PQuantML: Streamlining ML Model Compression to Deployment for Next-Gen Detector Systems

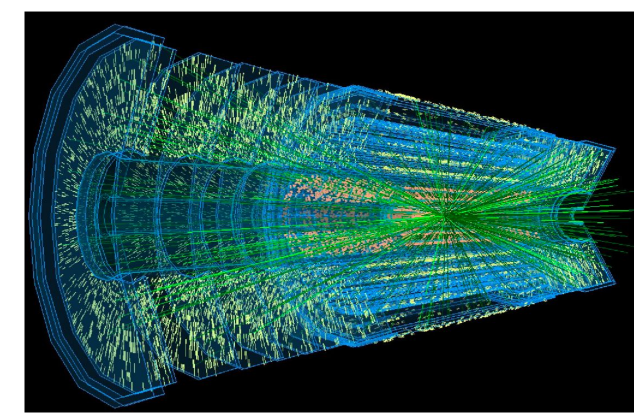
Speaker: Arghya Ranjan Das

PQuantML Team: Roope Niemi, Chang Sun, Anastasiia Petrovych, Enrico Lupi, Dimitrios Danopoulos, Arghya Ranjan Das, Miaoyuan Liu, Sebastian Dittmeier, Michael Kagan, Maurizio Pierini, Vladimir Loncar

Real-Time ML in High-Energy Physics

- Next-generation detector systems heavily rely on real-time machine learning.
- On-detector FPGAs/ASICs have tight DSP, BRAM… resource budgets.
- Compression (pruning + quantization) is mandatory for latency, power, and bandwidth.
- Use-cases in many areas: SmartPixels, Triggers, HEPT, ParT,..., Self-Driving Car…

SmartPixels



LHC triggers

Self-Driving Car

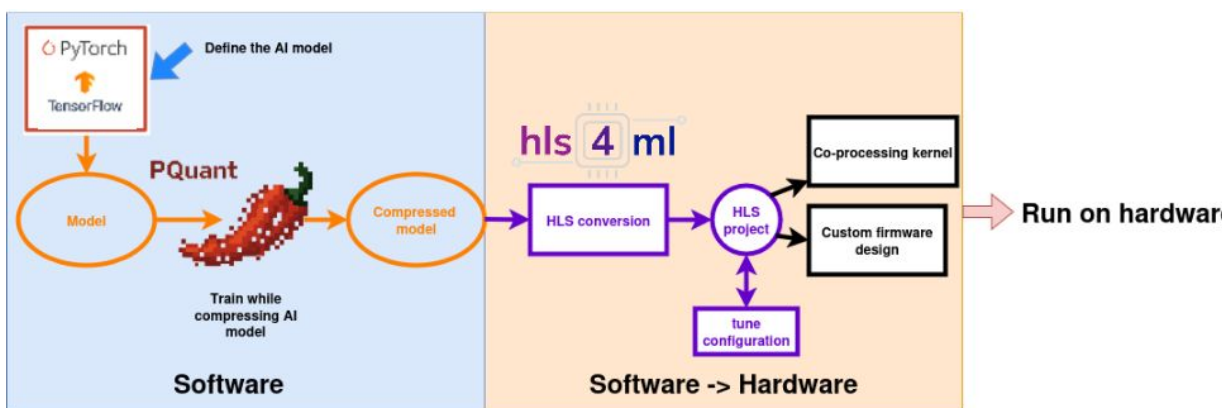
hls4ml

An open-source tool that bridges the gap between machine learning frameworks and hardware design.

It **translates trained ML models** into optimized High-Level Synthesis (HLS) code that can be synthesized into firmware for FPGAs and ASICs.

Provides a straightforward path from a trained model to a hardware implementation.

hls4ml: https://fastmachinelearning.org/hls4ml/ [arXiv:1804.06913]



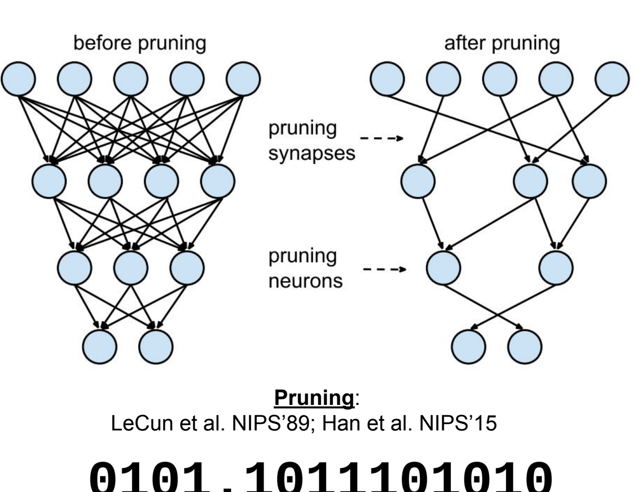
ML Model Compression

Refers to a suite of techniques designed to reduce the size, computational cost, and power consumption of ML models.

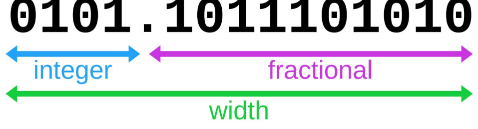
Primary Techniques:

- **Pruning**: Removal of model redundant weights/neurons⟶reduce number of params.
- **Quantization**: Reduction of numerical precision (number of bits) for weights and activations.
- *Other methods* includes Knowledge distillation, low-rank approximation, etc.

hls4ml handles deployment⟶Need dedicated tool for model compression



Pruning:
LeCun et al. NIPS'89; Han et al. NIPS'15



Quantization: hls4ml tutorial

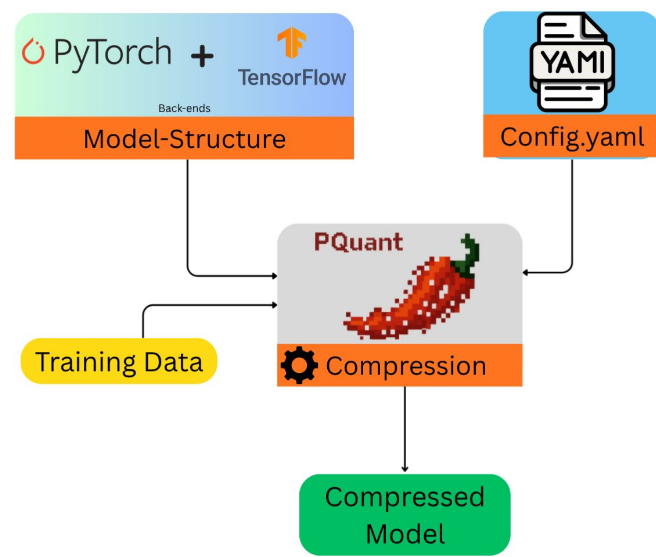
PQuantML

PQuantML

Unified-Toolchain: **Streamlines Pruning and Quantization**.

- **Backend-Agonistic** design: supports both **PyTorch** and **Tensorflow** out-of the box.
- No-detailed knowledge of the ML compression method is required.
- Pruning methods:
    - Unstructured Pruning
    - Structured Pruning (PDP and ActivationPruning)
    - N:M pruning (Wanda)
    - Hardware-aware resource optimization (MDMM framework)
- **Quantization Options:**
    - Fixed-point quantization
    - High-granularity quantization (per-weight bit control)
- Auto layer-wrapping, training, and cleanup.

Link to repository: https://github.com/nroope/PQuant



🌶️ PQuantML: Workflow

PQuantML has a very simple WorkFlow.

High-Level overview can be summarised as:

1. Entire PQuantML workflow is controlled by the **config.yaml**: (method of pruning, quantization, tuning epochs etc.)
2. **Extract the Layers** from model (PyTorch or TensorFlow)
3. Training is performed with a simple ***iterative_train*** function

```python
# pruning methods: "autosparse", "cl", "cs", "dst", "pdp", "wanda", "mdmm"
pruning_method = "mdmm"
config = get_default_config(pruning_method)
```

```python
# Replace layers with compressed layers
from pquant import add_compression_layers
input_shape = (256,3,32,32)
model = add_compression_layers(model, config, input_shape)
model
```

```python
from pquant import iterative_train

trained_model = iterative_train(model = model,
                                config = config,
                                train_func = train_func,
                                valid_func = valid_func,
                                trainloader = trainloader,
                                testloader = testloader,
                                device = device,
                                loss_func = loss_function,
                                optimizer = optimizer,
                                scheduler = scheduler
                                )
```

Compression is controlled by hyperparameters in the config file

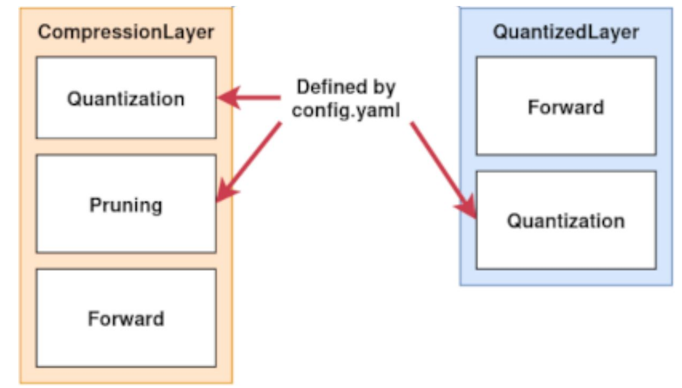
Compressed Layer: Replaces layers like: Linear, Conv1D, Conv2D etc. copying their weights and biases.

Quantized Layer: Replaces layers like: Relu, Tanh, AvgPool etc. with their quantized variants.

```yaml
pruning_parameters:
  disable_pruning_for_layers:
    -
  enable_pruning: true
  epsilon: 0.015
  pruning_method: pdp
  sparsity: 0.8
  temperature: 1.0e-05
  threshold_decay: 0.
  structured_pruning: false
```

```yaml
quantization_parameters:
  default_integer_bits: 0.
  default_fractional_bits: 7.
  enable_quantization: true
  hgq_gamma: 0.0003
  hgq_heterogeneous: True
  layer_specific: []
  use_high_granularity_quantization: false
  use_real_tanh: false
  use symmetric quantization: false
```

Example: pruning and quantization hyperparameters in config.yaml



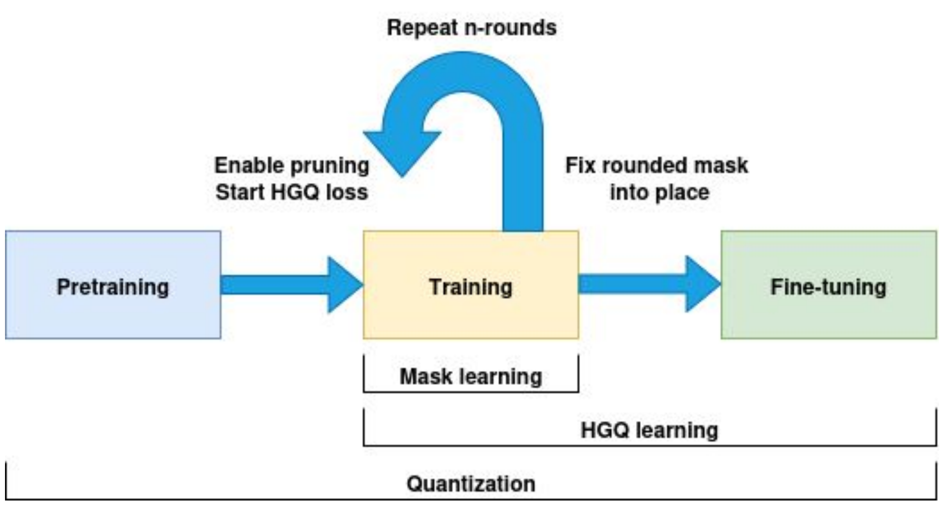
🌶️ PQuantML: Training

The training process in PQuantML is also controlled by the ***config.yaml*** file.

PQuantML provides a flexible three-stage training process to accommodate different compression methods

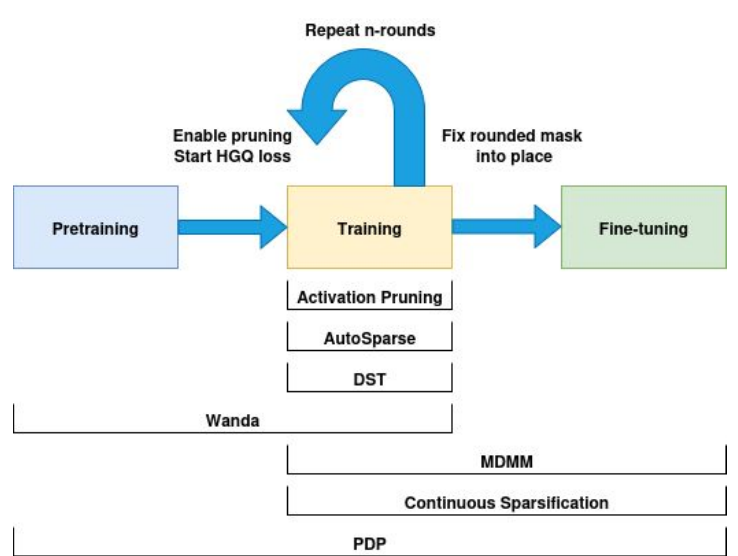
- **Pre-training**: Establishes a baseline model, often with initial quantization.
- **Training**: Main compression-aware phase to learn the pruning mask and apply quantization.
- **Fine-tuning**: Fixes the pruning mask and retrains remaining weights to recover accuracy.

```
iterative_train(
    model=model,
    config=config,
    train_func=training_function,
    valid_func=validation_function
    **kwargs
)
```



Default Pruning Methods

- PQuantML has implemented 7 different pruning mask-based methods, plus an optimizer-based method.
- These methods covers: unstructured, structured pruning and resource-aware compression methods.
- The available methods include:
  - Activation Pruning [arxiv:1903.04476]
  - AutoSparse [arxiv:2304.06941]
  - DST [arxiv:2005.06870]
  - Wanda [arxiv:2306.11695]
  - MDMM [NIPS 1987]
  - Continuous Sparsification [arxiv.org:1912.04427]
  - PDP [arxiv:2305.11203]



Pruning/Compression

Unstructured pruning:

- Removes individual weights, creating irregular sparse matrices.
- Can achieve very high sparsity.
- Results in a sparse, irregular matrix of weights→difficult to accelerate on hardware.

Structured pruning:

- Removes entire structural units (rows, columns, filters).
- Creates sparse matrices that are hardware-friendly.
- *PQuantML Support: PDP and Activation Pruning methods.*

Structured Pruning

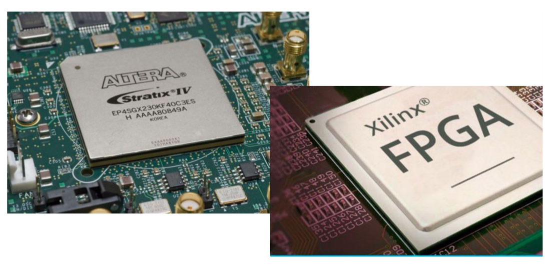
0.37	0.57	0.49	0.36	0.65
0.00	0.00	0.00	0.00	0.00
0.28	0.56	0.63	0.14	0.65
0.25	0.16	0.95	0.97	0.83
0.00	0.00	0.00	0.00	0.00

Resource-aware optimization:

Pruning is directly guided by hardware resource constraints (DSPs, BRAMs).

Using the **MDMM** framework, PQuantML optimizes for specific FPGA/ASIC resources like DSPs, BRAMs etc.

Also supported for convolution layers by compression of kernel pattern sets.



MDMM (Modified Differential method of Multiplier)

MDMM: Lagrange multiplier based **Multi-Objective Optimization** algorithm

Algorithm: Define the loss as:

$$\mathcal{L} = L_0(\boldsymbol{\theta}) + \sum_{\alpha=1}^{N}\left(\lambda_\alpha L_\alpha(\boldsymbol{\theta}) + \frac{c}{2}|L_\alpha(\boldsymbol{\theta})|^2\right)$$

Gradient descent on weights and gradient ascent on Lagrange multipliers.

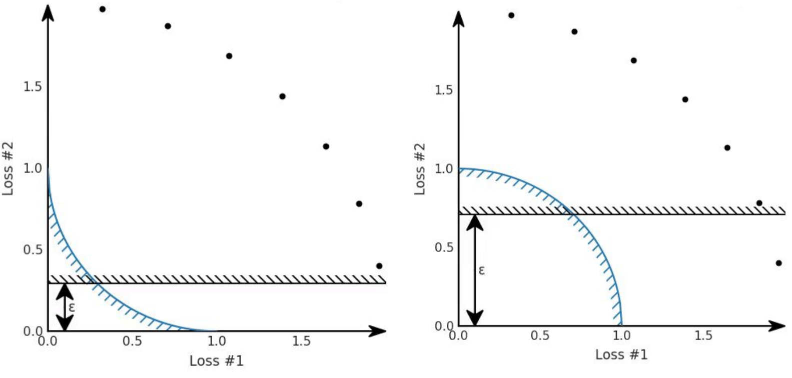
$$\dot{\theta_i} = -\frac{\partial}{\partial\theta_i}L_0(\boldsymbol{\theta}) - \sum_{\alpha=1}^{N}\lambda_\alpha\frac{\partial}{\partial\theta_i}L_\alpha(\boldsymbol{\theta})$$

$$\dot{\lambda}_i = L_i(\boldsymbol{\theta})$$

For Sparsification (unstructured): We define a loss-function that goes to zero whetarget sparsity s achieved

$$\mathcal{L} = (t^2 - s(\theta)^2) \cdot \sum |\theta_i| \quad \text{Where} \quad s(\boldsymbol{\theta}) = \sum_i \mathcal{I}(|\theta_i| < \epsilon)$$

MDMM first introduced in [Neural Information Processing Systems (NIPS 1987)]



J. Degrave & I. Korchunov

The **MDMM** is **very robust** in finding the **optimal balance** between competing losses/constraints.

FPGA Aware

Simply zeroing out weights doesn't free up hardware resources.

DSPs are still consumed: e.g. redundant multiplication by zero.

The PQuantML Solution: The `FPGAAwareSparsityMetric` in `mdmm.py` directly addresses this.

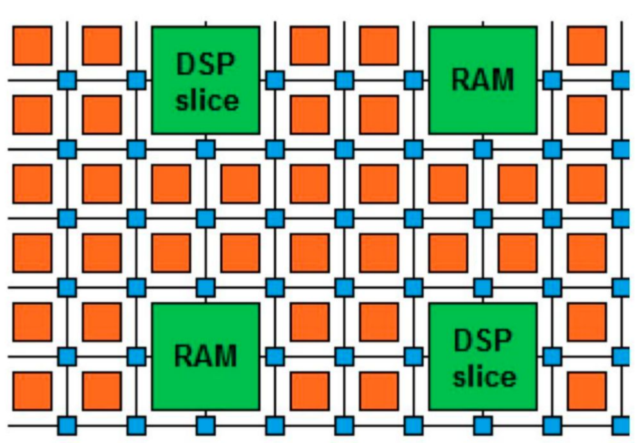
- Groups weights based on hardware Reuse Factor (RF).
- Calculates the L2 norm of the entire group.
- Prunes the whole group if its norm is below a threshold.

To save DSPs: Prune small, RF-sized groups of weights.

To save BRAMs: Prune larger chunks of weights that map to memory blocks.

For more details refer here [arxiv.org:2308.05170]

FPGA diagram



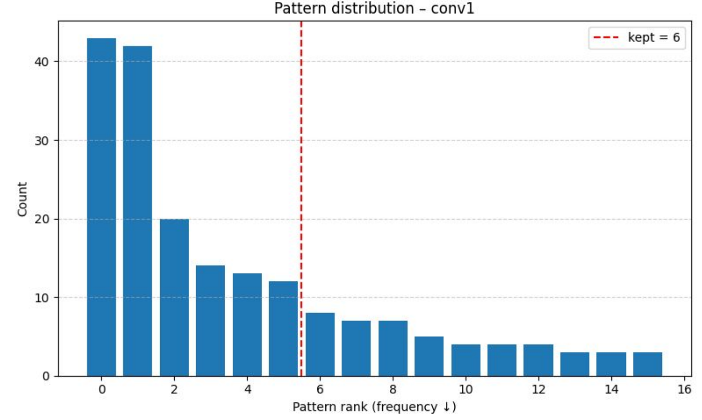
Pattern Based

The `PACAPatternMetric`: This metric is based on the concepts from the PACA paper.

1. **Identify Dominant Patterns:** Analyzes convolutional kernels and identifies a small, representative set of "***dominant patterns***" (e.g., *6 unique patterns*).
2. **Measure Penalty:** Define a "distance" to the nearest *dominant pattern* (e.g. cosine distance)
3. **Training:** MDMM encourages all kernels to morph into one of the few dominant patterns.

Reduced Memory Footprint: Instead of storing thousands of unique kernels, the hardware only needs to store the small set of dominant patterns.

Efficient Inference: During inference, the hardware performs a simple **lookup** to select the appropriate dominant pattern for



The number of unique patterns increases exponentially, with kernel sizes.

For a 3x3 kernel: There are 9 weight positions. The number of unique on/off patterns is 2^9=512.

For a 5x5 kernel: This number explodes to 2^25≈3.3×10^7 (over 33 million) unique patterns.

PQuantML in action: ResNet18 Example

ResNet18 model (Unstructured Sparcification)

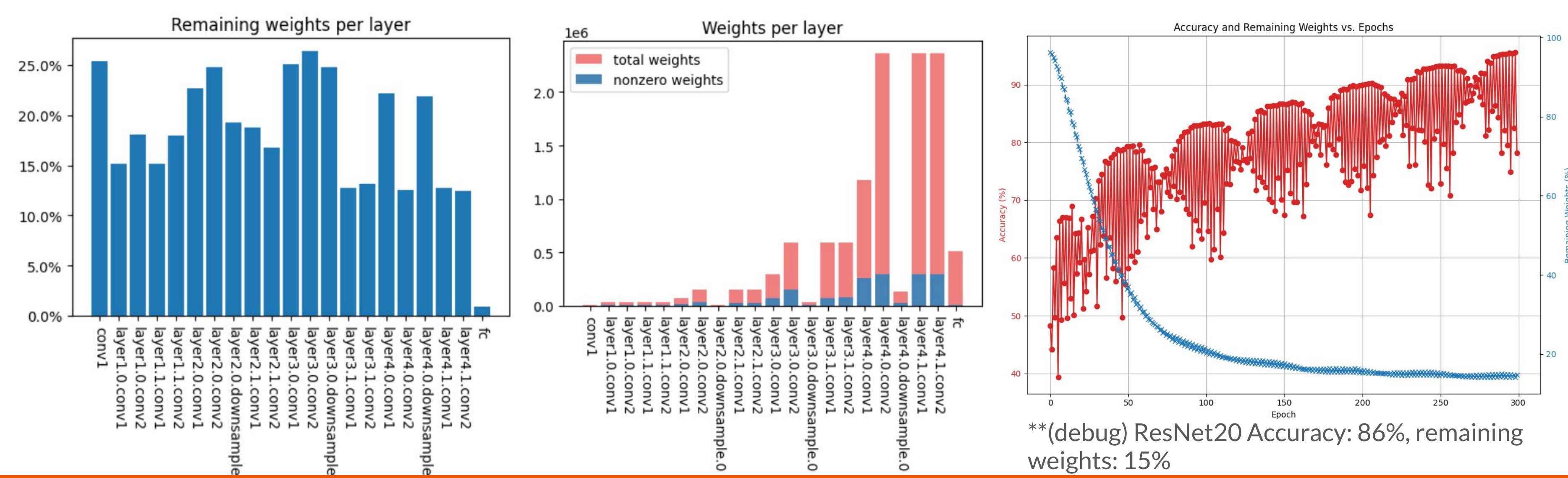
Currently at https://github.com/nroope/PQuant

At **PQuant/src/pquant/pruning_methods/mdmm.py**

```
class UnstructuredSparsityMetric:
```

- **Target sparsity**: 90%
- So: 10% remaining weights

Accuracy: 95.63%, remaining_weights: 14.15%**



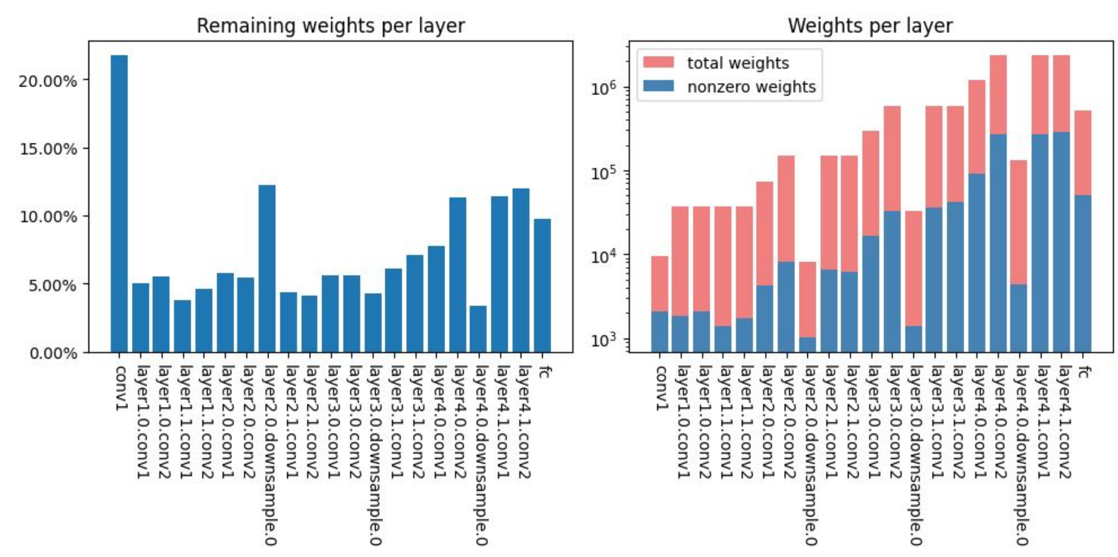
**(debug) ResNet20 Accuracy: 86%, remaining weights: 15%

Fine-Tuning Unstructured Sparcification

Unstructured pruning: Individual weights are zeroed out

During fine-tuning: The zero-ed weights are frozen. Only the non-zero weights are trained.

The fine-tuning step results in improved accuracy.

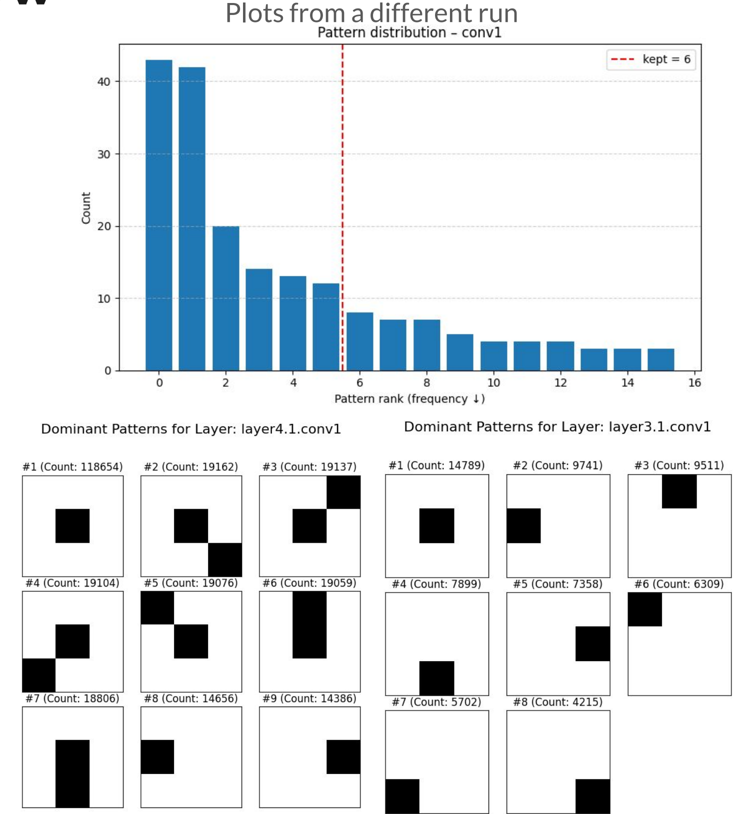


Pattern Compression Workflow

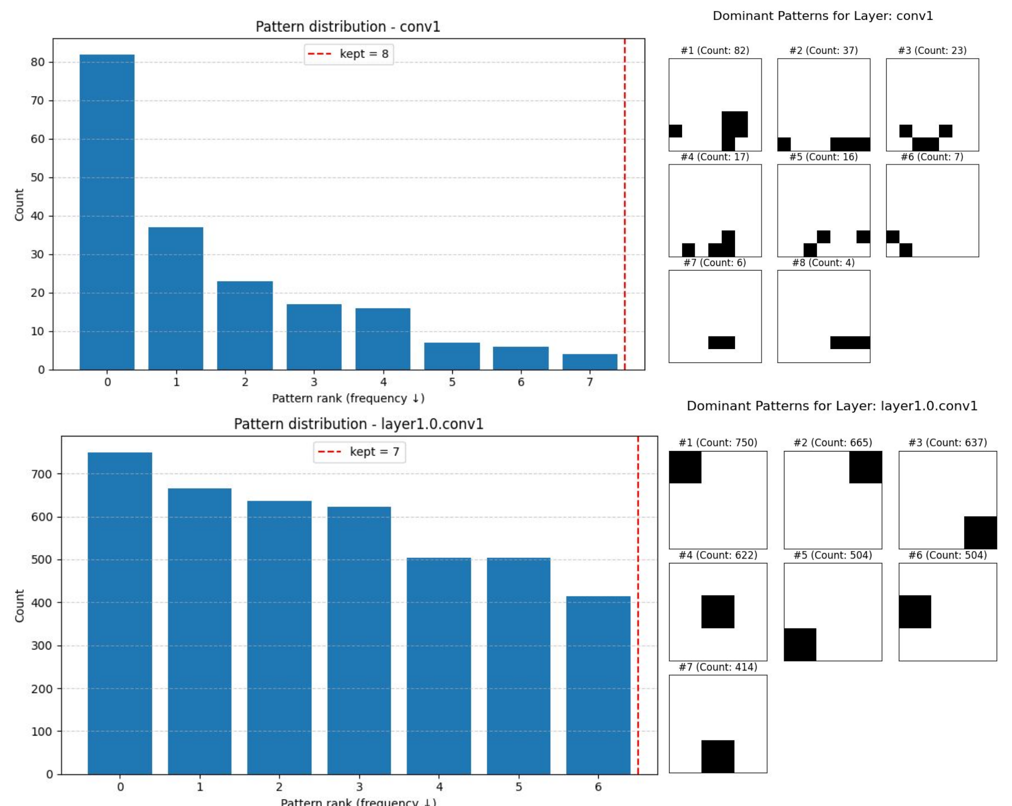
```
class PACAPatternMetric:
```

- Extract 2D patterns from convolution kernels
- Binary encoding:
  - 1 = non-zero weight
  - 0 = zeroed weight
- **Pattern analysis**: Check distribution of patterns and and select dominant set.
- **MDMM training**: minimize distance of each kernel's pattern to a pattern in dominant set.
- **Fine-tuning**: All patterns are projected to their closest pattern in dominant set and are frozen.

Plots from a different run



Fine-Tuning Pattern Compression



- After the training all the patterns are from the small dominant-set… what we wanted…

Conclusion

PQuantML provides a unified framework for model pruning and quantization:

- Backend-agnostic: works with both PyTorch and TensorFlow.
- Simple config-driven pipeline (yaml-controlled workflow).
- Supports unstructured, structured, and Resource-aware optimization (MDMM-PQuantML).
- Seamlessly integrates (in-dev) with **hls4ml** for FPGA/ASIC deployment.

Thank-you

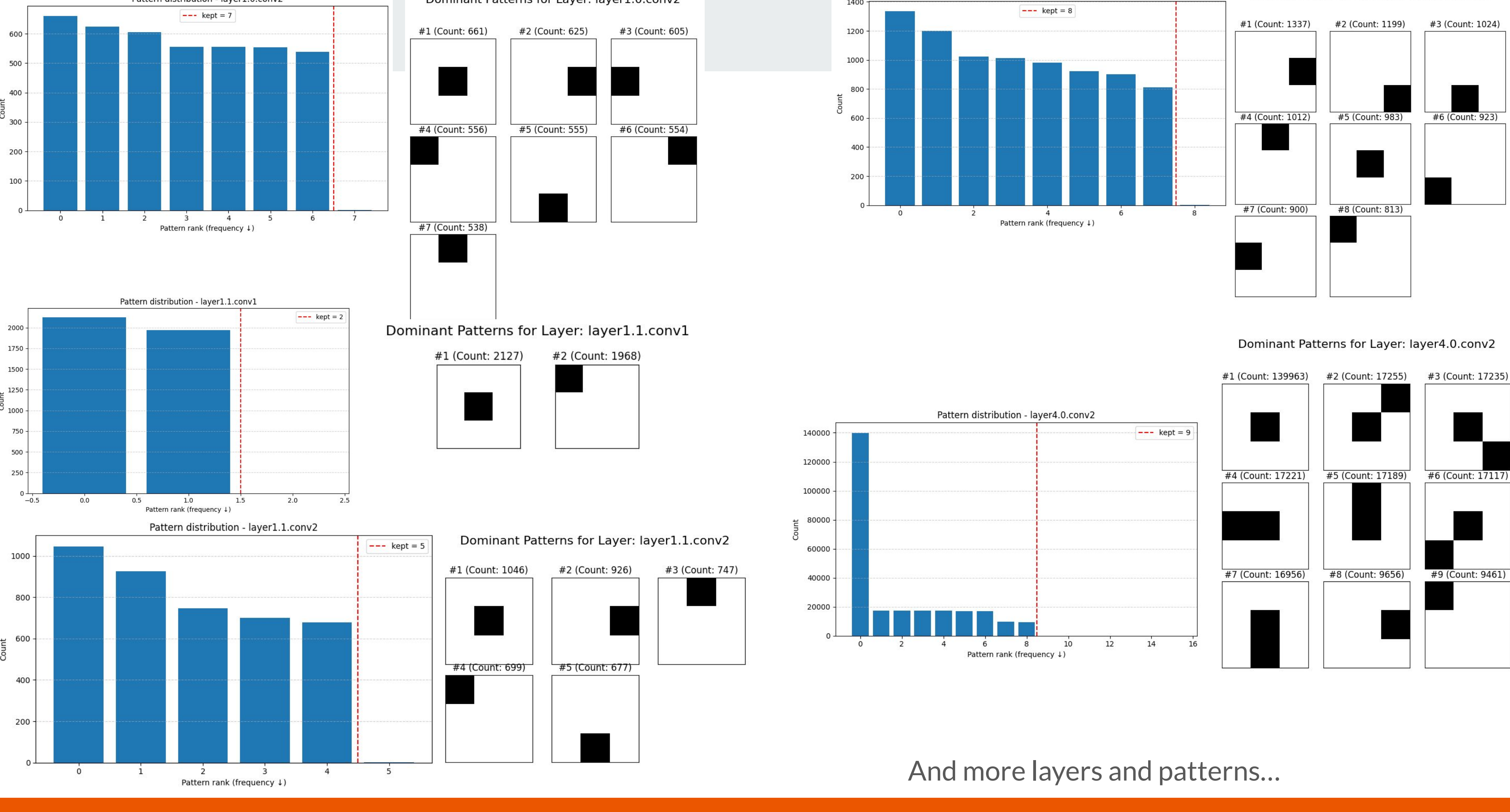
Questions?

PQuantML Repository: https://github.com/nroope/PQuant
MDMM Development Branch: https://github.com/ArghyaDas112358/PQuant/tree/mdmm/dev

Backup

Current and Near Teamwork

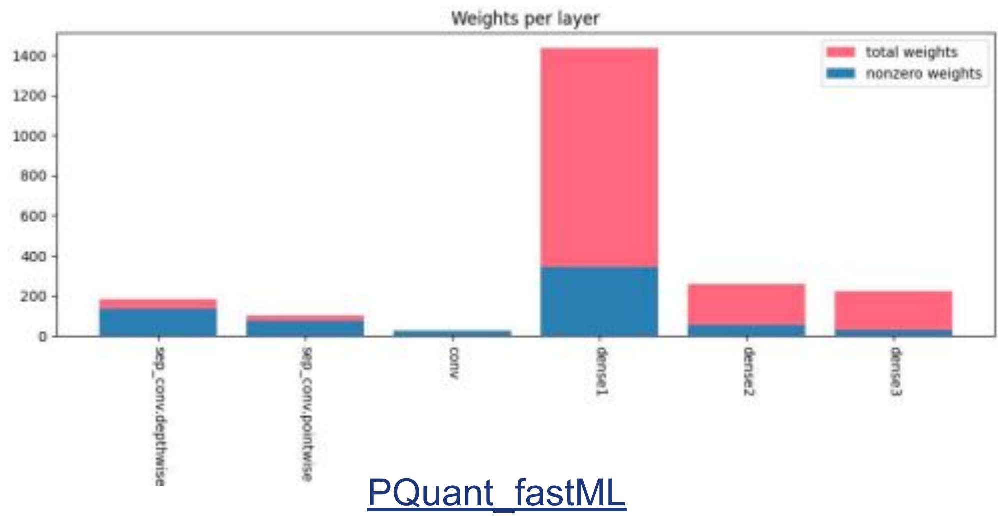
- Interfacing with **hls4ml** to enable a seamless workflow from compressed model to FPGA/ASIC deployment.
- Implementing **hyperparameter optimization** for automated tuning of compression settings.
- Integrating the **FitCompress algorithm** into the PyTorch backend to automatically find optimal configurations for quantization and sparsity.
- Preparing for the **first official release** of **PQuantML**.



And more layers and patterns...

Results: SmartPixels

- Pruned and quantized, ~2200 parameters
- 4 bit convolutions + their activations, 8 bit dense + their activations
- ~ 70% pruned



PQuant_fastML

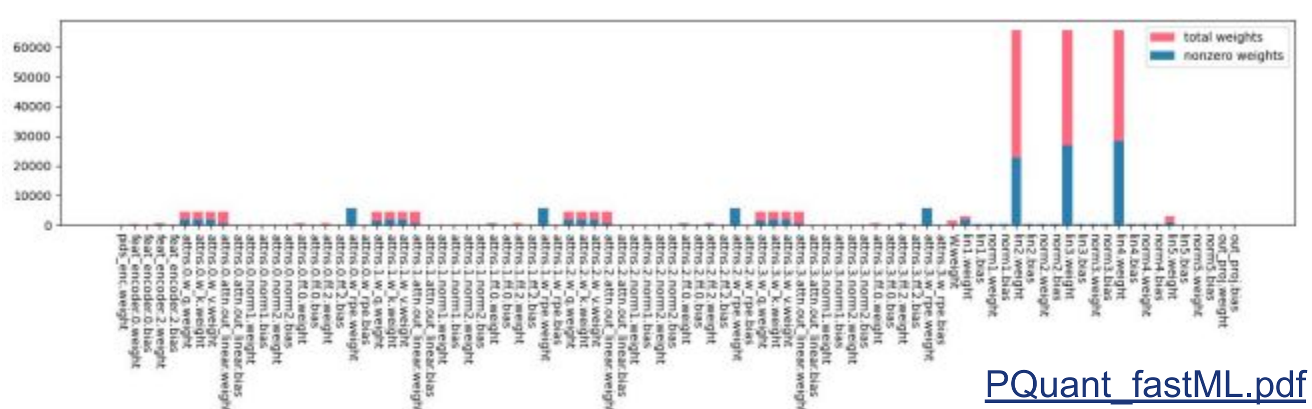
Results: LSH-Based Efficient Point Transformer (HEPT)

Pruned only
Pileup trainer:

- 43.90% remaining parameters (137,833 / 313,979)
- Test/auc: 38.71 (40.39)

Pruned only

- Tracking trainer 60k:
- 51.14% remaining parameters (168,461 / 329,400)
- Test accuracy@0.9: 86.44 (91.93)

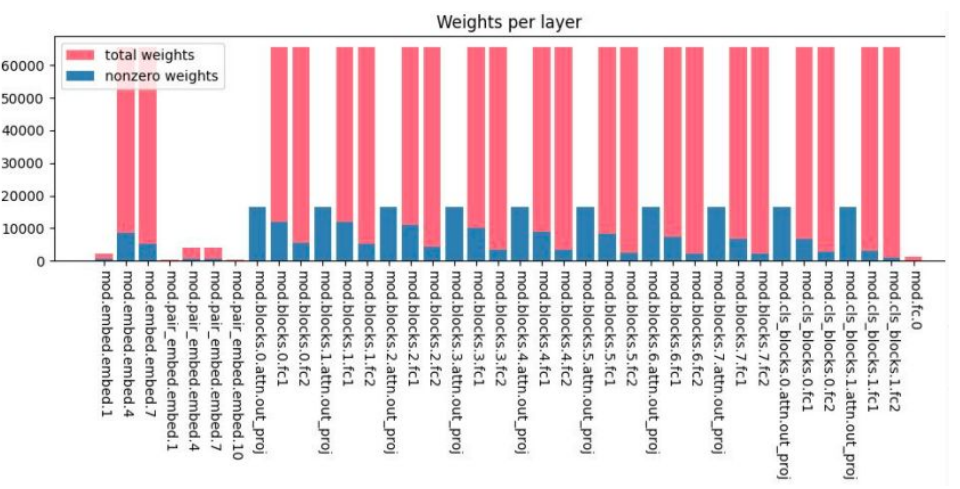


PQuant_fastML.pdf

Results: ParT

Only Pruning

- 38.51% remaining parameters
- Accuracy 84.71% (86.1%)



PQuant_fastML.pdf

Multi-Objective Optimization

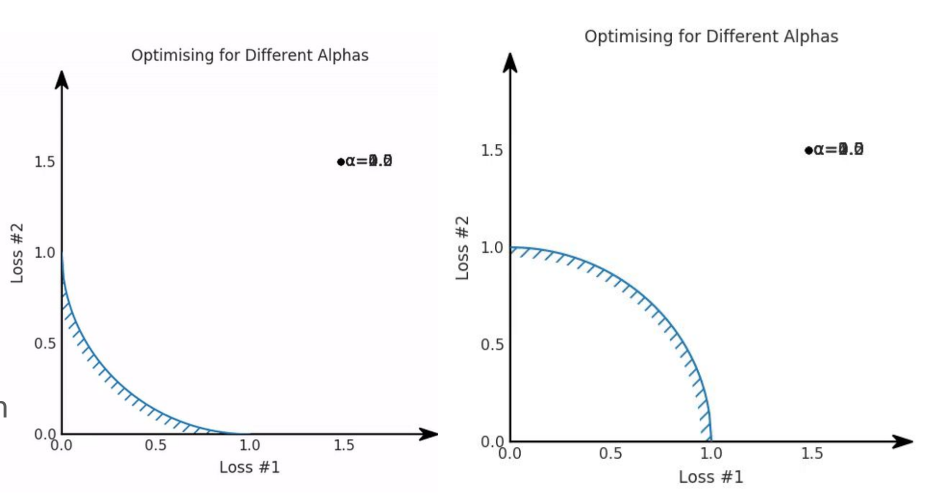
$$Loss = \mathcal{L} = L_0(\boldsymbol{\theta}) + \sum_{\alpha=1}^{N} \lambda_\alpha L_\alpha(\boldsymbol{\theta})$$

Penalty Method: (ex: l1/l2 regularizers)

$$\dot{\theta}_i = -\frac{\partial}{\partial \theta_i} L_0(\boldsymbol{\theta}) - \lambda_\alpha \sum_{\alpha=1}^{N} \frac{\partial}{\partial \theta_i} L_\alpha(\boldsymbol{\theta})$$

We want to optimize L0 subject to some constraint L1

- An optimal solution can be determined when the Pareto front is convex, but this does not hold for concave fronts.
- Generally, the Pareto front is unknown and is a mix of both the types.
- Thus there is no mathematical guarantee for optimizing multiple objectives simultaneously



J. Degrave & I. Korchunov

Thus we see move to another method of using lagrange multipliers

BDMM (Basic Differential method of Multiplier)

Lagrange Multipliers: Optimal solution when found would satisfy the condition

$$\frac{\partial \mathcal{L}}{\partial \theta_i} = 0 \quad \frac{\partial \mathcal{L}}{\partial \lambda_i} = 0$$

Pure gradient descent does not work with Lagrange multipliers due to saddle points. We perform gradient descent on weights and gradient ascent on Lagrange multipliers.

$$\dot{\theta}_i = -\frac{\partial}{\partial \theta_i} L_0(\boldsymbol{\theta}) - \sum_{\alpha=1}^{N} \lambda_\alpha \frac{\partial}{\partial \theta_i} L_\alpha(\boldsymbol{\theta})$$

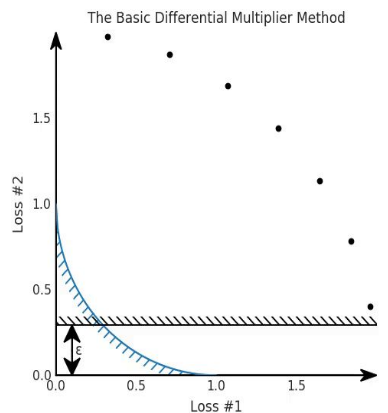
$$\dot{\lambda}_i = L_i(\boldsymbol{\theta})$$

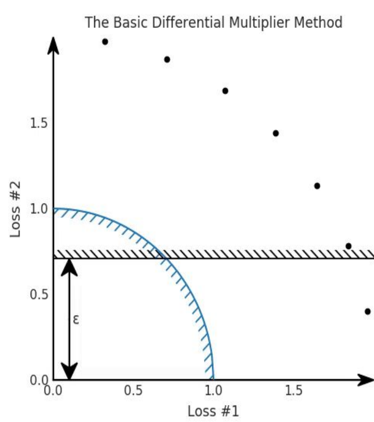
Equation of motion of the weights

$$\ddot{\theta}_i + \sum_j \left(\frac{\partial^2 L_0}{\partial \theta_i \partial \theta_j} + \sum_{\alpha=1}^{N} \lambda_\alpha \frac{\partial^2 L_\alpha}{\partial \theta_i \partial \theta_j} \right) \dot{\theta}_j + \sum_{\alpha=1}^{N} L_\alpha \frac{\partial L_\alpha}{\partial \theta_i} = 0$$

$$\dot{E} = -\sum_{i,j} \dot{\theta}_i \left(\frac{\partial^2 L_0}{\partial \theta_i \partial \theta_j} + \sum_{\alpha=1}^{N} \lambda_\alpha \frac{\partial^2 L_\alpha}{\partial \theta_i \partial \theta_j} \right) \dot{\theta}_j = -\sum_{i,j} \theta_i A_{ij} \dot{\theta}_j$$

We need to add some damping





J. Degrave & I. Korchunov

Pruning class structure

Currently Implemented a General Constraint Class

```
@keras.utils.register_keras_serializable(name = "Constraint")
class Constraint(keras.layers.Layer):
```

Using **Constraint** inequality class to set limits on the resource consumption

```
@keras.utils.register_keras_serializable(name = "EqualityConstraint")
class EqualityConstraint(Constraint):…

@keras.utils.register_keras_serializable(name = "LessThanOrEqualConstraint")
class LessThanOrEqualConstraint(Constraint):…

@keras.utils.register_keras_serializable(name = "GreaterThanOrEqualConstraint")
class GreaterThanOrEqualConstraint(Constraint):…
```

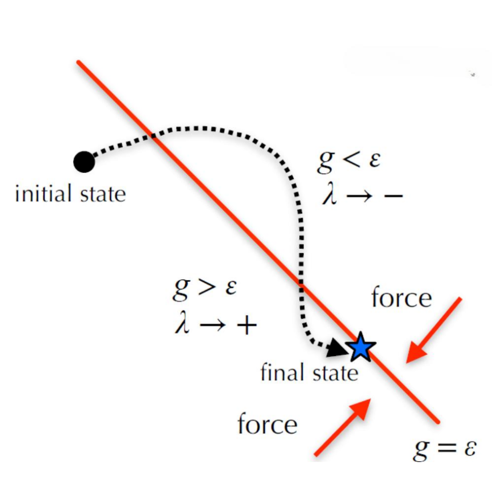
And the **Metric** function are at

```
class UnstructuredSparsityMetric:…

class FPGAAwareSparsityMetric:…

class PACAPatternMetric:…
```

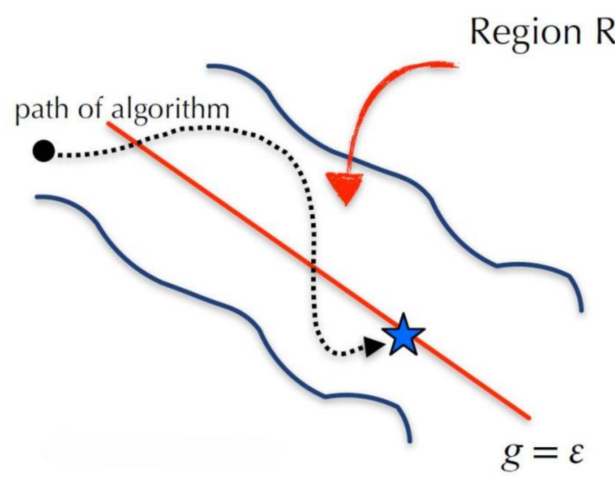
The general constraint class and the metric function are called in the **MDMM pruning method**

```
class MDMM(keras.layers.Layer):…
```

BDMM vs MDMM



BDMM



MDMM