TÉLÉCOM PARIS · SETI MASTER · LECTURE 03

# Pruning and Architecture Search

Remove unnecessary computation, then select a network shape under a budget

03

Granularity, criteria, recovery, structured pruning and hardware-aware architecture search

Wednesday 13 January 2027 · Session 3 of 8 · lecture + Lab 3

## Where we are

| S1 9 Dec | S2 6 Jan | S3 13 Jan | S4 27 Jan | S5 3 Feb | S6 10 Feb | S7 24 Feb | S8 3 Mar |
|---|---|---|---|---|---|---|---|
| L01 + Lab 1 | L02 + Lab 2 | **L03 + Lab 3** | L04 + Lab 4 | Lab 5 | Lab 6 | Lab 7 (I) | Lab 7 (II) |

### What we keep from S2

**The DPU contract**
INT8, signed, symmetric, per-tensor, power-of-two scales. Whatever we do today must still fit it.

**PTQ first, on representative data**
and escalate only when a stated tolerance is missed.

**One deployment path**
quantize → export batch 1 → `vai_c_xir` → XIR/VART → four measurement levels.

**Today's question**

Quantization made every number smaller – but kept every multiplication.

**Which weights, neurons or channels can we remove altogether – and does the DPU actually run faster when we do?**

**S2 changed the size of every number. Today we change how many numbers there are.**

THE THREAD – TODAY'S ANGLE

# How much useful intelligence can we obtain from a limited budget of computation, memory and energy?

A trained network is heavily over-parameterised: most of its weights can be removed with little or no loss – *if* we fine-tune afterwards.

But "removed" has two meanings: set to zero, or taken out of the architecture. Only one of them changes what a dense accelerator does. Today we learn to tell them apart – and to measure the difference.

## By the end of this session you can

### Compute

by hand

sparsity, parameters and MACs of a CNN before and after channel pruning; when a sparse format actually saves bytes.

### Explain

with the right words

family vs policy, granularity, criteria, layer-wise vs global allocation, one-shot vs iterative, why fine-tuning matters.

### Choose

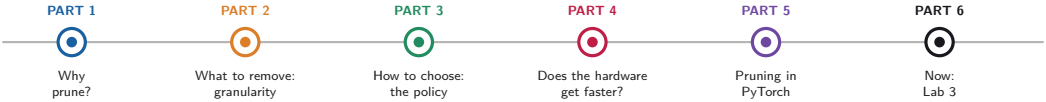
with a control

a pruning policy or a channel-width search under a budget – and check against an appropriate baseline.

### Measure

on the KV260

quantize, compile and run pruned candidates; read `WL`, `LdWB`, FPS and mJ/input – and explain what did not change.



PART 1

# Why prune?

Trained networks are redundant – by a large factor. What removing the redundancy can buy, and on which hardware.

1

# Most of a trained network is redundant

**9×**
fewer weights in AlexNet (61 M → 6.7 M), same ImageNet accuracy

**13×**
fewer weights in VGG-16 (138 M → 10.3 M), same accuracy

**35×**
smaller AlexNet file with pruning + quantization + Huffman: 240 MB → 6.9 MB

**49×**
smaller VGG-16 file: 552 MB → 11.3 MB

S. Han et al., *Learning both Weights and Connections*, NeurIPS 2015; *Deep Compression*, ICLR 2016.

## Why redundancy exists
we train large networks because they are easier to *optimise*, not because the final function needs every weight.

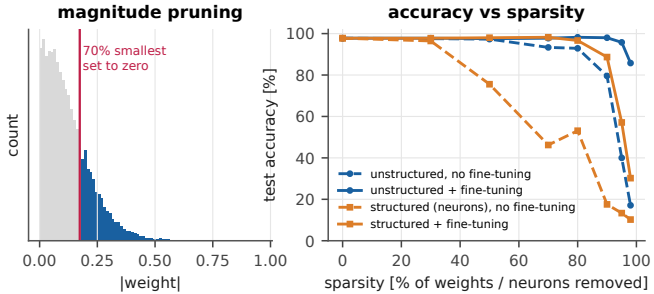
## Biology did it first
synaptic density in the human frontal cortex peaks around age 1–2, then declines by about a third by adulthood: unused connections are removed.

## The catch
"same accuracy" always means *after fine-tuning*, and "fewer weights" does not automatically mean "faster".

P. Huttenlocher, *Brain Research*, 1979; C. Walsh, *Nature* 502, 2013.

**Pruning exploits a gap between what training needs and what inference needs.**

## Recap from S1: pruning in one picture




**Unstructured** (blue): zero the smallest individual weights. With fine-tuning, 90% of them go at no loss – but every matrix keeps its shape.

**Structured** (orange): remove whole neurons. It hurts more before fine-tuning; after it, 70% of the neurons go at no loss – and the layers really are smaller.

Computed for this course: 3-layer MLP on scikit-learn `digits`; magnitude (L1) criteria; 15 epochs of fine-tuning.

**Today we open this picture: which unit, which criterion, how much, in which order – and what the DPU does with it.**

## What pruning can buy – depends on what you remove

| | **unstructured**, dense accelerator (DPU, GPU) | **unstructured**, sparse hardware or format | **structured**, any hardware |
|---|---|---|---|
| parameters stored (dense count) | unchanged | fewer nonzeros + indices | **fewer** |
| file size after compression | **smaller** (zeros compress) | **smaller** | **smaller** |
| MACs executed | unchanged | **fewer** (if the kernel skips) | **fewer** |
| weight traffic (DDR → chip) | unchanged | **lower** at high sparsity | **lower** |
| activation memory | unchanged | unchanged | **lower** (fewer channels) |
| latency, energy | unchanged | only above ≈90% sparsity | **lower**, if the layer mattered |

### Accuracy per removed weight

unstructured wins: it removes exactly the least useful numbers.

### Speed per removed weight

structured wins on the hardware you have: it produces a smaller *dense* network.

**The KV260 DPU is a dense INT8 engine. On it, only structure changes the work.**

PART 2

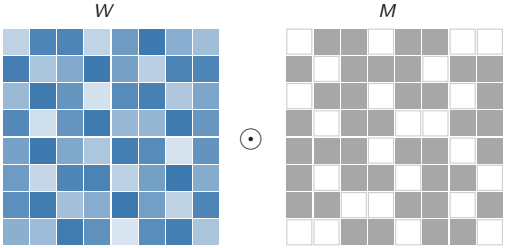
# What to remove: granularity

2

From single weights to whole channels – the storage and compute cost of zeros, and why structured pruning is a graph transformation.

## The core operation: a mask

$$W' = M \odot W, \qquad M_{ij} \in \{0, 1\}$$




Here the mask keeps the entries with the largest $|w|$: **magnitude pruning**. In PyTorch the mask is a buffer next to the weight; after `prune.remove()` the zeros are written into the weight itself.

### Sparsity

$s = 1 - \dfrac{\text{nnz}(W')}{N}$ fraction of zeros; the *density* is $1 - s$.

### Compression ratio (ideal)

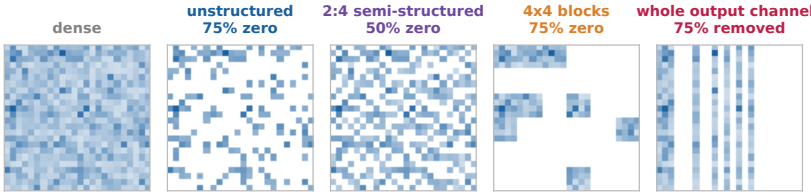
$\dfrac{N}{\text{nnz}(W')} = \dfrac{1}{1-s}$: 90% sparsity $\Rightarrow 10\times$ – *before* paying for indices.

### Who decides $M$

a **criterion** scores each unit, an **allocation** rule sets how many survive per layer, a **schedule** decides when.

**Every pruning method is a way of choosing $M$. What differs is which entries may be zeroed together.**

## Granularity: from single weights to whole channels




### Fine-grained

any weight may go. Best accuracy; irregular – needs a sparse format and kernel.

### N:M

keep 2 of every 4 weights. NVIDIA Ampere runs 2:4 at up to 2× math throughput.

### Blocks

remove $4 \times 4$ or $8 \times 8$ tiles, the tiles an accelerator loads.

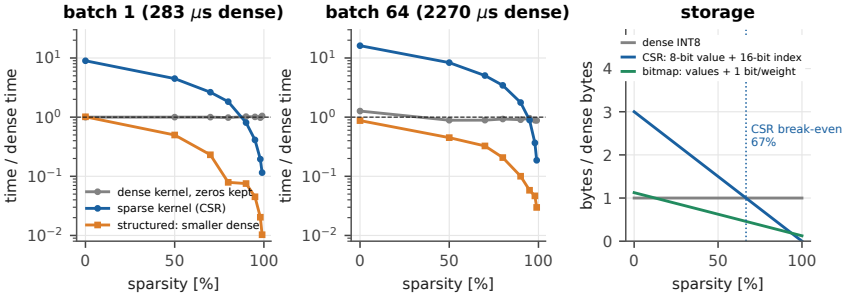
### Channels

remove whole rows of work: the only kind a dense engine turns into speed.

Computed for this course on a $128 \times 128$ layer of the `digits` MLP ($24 \times 24$ crop shown; white = pruned). Mishra et al., *Accelerating Sparse Deep Neural Networks*, arXiv:2104.08378, 2021.

**Finer granularity keeps more accuracy; coarser granularity keeps more speed. The hardware decides where to sit.**

## Zeros are not free: storage and compute, measured



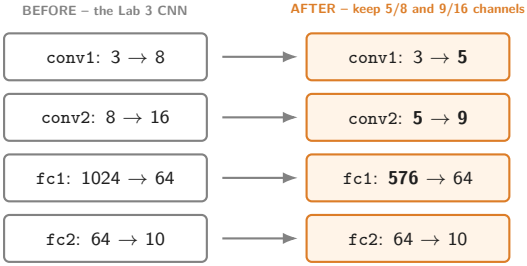

**Left, middle** (2048 × 2048 layer on a CPU): a dense kernel does not care how many zeros it multiplies (grey). A sparse kernel (CSR) pays for indirection: at 50% sparsity it is **4.5–8× slower**, and it only wins above ≈90% (batch 1) or ≈95% (batch 64). Removing whole rows (orange) scales with what is left.

**Right**: an 8-bit value with a 16-bit column index costs 3 bytes. CSR saves space only above 67% sparsity; a bitmap (1 bit per weight) is cheaper at moderate sparsity.

Measured for this course: NumPy/OpenBLAS dense vs SciPy CSR, float32, median of repeated runs, 2-core cloud VM. Absolute numbers depend on the machine; the crossovers are typical.

**Irregular sparsity pays on general-purpose hardware only when very high.**

# Structured pruning builds a smaller dense network




**Parameters**: 67 642 → 38 132 (−44%). **MACs**: 582 k → 279 k (−52%).
Removing a conv2 output channel deletes a $8 \times 8$ feature map, i.e. **64 input columns** of `fc1`: $16 \cdot 64 = 1024 \to 9 \cdot 64 = 576$.

### No zeros left

the pruned model is an ordinary network with smaller layers: any runtime, any accelerator runs less work.

### It is a new architecture

the checkpoint no longer loads into the original class: Lab 3's structured variants are exported as `SlimCNN`.

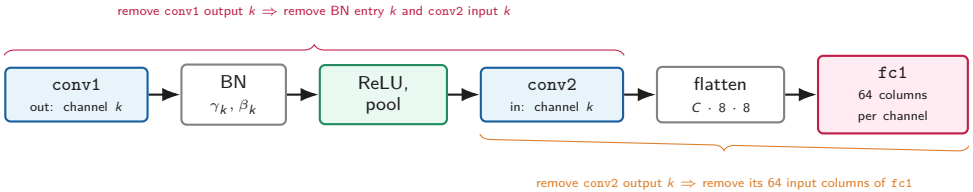
### Removing a neuron removes its fan-out

every consumer of that channel must be rebuilt consistently.

Counts computed from the Lab 3 architecture: $32 \times 32$ inputs, $3 \times 3$ convolutions with padding, $2 \times 2$ pooling after each convolution.

**Structured pruning does not produce a sparse model. It produces a smaller dense one.**

# Structured pruning is a graph transformation, not a tensor operation




## Propagate the indices

the *same* kept indices, sorted, must be applied to the producer's outputs, the normalisation, and every consumer's inputs: sorted, the kept channels keep their order.

## Branches must agree

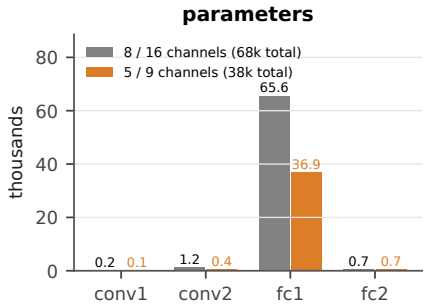
residual additions and concatenations tie channels together: prune them jointly, or not at all.

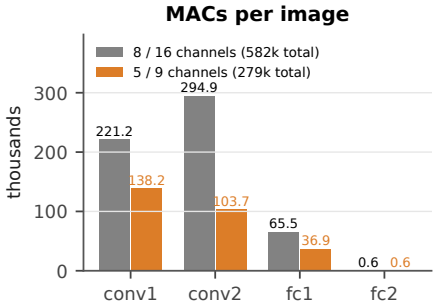
## Check the result

output shape still `[N,10]`, no NaN, accuracy above chance – the Lab 3 final task asks exactly this.

**A channel is not a tensor slice: it is a node in a graph, with producers and consumers that must all be rewritten.**

## Where do parameters and MACs live? The Lab 3 CNN




**Parameters**: `fc1` holds **97%** of them. Pruning `conv2` channels is the lever that shrinks `fc1` – and the model file.

**MACs**: the two convolutions do **89%** of the work with 2% of the parameters – each weight is reused at every pixel.

Computed from the Lab 3 architecture (see previous slide).

**Size and compute live in different layers. Decide which one your constraint is before choosing what to prune.**

## Hardware that likes zeros – and hardware that does not

### Sparse accelerators

EIE, ISCA 2016; SCNN, ISCA 2017

store only nonzeros with compressed indices and skip zero operands in the datapath. Built *for* fine-grained pruning – and for its irregular memory accesses.

### Semi-structured support

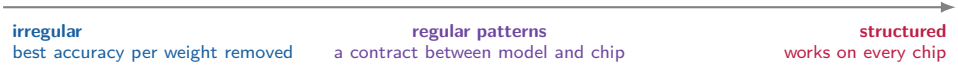
NVIDIA Ampere and later

2:4 sparse tensor cores: a 2-bit index per kept value, a fixed pattern, up to 2× math throughput. The model must be pruned to exactly that pattern.

### Dense INT8 engines

the KV260 DPU, TPU v1, most NPUs

a zero weight is multiplied like any other. Speed only comes from smaller tensors: fewer channels, smaller maps, fewer layers.




S. Han et al., ISCA 2016; A. Parashar et al., ISCA 2017; NVIDIA A100 architecture whitepaper, 2020.

**Sparsity is a contract between the model and the hardware – exactly like the INT8 contract of S2.**

PART 3

# How to choose: the pruning policy

Criterion, allocation, strength, schedule, constraints, recovery – and a random baseline to keep us honest.

3

## A family is not a policy: seven decisions

$$\mathcal{P} = (\text{family},\ \text{criterion},\ \text{allocation},\ \text{strength},\ \text{schedule},\ \text{constraints},\ \text{fine-tuning})$$

| Component | Question | In Lab 3 |
|---|---|---|
| family | what unit is removed? | individual weights; whole channels or neurons |
| criterion | how is importance estimated? | L1 magnitude, L2 magnitude, random baseline |
| allocation | how is the budget split across layers? | same ratio per layer; one global ranking (mean-normalised) |
| strength | how much is removed or kept? | `AMOUNT` (unstructured), `KEEP` (structured) |
| schedule | once, or gradually? | one-shot; iterative as an extension |
| constraints | which shapes are allowed? | at least one channel; optional `ROUND_TO_MULTIPLE` |
| recovery | how is accuracy recovered? | fine-tuning epochs and learning rate |

The board never sees the policy – only the final tensor shapes. Record the policy in `meta_pruned.json`, `prune_report.json` and your table *before* deploying.

**"Structured" or "unstructured" names a family. A result is only reproducible when all seven decisions are written down.**

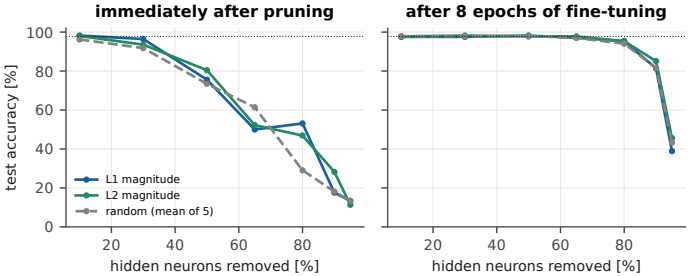
## Criteria: magnitude – and a random control to keep us honest

One score per output channel of $W \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times K \times K}$:

$$s_i^{L1} = \sum_{j,u,v} |W_{i,j,u,v}|$$

$$s_i^{L2} = \left( \sum_{j,u,v} W_{i,j,u,v}^2 \right)^{1/2}$$

Keep the top-$k$ channels. A **random** score is the control: a criterion that cannot beat it has not shown that it measures importance.

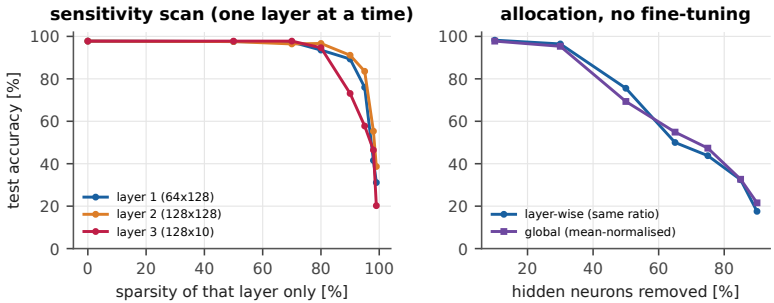



Before fine-tuning, the criteria differ – and random is not always worse. After 8 epochs of fine-tuning all three reach ≈98% up to 65% of the neurons removed: the *architecture* that remains matters more than which neurons were chosen.

Computed for this course: `digits` MLP, structured (neuron) pruning, same ratio in each hidden layer. See also Z. Liu et al., *Rethinking the Value of Network Pruning*, ICLR 2019.

**Judge a criterion after fine-tuning, and always next to a random baseline. Immediate accuracy is a useful but imperfect signal.**

## Layers are not equally sensitive – so how do we split the budget?



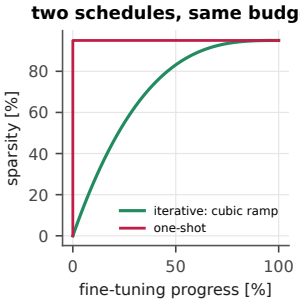

**Sensitivity scan**: prune one layer at a time and measure. The output layer (red) breaks first; the big middle layer tolerates the most. First and last layers are usually kept denser.

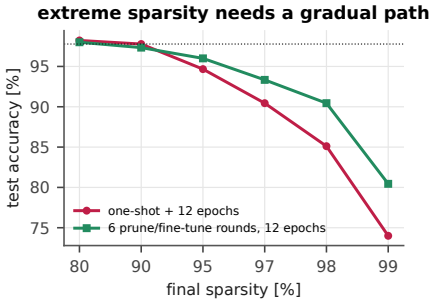
**Allocation**: *layer-wise* removes the same ratio everywhere; *global* ranks all channels together after normalising each layer's scores by their mean. Here global keeps 54 + 74 neurons at 50%: sometimes better, sometimes worse.

Computed for this course: `digits` MLP, L1 criteria, no fine-tuning. Lab 3: `STRUCTURED_POLICY=layerwise` vs `03c_prune_structured_global.sh` with `GLOBAL_NORMALIZE=mean`.

**Global is not automatically better: check every layer for collapse.**

## Schedules: one big cut, or many small ones?




**Left**: gradual pruning raises the sparsity along a cubic ramp $s_t = s_f\left(1-(1-t/T)^3\right)$ – fast at first, when many weights are useless, slowly at the end.

**Right**: same total fine-tuning (12 epochs). Up to 90% the schedule hardly matters; at 98% one-shot keeps 85% accuracy, six small rounds keep 90%. The network adapts between cuts.

Computed for this course: `digits` MLP, global unstructured magnitude pruning. M. Zhu, S. Gupta, *To prune, or not to prune*, arXiv:1710.01878, 2017.

**Moderate targets: prune once and fine-tune. Extreme targets: prune gradually and let the network adapt between cuts.**

PART 4

# Does the hardware get faster?

4

What changes in the compiled model and what does not, how the DPU sees channels, and why a faster DPU is not always a faster application.

## Follow the model through the toolchain: what changes?

| Quantity | unstructured (60% zeros) | structured (5/8, 9/16 channels) | where to read it |
|---|---|---|---|
| dense parameter count | unchanged | −44% | `summary_prune` |
| nonzero weights | −60% | −44% | `summary_prune` |
| estimated MACs | unchanged (dense) | −52% | `summary_prune` |
| XMODEL size | ≈ unchanged | smaller | `ls -lh` |
| compiled DPU workload | unchanged | smaller | `vaitrace WL` |
| weight traffic | unchanged | smaller | `vaitrace LdWB` |
| DPU time | unchanged | smaller – not proportionally | `HW_RT`, `xdputil` |
| application FPS | unchanged | maybe | VART app |

**Why unstructured changes nothing here**

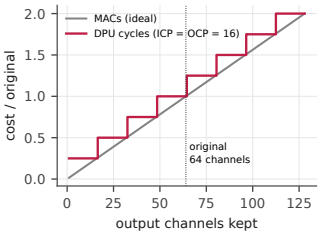
the INT8 XMODEL stores every weight, zeros included; the DPU streams and multiplies them all.

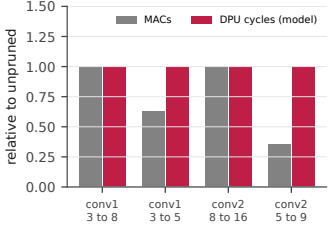
**Why structured may still disappoint**

the next slides: channel parallelism, and everything around the DPU.

**Predict each row before you measure it – the Lab 3 report asks you to explain every row that did not move.**

# The DPU computes channels in groups of 16




B4096 = 8 pixels × 16 input channels × 16 output channels × 2 ops per clock.

**First-order cost**

$\propto \lceil C_{\text{in}}/16 \rceil \cdot \lceil C_{\text{out}}/16 \rceil \cdot \lceil W/8 \rceil \cdot H \cdot K^2$: going from 64 to 50 channels saves nothing; 64 to 48 saves 25%.

**Lab 3's convolutions**

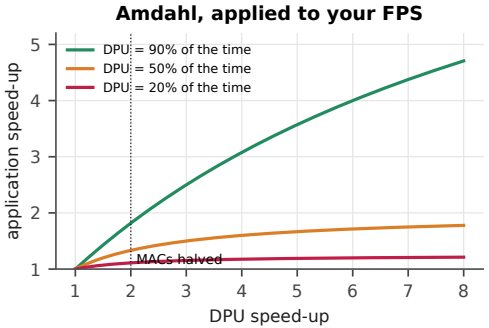
all widths are below 16: 8 → 5 and 16 → 9 cut MACs, not cycles. The gain must come from `fc1`: 1024 → 576 inputs.

Parallelism of the DPUCZDX8G B4096 from AMD PG338. The staircase is a first-order model for intuition, not a measurement – `vaitrace WL` and `HW_RT` give the real answer.

**On a parallel engine, prune in multiples of its parallelism – or you remove MACs the hardware was going to waste anyway.**

## A faster DPU is not always a faster application

$S_{\text{app}} = 1/\big((1-f) + f/S_{\text{DPU}}\big)$, with $f$ the fraction of the application's time spent *on the DPU*.




### Why $f$ is small in Lab 3

$32 \times 32$ inputs, a compact CNN, batch 1: the DPU finishes quickly, while Python, buffer copies, VART dispatch and post-processing take a fixed time.

### How to see it

compare `xdputil benchmark` (DPU only), VART FPS (submit/wait) and application FPS: the gaps between them are the part pruning cannot touch.

### When pruning pays

large inputs, wide layers, a DPU-bound pipeline. In TP05, measure the DPU fraction before predicting a gain.

G. Amdahl, AFIPS 1967. Curves are illustrative.

**Before you prune for speed, measure what fraction of the time the model actually takes.**

## Reading `vaitrace` for pruned models

```
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
```

| | |
|---|---|
| `WL` | compiled DPU workload (operations) |
| `HW_RT` | measured DPU execution time |
| `Effic` | fraction of the DPU peak achieved – not a DSP utilisation |
| `LdFM`, `StFM` | feature-map loads and stores |
| `LdWB` | weight and bias loads |
| `AvgBw` | average external-memory bandwidth |

### Expect for unstructured

the same `WL`, `LdWB` and `HW_RT` as the float baseline: zeros are streamed and multiplied.

### Expect for structured

lower `WL` and `LdWB`; `HW_RT` lower but not in proportion to MACs.

### Then connect the levels

`HW_RT` explains `xdputil`; `HW_RT` plus fixed overheads explains application FPS. Which one limits *your* FPS?

**`vaitrace` tells you whether pruning changed the compiled work. FPS alone cannot.**

## Energy per input: the number the frugal question asks for

$$E_{\text{input}}\,[\text{mJ}] = 1000 \cdot \bar{P}\,[\text{W}] \,/\, \text{FPS}$$
$$E_{\text{dyn}}\,[\text{mJ}] = 1000 \cdot (\bar{P} - P_{\text{idle}}) \,/\, \text{FPS}$$

**A worked example** illustrative numbers

board at 7.2 W while running 1 200 inputs/s, 6.4 W idle:
total $= 1000 \cdot 7.2/1200 = 6.0$ mJ/input
dynamic $= 1000 \cdot 0.8/1200 = 0.67$ mJ/input

Double the FPS at the same power: both halve.

**How Lab 3 measures it**

`MEASURE_IDLE=1 ./profile_energy_all_variants.sh ${TAG}` samples `xmutil xlnx_platformstats -p` during a controlled workload, then without it.

**Total or dynamic?**

the board's idle power dominates on a small model. Total energy answers "what does the battery see"; dynamic energy answers "what did the model cost".

**Same conditions**

same command, same input count, same board state, idle measured in the same session.

**Power is a rate; energy is what the model consumes per answer. A faster model at the same power is a frugal model.**

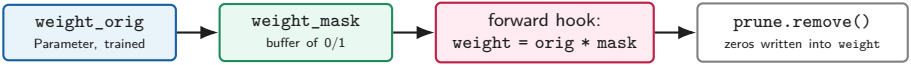
PART 5

# Pruning in PyTorch

Masks by re-parameterisation, global unstructured pruning, and the score–select–rebuild recipe for channels.

5

## `torch.nn.utils.prune`: a mask next to the weight




```python
import torch.nn.utils.prune as prune
conv = model.conv2

# zero the 60% smallest |w| of this layer
prune.l1_unstructured(conv, name="weight", amount=0.60)
print(conv.weight_mask.float().mean()) # density: 0.40

# ... fine-tune: masked weights stay at zero ...

prune.remove(conv, "weight") # materialize zeros for export
```

### Why a mask

gradients still flow to `weight_orig`, but the mask keeps pruned entries at zero during fine-tuning.

### Before PTQ and export

after recovery, call `prune.remove()` for an ordinary tensor. Later optimizer steps can regrow zeros unless masks are reapplied.

### Same shape

`ln_structured(..., dim=0)` zeroes whole filters – but still does not shrink the tensor.

**PyTorch pruning is mask-based. Shrinking the architecture is a separate step you write yourself.**

# Two recipes for the Lab 3 final task

## Global unstructured

```
params = [(m, "weight") for m in model.modules()
          if isinstance(m, (nn.Conv2d, nn.Linear))]
prune.global_unstructured(
    params,
    pruning_method=prune.L1Unstructured,
    amount=0.60) # one threshold for all
# Fine-tune while masks remain active.
# Remove reparameterizations only after recovery,
# or reapply saved masks after every optimizer step.
for m, name in params:
    prune.remove(m, name) # export stage
```

The list of layers is part of the experiment: excluding the last layer protects the most sensitive one. The guided runs (03b) prune every layer by the same amount and enforce the saved masks while fine-tuning.

## Structured: score, select, rebuild

```
w1, w2 = model.conv1.weight, model.conv2.weight
s1 = w1.detach().abs().sum(dim=(1, 2, 3)) # per out-ch
s2 = w2.detach().abs().sum(dim=(1, 2, 3))
s3 = model.fc1.weight.detach().abs().sum(dim=1)
k1 = topk_indices(s1, keep_ratio) # kept, sorted
k2 = topk_indices(s2, keep_ratio)
k3 = topk_indices(s3, keep_ratio)
slim = rebuild_structured_subnetwork(
    model, keep_conv1=k1, keep_conv2=k2, keep_fc1=k3)
```

The helper slices `conv2`'s inputs by `k1` and `fc1`'s input columns by `k2` – the propagation of Part 2.

**Unstructured: one call and a list of layers. Structured: your scores, your indices, and a consistent rebuild.**

## Count what you deploy

```python
def sparsity(model):
    zeros = total = 0
    for name, p in model.named_parameters():
        if name.endswith("weight"):
            zeros += (p == 0).sum().item()
            total += p.numel()
    return zeros / total

def macs_conv(c_in, c_out, k, h_out, w_out):
    return c_in * c_out * k * k * h_out * w_out
```

### From the exact model you export

count after `prune.remove()` and after fine-tuning – not from the mask of an earlier candidate.

### Dense vs nonzero

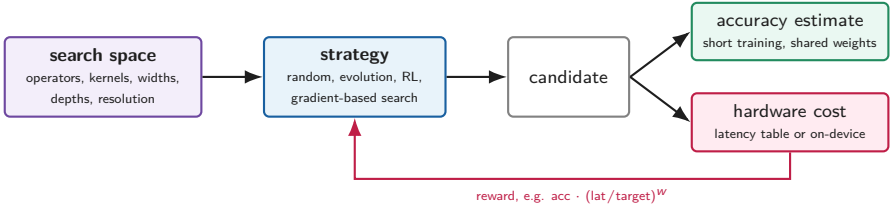
report both: the DPU executes the dense count; the file compresses the nonzero count.

### Metadata travels with the checkpoint

`meta_pruned.json`: `criterion`, `prune_policy`, `global_normalize`, `round_to_multiple`, `kept_indices`, `slim`.

**The model that is quantized, compiled and run must be the one your table describes. Verify it before PTQ.**

## Beyond pruning: space, strategy, evaluation




**Space → result**

Search supported operators and static shapes. Channel alignment affects efficiency; TP03 permits widths in steps of eight.

**Strategy → cost**

how many candidates must be trained before a good one appears.

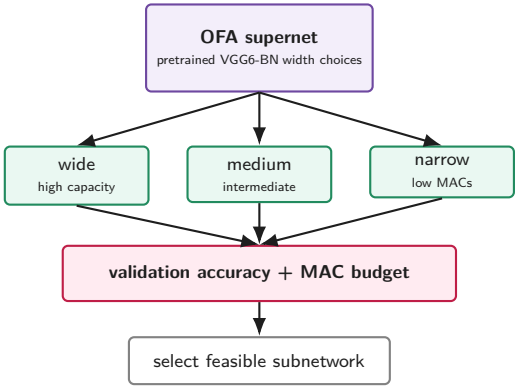
**Evaluation → truth**

proxy accuracy and proxy cost must rank candidates like the real thing.

T. Elsken et al., *Neural Architecture Search: A Survey*, JMLR 2019.

**TP03 searches channel widths under a MAC budget; validate the deployed result.**

## Once-for-All: one trained supernet, many subnetworks




**Train once, off-line**
The lab provides the baseline and width-sharing supernet. Its smaller paths learn from the largest path during preparation.

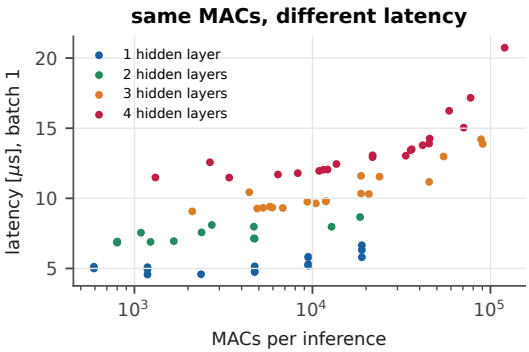
**Search under a budget**
Recalibrate BatchNorm, then rank widths on validation data. Fine-tune selected subnets; compare with the same VGG6-BN baseline.

**Measure the deployed result**
Select and rank on validation data. Report test accuracy for exported models, never for selection. Measure target latency.

H. Cai et al., *Once-for-All*, ICLR 2020; Lab 3 Experiment B: `provided/vaiopt/` and `09e_vaiopt_nas.sh`.

**TP03 shares weights across a width-only OFA search space.**

## MACs are not latency




In the measured CPU illustration, at $10^4$ MACs a 1-layer network answers in $\approx 5\,\mu$s, a 4-layer one in $\approx 12\,\mu$s: every layer adds a fixed cost – a call, a memory round trip, a synchronisation.

**DPU cost has its own structure**
channel parallelism, padding, instruction scheduling and memory traffic all affect latency.

**So measure the target**
latency lookup tables built on the device (ProxylessNAS, FBNet), or on-device measurement in the loop (MnasNet).

Measured for this course: NumPy MLPs, batch 1, median of 300 runs on a CPU. H. Cai et al., ICLR 2019; B. Wu et al., CVPR 2019; M. Tan et al., CVPR 2019.

**A search that minimises MACs optimises a proxy. A search that measures latency on the target optimises the product.**

PART 6

# Now: Lab 3

Two experiments, two baselines. A: the small CIFAR-10 CNN, several policies from the same checkpoint, two candidates fine-tuned, deployed and profiled on the KV260. B: VGG6-BN, a larger network, whose widths the Vitis AI Optimizer chooses.

6

## Lab 3 in one picture




### Train once, compare fairly

every candidate starts from the *same* `model_float.pth`. Different seeds would confound the policy with the training run.

### Explore cheaply, then commit

`03g_save_candidate.sh` keeps each candidate in `policy_study/`; fine-tune only the two you select.

### Deploy all five IDs

`float_base`, `pruned_*` and `ft_*`, one shared bundle, DPU fingerprint 0x101000056010407.

| Deployment ID | selected with | what it isolates |
|---|---|---|
| `float_base` | `SRC=float` | the unpruned reference, after the same PTQ |
| `pruned_unstructured` / `ft_unstructured` | `SRC=pruned\|ft VARIANT=unstructured` | zeros in dense tensors, before / after recovery |
| `pruned_structured` / `ft_structured` | `SRC=pruned\|ft VARIANT=structured` | a smaller dense network (`SlimCNN`) |

**The expensive steps – fine-tuning and deployment – happen only after the policy study has chosen what deserves them.**

## The structured candidates: one change at a time

| ID | criterion | allocation | command (keep 0.90) | compare with |
|---|---|---|---|---|
| S1 | L1 | layer-wise | `CRITERION=l1 STRUCTURED_POLICY=layerwise 03c_prune_structured.sh` | S2: criterion |
| S2 | L2 | layer-wise | `CRITERION=l2 STRUCTURED_POLICY=layerwise` | S1 |
| S3 | L1 | global | `CRITERION=l1 GLOBAL_NORMALIZE=mean 03c_prune_structured_global.sh` | S1: allocation |
| S4 | L2 | global | `CRITERION=l2 GLOBAL_NORMALIZE=mean` | S3, S2 |
| S5 | random | global | `CRITERION=random GLOBAL_NORMALIZE=mean` | S3/S4: the control |

### Same channels or same counts?

L1 and L2 may keep the same *number* of channels but not the same ones.

### Dominance

A dominates B if it has no lower accuracy and no more MACs, and is strictly better on one.

### Constraints last

`ROUND_TO_MULTIPLE=4/8` only after the unconstrained study: with 8 and 16 channels, few widths remain.

**Each row differs from another row by exactly one decision. That is what makes the table an experiment.**

## What your Lab 3 report must show

| Experiment A | XMODEL 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` | | | | | | | | |

### Plus, from `vaitrace`

`WL`, `HW_RT`, `Effic`, `LdFM`, `LdWB`, `StFM` for the float baseline and the two fine-tuned candidates.

### And the answers

Did unstructured pruning change `WL`? Is the drop in `HW_RT` proportional to the drop in MACs? Which measurement explains the application FPS? Experiment B gets its own table, against its own baseline.

**A pruning result is a row of this table and a sentence explaining it – including the columns that did not move.**

## Then: let the Vitis AI Optimizer choose the widths

### Iterative
sensitivity analysis: prune each layer alone, 10–90 %, measure; every layer gets the largest ratio within one tolerance, tuned to the MAC target.

### One-step
sample many sets of per-layer ratios at the MAC target, recalibrate BatchNorm, keep the best on validation.

### OFA (NAS)
a provided supernet; evolutionary search selects channel widths under a MAC budget (explained before the lab).

```
# VGG6-BN, 38.6 M MACs; baseline and
# supernet in provided/vaiopt/
bash scripts/09a_vaiopt_setup.sh $TAG
bash scripts/09b_vaiopt_sensitivity.sh $TAG
bash scripts/09c_vaiopt_prune.sh $TAG uniform 0.5
bash scripts/09c_vaiopt_prune.sh $TAG iterative 0.5
bash scripts/09e_vaiopt_nas.sh $TAG
bash scripts/09f_vaiopt_deploy.sh $TAG \
     base uniform_l1 iterative ofa_50
```

About 11 minutes of CPU time on a lab PC; start it while Part 2 runs on the board.

**Experiment B, a new baseline:** VGG6-BN, about 88 % (the small CNN: 58 %), 66× its MACs. Compare its candidates with its own base, never with Experiment A.

**All three remove whole channels – and all three constrain MACs, not the latency of this DPU.**

# Final task: one policy, implemented by you, on the board

```python
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 model, family, policy description
    ...
```

```
python3 prune_and_finetune.py --tag ${TAG} \
       --keep_ratio 0.75 --epochs 2 # or --amount 0.60
python3 quantize.py --model_dir outputs \
       --data_root ../../data --calib_steps 100
bash compile.sh outputs
export BOARD=root@maquette-07; bash copy_to_board.sh
```

**Recommended first**

structured layer-wise L1/L2 with `rebuild_structured_subnetwork`, or global unstructured L1 with `prune.global_unstructured`.

**One objective**

accuracy, size, MACs, FPS or energy – state it before you start.

**Checks**

output `[N,10]`, no NaN, sparsity or MACs changed as intended, accuracy well above chance.

**Global structured allocation and iterative schedules are extensions – after one complete deployment works.**

## Before you start the lab

### Checklist

**1. Read the code while the model trains**
Exercise 1: the unstructured masks, the channel scores, the global normalisation, the `conv1`→`conv2`→`fc1` propagation.

**2. One TAG, one float checkpoint**
never retrain the baseline between policies.

**3. Save every candidate**
a new run overwrites `prune_structured/`: `03g_save_candidate.sh` keeps a copy and prints its row.

**4. Verify before PTQ**
`meta_pruned.json` and `shape_table.txt` describe the model you deploy.

### Predictions to write down now

**Unstructured 80%**
accuracy before and after fine-tuning? `WL`?

**Structured, 5/8 and 9/16 channels**
MACs −52% – and DPU time?

**Random (S5)**
how far behind L1 before fine-tuning? after?

Lab 4 studies distillation on its own teacher/student pair; TP05 combines pruning with KD.

**Write the prediction before the command. A surprise you wrote down is a finding; one you did not is just a number.**

THE THREAD – WHAT TODAY ADDED

# How much useful intelligence can we obtain from a limited budget of computation, memory and energy?

Most weights can be removed, but only **structure** turns removal into speed on a dense accelerator – and only if the removed work was on the critical path.

Pruning can take away more than the network can recover alone. We also searched channel widths under a budget. Next session: a large network teaches a small one – knowledge distillation.

## Takeaways

| | |
|---|---|
| **Redundancy is real** | Trained networks tolerate removing most of their weights – after fine-tuning, which is part of the method. |
| **Granularity is a contract** | Fine-grained sparsity needs sparse hardware; structured pruning gives a smaller dense network that any hardware runs faster. |
| **A policy, not a family** | Criterion, allocation, strength, schedule, constraints, recovery – and a random control before believing a criterion. |
| **Check the hardware** | On the dense DPU, zeros do not skip MACs; channel tiling can hide arithmetic savings; and Amdahl caps what a faster DPU gives the application. |
| **Measure every level** | `WL`/`LdWB` for the compiled work, `xdputil`/VART/application FPS, and mJ per input. |

**Pruning is only frugal when the removed computation was actually being paid for.**

# Sources and further reading

## Foundations

- Y. LeCun et al., *Optimal Brain Damage*, NeurIPS 1990; B. Hassibi, D. Stork, *Optimal Brain Surgeon*, NeurIPS 1993.
- S. Han et al., *Learning both Weights and Connections*, NeurIPS 2015; *Deep Compression*, ICLR 2016.
- M. Zhu, S. Gupta, *To prune, or not to prune*, arXiv:1710.01878, 2017.
- J. Frankle, M. Carbin, *The Lottery Ticket Hypothesis*, ICLR 2019.
- Z. Liu et al., *Rethinking the Value of Network Pruning*, ICLR 2019.
- MIT HAN Lab 6.5940, lectures on pruning and sparsity: `https://hanlab.mit.edu/courses/2026-fall-65940`

## Criteria

- H. Li et al., *Pruning Filters for Efficient ConvNets*, ICLR 2017.
- Z. Liu et al., *Network Slimming*, ICCV 2017; J.-H. Luo et al., *ThiNet*, ICCV 2017.
- P. Molchanov et al., ICLR 2017 and CVPR 2019 (Taylor criteria); Y. He et al., *AMC*, ECCV 2018.

## Sparse hardware

- S. Han et al., *EIE*, ISCA 2016; A. Parashar et al., *SCNN*, ISCA 2017.
- A. Mishra et al., *Accelerating Sparse Deep Neural Networks*, arXiv:2104.08378, 2021 (2:4).
- G. Amdahl, AFIPS Spring Joint Computer Conference, 1967.

## Tools

- PyTorch, `torch.nn.utils.prune` documentation and pruning tutorial.
- AMD PG338 (DPUCZDX8G); UG1414 (Vitis AI User Guide): `vaitrace`, `xdputil`.

## Biology

- P. Huttenlocher, *Synaptic density in human frontal cortex*, Brain Research, 1979.

Figures marked "computed for this course" or "measured for this course" come from the course's figure scripts, which are not part of the student release.

FOR READING

# Supplementary material

Importance signals beyond magnitude, and what pruning really finds.

## Other importance signals

### Magnitude

Han et al. 2015; Li et al. 2017

$|w|$ or a channel norm. Free to compute, surprisingly strong; blind to how the next layer uses the value.

### BatchNorm scale

Network Slimming, ICCV 2017

train with an L1 penalty on the BN factors $\gamma_c$; channels whose $\gamma_c \to 0$ are switched off by the network itself.

### Gradient (Taylor)

Molchanov et al., ICLR 2017, CVPR 2019

estimate the loss change if a unit is removed: $|\, g \cdot w \,|$ accumulated over a few batches.

### Second order

LeCun 1990; Hassibi 1993

Optimal Brain Damage / Surgeon: the Hessian says which weight the loss is least sensitive to – and how to compensate the others.

### Activations

APoZ, 2016; ThiNet, ICCV 2017

a channel that is almost always zero after ReLU, or whose output the next layer can reconstruct, is removable.

### Learned / searched

AMC, ECCV 2018

an agent selects per-layer pruning ratios using hardware feedback – architecture search extends the policy.

**Every criterion is an estimate of "how much would the loss change without this unit?". Magnitude is the cheapest estimate – start there.**

# What pruning really finds

## The lottery ticket hypothesis

Frankle & Carbin, ICLR 2019

A dense network contains sparse subnetworks that, *reset to their original initialisation*, train to the same accuracy on their own. Pruning finds a "winning ticket"; the rest of the weights mostly helped optimisation.

## Rethinking the value of pruning

Liu et al., ICLR 2019

For structured pruning, training the pruned *architecture* from scratch often matches pruning + fine-tuning. What pruning discovers is a good shape – the connection to architecture search discussed before Lab 3.

## Pruning and quantization compose




Deep Compression chains pruning, quantization and entropy coding for its 35–49×. Lab 3 uses the same order: fine-tune in float, then calibrate the ranges of the *final* weights.

## Recovery is part of the method

fine-tuning epochs, learning rate, and even distillation from the unpruned model (S4) decide how much of the cut the network absorbs.

**Pruning is less about deleting weights than about finding a smaller network that can be trained to the same function.**