Embedded Artificial Intelligence

SETI Master

Lab 3 — Neural Network Pruning for Frugal Embedded AI

From pruning families to a deployable pruning policy

Author: Alaa Eddine Mazouz

Academic year: 2026–2027

Version: 1.22

Date: 26 October 2026

Télécom Paris

Work in the lab folder `TP03_pruning/`, which contains `scripts/` and `src/`.

Host shell. In each new host terminal, type `bash` and press Enter separately. Wait for its prompt before pasting the commands below; the teaching hosts default to zsh. Commands launched with `bash script.sh` already select Bash. The initial `cd TP03_pruning` assumes the course checkout root.

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.

Lab 3 in brief

Lab 3 runs two experiments, each with its own baseline. **Experiment A** (Parts 1–2) prunes the small CIFAR-10 CNN of Lab 2 (trained again here, about 58 % accuracy) with policies you choose, fine-tunes two candidates and measures them on the KV260: small enough to see what each pruning family does to the graph. **Experiment B** (Part 3) lets the Vitis AI Optimizer choose the layer widths of VGG6-BN, a larger network (about 88 %), and measures its candidates the same way. A pruned model is compared only with the baseline of its own experiment. The theory (granularity, criteria, layer-wise and global allocation, what the DPU does with zeros, `vaitrace`) is in the L03 slides; the container, PTQ, compilation and board access work as in Lab 2.

Part	Content	Steps
1	Experiment A: pruning policies on the small CNN, all from the same float checkpoint	1–4
2	Experiment A: fine-tune two candidates; deploy and measure five models on the KV260	5–9
3	Experiment B: automatic pruning and NAS of VGG6-BN with the Vitis AI Optimizer; reading the results of both experiments	A1–A6
4	Final task: your own pruning policy, deployed	—

Parts 1 to 3 only run provided scripts: the code you write is in Part 4, at the end, so it never blocks the pipeline. Note each result in its table as it is printed; answer the questions while the next command runs, or at the end (every result stays in `out/runs/${TAG}/`). The QCM at the end is for revision.

Before you start

```
# host
cd TP03_pruning
source scripts/00_source_env.sh
bash scripts/02_enter_container.sh          # ends with: [TP03] Container ready
```

Wait for the container-ready message, then run this in the container:

```
# container
export TAG=exp_pruning
```

- CIFAR-10 is not downloaded again: `00_source_env.sh` links `data/` to the dataset folder of the course (`datasets/` next to the lab folders, or the course's shared copy on the lab machines).
- Every script takes the TAG as its first argument. A new terminal or container shell does not know `TAG`: run `export TAG=exp_pruning` in each one. A script that finds no float model for its TAG stops and lists the TAGs that have one.

- Every candidate starts from the same `model_float.pth`: never retrain the baseline between policies.
- A pruning run replaces the current candidate of its family, in `prune_unstructured/` or `prune_structured/` under `out/runs/${TAG}/`; the previous one, with its fine-tuning, moves to `prune_<family>.previous/`, so that its fine-tuned model is never taken for the new one. The script `03g_save_candidate.sh` keeps a copy in `policy_study/` and prints its row (validation accuracy, widths, parameters, nonzeros, sparsity, MACs); with the TAG alone, it lists every saved row.
- Decisions (best epoch, policy, keep ratio) use the validation split held out from the training set (seed in `out/runs/${TAG}/meta.json`); the test accuracy is only reported.

Part 1 — Experiment A: pruning policies on the small CNN

Step 1 — Train the float baseline

```
# container
DATASET=cifar10 ARCH=cnn WIDTH=1 EPOCHS=5 bash scripts/03_train_float.sh ${TAG}
cat out/runs/${TAG}/float/summary_float.txt
```

Record the validation accuracy. The network: `conv1` 3 → 8, `conv2` 8 → 16 (each followed by a 2×2 max-pool), `fc1` 1024 → 64, `fc2` 64 → 10; 67,642 parameters (97 % in `fc1`) and 582,272 MACs (89 % in the convolutions).

Exercise 1 — Find the pruning code (while the model trains)

In `src/tp03_prune.py`, find (1) where the unstructured masks are created and kept for fine-tuning, (2) the L1, L2 and random channel scores, (3) the normalization of the scores before the global ranking, (4) where the kept `conv1` channels select the inputs of `conv2` and the kept `conv2` channels the columns of `fc1`. For each, say what would go wrong without it.

```
grep -n "^def " src/tp03_prune.py              # every function, with its line
```

Step 2 — Unstructured pruning: strength

L1 magnitude pruning of the weights of every `Conv2d` and `Linear` layer, at three amounts:

```
# container
for A in 0.20 0.50 0.80; do
  bash scripts/03b_prune_unstructured.sh ${TAG} ${A} && \
  bash scripts/03g_save_candidate.sh ${TAG} unstructured U_${A}
done
```

Amount	Val. acc.	Params	Nonzeros	Sparsity	MACs
0.20					
0.50					
0.80					

Exercise 2 — Zeros are not fewer operations

1. Where does the accuracy start to collapse? Which amount would you fine-tune?
2. Why do the parameter count and the dense MACs not change?

Step 3 — Structured pruning: criterion and allocation

Five policies at the same mild keep ratio, 0.90, each differing from another by one decision. The layer-wise script keeps the same fraction of every layer; the global script ranks the channels of both convolutions together, after dividing the scores of each layer by their mean.

```
# container
CRITERION=l1 bash scripts/03c_prune_structured.sh ${TAG} 0.90
bash scripts/03g_save_candidate.sh ${TAG} structured S1_l1_layerwise
CRITERION=l2 bash scripts/03c_prune_structured.sh ${TAG} 0.90
bash scripts/03g_save_candidate.sh ${TAG} structured S2_l2_layerwise
CRITERION=l1 bash scripts/03c_prune_structured_global.sh ${TAG} 0.90
bash scripts/03g_save_candidate.sh ${TAG} structured S3_l1_global
CRITERION=l2 bash scripts/03c_prune_structured_global.sh ${TAG} 0.90
bash scripts/03g_save_candidate.sh ${TAG} structured S4_l2_global
CRITERION=random bash scripts/03c_prune_structured_global.sh ${TAG} 0.90
bash scripts/03g_save_candidate.sh ${TAG} structured S5_random_global
```

ID	Criterion	Allocation	Val. acc.	C_1 / C_2	Params	MACs	MAC cut
S1	L1	layer-wise					
S2	L2	layer-wise					
S3	L1	global					
S4	L2	global					
S5	random	global					

C_1 and C_2 are the channels kept in `conv1` and `conv2`; the kept indices are in the file `meta_pruned.json` of each saved folder.

Exercise 3 — Compare the policies

1. Do L1 and L2 keep the same channels, or only the same number of channels?
2. How does the global allocation share the channels between the two layers? Is one layer over-pruned?
3. Does magnitude pruning beat the random control (S5) before fine-tuning?
4. Which candidates are non-dominated in the accuracy–MACs plane?

Step 4 — Strength of the best structured policy

Take your best magnitude policy of Step 3 (here global L1) to two stronger points:

```
# container
for K in 0.70 0.50; do
  CRITERION=l1 bash scripts/03c_prune_structured_global.sh ${TAG} ${K} && \
  bash scripts/03g_save_candidate.sh ${TAG} structured l1_global_${K}
done
bash scripts/03g_save_candidate.sh ${TAG}          # every saved row
```

With the 0.90 point of Step 3, find where the MACs fall at a limited accuracy cost and where the accuracy collapses, and choose the keep ratio to fine-tune: fine-tuning does not undo a collapse, so do not pick the smallest MACs for their own sake.

Optional, channel multiples. `ROUND_TO_MULTIPLE=4` rounds the kept counts to multiples of 4:

```
# container
CRITERION=l1 ROUND_TO_MULTIPLE=4 \
  bash scripts/03c_prune_structured_global.sh ${TAG} 0.70
```

Compare the widths with the unconstrained 0.70 point. Why is the keep ratio no longer 70 %? With 8 and 16 channels, does the constraint leave enough choices to be useful?

Part 2 — Experiment A: fine-tune, deploy, measure

Step 5 — Freeze two policies and fine-tune them

Run your two choices again, so that the standard folders hold the candidates that will be deployed (replace the values with yours), then fine-tune them (3 epochs, best epoch on validation; the unstructured masks stay enforced):

```
# container
bash scripts/03b_prune_unstructured.sh ${TAG} 0.60
CRITERION=l1 bash scripts/03c_prune_structured_global.sh ${TAG} 0.70
bash scripts/03d_finetune_pruned.sh ${TAG} unstructured
bash scripts/03d_finetune_pruned.sh ${TAG} structured
bash scripts/03e_show_summary.sh ${TAG}
```

`prune_<family>/finetune_summary.json` gives `best_val_acc`, `final_test_acc` and, for the unstructured model, `regrown_pruned_weights` (must be 0).

Optional longer recovery. The default is exactly three epochs with `PATIENCE=0`. After saving these results, you may compare a longer recovery from the same pruned checkpoint:

```
EPOCHS=20 PATIENCE=5 MIN_EPOCHS=5 MONITOR=val_loss \
  bash scripts/03d_finetune_pruned.sh ${TAG} unstructured
```

With patience enabled, `EPOCHS` is a maximum; otherwise it is exact. `MONITOR` selects validation accuracy or loss for stopping, but the saved checkpoint always has the best validation accuracy. `MIN_DELTA` is an absolute improvement threshold (accuracy is a fraction, not percent). The summary records `best_epoch`, `epochs_executed`, `stop_reason` and `train_seconds`. The test set never selects the stopping epoch; this optional comparison does not change the required exercises.

Exercise 4 — Did the selection survive fine-tuning?

1. How much accuracy did each family recover? Did the unstructured model keep its sparsity?
2. Could another candidate of your tables recover better? Why is the accuracy just after pruning a useful but imperfect selection signal?

Step 6 — What changes on the CPU

```
INCLUDE_FT=1 bash scripts/03f_profile_cpu.sh ${TAG}
```

Model-only PyTorch inference on one CPU thread (batch-1 latency, batch-32 throughput), also in `out/runs/${TAG}/tp03_cpu_profile.csv`. Use the rows `float`, `unstructured_ft` and `structured_ft`:

Model	Params	Sparsity	Dense MACs	B1 ms/img	B32 img/s
Float		0 %			
Unstructured, fine-tuned					
Structured, fine-tuned		—			

Exercise 5 — Sparsity is not speed

1. Did about 60 % sparsity reduce the dense MACs or the CPU latency? Why?
2. Is the structured speedup proportional to its MAC reduction? Write down what you expect on the KV260.

Step 7 — Quantize and compile five models

```
# container
bash scripts/06_prepare_eval_bundle.sh ${TAG}        # test images for every model
bash scripts/04f_ptq_compile_all_variants.sh ${TAG}
ls -lh out/runs/${TAG}/deploy/*/compile/model.xmodel
```

The bundle holds the whole test set, on which the PTQ accuracy is also measured, so the PTQ and FPGA columns compare the same images. `04f` quantizes and compiles `float_base`, `pruned_unstructured`, `ft_unstructured`, `pruned_structured` and `ft_structured`, one folder each in `out/runs/${TAG}/deploy/` (PTQ accuracy in `model_info.txt`); compilation stops if `arch.json` is not the DPU of the KV260 (fingerprint `0x101000056010407`).

Step 8 — Run on the KV260

```
# container or host
export BOARD=maquette-07                          # the name of your board, as in Lab 2
bash scripts/07b_copy_all_to_board.sh ${TAG}
ssh root@${BOARD#*@}                          # BOARD without a user@ part, as in Lab 2
# KV260
export TAG=exp_pruning                        # the same TAG as in the container
cd /home/root/c5/TP03                         # one folder per model: ${TAG}_<model>/
./run_all_fpga_variants.sh ${TAG}             # accuracy and FPS -> fpga_results.md
for d in ${TAG}_*/; do
  echo "== ${d}"; xdputil benchmark ${d}model.xmodel -i -1 1
done
MEASURE_IDLE=1 ./profile_energy_all_variants.sh ${TAG}    # SOM W and mJ/input
```

Each default `xdputil benchmark` takes about 60 seconds per model; a loop over many variants therefore takes several minutes. Power readings cover the SOM, excluding the carrier and USB camera. The energy helper records the idle baseline, workload samples, interval and counted inputs; idle-subtracted energy is an estimate for that workload, not DPU-only energy.

`run_all_fpga_variants.sh` ends with table A (the five models of Experiment A) and table D (their accuracy on the host and on the board); the columns are explained at the end of Part 3, where both experiments are read together. Note here what the tables do not give: `xdputil` and energy.

Model	Size (KB)	PTQ acc.	FPGA acc.	FPS app	FPS VART	FPS xdputil	Avg W	mJ/ input
float_base								
pruned_unstructured								
ft_unstructured								
pruned_structured								
ft_structured								

Step 9 — What the DPU executes: `vaitrace`

Profile the float baseline and the two fine-tuned models (fields as in Lab 2 and L03; older releases print `RT` and `Perf` instead of `HW_RT` and `Effic`):

```
# KV260, in /home/root/c5/TP03
for m in float_base ft_unstructured ft_structured; do
  (cd ${TAG}_${m} && source ./setup_env.sh && \
   python3 vai_trace_fpga.py --workload_seconds 5 --trace_time 15 -- \
     --xmodel model.xmodel --bundle eval_bundle.npz --bundle_meta bundle_meta.json \
     --model_info model_info.json --warmup 20 && \
   cat vai_trace/profile_summary.txt)
done
```

Model	WL	HW_RT	Effic	LdWB	LdFM	StFM
float_base						
ft_unstructured						
ft_structured						

Exercise 6 — Read the DPU profiles

1. Did unstructured pruning change `WL`, `LdWB` or `HW_RT`? Why?
2. Is the drop of `HW_RT` of the structured model proportional to its drop of MACs?
3. Which measurement explains the application FPS: `HW_RT`, `xdputil`, or the time spent around the DPU? Compare `HW_RT` with the time per input of the application.

Part 3 — Experiment B: automatic pruning and NAS with the Vitis AI Optimizer

Why a second network. The small CNN shows clearly what pruning does to a graph, but it is a poor subject for an automatic method and for the board: 58 % accuracy, two convolutions narrower than the DPU's groups of 16 channels, and so little computation (0.58 M MACs, about 1 µs at the DPU's peak) that every board time is dominated by fixed costs. VGG6-BN (88 %, 38.6 M MACs, six convolutions up to 128 channels) gives the Optimizer layers to choose between and gives the DPU real work. It is a new baseline, not a continuation of Experiment A: compare its candidates with `base`, never with the small CNN.

So far you chose one ratio for every layer, or one global threshold. The Vitis AI Optimizer of the container (`pytorch_nndct`) chooses the width of every layer itself. All its methods remove whole output channels, so its models deploy like your structured ones.

Method	How it chooses the widths	Cost
Iterative	*Sensitivity analysis*: prunes each layer alone (10 to 90 % of its channels) and measures the accuracy; then gives every layer the largest ratio whose loss stays under one common tolerance, adjusted until the MAC target is met	one evaluation per layer and ratio, then fine-tuning
One-step	Samples many sets of per-layer ratios at the MAC target, recalibrates the BatchNorm statistics of each sub-network on a few training batches, keeps the best on validation images	one recalibration and evaluation per sample, then fine-tuning
OFA (NAS)	A *supernet* trained once so that all its sub-networks (other widths) work; an evolutionary search picks the best sub-network for each MAC budget	supernet training (done once, provided); search, short fine-tuning

The network is **VGG6-BN**: six 3×3 Conv–BN–ReLU blocks of widths 32-32-64-64-128-128, a 2×2 max-pool after blocks 2 and 4, global average pooling and one linear layer; 288,746 parameters, 38.6 M MACs (`src/tp03_vaiopt_models.py`). Its trained baseline and supernet take longer than a session on the lab CPUs: they are provided in `provided/vaiopt/`, with their accuracies and training times in `reference.json`. Each candidate is a folder `out/runs/${TAG}/vaiopt/<name>/` (`model.pth`; `meta.json` with widths, MACs and accuracies). Steps A1 to A6 without the optional A4 take about 11 minutes of computation on a lab PC, and A1 to A5 do not use the board: you can start them while Part 2 runs on the KV260.

Step A1 — Install the baseline

```
# container
bash scripts/09a_vaiopt_setup.sh ${TAG}
```

Record the widths, MACs and validation and test accuracies of candidate `base`.

Step A2 — Sensitivity analysis

```
# container
bash scripts/09b_vaiopt_sensitivity.sh ${TAG}      # also vaiopt/sensitivity.png
```

For each convolution, the validation accuracy (2,000 images) when only that layer loses 10, 20, . . . , 90 % of its output channels. Share of the MACs: `conv1` 2.3 %; `conv2`, `conv4`, `conv6` 24.4 % each; `conv3`, `conv5` 12.2 % each.

Exercise 7 — Read the sensitivity curves

1. Which layers lose half of their channels at little cost, and which collapse first?
2. Relate the robust layers to their position and to their share of the MACs.
3. What does a uniform ratio do to the most sensitive layer?

Step A3 — Uniform versus sensitivity-based pruning

Remove 50 % of the MACs in two ways, each followed by three epochs of fine-tuning (channel counts multiple of 8):

```
# container
bash scripts/09c_vaiopt_prune.sh ${TAG} uniform   0.5   # same ratio in every layer
bash scripts/09c_vaiopt_prune.sh ${TAG} iterative 0.5   # per-layer ratios from A2
```

Candidate	Widths	MAC cut	Val. before FT	Val. after FT
`uniform_l1`				
`iterative`				

Exercise 8 — Automatic per-layer ratios

1. Which layers did the iterative method prune more than the uniform policy? Is it consistent with Exercise 7?
2. Compare the requested and the obtained MAC cuts. Where does the difference come from?
3. Which candidate is better before fine-tuning, and after it?

Step A4 — One-step search (optional)

```
# container
bash scripts/09d_vaiopt_search.sh ${TAG} 0.5        # fine-tunes the best of 30
TOP=3 bash scripts/09d_vaiopt_search.sh ${TAG} 0.5  # fine-tunes the 3 best
```

`vaiopt/one_step_search_0.5.csv` lists every sampled sub-network: its score after BatchNorm recalibration, its MACs and its per-layer ratios.

Exercise 9 (optional) — Is the search score a good proxy?

1. Why must the BatchNorm statistics be recalibrated before a pruned sub-network is scored?
2. With `TOP=3`: is the best-scored sub-network still the best after fine-tuning?

Step A5 — Neural architecture search with OFA

The provided supernet covers 25, 50, 75 and 100 % of the widths of `conv2` to `conv5` (multiples of 8; the optimizer keeps the first and last convolutions): 256 sub-networks from 6.8 to 38.6 M MACs (the optimizer prints its own count, which adds the BatchNorm operations: 6.9 to 38.9 M). It was trained with the sandwich rule: at every batch, the largest sub-network learns from the labels, the smallest and two random ones from the largest one's outputs.

```
# container
bash scripts/09e_vaiopt_nas.sh ${TAG}        # budgets: 50 % and 75 % of the MACs
```

The search scores 16 sub-networks, then four generations of 12 mutations and crossovers, each after BatchNorm recalibration on 2,000 validation images, and keeps the accuracy–MACs front (`vaiopt/nas_pareto.{csv,png}`). `ofa_50` is the best point of the front up to 50 % of the largest MACs, `ofa_75` the best between 50 and 75 %; each is extracted as a normal model, recalibrated and fine-tuned for two epochs.

Exercise 10 — Search versus pruning

1. Describe the front. Where do extra MACs stop paying?
2. At about 50 % of the MACs, compare `ofa_50` with `uniform_l1` and `iterative`: accuracy, widths, fine-tuning needed.
3. The supernet is expensive to train (time in `reference.json`) but trained once. When is OFA worth it compared with pruning one model?

Step A6 — Deploy and measure on the KV260

```
# container: PTQ and compilation (about 75 s per model), then vaiopt/summary.md
bash scripts/09f_vaiopt_deploy.sh ${TAG} base uniform_l1 iterative ofa_50
# container or host (test bundle of Step 7, BOARD as in Step 8)
bash scripts/09g_vaiopt_copy_to_board.sh ${TAG}
# KV260, in /home/root/c5/TP03: runs the guided and the automatic models
./run_all_fpga_variants.sh ${TAG}
for d in ${TAG}_vaiopt_*/; do
  echo "== ${d}"; xdputil benchmark ${d}model.xmodel -i -1 1
done
```

Add `one_step` or `ofa_75` to the `09f` command if you have time. The board run now prints table B (Experiment B, every change relative to `base`) and table C (the two baselines) after tables A and D.

Candidate	Widths	MACs (M)	Test acc.	FPGA acc.	FPS app	FPS VART	FPS xdputil
`base`	32-32-64-64-128-128	38.6					
`uniform_l1`							
`iterative`							
`ofa_50`							

Exercise 11 — MACs are not latency

1. Is the gain of `FPS xdputil`, the closest to the DPU's own time, proportional to the MAC reduction?
2. `uniform_l1`, `iterative` and `ofa_50` have similar MACs but different widths. Do they run at the same speed on a DPU that processes 16 input and 16 output channels per cycle?
3. Optional: prune with channel counts multiple of 16, deploy, and compare with `iterative`:

```
# container
DIV=16 NAME=iterative16 bash scripts/09c_vaiopt_prune.sh ${TAG} iterative 0.5
bash scripts/09f_vaiopt_deploy.sh ${TAG} iterative16
bash scripts/09g_vaiopt_copy_to_board.sh ${TAG} iterative16
```

4. The optimizer constrains MACs. What would a search need in order to constrain the latency on this DPU instead?

Reading the results of the two experiments

After Step A6, `run_all_fpga_variants.sh` ends with four tables, also in `fpga_results.md`; `fpga_results.csv` and `fpga_results.json` keep every field of every model.

Table	Content
A	Experiment A, the small CNN: `baseline` (`float_base`), then the unstructured and structured models before and after fine-tuning (FT). Every change is relative to `baseline`.
B	Experiment B, VGG6-BN: `base` and the Optimizer candidates. Every change is relative to `base`.
C	The two baselines side by side, and their ratio: a choice of architecture, not a pruning result.
D	Each model's accuracy as a float checkpoint, after PTQ on the host, and on the KV260, on the same 10,000 test images.

- **Weights stored, weights nonzero, MACs.** Unstructured pruning sets weights to zero inside tensors of unchanged shape: the stored weights, the MACs (counted from the layer shapes) and the XMODEL stay those of the baseline, and the DPU multiplies the zeros. Structured pruning removes whole channels: fewer parameters are stored, and the saving is in the shapes; incidental zeros may remain. The reported parameter and nonzero counts include biases, dense MACs count layer shapes, and CPU latency is a host measurement.
- **Fine-tuning.** The weights left by pruning were trained together with the removed ones; a few epochs let them compensate (the unstructured masks stay enforced). Fine-tuning cannot restore capacity that is gone (Step 4).
- **FPS.** *app FPS* divides the number of images by the sum of the per-image times of the board application: conversion of the `uint8` image into the INT8 input on the Arm, copy, VART call, argmax. *VART FPS* counts the VART call alone (the DPU run and the runtime around it); table C adds the *wall FPS* of the whole Python loop. None of them is the DPU's own time, which `xdputil` and `HW_RT` approach. A speedup divides by the FPS of the baseline of the same table.
- **XMODEL KiB** is the compiled file: the INT8 weights (one byte each), the DPU instructions and metadata.
- **FPGA and PTQ accuracies** that agree mean the same number of correct images, not necessarily the same prediction for every image.

Exercise 12 — Interpret the two experiments

1. **Inside table A.** Compare the MAC cut of `structured + FT` with its app and VART speedups. Convert each FPS into a time per image (1000/FPS ms) and split the app time into the VART call and the rest. Which part did pruning change? Estimate the DPU computation at its peak (2,048 MACs per cycle at 300 MHz) and compare it with the VART time.

2. **Inside table B.** The same for the candidate with the largest MAC cut. Why is the VART speedup larger than in table A, and the app speedup still small?

3. **Across experiments (table C).** VGG6-BN computes about 66 times more MACs than the small CNN and is 30 points more accurate, yet its app FPS is only about 7 % lower. Explain it with your answer to question 1. Why does this compare two architectures and say nothing about pruning?

4. **Is pruning VGG6-BN worth it on this board?** From the measurements, which model would you deploy for (a) a 30 FPS camera, (b) classifying a large set of stored images, (c) a tight memory budget? Under which conditions would pruning VGG6-BN pay off in throughput?

5. **Same accuracy, same predictions?** The host does not keep its PTQ predictions image by image, but the bundle holds those of the float model of Experiment A. Count the test images on which the board and the float model predict different classes:

```
# KV260, in /home/root/c5/TP03
python3 - ${TAG}_float_base/fpga_predictions.csv <<'EOF'
import csv, sys
rows = list(csv.DictReader(open(sys.argv[1])))
n = sum(r["predicted_label"] != r["float_model_predicted_label"] for r in rows)
print(n, "of", len(rows), "images predicted differently")
EOF
```

   How can the two accuracies be within a few tenths of a point while some predictions differ? What would you need to show that the PTQ model and the board predict the same class for every image?

6. **Fair comparisons.** `uniform_l1` is the layer-wise L1 policy of Part 1 (S1) applied to VGG6-BN. Which rows of table B compare a manual policy with the Optimizer fairly? Which comparisons between a row of table A and a row of table B would be unfair, and why?

7. **XMODEL size.** From the XMODEL sizes and the stored weights of tables A and B, estimate the part of each file that is not weights. Why does it not shrink with pruning?

Optional, the same budget for both approaches. `uniform_l1` and `iterative` reached slightly different MAC cuts. To compare them at the same cut, prune uniformly at the cut that `iterative` obtained (the value below is an example), deploy, and add the row to table B:

```
# container
NAME=uniform_matched bash scripts/09c_vaiopt_prune.sh ${TAG} uniform 0.48
bash scripts/09f_vaiopt_deploy.sh ${TAG} uniform_matched
bash scripts/09g_vaiopt_copy_to_board.sh ${TAG} uniform_matched
```

Part 4 — Final task: your own pruning policy

Final task — one policy, implemented by you, on the board

Choose a policy from your measurements, implement it, fine-tune, quantize and compile the model, run it on the KV260, and state the trade-off you obtained for one objective chosen beforehand: accuracy, size, MACs, FPS or energy. Start with structured layer-wise L1/L2 (with the provided rebuild helper) or global unstructured L1; global structured allocation, per-layer ratios and iterative schedules are extensions once one deployment works.

The folder `challenge_frugal_pruning/student/` provides everything (data, evaluation, fine-tuning, PTQ, compilation, copy, board run) except one function of `prune_and_finetune.py`:

```
def apply_my_pruning_policy(dense_model, args):
    # 1. compute importance scores
    # 2. select the weights or channels to keep
    # 3. prune, or rebuild the smaller model
    # 4. return the model, its family and a description of the policy
    ...
```

Its `README.md` gives a starting pattern for each path: structured (one L1 or L2 score per output channel, `topk_indices`, then `rebuild_structured_subnetwork` from `common.py`, which rebuilds `conv2` and `fc1` consistently) and unstructured (`prune.global_unstructured` over the `(module, "weight")` pairs you choose; the program keeps the masks during fine-tuning and removes them before PTQ).

```
# container
cd challenge_frugal_pruning/student
python3 prune_and_finetune.py --tag ${TAG} --keep_ratio 0.75 --epochs 2
#   unstructured: --amount 0.60 instead of --keep_ratio
cat outputs/model_config.json outputs/results_host.json
python3 quantize.py --model_dir outputs --data_root ../../data --calib_steps 100
bash compile.sh outputs
bash copy_to_board.sh                          # BOARD exported as in Step 8
# KV260 (ssh root@${BOARD#*@})
cd /home/root/c5/TP03/challenge && source ./setup_env.sh
python3 run_xmodel.py
xdputil benchmark model.xmodel -i -1 1
```

`--tag` is the TAG of your guided run: its float checkpoint is the starting point. Before going further, check that the output shape is `[N,10]` with no NaN or infinity, that the parameters and MACs fell (structured) or the sparsity rose (unstructured), and that the accuracy stays well above chance (10 %). `run_xmodel.py` uses the same test images, in the same order, as the guided bundle, so its accuracy compares directly with Step 8.

Model	Val. acc. after pruning	Test acc. after FT	Params	MACs	Size (KB)	FPGA acc.	FPGA FPS
Guided float baseline	—						
Your model							

Self-assessment QCM

QCM — Pruning, model reconstruction, and deployment

Select the **best** answer. There is one correct answer per question.

1. The sparsity of a tensor containing N parameters and $\text{nnz}(W)$ non-zero values is:

   A. $s = \dfrac{\text{nnz}(W)}{N}$

   B. $s = 1 - \dfrac{\text{nnz}(W)}{N}$

   C. $s = \dfrac{N}{\text{nnz}(W)}$

   D. $s = 1 - \dfrac{N}{\text{nnz}(W)}$

2. For a convolutional weight tensor

   $$W \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times K \times K},$$

   which expression computes an L1 importance score for every output channel?

   A. `W.abs().sum(dim=0)`

   B. `W.abs().sum(dim=(1,2,3))`

   C. `W.abs().sum(dim=(0,2,3))`

   D. `W.abs().mean()`

3. Consider the following code:

   ```
   prune.l1_unstructured(layer, name="weight", amount=0.6)
   prune.remove(layer, "weight")
   ```

   What is true after these two instructions?

   A. The layer has 60% fewer output channels.

   B. Approximately 60% of its weights are zero, but its tensor dimensions remain unchanged.

   C. The layer is automatically converted into a sparse DPU operator.

   D. The zero weights are restored by `prune.remove`.

4. A network contains:

   ```
   conv1: 3 -> 16
   conv2: 16 -> 32
   ```

   After structured pruning, only 10 output channels of `conv1` are kept. What must be true in the rebuilt model?

   A. `conv2.in_channels = 10`

   B. `conv2.out_channels = 10`

   C. `conv1.in_channels = 10`

   D. No change is required in `conv2`.

5. The approximate number of MAC operations in a dense convolution is:

$$\text{MACs} = H_{\text{out}} W_{\text{out}} C_{\text{out}} C_{\text{in}} K_h K_w.$$

If structured pruning halves both C_{in} and C_{out}, while all other dimensions remain unchanged, the new MAC count is approximately:

A. 50% of the original
B. 25% of the original
C. 75% of the original
D. unchanged

6. An iterative experiment should retain $k_t = 0.60$ of the original channels at step t. The previous step retained $k_{t-1} = 0.75$. What fraction of the *current* channels should be retained?

$$r_t = \frac{k_t}{k_{t-1}}$$

A. 0.45
B. 0.60
C. 0.75
D. 0.80

7. A structured-pruned model has fewer parameters and fewer estimated MACs, but nearly unchanged application FPS on the KV260. Which conclusion is the most technically sound?

A. The pruning implementation must be incorrect.
B. MAC reduction always guarantees proportional FPGA speedup.
C. The workload may be too small, with runtime dominated by VART dispatch, transfers, buffer handling, or low DPU utilization.
D. Structured pruning only affects training time.

Answers

1–B,	2–B,	3–B,	4–A,	5–B,	6–D,	7–C

A useful pruning result must connect three levels:

policy → PyTorch tensors and architecture → measured deployment behaviour.

High sparsity alone is not sufficient evidence of frugal inference.