Real Time 2026 Tutorial

SNAC-Pack

Surrogate Neural Architecture Codesign Package

Dmitri Demler

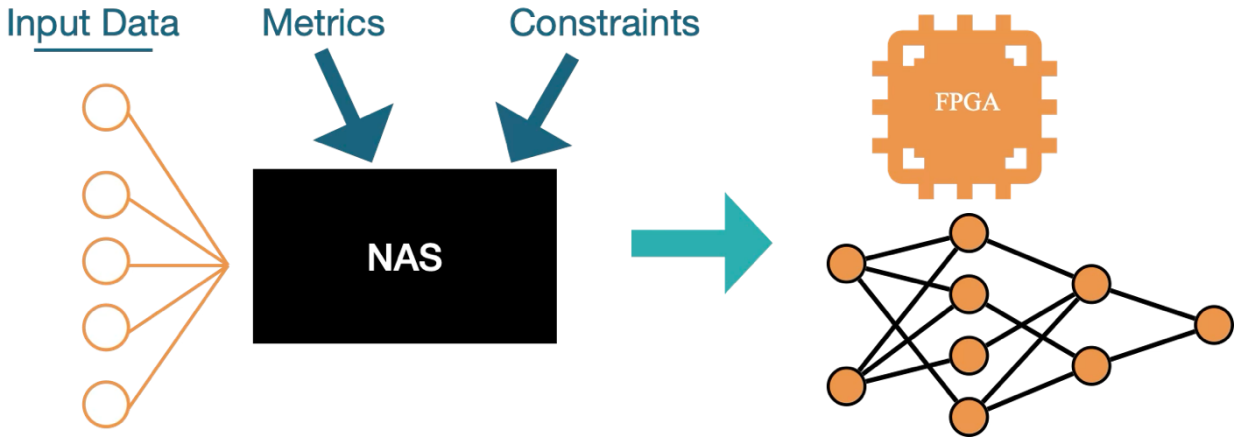
University of California San Diego & ETH Zürich

Plan for today

1.	**The problem**	Why deploying ML on FPGAs is still hard
2.	**Neural Architecture Codesign (NAC)**	Global Search (NSGA-II) & Local Search (Pruning/QAT)
3.	**SNAC-Pack pipeline**	Accelerating search with hardware surrogates.
4.	**The YAML interface**	What you'll edit in today's hands-on
5.	**Case Study:**	Exploring the Jet classification search space.
6.	**MCP showcase**	Running SNAC-Pack through an AI agent

Goal

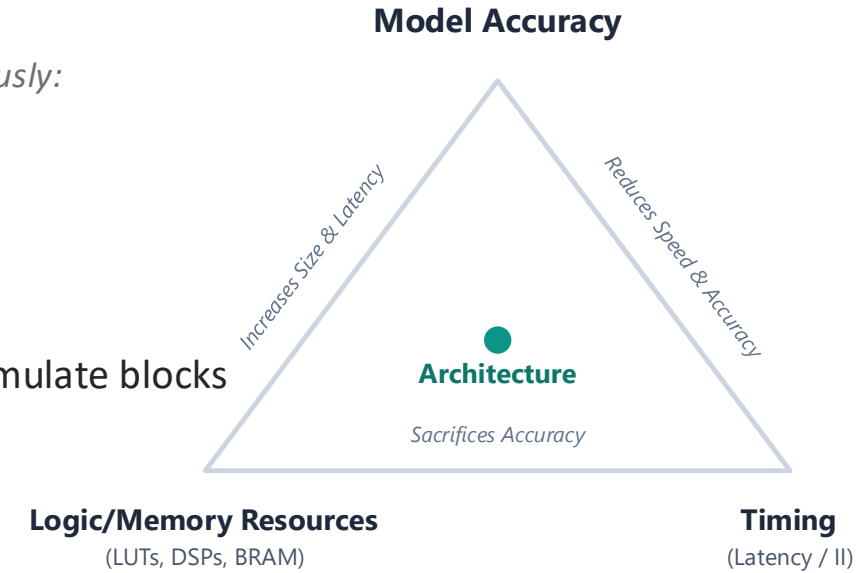
- Help non-ML scientists make cutting edge models through our neural architecture codesign pipeline
- **Ultimate goal:** User specifies the input data, metrics for evaluation, and their constraints and we output a well performing synthesized model



Deploying ML on FPGAs is a multi-constraint problem

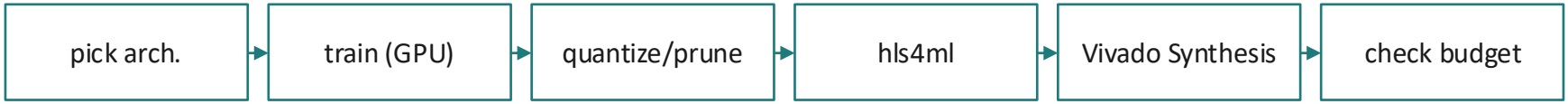
Every architectural choice affects all of these simultaneously:

- **LUTs:** look-up tables (logic fabric)
- **DSPs:** digital signal processor. multiply-accumulate blocks
- **BRAM:** on-chip block RAM
- **Flip-flops:** registers / pipeline state
- **Latency / II:** clock cycles per inference



Layer types, widths, depth, activations, quantization and pruning all interact and they interact differently for every target FPGA.

The manual workflow doesn't scale



loop until it fits taking months for a single qubit-readout model

Months

of design-space exploration per readout model

Manual tuning

for every architecture variation

Minutes–hours

of Vivado synthesis per trial

Hardware-aware codesign

Manual design

Pick an architecture, then hope it fits.

- Hardware feedback comes only at the end
- Re-pick, retrain, re-synthesise, repeat
- Optimises accuracy first, cost second

Codesign

Search architectures with the FPGA in the loop.

- Hardware cost becomes a search objective
- Multi-objective: accuracy + LUTs + DSPs + …
- Output is a Pareto set, not one model

Neural Architecture Search

NAS automates the choice of model architecture.

Search space

What architectures are allowed?

e.g. 4–8 layers, widths in {32, 64, 128}, ReLU / Tanh, with/without BatchNorm.

Search strategy

How do we explore the space?

We use NSGA-II, a genetic algorithm that maintains a Pareto-optimal population.

Evaluation

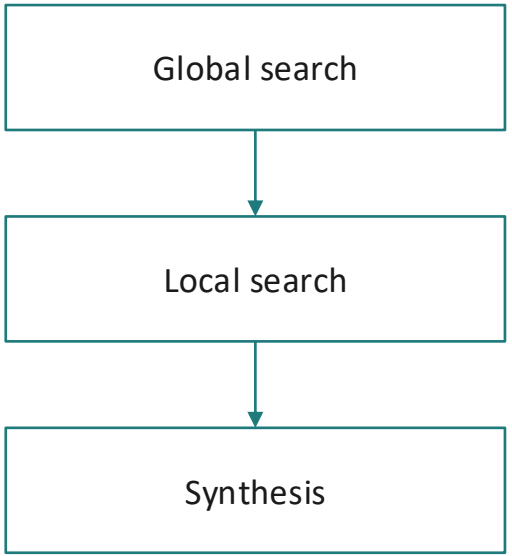
How do we score each candidate?

Validation accuracy + hardware cost (LUTs, DSPs, latency, …).

Neural Architecture Codesign

NAC consists of three stages.

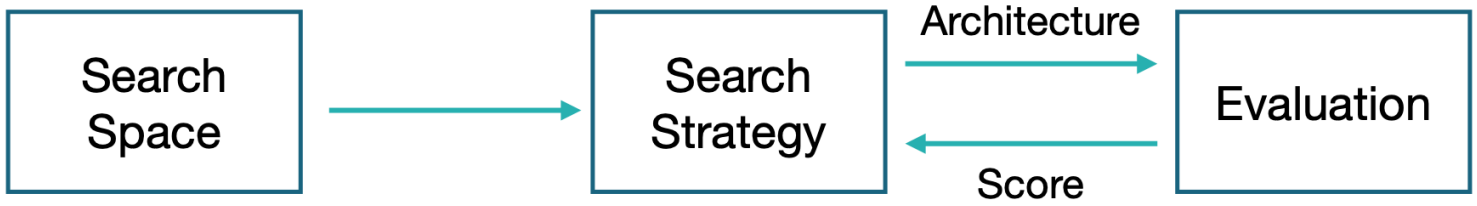
1. Global architecture search
2. Local architecture search
3. hls4ml synthesis

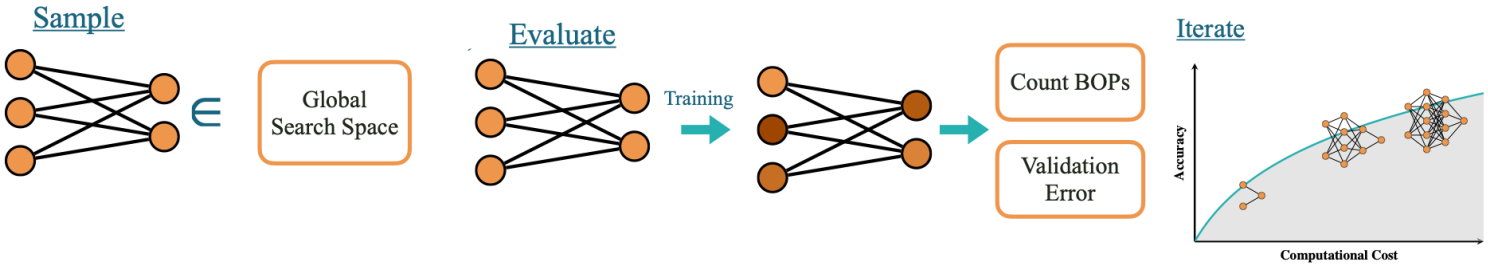


https://arxiv.org/abs/2501.05515

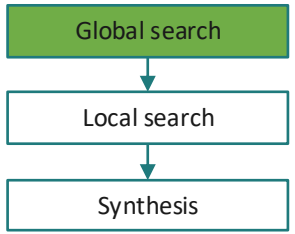
NAC global search

Three blocks (search space → search strategy → evaluation) feed each other in a loop: candidates flow right, scores flow back left.



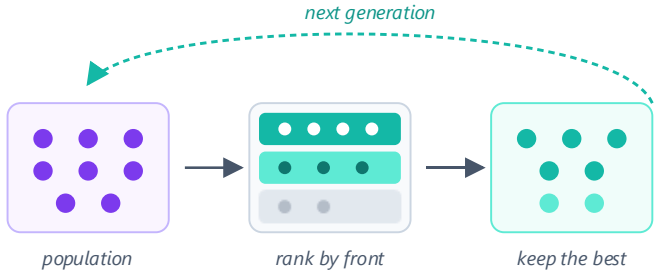


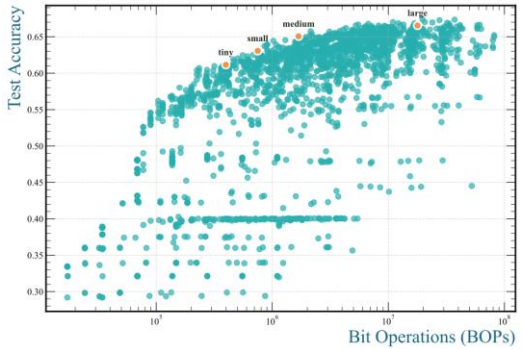
Stage 1 — global search with NSGA-II



NSGA-II keeps a population of architectures and evolves it across generations.

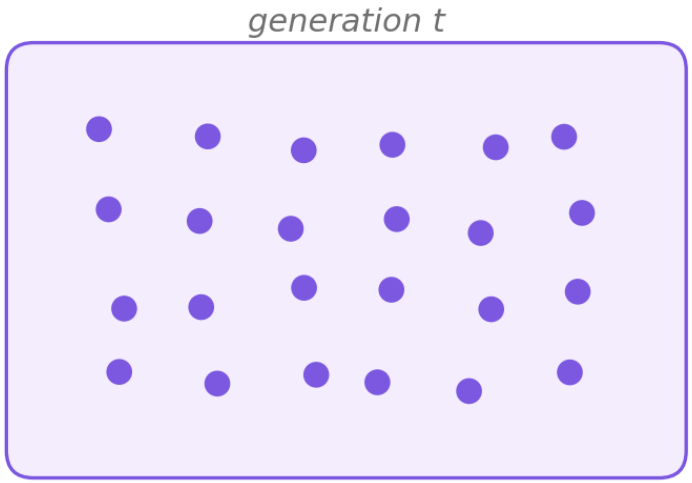
- Mutation and crossover propose new architectures
- Each candidate is trained briefly and scored
- Non-dominated solutions survive to the next generation
- Output is a Pareto front, not a single "best" model





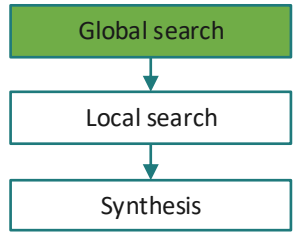
NSGA-II step 1: start with a population

Each dot is a different MLP. NSGA-II evolves the whole cloud, not one model.

