Embedded Artificial Intelligence

SETI Master

Lab 5 — End-to-End Frugal AI on the KV260

Pruning, knowledge distillation, PTQ and VART

Author: Alaa Eddine Mazouz

Academic year: 2026–2027

Version: 13.11

Date: 26 October 2026

Télécom Paris

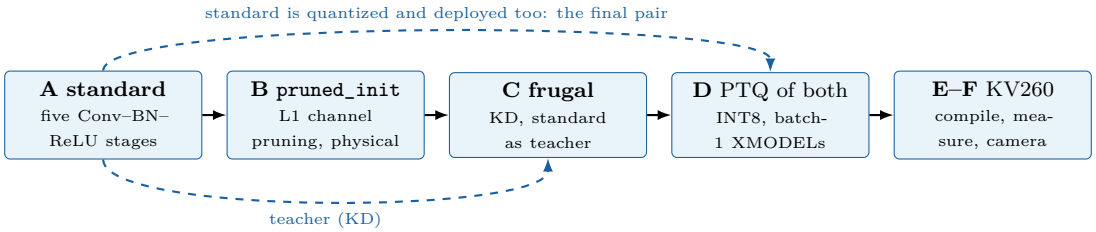
Work in the lab folder `TP05_frugal_kv260/`, which contains `scripts/` and `src/`.

Host shell. Type `bash` once in every new host terminal before the course commands that use `source`; the teaching hosts default to zsh. Commands launched with `bash script.sh` already select Bash.

Raw datasets. Raw data are shared in the course-root `datasets/` folder; a standalone lab uses `$HOME/.cache/c5/datasets` (or `$XDG_CACHE_HOME/c5/datasets`). Set `C5_DATA=/absolute/path` before sourcing `scripts/00_source_env.sh` to override it. Existing datasets stay in place; experiment outputs stay in this lab.

What you will produce

Two deployable models of a rock–paper–scissors classifier: **standard**, a five-stage DPU-friendly CNN, and **frugal**, a physically channel-pruned copy of standard recovered with knowledge distillation (KD), standard being the teacher. Both are quantized (PTQ), compiled, measured on the KV260 and used in a live USB-camera demo. The question is not only "how much smaller is frugal?" but "what does the smaller model actually buy on this board, and why?". The ideas (where the time goes in an application, the preprocessing contract and measurement scope) are in L02 and the course tools guide; pruning, KD and PTQ are those of Labs 2–4.



Plan

Part	Content
0	setup and data
A–D	train standard, prune it, recover frugal with KD, quantize both (reference code)
E–F	compile, copy, measure on the KV260; live demo
—	discussion of the final pair
Your code	the four parts A–D in `your_code/`, checked, then the pipeline again with your code

Parts A–F run the provided reference code of `src/`: the pipeline never waits for your code. The code you write comes last, in `your_code/`, where it replaces the reference parts one by one. `pruned_init`, the model between pruning and KD, is not a third final model: the final comparison contains standard and frugal only.

Model contract

Fixed	RGB 128×128 input; 3 logits (rock, paper, scissors); five sequential Conv–BN–ReLU stages; strides $[2, 2, 2, 2, 1]$; average pooling to 1×1; linear classifier.
Adjustable	the five channel widths (`configs/standard.json`); training settings; frugal `keep_ratio`; KD temperature and CE weight.

Defaults: widths `[16,32,48,64,96]`; `keep_ratio` 0.3 gives `[8,8,16,16,32]` (multiples of eight), about 11× fewer parameters and 7× fewer MACs.

