KEYWORDS SPOTTER FOR EDGE

Scarpino Luciano

1. Methodology

I design a compact KWS pipeline: audio is standardized (mono, 16 kHz, fixed 1 s window), converted to MFCCs by an ONNX **frontend**, and classified by a lightweight CNN exported as a separate ONNX graph. Hyperparameters are selected in two steps: (i) grid search to find a good accuracy region, and (ii) targeted manual refinement to satisfy deployment constraints (size + latency). The selected checkpoint is the one with best validation accuracy.

2. Feature Extraction Hyperparameters

MFCC choices affect **latency** mainly through (i) how many frames are computed per second and (ii) how expensive each frame is. For a 1 s signal (T), the number of frames is approximately $N \approx \lfloor \frac{T-L}{H} \rfloor + 1$ where L is the frame length and H the frame step (hop). Smaller H (more overlap) increases N, which increases MFCC compute and also increases the CNN compute because the time axis becomes longer. In my final setup I enforce $L = H$ (no overlap), which keeps the feature sequence short and avoids unnecessary redundancy. Per-frame cost is driven by $n_{fft} = L \cdot f_s$ (here $f_s = 16$ kHz): longer frames increase FFT cost and can increase the ONNX frontend size due to larger constant matrices (mel filterbank and DCT). n_{mels} and n_{mfcc} increase feature richness and robustness, but also increase frontend compute and typically increase frontend size (bigger mel/DCT matrices) and CNN compute (more frequency bins). Therefore, feature extraction is a three-way trade-off: **temporal resolution** (H), **spectral resolution** (n_{mels}, n_{mfcc}), and **compute/size** (frontend constants + downstream MACs). I set $L = H = 0.032$ s (512 samples), producing **31 frames** per 1 s. This improves temporal resolution compared to coarser hops (useful for short commands) while keeping compute manageable thanks to (i) no overlap and (ii) a carefully scaled CNN. Since smaller L reduces per-frame spectral detail, I increase $n_{mels} = 20$ **and** $n_{mfcc} = 20$ to preserve discriminative information despite short windows. I choose $f_{min} = 40$ **Hz** to attenuate very-low-frequency components (rumble/DC-like energy) that are common in real recordings and tend to harm calibration; $f_{max} = 6000$ **Hz** keeps most speech information while discarding high-frequency bands that are less informative for these two commands and may add noise sensitivity.

Parameter	Value
Sampling rate	16000 Hz
Frame length (L)	0.032 s (512 samples)
Frame step (H)	0.032 s (512 samples)
Frames per 1 s	31
Num mel bins	20
Num MFCC	20
$f_{\min}$ / $f_{\max}$	40 Hz / 6000 Hz

3. Training Hyperparameters

I train with Adam and cosine annealing for 30 epochs. Robustness is improved by **waveform-level augmentation** before MFCC: random time shift, small gain variation, and additive noise with controlled SNR. These choices specifically target **confidence stability**: the deployed system triggers **up** only when the model is very confident, so maintaining a high **pass@0.999 (correct)** requires reducing sensitivity to small real-world perturbations. I therefore use conservative augmentation ranges that diversify the data without destroying the keyword structure: shift up to 30 ms (alignment jitter), gain 0.97–1.03 (microphone variability), and noise with SNR 35–55 dB (mild background), applied with probability 0.6 when augmentation is active. BatchNorm is frozen at epoch 6 to reduce late-epoch instability on the small dataset.

Parameter	Value
Epochs	30 (best ckpt by val_acc)
Batch size	64
Optimizer	Adam
Learning rate	1e−3
Weight decay	0
Scheduler	CosineAnnealing (η_{min} = 1e−5)
Loss	CrossEntropy
Augmentation	p=0.20, shift 30 ms, SNR 35–55 dB, gain 0.97–1.03, noise prob 0.6
BN freeze	at epoch 6
Seeds	torch/numpy/python = 0

Table 2: Final training setup.

4. Model Architecture and Quantization

The input MFCC tensor is processed by a compact CNN: a 3×3 stem conv (stride (2, 1) to downsample frequency only), five **depthwise-separable residual blocks**, global average pooling, and a final linear classifier. A key manual knob is the **width multiplier** (w), which scales the channel count $c_0 = \max(16, \lfloor 256w \rfloor)$. This is a **gamechanger** for the latency–accuracy trade-off: channel width affects almost all convolutions, so **model latency and parameter size grow roughly with c_0^2** for pointwise (1×1) layers, while accuracy typically saturates beyond a certain width. I set $w = 0.28$ to keep inference fast while still providing enough capacity to exploit the richer MFCC features.

I export **frontend** and **model** as separate ONNX graphs and apply static INT8 quantization (QDQ) with entropy calibration and per-channel weights. Quantization reduces size and can improve speed; it may slightly affect confidence calibration, so I also monitor pass@0.999.

5. Results and Discussion

The selected checkpoint achieves **99.50%** test accuracy. The Float32 ONNX footprint is **198.2 KB** (frontend **40.5 KB**

+ model **157.7 KB**). After static INT8 quantization, accuracy remains **99.50%** while the model shrinks to **84.1 KB** and the total to **124.6 KB**. Latency (approx.) is **5.0 ms** end-to-end (**3.4 ms** model + **1.5 ms** frontend).
I also report **pass@0.999 (correct)**: fraction of test samples correctly classified with top softmax probability > 0.999 (relevant because `up` is triggered only above this threshold). PyTorch results are **0.28**; ONNX results are **0.28** (Float32) and **0.275** (INT8), showing a minor confidence drop after quantization while preserving top-1 accuracy.

Format	Acc (%)	Total ONNX (KB)	Latency (ms)
Float32	99.50	198.2	-
INT8	99.50	124.6	5.0

Table 3: Final ONNX metrics (frontend + model). Latency is approximate.