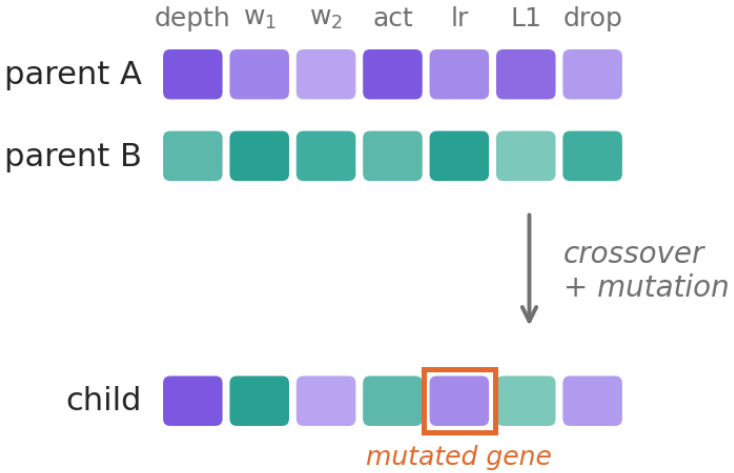


Population: N candidate architectures

NSGA-II step 2: make offspring



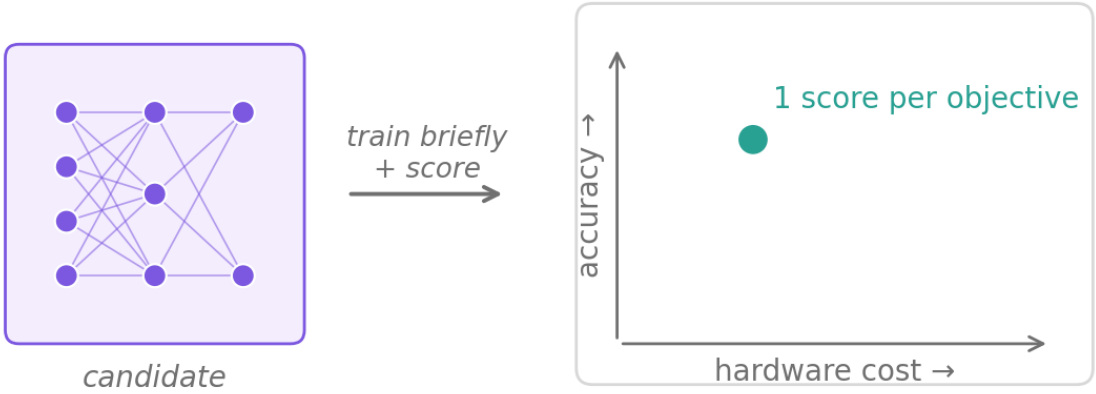
Pick two parents. Splice their gene strips, flip one gene at random.



Genome = architectural choices. Mix two parents, mutate a gene

NSGA-II step 3: evaluate

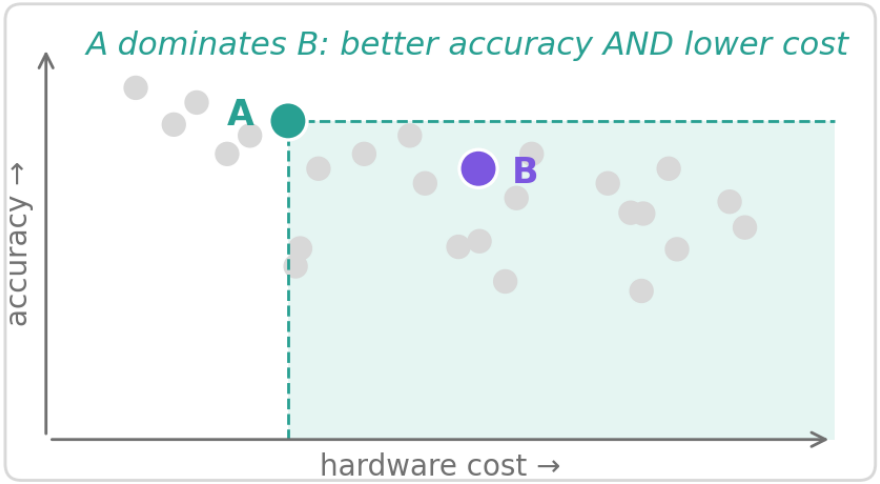
Train the child briefly, plot it in objective space (accuracy vs hardware cost).



Each candidate is trained briefly and plotted in objective space

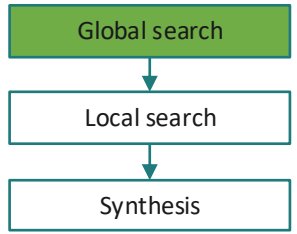
NSGA-II step 4: Pareto dominance

A dominates B if it wins on every axis. The shaded region is everything A dominates.

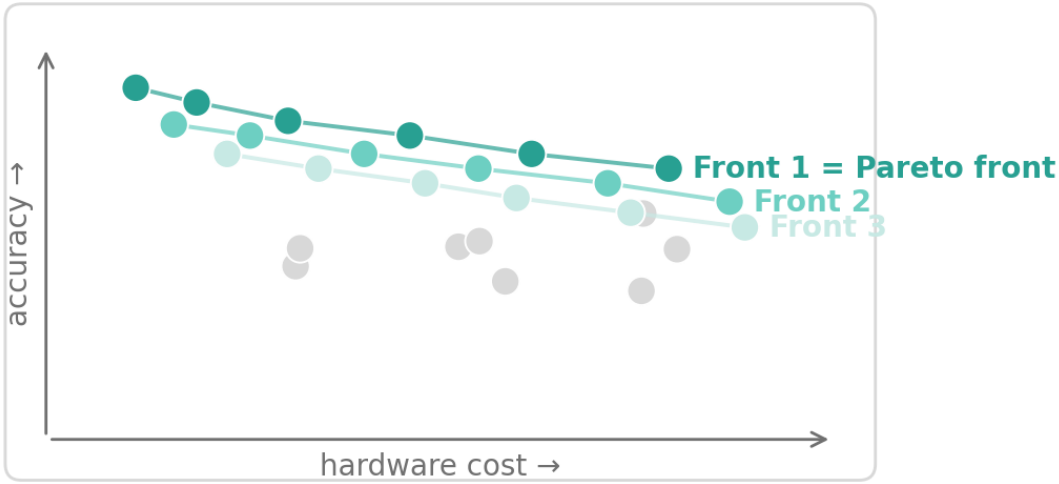


Dominance: better o every axis (at least one strictly)

NSGA-II step 5: non-dominated sort

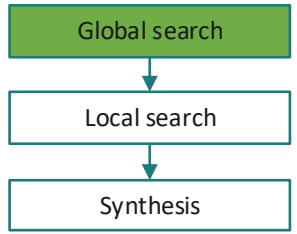


Peel off the Pareto front, then the next non-dominated layer, and so on.

