Live Audio Demo (Optional)

This notebook demonstrates how to:

- Record or load a short audio clip containing a spoken keyword.
- Compute MFCC features in the same way as training.
- Run inference with the trained CNN and SNN.

In [1]:
```
# Cell 1: Imports and paths
from pathlib import Path

import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
import soundfile as sf

import torchaudio
import torchaudio.transforms as T
import snntorch as snn
from snntorch import surrogate

PROJECT_ROOT = Path.cwd().resolve()
MODEL_DIR = PROJECT_ROOT / "saved_models"

print("PROJECT_ROOT:", PROJECT_ROOT)
print("MODEL_DIR exists:", MODEL_DIR.exists())

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

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

```
N_MFCC = 40
```

```
PROJECT_ROOT: /Users/maddy/Desktop/PLEP/Project/CS-576-Final-Project_backup
MODEL_DIR exists: True
Using device: cpu
```

In [2]:

```
# Cell 2: Model definitions and loading
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)
        x = self.features(x)
        x = torch.flatten(x, 1)
        Fdim = x.shape[1]
        if Fdim > self.flatten_dim:
            x = x[:, :self.flatten_dim]
        elif Fdim < self.flatten_dim:
            pad = self.flatten_dim - Fdim
            x = F.pad(x, (0, pad))
        return self.classifier(x)

spike_grad = surrogate.fast_sigmoid()
```

```
class SNN_KWS(nn.Module):
    def __init__(self, base_cnn: CNN_KWS, num_steps: int = 50, beta: float = 0.95):
        super().__init__()
        self.num_steps = num_steps
        self.features = base_cnn.features
        self.fc1 = base_cnn.classifier[0]
        self.fc2 = base_cnn.classifier[2]

        self.lif1 = snn.Leaky(beta=beta, spike_grad=spike_grad)
        self.lif2 = snn.Leaky(beta=beta, spike_grad=spike_grad)

    def forward(self, x):
        spk2_rec = []
        mem1 = self.lif1.init_leaky()
        mem2 = self.lif2.init_leaky()

        x = x.unsqueeze(1)

        for _ in range(self.num_steps):
            cur = self.features(x)
            cur = torch.flatten(cur, 1)
            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)
```

In [3]:

```
# Cell 3: Load trained models
cnn_ckpt_path = MODEL_DIR / "baseline_cnn_kws_vfinal.pt"
snn_ckpt_path = MODEL_DIR / "snn_kws_beta0.95_T50.pt"

print("CNN checkpoint exists:", cnn_ckpt_path.exists())
print("SNN checkpoint exists:", snn_ckpt_path.exists())

flatten_dim_ckpt = 3840
cnn_model = CNN_KWS(num_classes=6, flatten_dim=flatten_dim_ckpt).to(device)
```

```
cnn_state = torch.load(cnn_ckpt_path, map_location=device)
cnn_model.load_state_dict(cnn_state)
cnn_model.eval()

snn_model = SNN_KWS(cnn_model, num_steps=50, beta=0.95).to(device)
if snn_ckpt_path.exists():
    snn_state = torch.load(snn_ckpt_path, map_location=device)
    snn_model.load_state_dict(snn_state)
    print("Loaded SNN weights from snn_kws_beta0.95_T50.pt")
else:
    print("SNN checkpoint not found, using CNN weights only.")
snn_model.eval()
```

```
CNN checkpoint exists: True
SNN checkpoint exists: True
Loaded SNN weights from snn_kws_beta0.95_T50.pt
```

Out[3]:

```
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, ceil_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, ceil_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 [4]:

```
# Cell 4: MFCC helper
mfcc_transform = T.MFCC(
    sample_rate=SAMPLE_RATE,
    n_mfcc=N_MFCC,
    melkwargs={
        "n_fft": 400,
        "hop_length": 160,
        "n_mels": 40,
```

```
        "center": False,
    },
)

def wav_to_mfcc(path: Path):
    waveform, sr = sf.read(str(path))
    waveform = torch.tensor(waveform).float().unsqueeze(0)
    if sr != SAMPLE_RATE:
        waveform = torchaudio.functional.resample(waveform, sr, SAMPLE_RATE)
    mfcc = mfcc_transform(waveform).squeeze(0)
    mfcc = (mfcc - mfcc.mean()) / (mfcc.std() + 1e-6)
    mfcc = torch.clamp(mfcc, -2.0, 2.0)
    return waveform.squeeze(0), mfcc
```

In [5]:
```
# Cell 5: Optional live recording using sounddevice
import importlib

have_sd = importlib.util.find_spec("sounddevice") is not None
print("sounddevice available:", have_sd)

if not have_sd:
    print("If you want live recording, install sounddevice:")
    print("  pip install sounddevice")
```

```
sounddevice available: True
```

In [8]:
```
# Cell 6: Manual Start/Stop Audio Recording
import time
import sounddevice as sd
import soundfile as sf
import numpy as np
from pathlib import Path as _Path

def record_audio_manual(fs=SAMPLE_RATE, out_path="live_record.wav"):
    """
    Manual recording:
    - Press Enter to start
    - Press Enter again to stop
    """
```

```
    input("Press ENTER to start recording...")
    print("Recording... Press ENTER again to stop.")

    sd.default.samplerate = fs
    sd.default.channels = 1

    frames = []

    # Callback that stores microphone audio frames
    def callback(indata, frames_count, time_info, status):
        frames.append(indata.copy())

    # Open audio stream
    with sd.InputStream(callback=callback):
        input()  # Wait until user hits ENTER again to stop

    print("Stopping recording...")
    audio = np.concatenate(frames, axis=0)

    sf.write(out_path, audio, fs)
    print(f"Saved recording to {out_path}")
    return _Path(out_path)


# Use manual recorder
if have_sd:
    audio_path = record_audio_manual()
else:
    raise RuntimeError("sounddevice is not installed; use Option B instead.")
```

```
Recording... Press ENTER again to stop.
Stopping recording...
Saved recording to live_record.wav
```

In [9]:

```
# Cell 7: Run inference and visualize
if not audio_path.exists():
    raise FileNotFoundError(
        f"{audio_path} does not exist. Either record with sounddevice or place a file with this nam
    )
```

```
waveform, mfcc = wav_to_mfcc(audio_path)
mfcc_batch = mfcc.unsqueeze(0).to(device)

with torch.no_grad():
    logits_cnn = cnn_model(mfcc_batch)
    pred_cnn = logits_cnn.argmax(dim=1).item()

    out_TBC = snn_model(mfcc_batch)
    logits_snn = out_TBC.sum(dim=0).squeeze(0)
    pred_snn = logits_snn.argmax().item()

print("CNN prediction:", CLASSES[pred_cnn])
print("SNN prediction:", CLASSES[pred_snn])

duration = len(waveform) / SAMPLE_RATE
t = np.linspace(0, duration, len(waveform))

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].plot(t, waveform.numpy())
axes[0].set_title("Waveform")
axes[0].set_xlabel("Time (s)")
axes[0].set_ylabel("Amplitude")

im = axes[1].imshow(mfcc.numpy(), aspect="auto", origin="lower")
axes[1].set_title("MFCC")
axes[1].set_xlabel("Time frames")
axes[1].set_ylabel("MFCC bins")
fig.colorbar(im, ax=axes[1])

plt.tight_layout()
plt.show()
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[9], line 14
     11 logits_cnn = cnn_model(mfcc_batch)
     12 pred_cnn = logits_cnn.argmax(dim=1).item()
---> 14 out_TBC = snn_model(mfcc_batch)
```

```
     15 logits_snn = out_TBC.sum(dim=0).squeeze(0)
     16 pred_snn = logits_snn.argmax().item()

File /opt/miniconda3/envs/cs576/lib/python3.10/site-packages/torch/nn/modules/module.py:1775, in Mo
dule._wrapped_call_impl(self, *args, **kwargs)
   1773     return self._compiled_call_impl(*args, **kwargs)  # type: ignore[misc]
   1774 else:
-> 1775     return self._call_impl(*args, **kwargs)

File /opt/miniconda3/envs/cs576/lib/python3.10/site-packages/torch/nn/modules/module.py:1786, in Mo
dule._call_impl(self, *args, **kwargs)
   1781 # If we don't have any hooks, we want to skip the rest of the logic in
   1782 # this function, and just call forward.
   1783 if not (self._backward_hooks or self._backward_pre_hooks or self._forward_hooks or self._fo
rward_pre_hooks
   1784         or _global_backward_pre_hooks or _global_backward_hooks
   1785         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1786     return forward_call(*args, **kwargs)
   1788 result = None
   1789 called_always_called_hooks = set()

Cell In[2], line 57, in SNN_KWS.forward(self, x)
     55 cur = self.features(x)
     56 cur = torch.flatten(cur, 1)
---> 57 cur = F.relu(self.fc1(cur))
     58 spk1, mem1 = self.lif1(cur, mem1)
     59 cur2 = self.fc2(spk1)

File /opt/miniconda3/envs/cs576/lib/python3.10/site-packages/torch/nn/modules/module.py:1775, in Mo
dule._wrapped_call_impl(self, *args, **kwargs)
   1773     return self._compiled_call_impl(*args, **kwargs)  # type: ignore[misc]
   1774 else:
-> 1775     return self._call_impl(*args, **kwargs)

File /opt/miniconda3/envs/cs576/lib/python3.10/site-packages/torch/nn/modules/module.py:1786, in Mo
dule._call_impl(self, *args, **kwargs)
   1781 # If we don't have any hooks, we want to skip the rest of the logic in
   1782 # this function, and just call forward.
   1783 if not (self._backward_hooks or self._backward_pre_hooks or self._forward_hooks or self._fo
```

```
rward_pre_hooks
   1784         or _global_backward_pre_hooks or _global_backward_hooks
   1785         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1786     return forward_call(*args, **kwargs)
   1788 result = None
   1789 called_always_called_hooks = set()

File /opt/miniconda3/envs/cs576/lib/python3.10/site-packages/torch/nn/modules/linear.py:134, in Lin
ear.forward(self, input)
    130 def forward(self, input: Tensor) -> Tensor:
    131     """
    132     Runs the forward pass.
    133     """
--> 134     return F.linear(input, self.weight, self.bias)

RuntimeError: mat1 and mat2 shapes cannot be multiplied (1x72640 and 3840x64)
```

In [ ]: