCC,
        n_fft=400,
        hop_length=160,
        n_mels=40,
    )  # shape -> (40, T)

    # ---- Normalize ----
    mfcc = torch.tensor(mfcc, dtype=torch.float32)
    mfcc = (mfcc - mfcc.mean()) / (mfcc.std() + 1e-5)

    # ---- Optional clamp ----
    mfcc = torch.clamp(mfcc, -2.0, 2.0)

    return mfcc
```

In [31]:
```
class CNN_KWS(nn.Module):
    def __init__(self, num_classes=6, flatten_dim=3840):
        super().__init__()
        self.flatten_dim = flatten_dim

        self.features = nn.Sequential(
            nn.Conv2d(1, 8, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(8, 16, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )

        self.classifier = nn.Sequential(
            nn.Linear(self.flatten_dim, 64),
            nn.ReLU(),
            nn.Linear(64, num_classes),
        )

    def forward(self, x):
        x = x.unsqueeze(1)           # [B,1,40,T]
        x = self.features(x)
        x = torch.flatten(x, 1)

        F = x.shape[1]
        if F > self.flatten_dim:
            x = x[:, :self.flatten_dim]           # crop
        elif F < self.flatten_dim:
            pad = self.flatten_dim - F
            x = F.pad(x, (0, pad))                # pad

        return self.classifier(x)
```

In [32]:
```
import snntorch as snn
from snntorch import surrogate

spike_grad = surrogate.fast_sigmoid()

class SNN_KWS(nn.Module):
    def __init__(self, base_cnn: CNN_KWS, num_steps: int = 50, beta: f
        super().__init__()
        self.num_steps = num_steps

        # Reuse trained CNN layers
        self.features = base_cnn.features
        self.fc1 = base_cnn.classifier[0]      # Linear(flatten_dim →
        self.fc2 = base_cnn.classifier[2]      # Linear(64 → num_class

        # Get the expected flatten dimension from fc1
        self.flatten_dim = self.fc1.in_features   # usually 3840 in yo
```

```
        # SNN layers
        self.lif1 = snn.Leaky(beta=beta, spike_grad=spike_grad)
        self.lif2 = snn.Leaky(beta=beta, spike_grad=spike_grad)

    def _fix_feature_dim(self, x):
        """
        Ensure x has shape [B, flatten_dim]
        Crop if too large, pad if too small.
        """
        B, F_in = x.shape

        if F_in > self.flatten_dim:
            return x[:, :self.flatten_dim]

        elif F_in < self.flatten_dim:
            pad = self.flatten_dim - F_in
            return F.pad(x, (0, pad))

        else:
            return x

    def forward(self, x):
        """
        x: [B, 40, T]
        Output: [T, B, num_classes]
        """
        spk2_rec = []
        mem1 = self.lif1.init_leaky()
        mem2 = self.lif2.init_leaky()

        x = x.unsqueeze(1)  # [B,1,40,T]

        for _ in range(self.num_steps):

            # CNN feature extractor
            cur = self.features(x)
            cur = torch.flatten(cur, 1)  # [B, F_in]

            # Fix dimension mismatch
            cur = self._fix_feature_dim(cur)  # -> [B, flatten_dim]

            # Fully connected → LIF → fc2 → LIF
            cur = F.relu(self.fc1(cur))
            spk1, mem1 = self.lif1(cur, mem1)

            cur2 = self.fc2(spk1)
            spk2, mem2 = self.lif2(cur2, mem2)

            spk2_rec.append(spk2)

        return torch.stack(spk2_rec)  # [T, B, C]
```

In [33]:
```
# %%
cnn_path = MODEL_DIR / "baseline_cnn_kws_vfinal.pt"
snn_path = MODEL_DIR / "snn_kws_model.pt"

print("CNN model file exists:", cnn_path.exists())
print("SNN model file exists:", snn_path.exists())

# Load checkpoint to read original flatten dim
cnn_state = torch.load(cnn_path, map_location=device)
flatten_dim_ckpt = cnn_state["classifier.0.weight"].shape[1]
print("Flatten dim in checkpoint =", flatten_dim_ckpt)

# Rebuild CNN using correct flatten dim
cnn_model = CNN_KWS(num_classes=6, flatten_dim=flatten_dim_ckpt).to(de
cnn_model.load_state_dict(cnn_state)
cnn_model.eval()
print("CNN model loaded successfully.")

# Build SNN model using CNN weights
snn_model = SNN_KWS(cnn_model, num_steps=50, beta=0.95).to(device)

# Load SNN weights if exist
if snn_path.exists():
    snn_state = torch.load(snn_path, map_location=device)
    snn_model.load_state_dict(snn_state)
    print("Loaded SNN weights from snn_kws_model.pt")
else:
    print("No separate SNN weights found; using CNN weights in SNN wra

snn_model.eval()
```

```
CNN model file exists: True
SNN model file exists: True
Flatten dim in checkpoint = 3840
CNN model loaded successfully.
Loaded SNN weights from snn_kws_model.pt
```

Out[33]:
```
SNN_KWS(
  (features): Sequential(
    (0): Conv2d(1, 8, kernel_size=(5, 5), stride=(1, 1), padding=(2,
2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ce
il_mode=False)
    (3): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ce
il_mode=False)
  )
  (fc1): Linear(in_features=3840, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=6, bias=True)
  (lif1): Leaky()
  (lif2): Leaky()
)
```

In [34]:
```
# %%
import random

# restrict to the same keywords you used in training
CLASSES = ["yes", "no", "go", "stop", "down", "up"]

def pick_example(label: str = None) -> Path:
    labels = [d for d in DATA_DIR.iterdir() if d.is_dir() and d.name i
    if label is None:
        label_dir = random.choice(labels)
    else:
        label_dir = DATA_DIR / label
    files = sorted([p for p in label_dir.glob("*.wav")])
    return random.choice(files)

example_path = pick_example("yes")  # or None for random
print("Example file:", example_path)

mfcc = wav_to_mfcc(example_path)  # [40,T]
mfcc_batch = mfcc.unsqueeze(0).to(device)  # [1,40,T]

with torch.no_grad():
    out_TBC = snn_model(mfcc_batch)   # [T,1,C]
    logits = out_TBC.sum(dim=0).squeeze(0)  # [C]
    pred_idx = logits.argmax().item()

print("Predicted class:", CLASSES[pred_idx])
```

```
Example file: /Users/maddy/Desktop/PLEP/Project/CS-576-Final-Project/sa
mple_data/speech_commands_v0.02/yes/2748cce7_nohash_0.wav
Predicted class: up
```

In [35]:
```
# %%
```

```
def extract_cnn_features(x: torch.Tensor, model: CNN_KWS) -> torch.Ten
    """
    x: [B,40,T]
    Returns: [B,64] feature vector before final linear layer.
    """
    with torch.no_grad():
        x_ = x.unsqueeze(1)          # [B,1,40,T]
        h = model.features(x_)
        h = torch.flatten(h, 1)

        # --- FIX FEATURE DIMENSION ---
        flatten_dim = model.classifier[0].in_features  # e.g., 3840
        F_in = h.shape[1]

        if F_in > flatten_dim:
            h = h[:, :flatten_dim]
        elif F_in < flatten_dim:
            pad = flatten_dim - F_in
            h = F.pad(h, (0, pad))

        # first FC + ReLU
        fc1 = model.classifier[0]
        h = F.relu(fc1(h))

    return h  # [B,64]


# Get feature vector for the same example
feat = extract_cnn_features(mfcc_batch, cnn_model)  # [1,64]
feat_np = feat.cpu().numpy().flatten()

# Get fc2 weights and bias
fc2 = cnn_model.classifier[2]
W = fc2.weight.detach().cpu().numpy()  # [6,64]
b = fc2.bias.detach().cpu().numpy()    # [6]

W_T = W  # [64,6]

num_classes = W.shape[0]
print("Feature dim:", feat_np.shape, "| num_classes:", num_classes)
```

```
Feature dim: (64,) | num_classes: 6
```

In [36]:

```
# %%
with nengo.Network(seed=0) as net:

    # 1) Input feature vector (64-dim)
    inp = nengo.Node(feat_np)  # constant for this example

    # 2) LIF Ensemble representing 64 features
    ens = nengo.Ensemble(
        n_neurons=64,
        dimensions=64,
```

```
        neuron_type=nengo.LIF(),    # Loihi-friendly neuron
        max_rates=nengo.dists.Uniform(100, 200),
    )

    # 3) Output node (6 class logits)
    out = nengo.Node(size_in=num_classes)

    # 4) Connect input → ensemble
    nengo.Connection(inp, ens, synapse=None)

    # 5) Connect ensemble.neurons → output using CNN weights
    #    W_T is shape (6,64), perfect for (post=6, pre=64)
    scale = 0.01
    nengo.Connection(
        ens.neurons,
        out,
        transform=W_T * scale,
        synapse=0.005,
    )

    # 6) Probe the output
    p_out = nengo.Probe(out, synapse=0.01)

# --- Run Loihi emulator ---
with nengo_loihi.Simulator(net) as sim:
    sim.run(0.1)  # simulate 100 ms

# --- Extract results ---
logits_loihi = sim.data[p_out][-1]
pred_loihi = int(np.argmax(logits_loihi))

print("Logits (Loihi):", logits_loihi)
print("Predicted class (Loihi):", CLASSES[pred_loihi])
```

```
Logits (Loihi): [-0.44104002 -0.68619463 -0.00359438 -0.45080814 -0.642
58878 -0.19853477]
Predicted class (Loihi): go
```

In [ ]:

In [37]:
```
# %%
import nengo
import nengo_loihi
import numpy as np

def run_loihi_for_feature(feat_vec: np.ndarray, W: np.ndarray, sim_tim
    """
    Run the 64-D feature vector through a tiny Loihi-emulated classifi

    feat_vec: shape [64]
    W:        shape [6, 64]  (same as cnn_model.classifier[2].weight)
    sim_time: simulation time in seconds (e.g., 0.1 = 100 ms)
```

```
    Returns:
        logits_loihi: numpy array [6]
    """
    assert feat_vec.shape == (64,), f"Expected feature shape (64,), go
    assert W.shape[0] == 6, f"Expected W to have 6 rows (classes), got
    assert W.shape[1] == 64, f"Expected W to have 64 columns (features

    num_classes = W.shape[0]

    with nengo.Network(seed=0) as net:
        # 1) Constant input node: always outputs feat_vec
        inp = nengo.Node(output=lambda t: feat_vec)

        # 2) LIF ensemble representing the 64-D feature
        ens = nengo.Ensemble(
            n_neurons=64,
            dimensions=64,
            neuron_type=nengo.LIF(),
        )

        # 3) Output node: 6-D classification logits
        out = nengo.Node(size_in=num_classes)

        # 4) Input → ensemble (identity)
        nengo.Connection(inp, ens, synapse=None)

        # 5) Ensemble neurons → output using W (6 x 64)
        nengo.Connection(
            ens.neurons,
            out,
            transform=W,
            synapse=0.01,
        )

        # 6) Probe output with a small synapse for smoothing
        p_out = nengo.Probe(out, synapse=0.01)

    # Run Loihi emulator
    with nengo_loihi.Simulator(net) as sim:
        sim.run(sim_time)
        logits_loihi = sim.data[p_out][-1]  # last timestep

    # Clean up any numerical weirdness
    logits_loihi = np.nan_to_num(logits_loihi)

    return logits_loihi
```

In [38]:

```
# %%
from typing import Tuple

def eval_loihi_classifier(
    loader,
```

```
    cnn_model: CNN_KWS,
    W: np.ndarray,
    device: torch.device,
    max_samples: int = 50,
    sim_time: float = 0.1,
) -> Tuple[float, float, int]:
    """
    Compare CNN classifier vs Loihi emulator on a subset of the test s

    Returns:
        cnn_acc:    CNN head accuracy on the subset
        loihi_acc:  Loihi classifier accuracy on the subset
        total:      number of samples evaluated
    """
    cnn_model.eval()

    total = 0
    correct_cnn = 0
    correct_loihi = 0

    for mfcc_batch, y_batch in loader:
        mfcc_batch = mfcc_batch.to(device)
        y_batch_np = y_batch.numpy()

        with torch.no_grad():
            # [B,64] features from CNN
            feats = extract_cnn_features(mfcc_batch, cnn_model)
            # CNN head logits [B,6]
            fc2 = cnn_model.classifier[2]
            logits_cnn = fc2(feats)
            preds_cnn = logits_cnn.argmax(dim=1).cpu().numpy()

        batch_size = feats.size(0)

        for i in range(batch_size):
            feat_np = feats[i].cpu().numpy()
            label = int(y_batch_np[i])

            # PyTorch CNN prediction
            if preds_cnn[i] == label:
                correct_cnn += 1

            # Loihi prediction
            logits_loihi = run_loihi_for_feature(
                feat_vec=feat_np,
                W=W,
                sim_time=sim_time,
            )
            pred_loihi = int(np.argmax(logits_loihi))

            if pred_loihi == label:
                correct_loihi += 1
```

```
            total += 1
            if total >= max_samples:
                cnn_acc = correct_cnn / total
                loihi_acc = correct_loihi / total
                return cnn_acc, loihi_acc, total

    cnn_acc = correct_cnn / max(total, 1)
    loihi_acc = correct_loihi / max(total, 1)
    return cnn_acc, loihi_acc, total
```

In [39]:

```
# %%
import torchaudio
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset
from pathlib import Path
import soundfile as sf

DATA_DIR = PROJECT_ROOT / "sample_data/speech_commands_v0.02"

CLASSES = ["yes", "no", "go", "stop", "down", "up"]
SAMPLE_RATE = 16000
N_MFCC = 40

mfcc_transform = torchaudio.transforms.MFCC(
    sample_rate=SAMPLE_RATE,
    n_mfcc=N_MFCC,
    melkwargs={
        "n_fft": 400,
        "hop_length": 160,
        "n_mels": 40,
        "center": False,
    },
)

# ------------------------------
# FIXED: safe MFCC loader (soundfile)
# ------------------------------
def wav_to_mfcc(path: Path) -> torch.Tensor:
    waveform, sr = sf.read(str(path))
    waveform = torch.tensor(waveform).float().unsqueeze(0)

    if sr != SAMPLE_RATE:
        waveform = torchaudio.functional.resample(waveform, sr, SAMPLE

    mfcc = mfcc_transform(waveform).squeeze(0)
    mfcc = (mfcc - mfcc.mean()) / (mfcc.std() + 1e-5)
    return mfcc


# ------------------------------
# FIXED: Dataset now receives file LIST, not folder
```

```
# ------------------------------
class KWS_Dataset(Dataset):
    def __init__(self, file_list, classes):
        self.files = file_list
        self.classes = classes

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

    def __getitem__(self, idx):
        path = self.files[idx]
        mfcc = wav_to_mfcc(path)
        label = path.parent.name
        y = self.classes.index(label)
        return mfcc, y


# ------------------------------
# FIX: Collect all wav files
# ------------------------------
file_list = []
for label in CLASSES:
    folder = DATA_DIR / label
    file_list.extend(sorted(folder.glob("*.wav")))

print("Total WAV files loaded:", len(file_list))


# ------------------------------
# Create test_loader properly
# ------------------------------
test_dataset = KWS_Dataset(file_list, CLASSES)

def pad_collate(batch):
    xs, ys = zip(*batch)
    max_t = max(x.shape[1] for x in xs)
    xs = [F.pad(x, (0, max_t - x.shape[1])) for x in xs]
    xs = torch.stack(xs)
    ys = torch.tensor(ys)
    return xs, ys

test_loader = DataLoader(
    test_dataset,
    batch_size=1,
    shuffle=True,
    collate_fn=pad_collate,
)

print("Loaded test samples:", len(test_dataset))
```

```
Total WAV files loaded: 23377
Loaded test samples: 23377
```

In [40]:
```
# %%
max_samples = 50   # adjust if you want more / fewer
sim_time = 0.1     # 100 ms per sample

cnn_acc, loihi_acc, total = eval_loihi_classifier(
    loader=test_loader,
    cnn_model=cnn_model,
    W=W,
    device=device,
    max_samples=max_samples,
    sim_time=sim_time,
)

print(f"Evaluated on {total} test samples")
print(f"CNN head accuracy:    {cnn_acc*100:.2f}%")
print(f"Loihi classifier acc: {loihi_acc*100:.2f}%")
```

```
Evaluated on 50 test samples
CNN head accuracy:    10.00%
Loihi classifier acc: 12.00%
```

In [ ]:
```

```