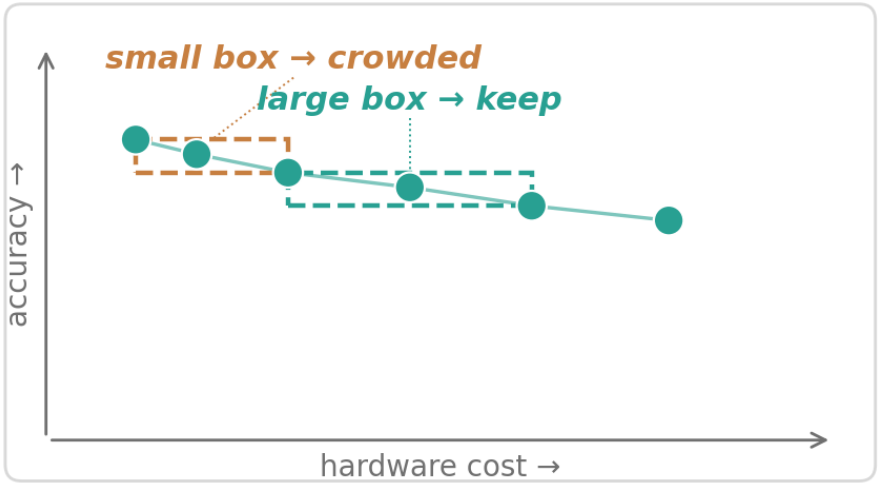


Non-dominated sort: peel off the Pareto front, repeat

NSGA-II step 6: crowding distance



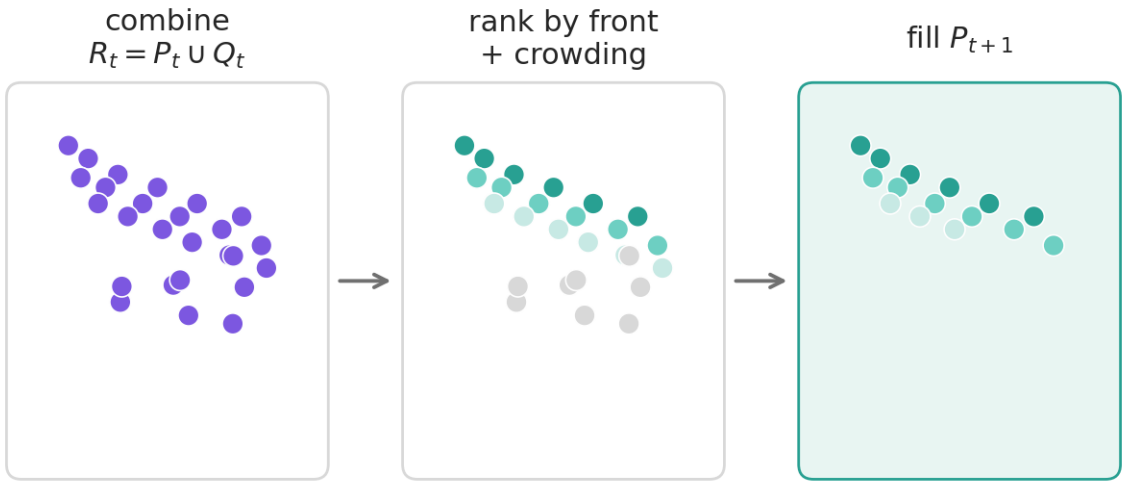
Within a front, prefer isolated points so the front stays spread out.



Crowding distance = box around each points neighbors. Bigger wins.

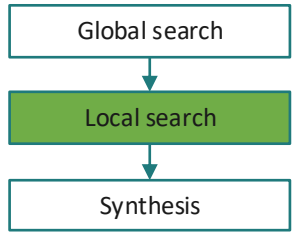
NSGA-II step 7: fill the next generation

Combine parents + offspring, take whole fronts in order, settle the last tie with crowding.

