

High Granularity Quantization



Chang Sun

Paper: <https://arxiv.org/abs/2405.00645>
Repository: <https://github.com/calad0i/HGQ>
Full Examples: <https://github.com/calad0i/HGQ-demos>

Target Audience

- You need neural networks running on FPGAs with super low latency
 - e.g.: LHC L1 triggers
- You are familiar with python

Motivation: FastML@L1

- Issue

- $O(100\text{ns})$ latency
- Limited on-chip resource

GAs to retrieve full hit data provides a
the first-level muon trigger's performance.
a latency within $O(100\text{ ns})$ are required.
new system is a fast tracking algorithm

system in a particle detector at the CERN
extreme environments in which one can
s. Latency is restricted to $O(1)\mu\text{s}$, gov-
of particle collisions and the number of
system consists of a limited amount of

to test in order to assess the time pedestal a
has however to provide robust and reliable l
maximum latency within a few microseconds
of spurious signal combinations.

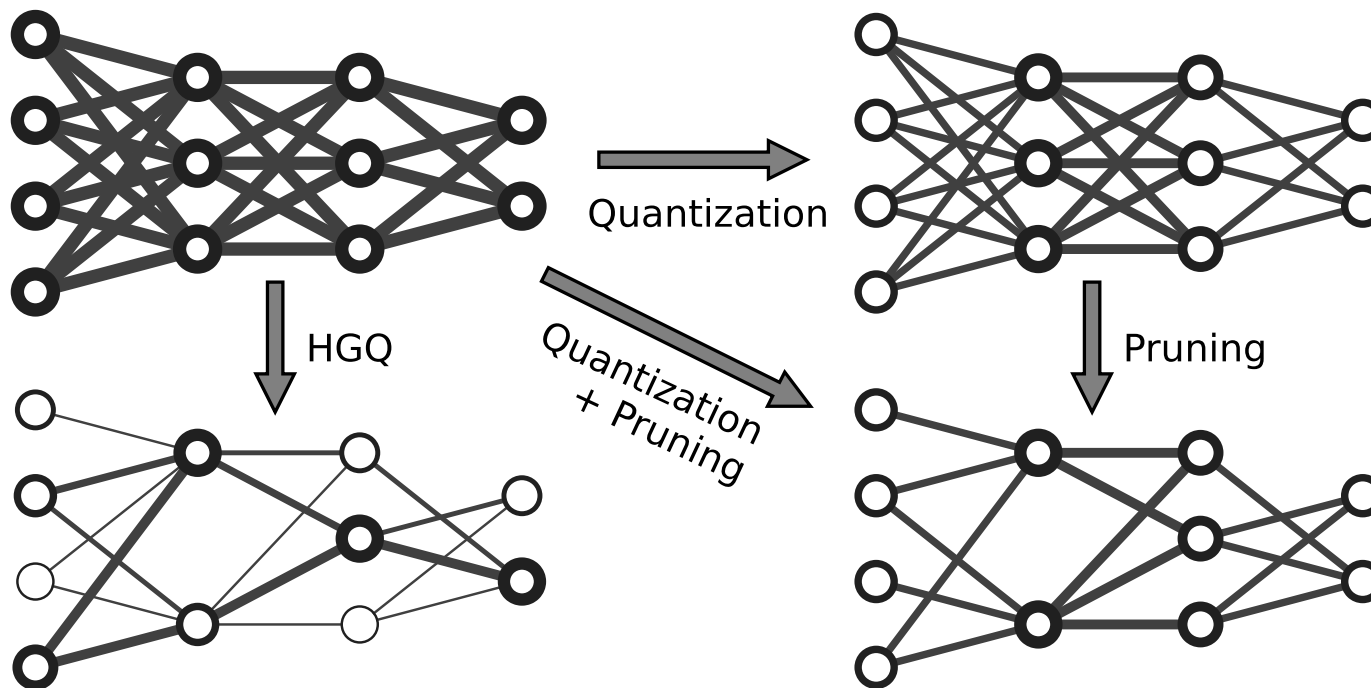
ly analytical approaches to the problem can

FastML@L1

- Issue
 - $O(100\text{ns})$ latency
 - Tight onboard resource constraint
- Current approaches
 - Use FPGAs with latency strategy on hls4ml
 - Smaller networks
 - Network compression
 - Quantization
 - Pruning

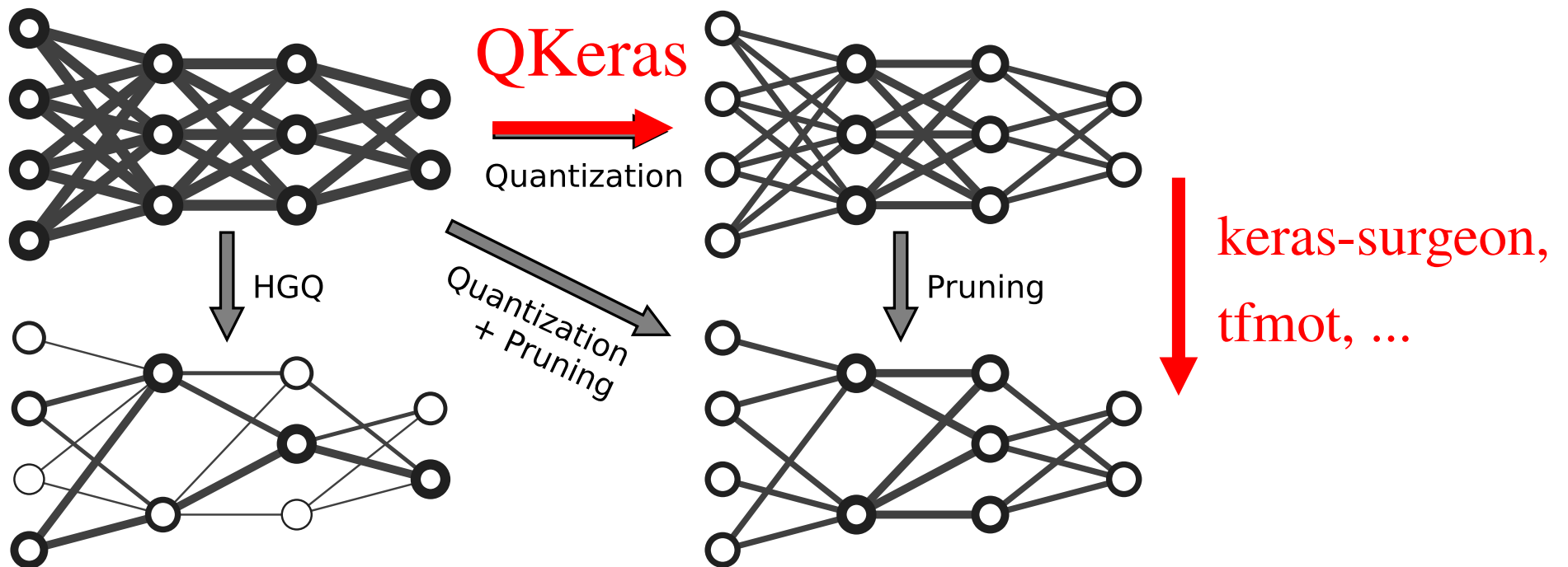
What does HGQ do

- HGQ optimizes the bitwidths of weights or activations at arbitrary fine granularity with gradient descents
 - Pruning is automatically done as $bw \rightarrow 0$
 - You can benefit from any small bitwidth anywhere, not only regular int 4/8/16



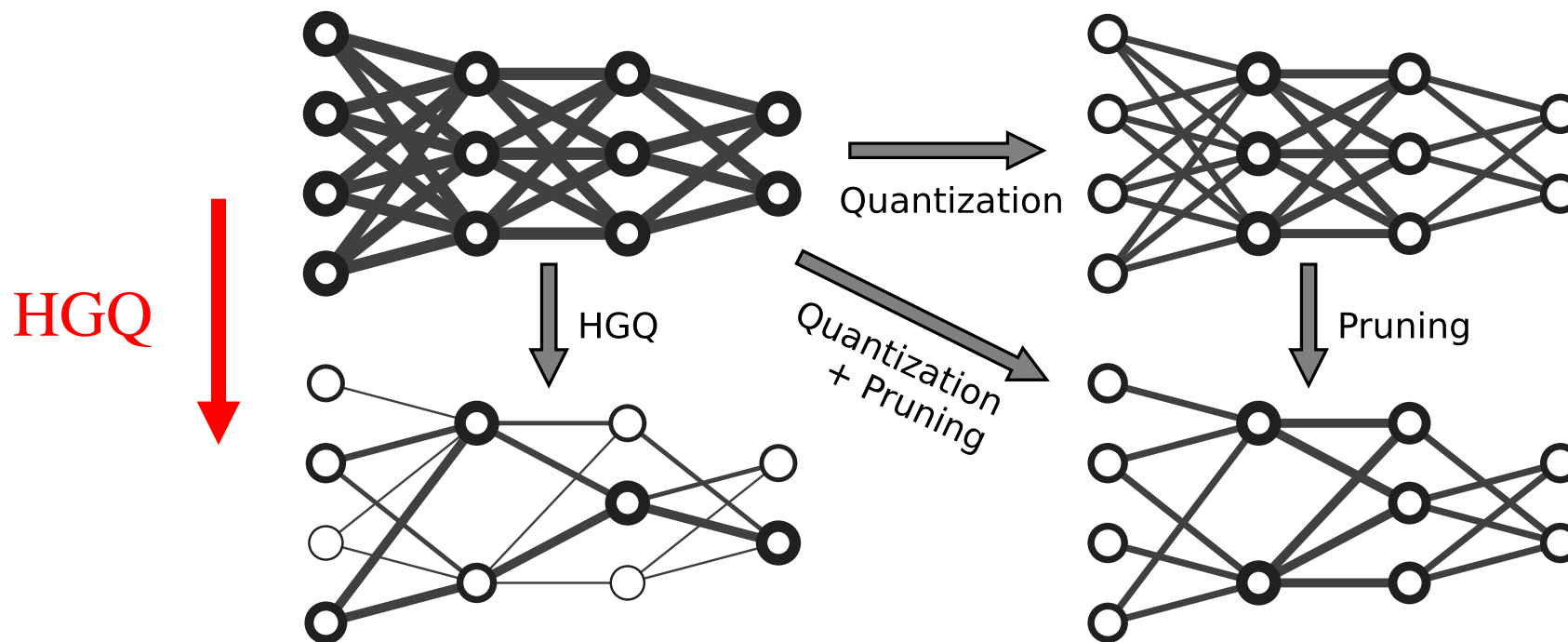
What does HGQ do

- HGQ optimizes the bitwidths of weights or activations at arbitrary fine granularity with gradient descents
 - Pruning is automatically done as $bw \rightarrow 0$
 - You can benefit from any small bitwidth anywhere, not only regular int 4/8/16



What does HGQ do

- HGQ optimizes the bitwidths of weights or activations at arbitrary fine granularity with gradient descents
 - Pruning is automatically done as $bw \rightarrow 0$
 - You can benefit from any small bitwidth anywhere, not only regular int 4/8/16



What is HGQ

- An adaptive QAT algorithm with differentiable bitwidth, and a production-ready framework implementing it

What does HGQ

- HGQ algorithm optimizes the bitwidths of weights or activations at arbitrary fine granularity with gradient descents
 - Any parameter anywhere, like, per-parameter for fully unrolled ones
 - We can benefit from any small bitwidth with FPGAs, not only regular int 4/8/16
 - Pruning is automatically done as $bw \rightarrow 0$
 - “scale invariance”: resource utilization no longer scales with layer size

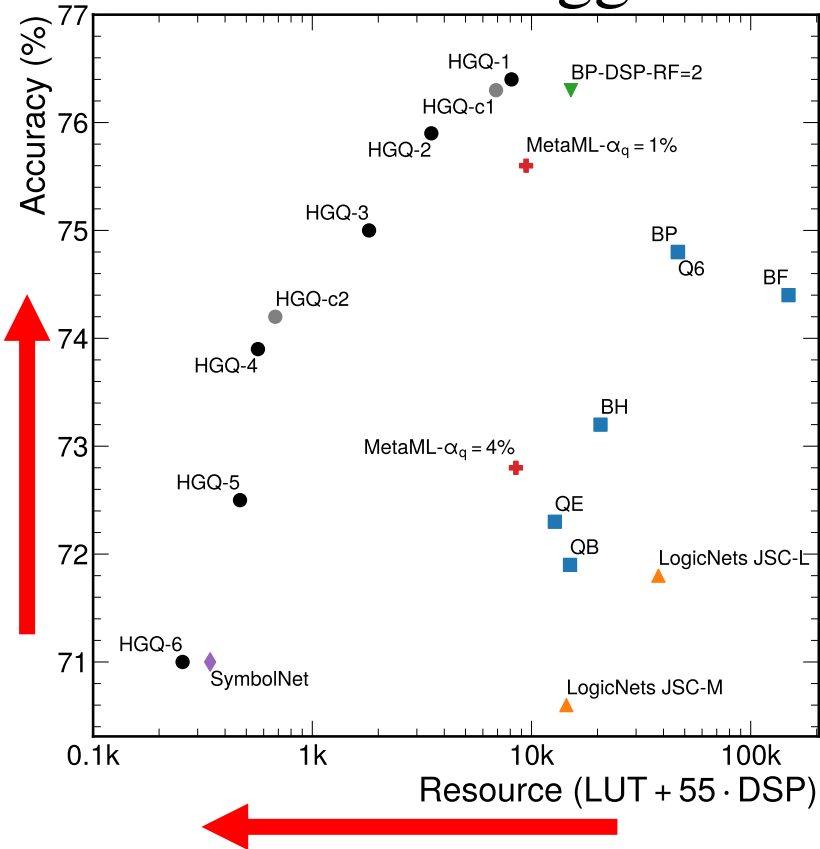
What does HGQ do

- HGQ framework will let hls4ml generate firmware with exactly the same results as in python
 - This is not given for a general QKeras → hls4ml conversion
- HGQ framework offers accurate train-time resource consumption estimation
 - RTL Synthesis is extremely time consuming. This will give we an idea at early stage on how large the firmware will be.

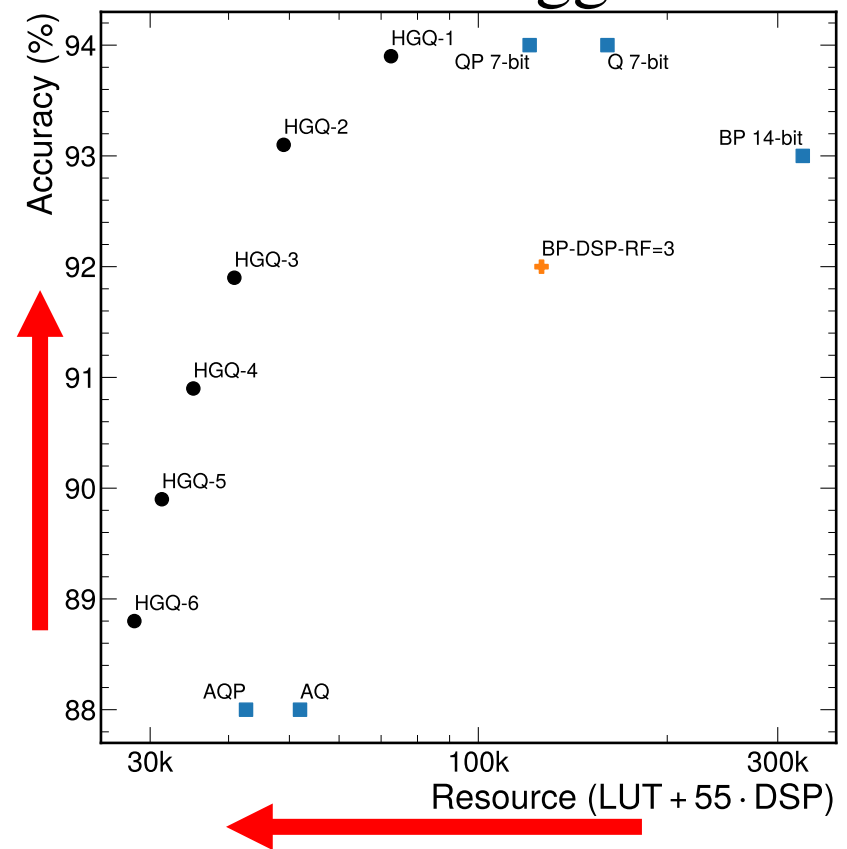
HGQ vs Qkeras and others

- Performance – Resource trade-off (in one run)

Small Jet Tagger



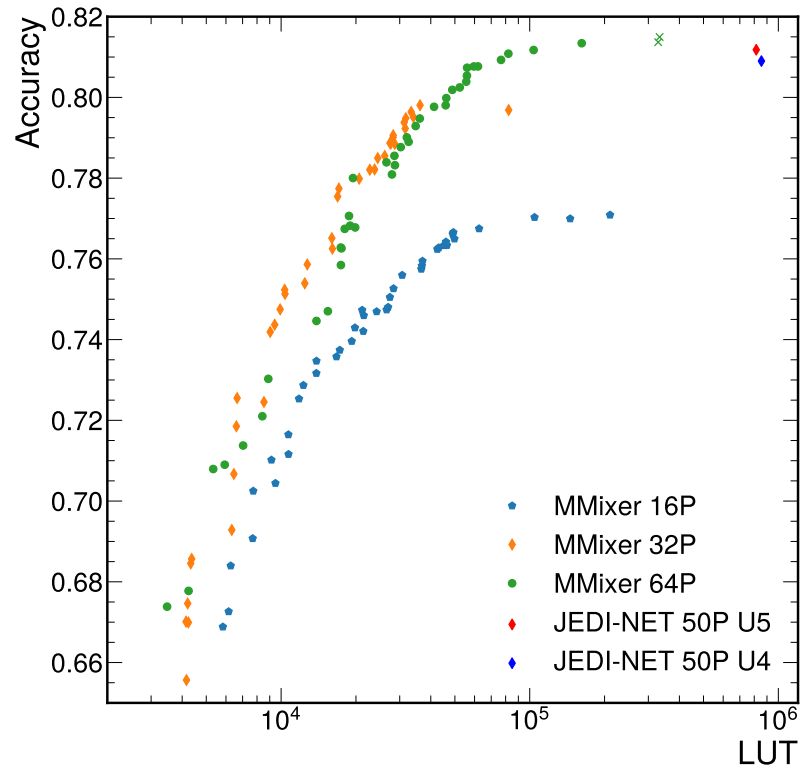
SVHN Tagger



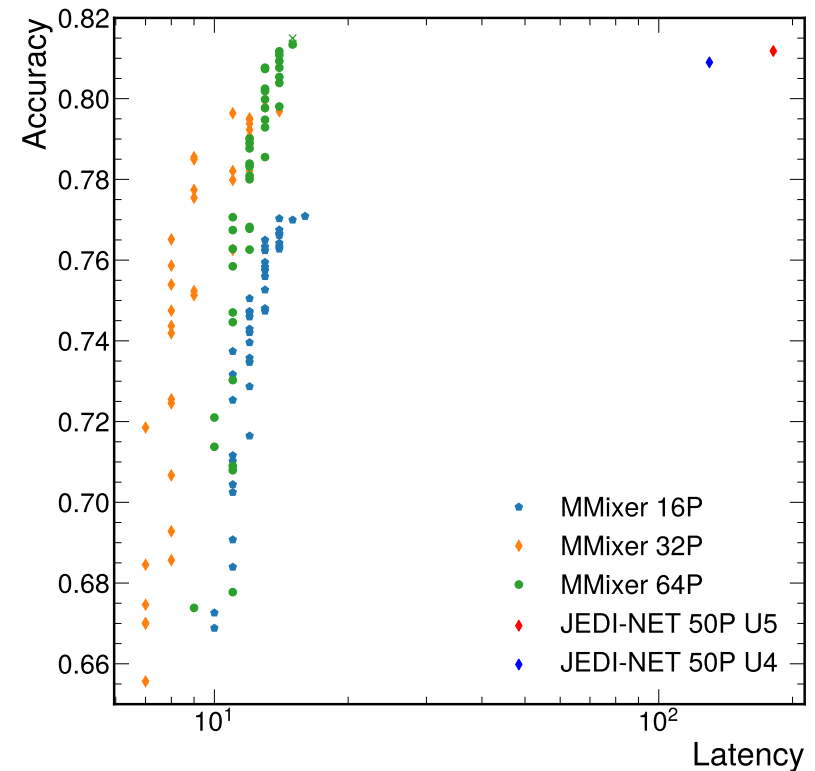
HGQ vs Qkeras and others

- Performance – Resource trade-off (in one run)

Large Jet Tagger Resource

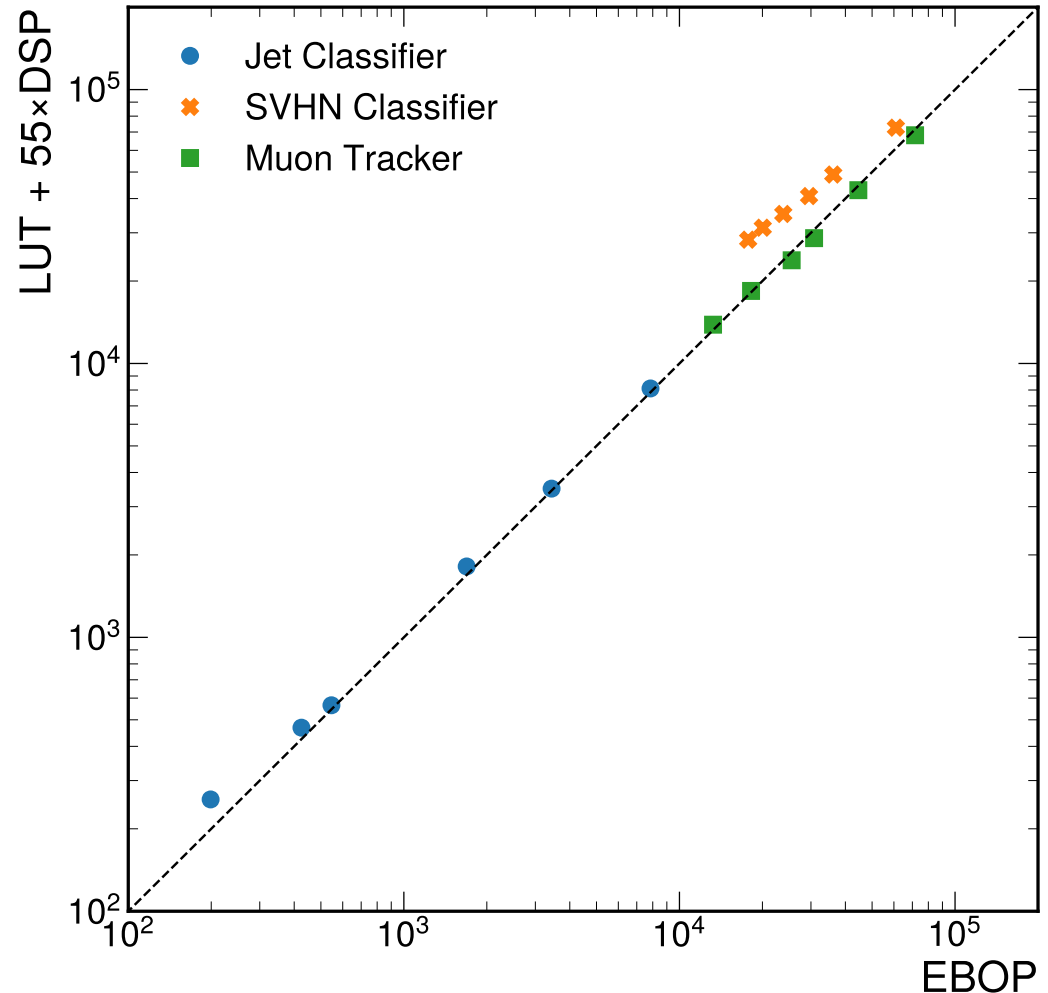


Large Jet Tagger Latency



HGQ

- With good resource estimation on the fly
 - EBOP is the estimator
- Having resource estimation at early stage is useful for software-hardware co-design
 - One don't need to wait for hours for vivado/vitis synth



HGQ

- “Scale invariant”: Resource does not scale with “model size on paper”
 - With the automatic pruning, similar submodel will be used no matter how big it was.
 - Can be used as NAS that sample subnetworks from a supernetwork.

How does HGQ work – Gradients for BW

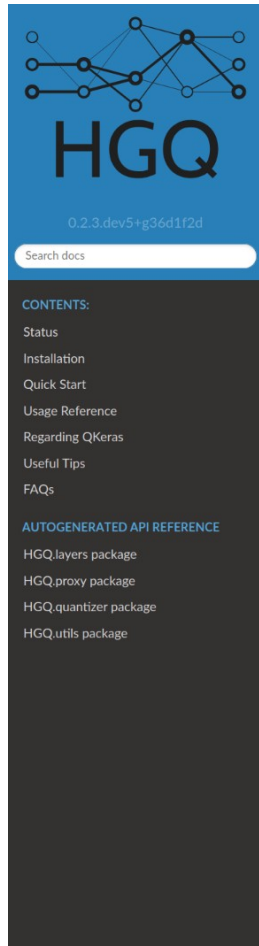
- The model keeps only the number of float bits, \mathbf{f} . The number of integer bits are determined passively.
- We have a surrogate gradient for \mathbf{f} from the model loss:
 - $\frac{\partial \delta_f}{\partial \mathbf{f}} \leftarrow -\log 2 \cdot \delta_f$, where $\delta_f \equiv x - f^q(x)$ is the quantization error
 - See full derivation in the paper
- If \mathbf{f} is small enough, the output value is constantly zero, and we effectively pruned the corresponding parameter(s).

How does HGQ work – Gradients for BW

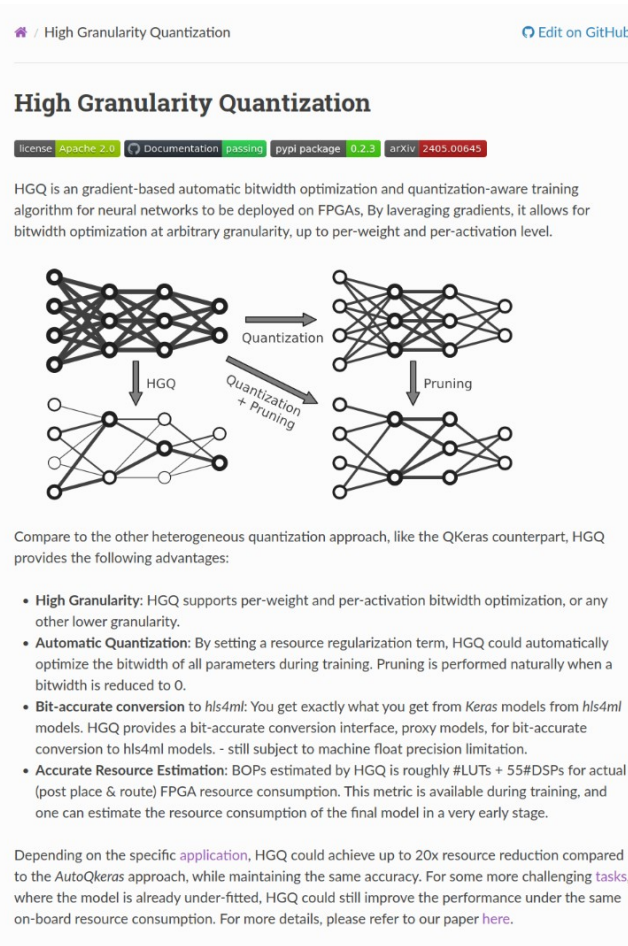
- The gradient in the previous page encourages large \mathbf{f} , and we need to keep it down to optimize for resource
- We use Effective Bit-Operations (EBOPs) as the regulation term
 - Basically BOPs with real bitwidth on a per-parameter base and ignoring accumulations
- And add a small L1 loss on \mathbf{f} everywhere
 - for some parameter does not result in additional EBOPs
- Final loss: $\mathcal{L} = \mathcal{L}_{\text{base}} + \beta \cdot \overline{\text{EBOPs}} + \gamma \cdot \text{L1}_{\text{norm}}$

How to use HGQ

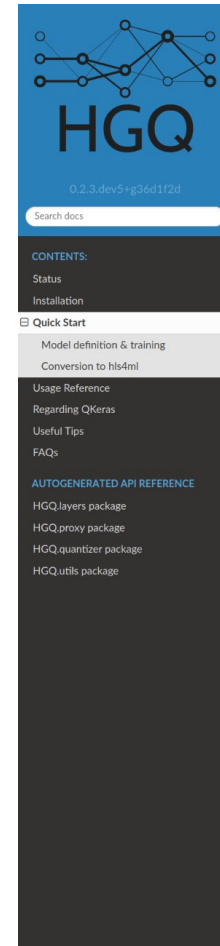
- Documentation: <https://calad0i.github.io/HGQ/>



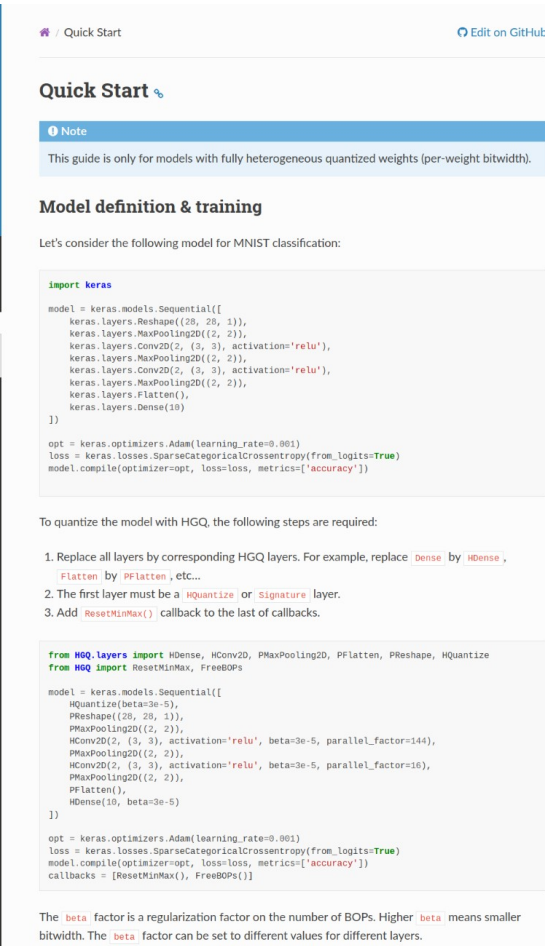
The image shows the top part of the HGQ documentation website. It features the HGQ logo, the version number 0.2.3.dev5+g36d1f2d, a search bar, and a navigation menu with links to Contents, Status, Installation, Quick Start, Usage Reference, Regarding QKeras, Useful Tips, FAQs, and an Auto-generated API Reference section.



This section of the documentation explains the High Granularity Quantization (HGQ) process. It includes a diagram showing a neural network being processed through Quantization, HGQ, and Pruning to reach a final state. The text describes HGQ as a gradient-based automatic bitwidth optimization and quantization-aware training algorithm for neural networks to be deployed on FPGAs. It lists advantages such as High Granularity, Automatic Quantization, Bit-accurate conversion to hls4ml, and Accurate Resource Estimation. A note at the bottom states that HGQ can achieve up to 20x resource reduction compared to the AutoQKeras approach.



This section shows the 'Quick Start' page of the HGQ documentation. It includes the HGQ logo, version number, a search bar, and a navigation menu with links to Contents, Status, Installation, Quick Start, Model definition & training, Usage Reference, Regarding QKeras, Useful Tips, and FAQs. The 'AUTOGENERATED API REFERENCE' section lists packages like HGQ.layers, HGQ.proxy, HGQ.quantizer, and HGQ.utils.



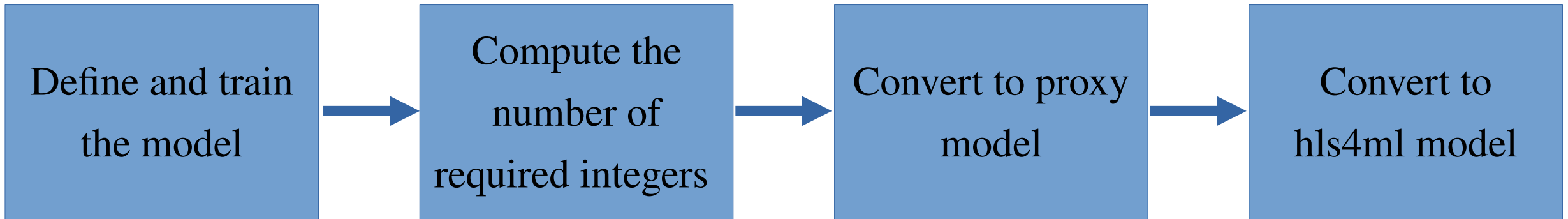
This section contains the 'Quick Start' content, including a note that the guide is for models with fully heterogeneous quantized weights, a code example for model definition and training, and a list of steps to quantize a model. The code example shows the import of HGQ modules and the use of HQuantize and HConv2D layers. The steps include replacing layers with HGQ counterparts, ensuring the first layer is a HQuantize or Signature layer, and adding a ResetMinMax callback.

How to use HGQ

- Documentation: <https://calad0i.github.io/HGQ/>
- Repository and complete examples:
 - <https://calad0i.github.io/HGQ/>
 - <https://github.com/calad0i/HGQ-demos>

How to use HGQ

- Documentation: <https://calad0i.github.io/HGQ/>
- Repository and complete examples:
 - <https://calad0i.github.io/HGQ/>
 - <https://github.com/calad0i/HGQ-demos>
- Interactive Example: <https://www.kaggle.com/code/calad0i/small-jet-tagger-with-hgq-1>



End of slides – Let's go to the code

S-QUARK: Scalable Quantization-Aware Realtime Keras (HGQ v2)

Project Page: <https://github.com/calad0i/s-quark> (plan to beta in 2 weeks)

- Everything from HGQ v1
 - HGQ itself for all common layers ✓
 - Bit accurate conversion and synthesis ⚠
 - EBOPs for resource estimation ✓
- Multi-backend support
 - Both in terms of training ✓ and synthesis ✗
- More quantizers
 - v1 can do fixed integer with wrap overflow, add saturation based modes ✓
 - QKeras emulation ⚠
 - And minifloat with differentiable bitwidth ✓
- Others
 - Full jit compile for TF and Jax ✓
 - QKeras compatible API interface ⚠

Backups

How does HGQ work - Quantizer

- Given **b** bits and **i** integer bits: define **f = b-i**
 - ⚠️ excluding the sign bit if presents (e.g., included in ap_fixed in vivado_hls)
- The (signed) QKeras quantizer works as (SAT overflow mode)
 - $v_q = \text{clip}(\text{round}(2^f \times v) / 2^f, -2^i, 2^i - 2^{-f})$

How does HGQ work - Quantizer

- Given **b** bits and **i** integer bits: define **f = b-i**
 - ⚠️ excluding the sign bit if presents (e.g., included in ap_fixed in vivado_hls)
- The HGQ quantizer works as (WRAP overflow mode)
 - Train time
 - $v_q = \text{round}(2^f \times v) / 2^f$
 - $i = \max(\lfloor \log_2 |v_{\max}^q| \rfloor + 1, \lceil \log_2 |v_{\min}^q| \rceil)$
 - Test time
 - $v_q = \text{wrap}(\text{round}(2^f \times v) / 2^f, -2^i, 2^i - 2^{-f})$

How does HGQ work - Quantizer

- Why the trouble?

How does HGQ work - Quantizer

- Why the trouble? – Saturation is expensive

1. Using the AP_SAT* modes can result in higher resource usage as extra logic will be needed to perform saturation and this extra cost can be as high as 20% additional LUT usage.

- And the difference can be enormous!

- Example: `hls4ml/example-models/keras/qkeras_3layer`

- AP_WRAP mode: **9** clk@5ns, 31439 LUT

- AP_SAT: **16** clk@5ns, 27263 LUT




There is another pattern where resource changes a lot but not this much in latency

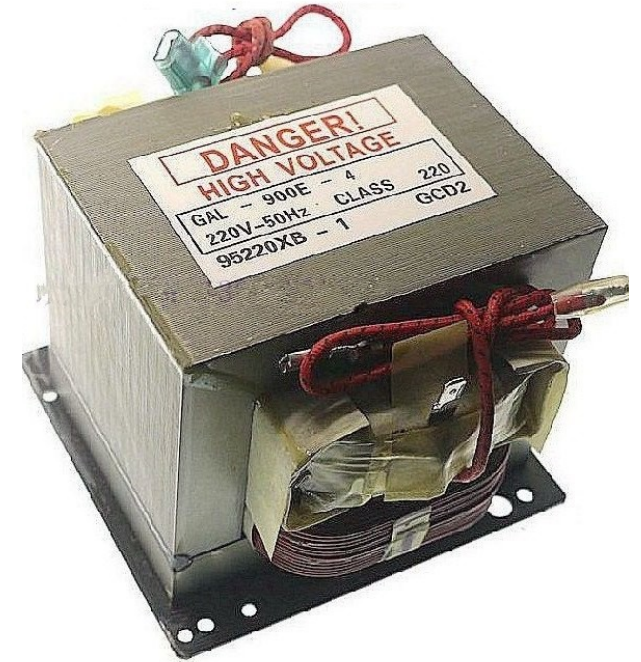
Quantizers

- Fixed-point numbers
 - Two parameterizations (it seems that 1 is for activation, and 2 is better for weights)
 - keep_negative, integers, float
 - keep_negative, width, integers
 - Round mode: floor, (stochastic) round, (stochastic) round_conv
 - Overflow mode: wrap around, saturation, and symmetric saturation
- Minifloat ([2311.12359](#))
 - Type parameterized by #bits of Mantissa, Exponent, and Zero point of Exponent
 - All have gradient, of course
 - IEEE-754 like, with subnormal support
 - But no special numbers like NaN or +/-inf
 - With hls support
 - Conversion from/to fixed point in runtime, multiply with fixed point → fixed point
 - (planned) multiply with minifloat to fixed-point

Layer support

Everything is fully quantized and (in theory) hls4ml-friendly

- Supported  : Currently no hls4ml support
 - Dense (with fused batchnorm)
 - EinsumDense (with fused batchnorm) 
 - Conv*D
 - BatchNormalization
 - UnaryActivation
 - Softmax
 - MultiHeadAttention with softmax attention 
- No need to implement fully passive layers
 - No need to pass bitwidth info across layers



Layer support

- Planned
 - HLS codegen for einsum dense
 - General support for latency & parallel io
 - Support specific patterns for latency & stream io
 - Pooling layers with EBOPs
 - Test and finalize Softmax and MultiHeadAttention
 - Train some practical model
 - And add cossim attention (as used in [2111.09883](#)).
 - Masked Average Pooling