0 – Setup and data

The dataset is the public Rock–Paper–Scissors set (computer-generated hands on a white background): 2520 training images, 15 % of them held out for validation, and a separate test set of 372 images, evaluated once the two models are frozen. Its two archives (about 230 MB) are downloaded once, into the dataset folder of the course (`datasets/rps/` next to the lab folders, as in the previous labs), or taken from the course's shared copy when the lab machine has one. From the Lab 5 folder, on the host:

```
# host, in the Lab 5 folder
source scripts/00_source_env.sh
bash scripts/enter_container.sh
```

`enter_container.sh` opens the Vitis AI container as in the previous labs and checks that `torch` and `pytorch_nndct` import. Every host command of this lab then runs in this container shell, the board-transfer scripts (`01b`, `F2`, `G`, `H`) included:

```
# container
bash scripts/01_check_setup.sh
bash scripts/02_prepare_dataset.sh
```

A to D – Standard, frugal, and their INT8 versions

```
# container
bash scripts/A2_train_standard.sh # standard, selected on validation
bash scripts/B2_create_pruned.sh # pruned_init: L1 channel pruning, physical
bash scripts/C2_train_frugal.sh # frugal: pruned_init recovered with KD
bash scripts/D2_quantize_models.sh # PTQ of both models
```

The JSON files remain the training defaults. For an explicit alternative budget, prefix `A2` or `C2` with `EPOCHS`, `BATCH_SIZE`, `LR`, `SEED` or `DEVICE`, for example:

```
EPOCHS=12 BATCH_SIZE=64 LR=0.0005 SEED=123 bash scripts/A2_train_standard.sh
```

Python trainers also accept `--epochs`, `--batch-size`, `--lr`, `--seed` and `--device`. Only specified values override JSON; its source is unchanged. The output `effective_config.json` records the resolved settings. These phases retain fixed budgets and validation-macro-recall checkpoint selection; record actual epochs when comparing runs.

- **A.** The CNN of `src/tp05_models.py`, trained with the widths of `configs/standard.json`.
- **B.** Each output filter j of a convolution is ranked by $s_j = \sum_{c,u,v} |W_{j,c,u,v}|$ (L1, L03), the strongest are kept, and their sorted indices slice the Conv rows, the BatchNorm vectors, the next Conv's input channels and the classifier's columns (`src/structured_pruning.py`): the tensors become narrower, not masked. With `keep_ratio=0.3`, expect `pruned_init` to collapse (often to one predicted class).
- **C.** KD recovery (L04) with standard as frozen teacher, $p_t = \mathrm{softmax}(z_t/T)$, $p_s = \mathrm{softmax}(z_s/T)$: $L = w_{CE}\, L_{CE}(z_s, y) + (1 - w_{CE})\, T^2\, D_{KL}(p_t \,\|\, p_s)$, with $w_{CE} =$ `ce_weight` of `configs/frugal.json` (0.5).
- **D.** PTQ of both models (Lab 2): calibration on training images, INT8 test on the test set, batch-1 XMODEL.

The two float models are now fixed: `work/outputs/standard/float_model.pt` and `work/outputs/frugal/float_model.pt`.

E – Compile for your KV260

The board is given as in Lab 2 (`BOARD`: its network name, such as `maquette-07`, or its IP address; account `root`). The DPU fingerprint is read from your board before compiling:

```
# container
export BOARD=maquette-07 # the name of your board, once per terminal
bash scripts/01b_sync_arch_from_board.sh
bash scripts/E_compile_models.sh
```

The compiled models are `work/compiled/standard/standard.xmodel` and `work/compiled/frugal/frugal.xmodel`, copied to `xmodels/` in the board folder.

F – Copy, evaluate and benchmark on KV260

```
# container
bash scripts/F1_prepare_board_bundle.sh
bash scripts/F2_copy_to_board.sh
```

If changing `TP05_BOARD_DIR`, set it before F1 and keep it for F2 and result collection: F1 embeds that destination in `setup_env.sh`. Destination on the board: `/home/root/c5/TP05`.

On the KV260 (`ssh root@${BOARD#*@}`, `BOARD` without a `user@` part as in Lab 2):

```
# KV260
cd /home/root/c5/TP05
source ./setup_env.sh
bash board/check_board.sh
bash board/run_all_board_variants.sh
```

This command evaluates the full public test set and runs an **uncapped in-memory benchmark**. Camera capture, display, JPEG encoding and intentional sleeps are excluded.

What the speed numbers measure

For each model, `board/benchmark_vart.py` writes three average timings to `results/<model>_kv260_benchmark.json`:

- `avg_input_quant_ms`: Python conversion of the float image to INT8 on the Arm CPU;
- `avg_dpu_ms`: `execute_async` + `wait`, i.e. the DPU run plus the runtime dispatch, what Lab 4 called the VART call (**DPU FPS** = 1000/`avg_dpu_ms`; separate hardware execution time with the profiler, see question 3);
- `avg_in_memory_pipeline_ms`: input quantization + DPU + output dequantization on a preloaded image, without camera, decoding or resizing (**runtime FPS**).

Use these numbers for the model comparison. The **live FPS** of the camera demo describes the application (capture, crop, JPEG, browser) and includes camera and application costs; it is not a model metric.

Live demo: choose one model or compare both

One launcher, the model named explicitly:

```
# KV260, in /home/root/c5/TP05
bash board/run_demo.sh standard
bash board/run_demo.sh frugal
bash board/run_demo.sh compare
```

Open `http://maquette-07:8080/` (the name of your board, or its IP address). The browser shows a clean dashboard above the camera image. For each model, **DPU FPS** is displayed directly next to DPU latency. **Live FPS** is shown separately because the camera/application rate can remain similar even when DPU FPS differs.

Camera/display options are passed on the same command line. For example:

```
# KV260
bash board/run_demo.sh compare --camera-width 1280 --camera-height 720 \
  --stream-width 960 --roi-scale 0.55 --jpeg-quality 80
```

The classifier has only rock, paper and scissors classes, with no background class: an empty scene can still receive a high softmax score. Test a real gesture in the ROI; a high score alone does not establish recognition quality.

Useful options are `--camera-width`, `--camera-height`, `--camera-fps`, `--stream-width`, `--roi-scale` and `--jpeg-quality`. `--camera-fps 0` means that the lab code does not request an FPS cap; the camera/USB mode may still impose one. Camera resolution affects capture/display cost, but the compiled network input remains **128x128**: the selected ROI is resized before DPU inference.

To measure the live behaviour instead of judging it by eye, hold one gesture in the square and record a fixed number of frames with `standard` or `frugal` (the `compare` mode records no histogram); the JSON contains a `prediction_histogram`:

```
# KV260
bash board/run_demo.sh frugal --frames 300 --output-json results/live_frugal_rock.json
```

Final result and discussion

At the end of `board/run_all_board_variants.sh`, the KV260 itself prints a table containing **INT8 accuracy, DPU ms, DPU FPS and in-memory runtime FPS** for standard and frugal. You therefore see the board numbers immediately.

Back in the container shell, with `BOARD` still set, run:

```
# container
bash scripts/G_summarize_results.sh
```

The script fetches the four KV260 result JSON files and prints the complete final report. It also writes `work/final_summary.md` and `work/final_summary.json`. A **final** report is never printed with empty KV260 columns: if the board results cannot be found, the script stops and tells you what to run or copy.

Accuracy can vary by several percentage points across seeds and recovery runs; compare the recorded split, seed and configuration rather than expecting a fixed reference score.

Discuss only the final pair:

1. How much smaller is frugal in parameters, MACs and XMODEL size? What FP32 accuracy is lost? What does PTQ change? Does KV260 INT8 accuracy agree with the host?

2. **Where does the time go?** For each model, compare `avg_input_quant_ms`, `avg_dpu_ms` and `avg_in_memory_pipeline_ms`. Is the DPU speedup of frugal proportional to its MAC reduction?

3. Estimate the pure compute time of each model on the DPU: the B4096 core performs 2048 MAC per cycle at 300 MHz. Compare with `avg_dpu_ms`. What does the difference say about fixed costs (dispatch, instruction and weight loading, synchronisation) for models this small? At which model size would you expect pruning to pay off in latency?

4. Which stage would you optimize first to raise the runtime FPS? Hint: the camera delivers `uint8` pixels; how many different INT8 input values can the network actually receive?

5. Does the live demo recognise *your* hand as reliably as the test accuracy suggests? What differs between the public images and the camera view?

6. Which model would you deploy, and why? Name what frugal buys on this board (memory, XMODEL size, headroom for other tasks, energy per inference, estimated only from SOM power and throughput measured over the same workload interval; the power tools of Labs 2–4 exclude the carrier and USB camera) and what it does not.

Your code: the four parts, then the pipeline again

Implement the four parts in `your_code/`: they are the files of `src/` with their core removed. The checks test only the behaviour the pipeline needs, not a reference parameter count or a coding style (Part B uses its own synthetic model sizes; Part D a fake quantizer).

Part	File	What you write	Check
A	`your_code/tp05_models.py`	TODO A1–A3: `ConvBNReLU` and the model; keep the attributes `conv`, `bn`, `relu` (Part B uses them)	`A1`
B	`your_code/structured_pruning.py`	`select_output_channels()`, `copy_pruned_weights()`: the ranking and the physical slicing above, indices sorted	`B1`
C	`your_code/train_frugal_recovery.py`	`kd_loss()`: the loss of Part C (student log-softmax, teacher softmax, `batchmean`, T^2, the weights)	`C1`
D	`your_code/quantize_nndct.py`	the NNDCT quantizer, the calibration or test data per mode, the export	`D1`

```
# container
bash scripts/A1_check_model.sh # then B1, C1, D1 (each tests your_code/)
bash scripts/check_all_todos.sh # the four checks
bash scripts/I_use_your_code.sh # your files replace the reference ones in src/
```

Then run the pipeline again with your code (`A2`, `B2`, `C2`, `D2`, `E`, `F1`, `F2`, the board, `G`; a few minutes of CPU time for these small models) and compare your pair with the reference pair: before replacing the code, `I_use_your_code.sh` keeps the results of the reference pipeline in `work/reference_run/` (compare `work/final_summary.md` with `work/reference_run/final_summary.md`). `bash scripts/I_use_your_code.sh --restore` puts the reference code back.

Validation and test. With equally many examples per class, macro recall equals accuracy on each split. A validation–test gap therefore cannot be explained by the metric names alone. Validation is sampled from the public training archive; the official test archive is separate. Report both confusion matrices and keep test data out of model selection. To intentionally regenerate an existing split, use `OVERWRITE\_DATA=1 bash scripts/02\_prepare\_dataset.sh`.

Optional – adapt the model to your camera

The public images are computer-generated hands on a white background; the camera sees real hands, real light and a real background. If the live demo is unreliable, collect a small training set with the deployment camera itself.

On the KV260 (stop the live demo first: it uses the same port):

```
# KV260
cd /home/root/c5/TP05
source ./setup_env.sh
```

```
python3 board/capture_rps.py --session s1
```

Use the same camera and crop settings as for the live demo (`--camera-width`, `--camera-height`, `--roi-scale`; the defaults are the same in both tools). Open `http://maquette-07:8080/` to see the camera and the capture square (the exact crop that the network receives), then follow the terminal: one class at a time, press Enter, show the gesture and keep moving the hand (position, rotation, distance, angle). About 40 images per class are enough to see an effect; diversity matters more than the count. Several group members, sleeves and backgrounds help even more (use one `--session` name per capture run).

Back in the container shell on the host:

```
# container
bash scripts/H_fetch_camera_captures.sh
OVERWRITE_DATA=1 bash scripts/02_prepare_dataset.sh
```

The camera images are added to **TRAIN only**: validation and the public test set are unchanged, so the before/after test accuracies remain comparable. Then run the same pipeline again (`A2`, `B2`, `C2`, `D2`, `E`, `F1`, `F2`; a few minutes of CPU time for these small models) and compare the live behaviour before and after with the `--frames`/`--output-json` measurement above. Which changed more: the public test accuracy or the live accuracy? What does this say about the data a frugal model must be trained on?

Optional optimization if you finish early

You may change the standard widths, `keep_ratio`, T or w_{CE}, then rerun the same pipeline. The definition does not change: **frugal is always physically pruned from your trained standard model and then KD-recovered**. Keep only your final standard/frugal pair for submission. Two directions connect directly to the discussion: make the input quantization of `board/vart_common.py` cheaper (for instance a 256-entry lookup table from `uint8` pixels to INT8 values) and measure the runtime FPS again; or train a wider standard model and check from which size the frugal DPU latency starts to follow the MAC reduction.