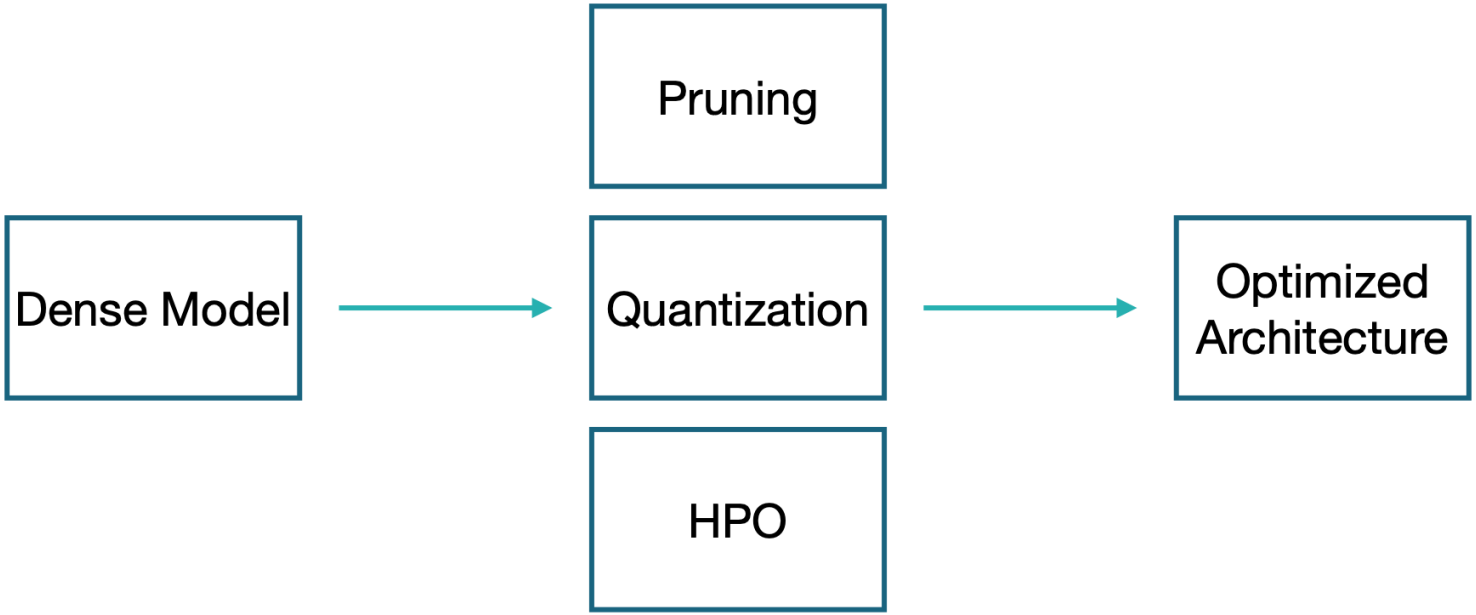


Take whole fronts until full. Break the last tie by crowding distance

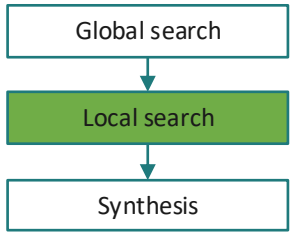
NAC: local search



Take a single architecture from the global stage and compress it.



Quantization: fewer bits per number



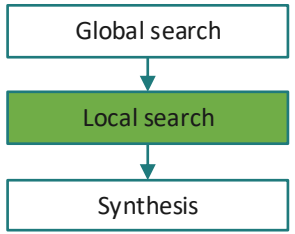
Replace 32-bit floats with low-precision fixed-point numbers → smaller multipliers, less memory, lower latency, in exchange for some accuracy.

- Standard ML training uses 32-bit floats (FP32)
- On FPGAs we use fixed-point: ap_fixed<W, I> — W total bits, I integer bits
- Smaller W → smaller multipliers, fewer LUTs / DSPs
- Too few bits → rounding error → accuracy drops
- Quantization-aware training (QAT) recovers accuracy by simulating the rounding during training

Representation	Bits	Value
FP32	32	0.72341
ap_fixed<16,6>	16	0.72363
ap_fixed<8,3>	8	0.71875
ap_fixed<4,1>	4	0.75000

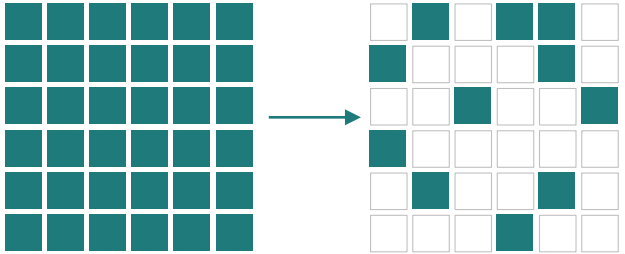
Same weight, four precisions

Pruning: remove weights you don't need

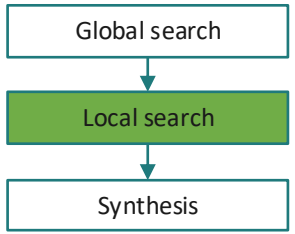


Most networks are over-parameterized: many weights contribute almost nothing. Set them to zero and the model keeps working well.

- Train the model, rank weights by magnitude
- Zero out the smallest fraction (e.g. bottom 50%)
- Re-train to recover accuracy
- Repeat → "iterative magnitude pruning"
- Typical: 70–95% sparsity with little accuracy loss
- Every pruned weight is one multiply you don't have to implement



Iterative magnitude pruning

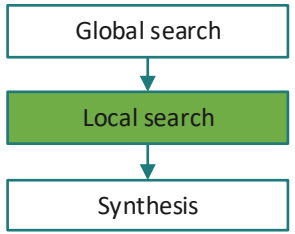


Prune a little, retrain, repeat. gentler than removing everything at once.

1. Train the dense network to convergence
2. Rank weights within each layer by magnitude |w|
3. Zero out the smallest p% (e.g. 20%)
4. Fine-tune the surviving (unmasked) weights for a few epochs
5. Raise the target sparsity and loop back to step 2

Typical schedule: 50% → 70% → 85% → 95% sparsity over 5–10 iterations.

Iterative pruning vs. One-shot pruning



One-shot to 90%

Drop 90% of weights in a single step.

- Many useful weights sit just above the cutoff and they get cut too
- Network has no chance to adapt. whole layers can lose all of their important paths at once
- Final fine-tune often can't recover
- **Result: large accuracy drop, sometimes unrecoverable**

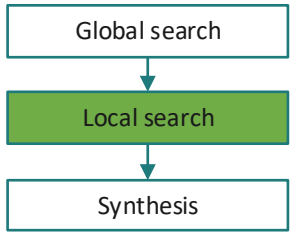
Iterative to 90%

Drop a small fraction each round, then re-train.

- Only the genuinely small weights are removed each round
- Survivors re-train, their magnitudes redistribute
- Cost: more training rounds (~5–10× a single fine-tune)
- **Result: same 90% sparsity, near-zero accuracy loss**

SNAC-Pack's local search uses iterative magnitude pruning

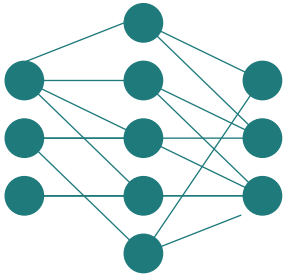
Structured vs unstructured pruning



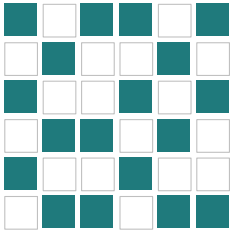
Same idea but two patterns: they trade granularity off against hardware friendliness.

Unstructured

Zero out individual weights — single edges disappear.

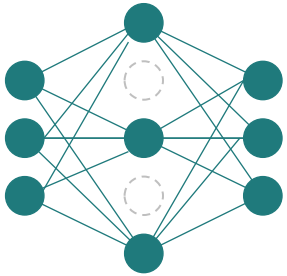


- Any single weight can be removed
- Sparse weight matrix but needs hardware that can skip zeros
- hls4ml maps each surviving weight to its own multiplier

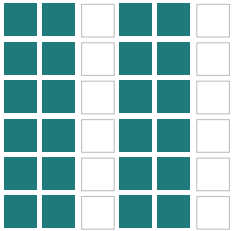


Structured

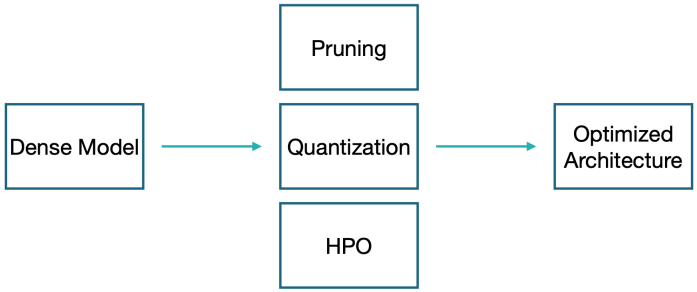
Remove whole neurons (or filters / channels) at once.

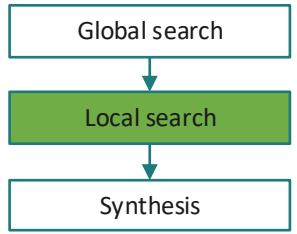


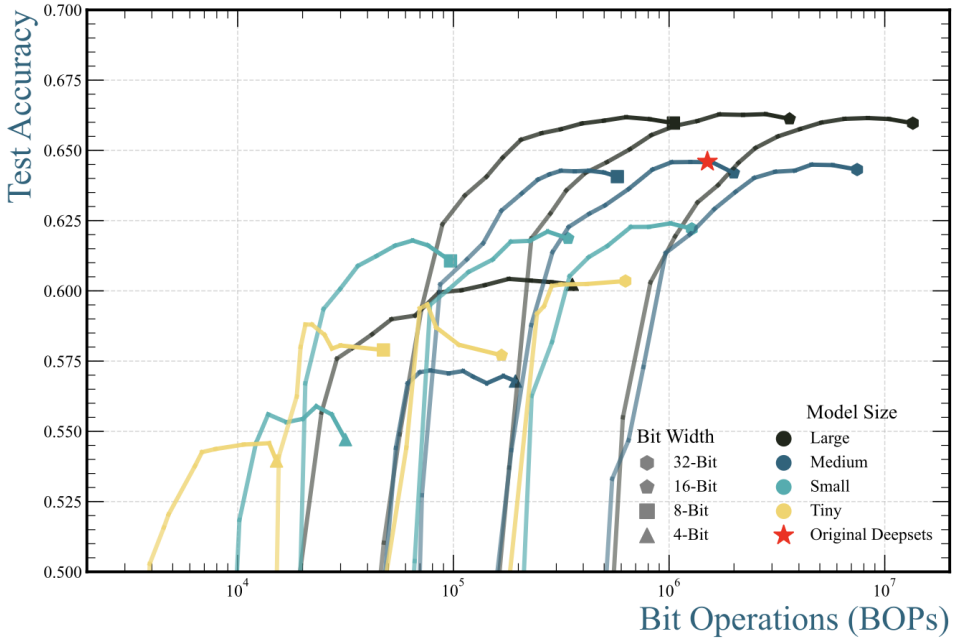
- Entire neurons (or filters) are removed
- Layer widths actually shrink
- Smaller dense network which is easy to accelerate anywhere



NAC: local search

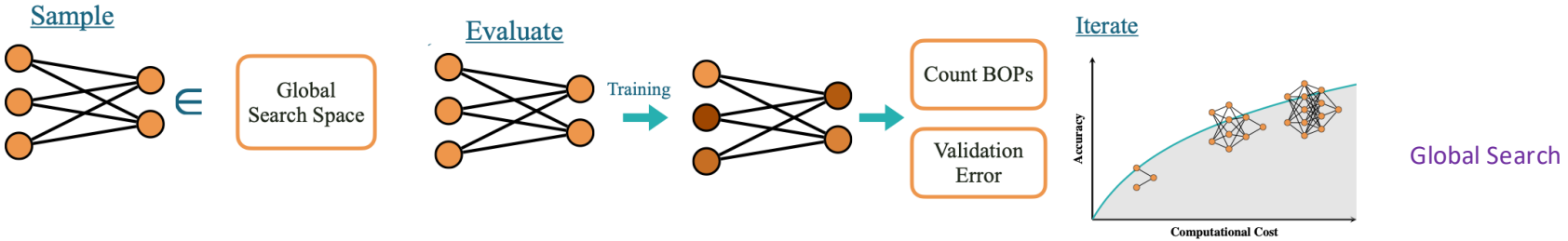




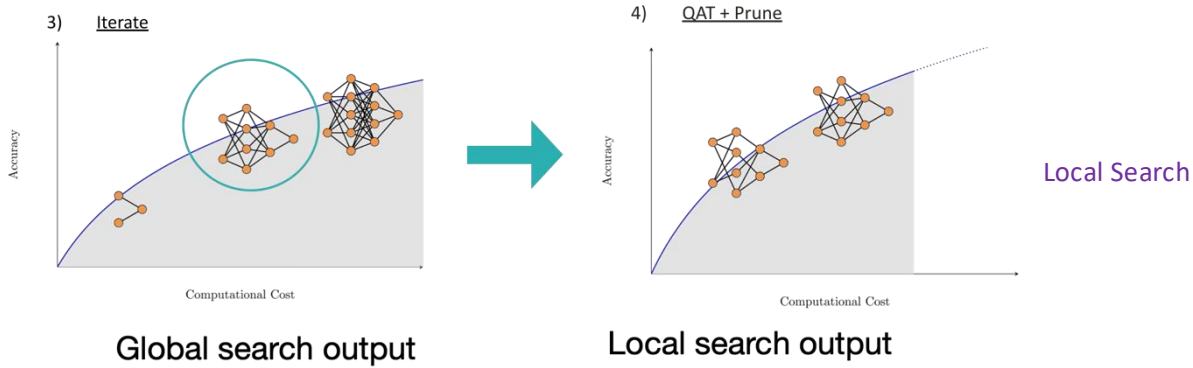


NAC Pipeline

Three stage: Global search, Local search, synthesis



Global Search



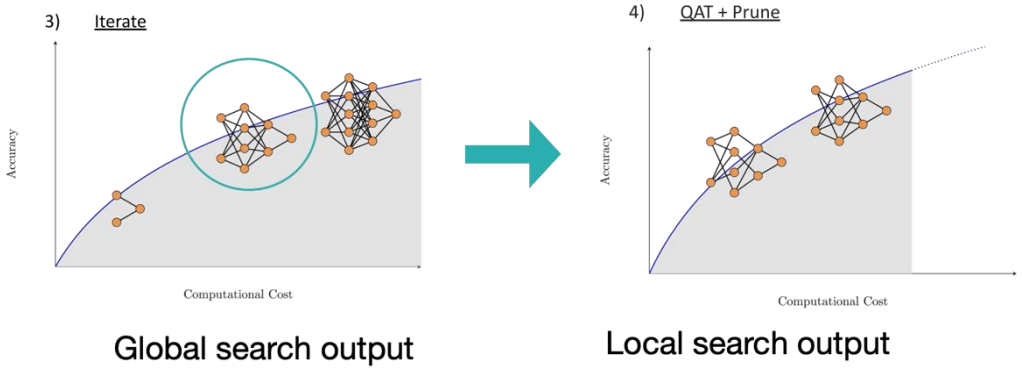
Global search output

Local search output

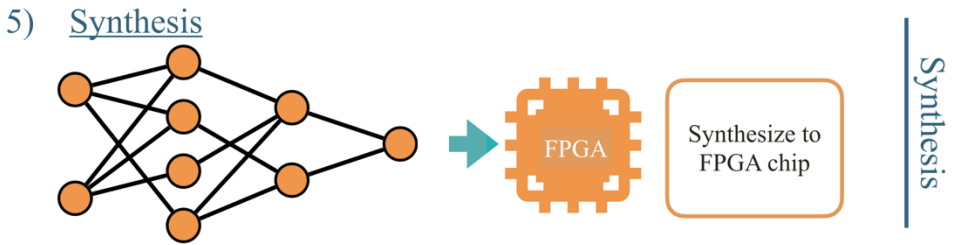
Local Search

NAC Pipeline

Three stage: Global search, Local search, synthesis



Local Search

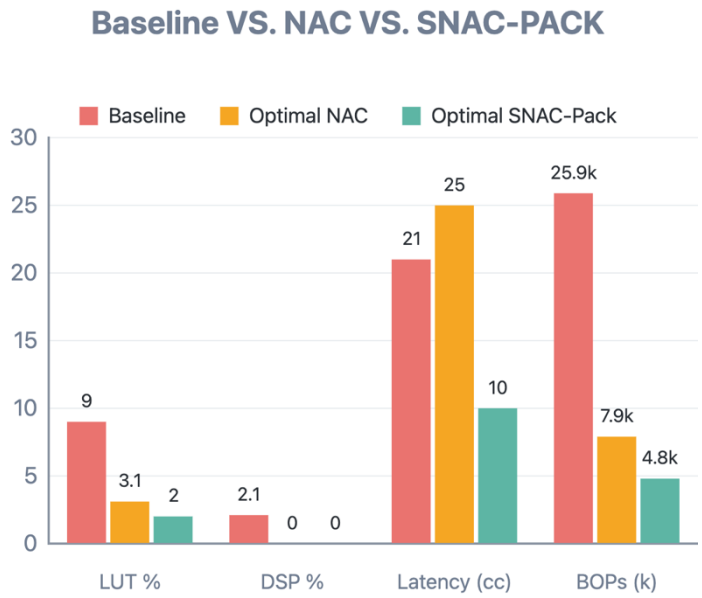


Synthesis

Why we can't just count operations (BOPs)

Most NAS tools score candidates with cheap software proxies such as FLOPs or bit operations (BOPs).

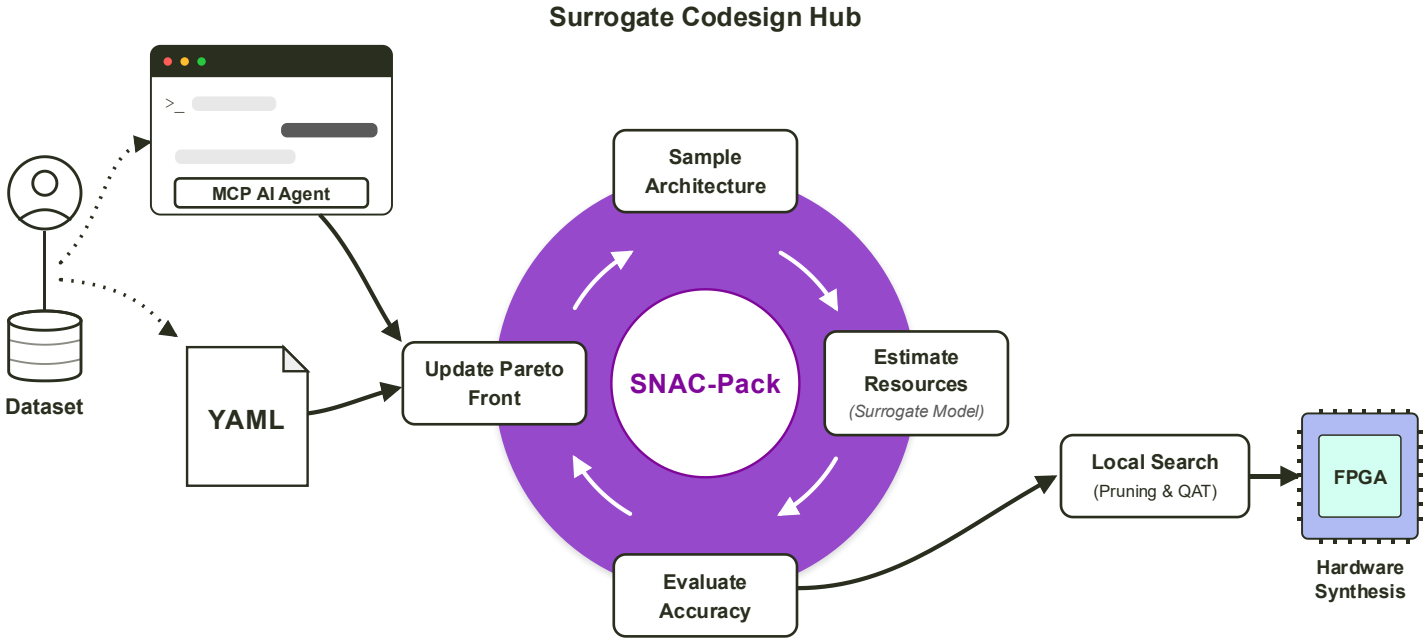
Problem. BOPs ≠ LUTs, DSPs, BRAM, FFs or cycles. Two models with similar BOPs can have very different post-synthesis cost.



http://arxiv.org/abs/2605.16138

SNAC-Pack pipeline

Same two-stage idea as NAC, but the global loop now scores trials with a learned hardware surrogate instead of just BOPs.



arxiv.org/abs/2605.16138

The surrogate: hardware cost in milliseconds

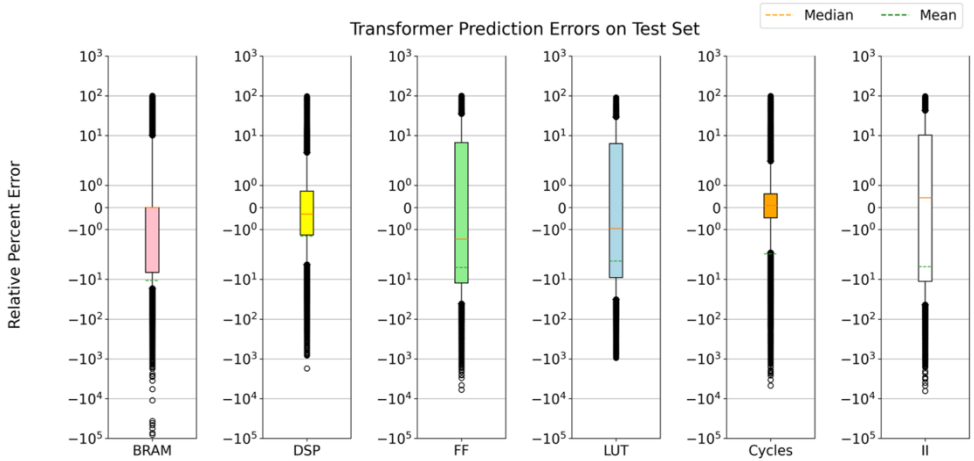
Replaces a Vivado run with a learned predictor which is trained on real hls4ml syntheses so it knows what LUTs/DSPs/BRAM/FFs/latency a given Keras model and HLS config will actually use.

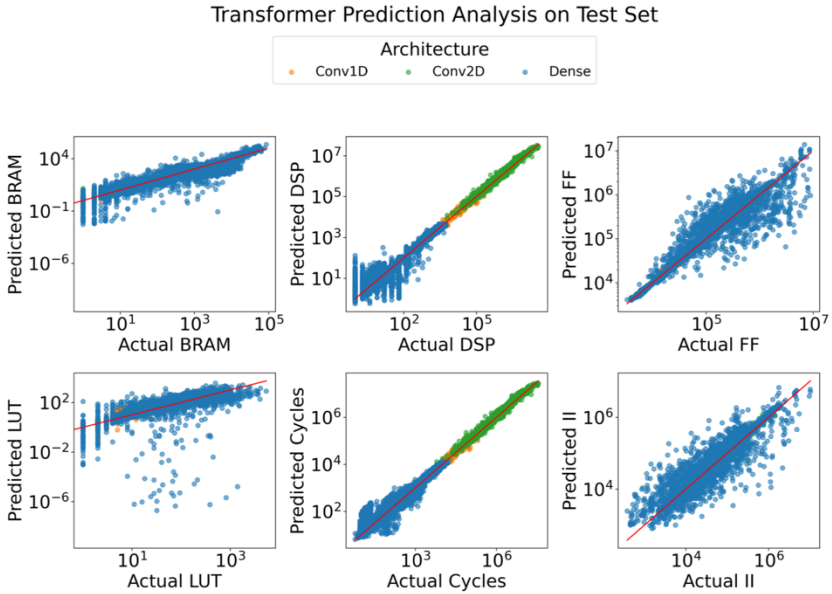
arxiv.org/abs/2408.05314

The surrogate: Rule4ml & wa_hls4ml

- Dataset:
  - 683,176 NNs
  - Dense 2-20 layer, resource and latency, io_parallel
  - Conv 3-7 layer, resource, io_stream
  - Targeting the Pynq-Z2, ZCU102, and Alveo-U200 & U250, FPGA boards
- Surrogate Model:
  - Evaluated LSTM, Transformer, Random Forest, Gradient Boosted Trees, and Multi-Layer Perceptrons (MLP) and most recently Graph Neural Networks (GNNs)
  - The final selected architecture for the predictors is MLP/GNN

The surrogate: wa_hls4ml





http://arxiv.org/abs/2511.05615

What you'll edit: one YAML file

*Datasets, search space, objectives, hardware, compression **all in one config.***

t3_config.yaml

```yaml
dataset:
  name: qubit
  window_size: 400

search:
  n_trials: 500
  objectives: [fidelity, resources, cc]
  use_hw_metrics: true

local_search:
  precision_pairs: [(16,6), (8,3), (4,1)]
  pruning_iterations: 10

synthesis:
  board: zcu102
  reuse_factor: 8
```

`dataset`

task and preprocessing options

`search`

trial budget, objectives, hardware mode

`local_search`

precision pairs and pruning schedule

`synthesis`

board, reuse factor, HLS strategy

What gets searched: jet-classification space

A concrete example of what lives inside the YAML search_space block.

Parameter	Search options
Number of layers	`{4, 5, 6, 7, 8}`
Layer 1 width	`{64, 120, 128}`
Layer 2 width	`{32, 60, 64}`
Activation function	`{ReLU, Tanh, Sigmoid}`
Batch normalization	`{True, False}`
Learning rate	`{0.001, 0.0015, 0.002}`
L1 regularization	`{0, 1e-6, 1e-5, 1e-4}`
Dropout rate	`{0, 0.05, 0.1}`

Other tasks (qubit, Bragg peak) have their own search blocks, but the structure is the same.

Bonus: an AI-agent front-end (optional)

Same YAML pipeline, exposed as Model Context Protocol tools for agentic agents.

User

“Search a small MLP for jet tagging on VU13P, 500 trials.”

Agent (MCP)

```
→ create_search_config(...)
→ run_search_pipeline_from_spec(...)
→ run_local_search(...)
→ read_search_results(...)
```

Same tool surface

Global and local search are all callable as MCP tools.

Sandboxed paths

The MCP layer keeps file access scoped to the repo.

Today → YAML only

We'll use YAML directly in the hands-on tutorial.

Questions!


Snac pack
mmm...delicious
MADE WITH REAL MILK
Super Neural Architecture Codesign Package
PUDDING
4-3.25 OZ CUPS NET WT 13 OZ (368g)
100 Calories PER PUDDING CUP
MADE WITH Resource Estimation

Accessing the SNAC-Pack Tutorial

- Join the hls4ml-tutorial team of Github
  - If not already member: fill out this form: https://forms.gle/Mk7EvDvVfY9wsVbMA
- Once joined, go to https://snac-tutorials.fastmachinelearning.org/
- If you want to run locally instead:
  - https://github.com/fastmachinelearning/nac-opt/tree/Real-time-tutorial

Form if not on github team: