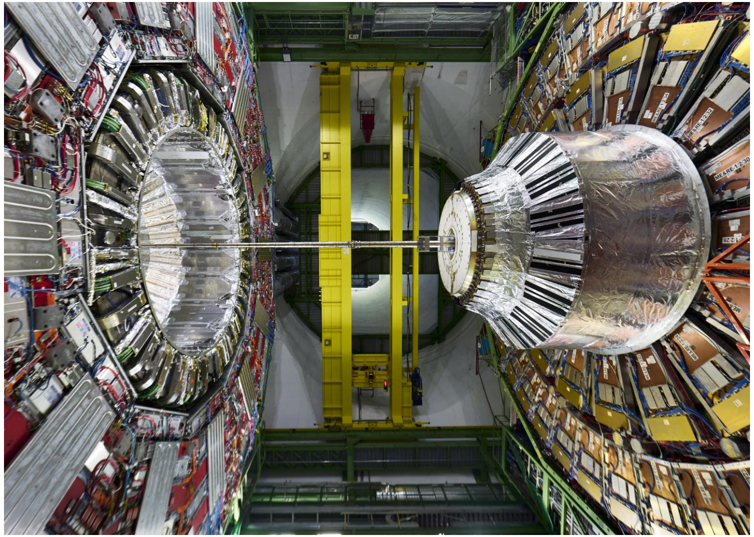
Autoencoder Optimization for Data Compression in front-end ASICs

Quinlan Bock

Advisors: Nhan Tran, Ben Hawks

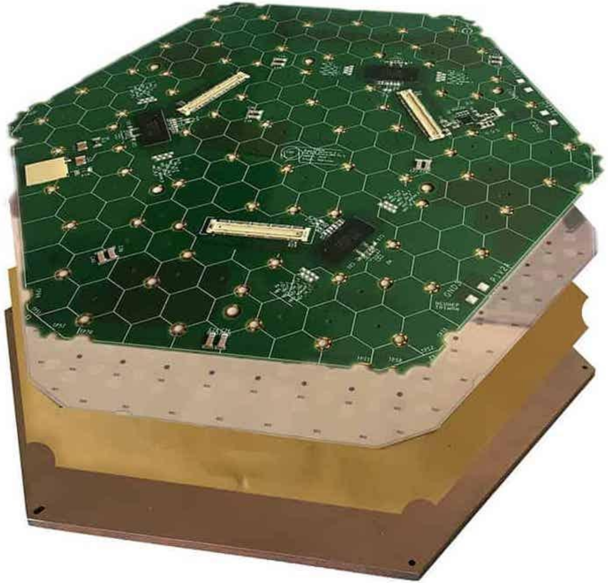
Background - CMS

- Compact Muon Solenoid Experiment (CMS) at CERN's LHC upgraded with high-granularity calorimeters (HGCAL)
- HGCAL includes ~6.5 million readout channels
- Collisions occur at a rate of 40Mhz
- High radiation environment creates undesirable faults
- Transport data from collisions to trigger systems
- Trigger systems decide data represents interesting particles and if to store for offline analysis
- Reduce latency and improve throughput by using ASICs and FPGAs for compression and triggering



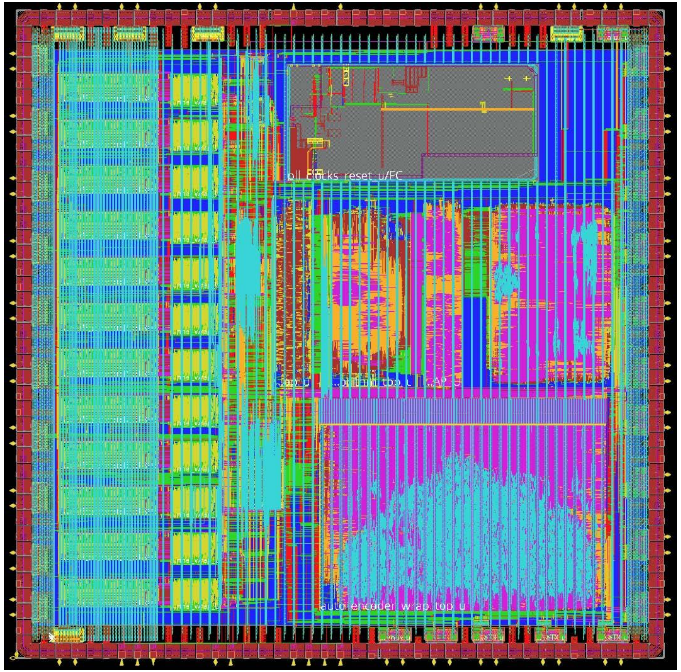
Background - Need for Data Compression

- Latency too high to process all the data from HGCAL uncompressed
- Compression algorithm that can handle the unstructured data
- Data compression allows more data transmit further down the data pipeline
- Latency requirements higher near triggering and storage where data can be uncompressed and processed



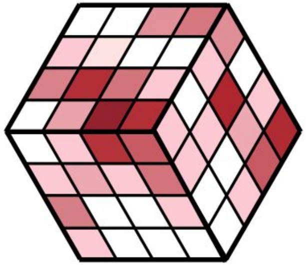
Background - Speed, Size, and Power Constraints

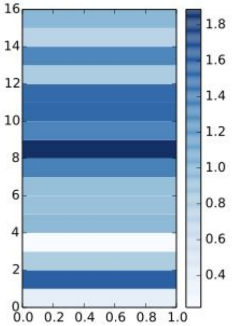
- Data compression must be quick to handle amount and frequency of the data being generated
- Compression algorithm must run on ASICs for latency
- ASICs limits on power consumption, and physical size limits size compression algorithm
- Places an upper bound on complexity of compression algorithm



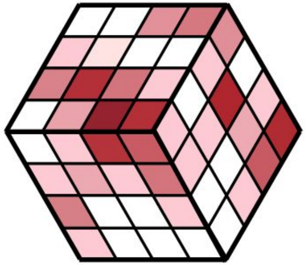
Background - Auto Encoders

- Autoencoders: specific neural network where output of the network is the same size as the input
- Minimize the loss between the input and the output
- Network consists of an encoder and a decoder
- Encoder feeds input to neural network layers and produces smaller encoded tensor
- Decoder takes encoded tensor and works to recreate the input to the encoder


Detector Input

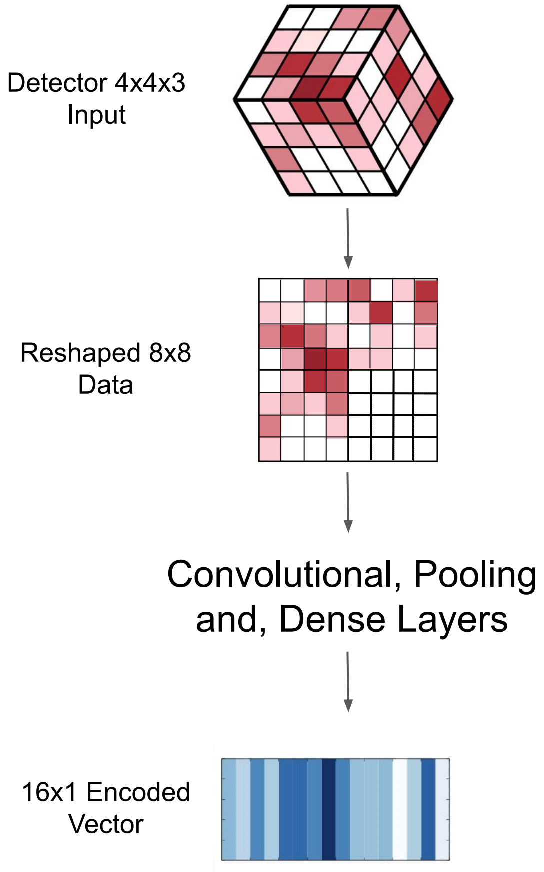


Encoded Vector


Decoded Output

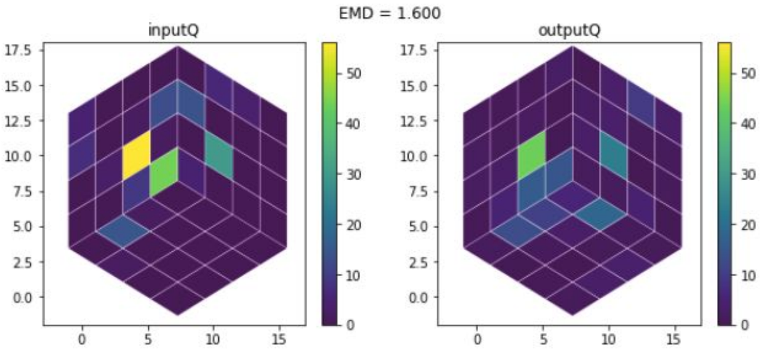
ECON-T

- The ECON-T contains an autoencoder model developed specifically for encoding data from HGCAL
- Written in python using Tensorflow and Keras
- Each model takes as its input the 4x4x3 shaped data and produces an encoded vector of size 16
- Model layers can include CNN layers, pooling layers, and dense layers
- Various functions used to compute loss however telescoping loss is used for original training
- Earth Mover's Distance (EMD) used as additional metric to quantify the distance between the input and reconstructed input



EMD

- Distance function between two probability functions
- Amount of shading needed to change for images to match

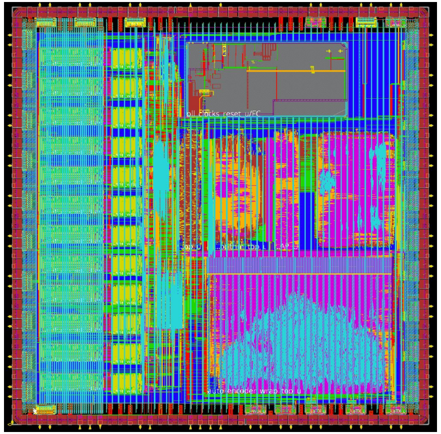


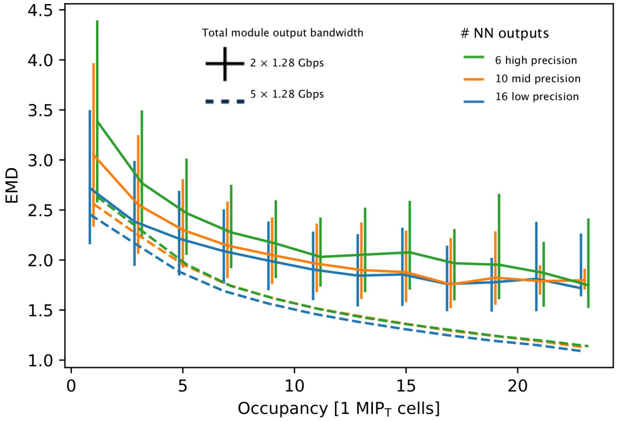
Telescoping Loss

- Associate shape information to each trigger cell (TC)
- Form 4x4, 2x2 super cell (SC) groupings
- Weight TCs less corresponding to how many SCs they appear in
- Mask SC groupings and add differences between input and reconstructed TCs, and SCs
- Associates the loss at multiple scales

Multi-objective optimization

- Hardware optimization for size, latency, OPs, and power
- Software optimization for reconstruction performance (EMD, telescoping loss)
- Optimizing both hardware and software can not be done separately since they both impact each other
- Nessicates co-design process where hardware and software optimization are integrated





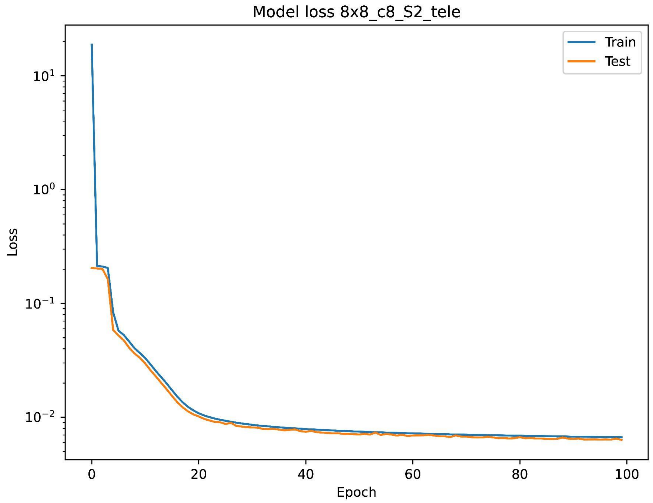
ECON-T - Baseline Model Architecture

- 4x4x3 data reshaped into 8x8x1 input
- Model architecture is very small amounting to just 4,977 trainable parameters
- Trained for 100 epochs with batches of size 800
- Achieves EMD of 1.067

```
Model: "encoder"
_________________________________________________________________
 Layer (type)                    Output Shape            Param #
=================================================================
 input_1 (InputLayer)            [(None, 8, 8, 1)]       0
 conv2d (Conv2D)                 (None, 4, 4, 8)         80
 flatten (Flatten)               (None, 128)             0
 encoded_vector (Dense)          (None, 16)              2064
=================================================================
Total params: 2,144
Trainable params: 2,144
Non-trainable params: 0
_________________________________________________________________
Model: "decoder"
_________________________________________________________________
 Layer (type)                    Output Shape            Param #
=================================================================
 decoder_input (InputLayer)      [(None, 16)]            0
 dense (Dense)                   (None, 128)             2176
 reshape (Reshape)               (None, 4, 4, 8)         0
 conv2d_transpose (Conv2DTranspose)  (None, 8, 8, 8)     584
 conv2d_transpose_1 (Conv2DTranspose)  (None, 8, 8, 1)   73
 decoder_output (Activation)  (None, 8, 8, 1)   0
=================================================================
Total params: 2,833
Trainable params: 2,833
Non-trainable params: 0
_________________________________________________________________
```

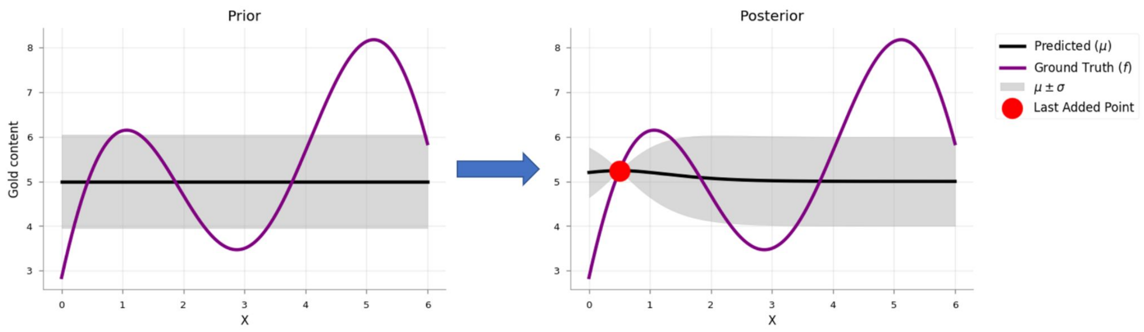
Hyperparameter Optimization and Model Architecture Exploration

- The baseline model architecture created through human trial and error, and heuristics behind autoencoders
- Optimization explores architectures and hyperparameters that provide better performance
- Search is automated and iterative,
- On each iteration a set of parameters is chosen based off of previous trials to:
    - Exploit space of parameters of which it knows produces good performance and optimize that performance further
    - Explore the spaces where there is uncertainty in whether that parameterization will produce a good model



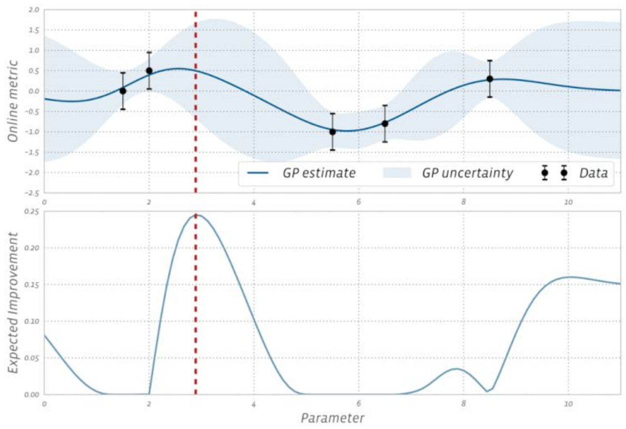
Bayesian Optimization

- Works upon Bayes Rule
- Start with surrogate model representing prior belief about model parameters and performance
- This model is updated iteratively after the new parameterizations are sampled
- Each new parameterization is used to create a network and train that network for a certain amount of epochs to produce a EMD metric
- The surrogate model is then updated based upon the EMD



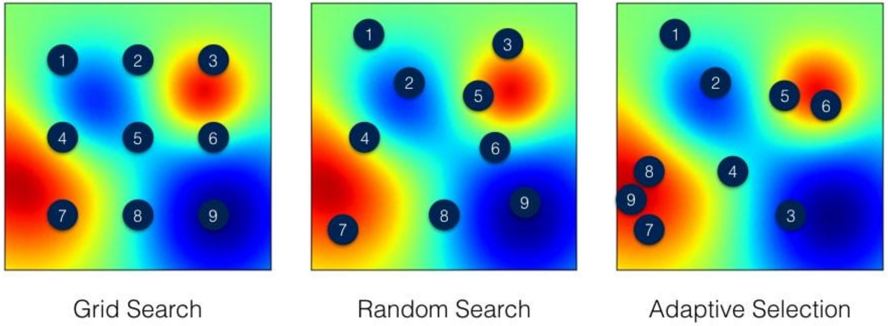
Ax

- Framework for hyperparameter search using Bayesian Optimization
- Need to provide:
    - Definition of each hyperparameters to be optimized and domain parameters
    - Function with parameterization as input that constructs the network, does training, and returns a metric optimize (EMD)
- No support for conditional constraints on hyperparameters
- Ax can’t choose number of each type of layer
- Hybrid approach: do grid search for each combination of CNN and dense layers then use Ax for hyperparameter optimization within each grid cells



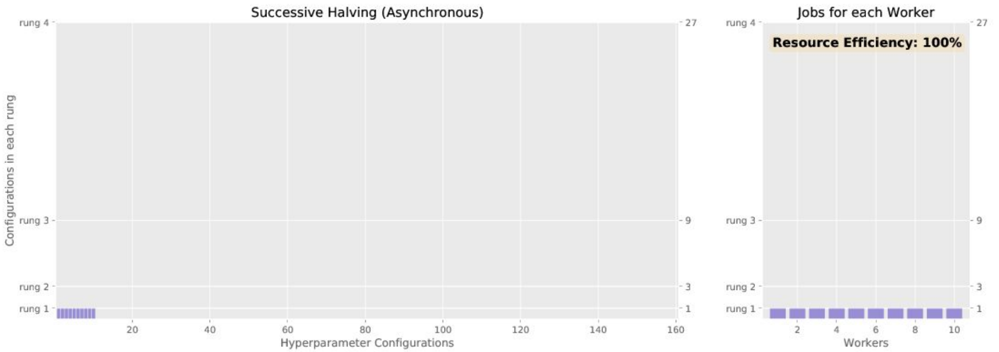
Adaptive ASHA

- Adaptive Asynchronous Successive Halving Algorithm
- SHA runs all model trials for a set duration in first iteration then discards lower performing half of trials
- Process repeated until models are narrowed down to threshold and stopping conditions are reached
- Instead of the algorithm waiting for full information to be obtained for all models at each iteration, only the a minimum amount of information is need to move to further testing
- Results in models that don’t get hung up on underutilized models that are still training



Determined.ai

- Uses Adaptive ASHA for hyperparameter optimization
- Combination of CLI and high level web interface tools allows visualization of training metrics and multiple experiments in real time
- Conditional hyperparameter constraint support means in addition to searching hyperparameters the algorithm can also search number of layers
- Need to provide:
    - Hyperparameters and their domains and constraints
    - Functions to load training/testing data and build optimizer, model, and callbacks



Compute for Model Training and Optimization

- Single ECON-T autoencoder model can be trained locally
- Local compute for hyperparameter optimization isn’t possible
- Both Ax, and Determined.ai optimizations are run on Fermilab’s Elastic Analysis Facility
- Allows training through JupyterHub and Determined’s own web interface
- Thanks to Burt Holzman and Ben Hawks for helping setting up user accounts and Determined instance

Hyperparameter Domains

- Up to 3 CNN layers and 3 Dense layers
- For each CNN layer
  - Number of filters in the layer
  - Kernel size
  - Stride
  - Whether to include pooling layer after
- For each Dense layer
  - Number of units in the layer
- Future:
  - Batch size
  - Learning rate

```
for i in range(0, cnn_layers+1):
        ax_parameters.append({"name": f"filters_{i}",
                          "type": "choice",
                              "is_ordered": True,
                              "value_type": "int",
                              "values": [0,2,4,8,16,32,64]})
        ax_parameters.append({"name": f"kernel_{i}",
                                        "type": "choice",
                                        "is_ordered": True,
                                        "value_type": "int",
                                        "values": [1,3,5]})
        ax_parameters.append({"name": f"pooling_{i}",
                                        "type": "choice",
                                        "is_ordered": True,
                                        "value_type": "bool",
                                        "values": [True,False]})
        ax_parameters.append({"name": f"stride_{i}",
                                        "type": "choice",
                                        "is_ordered": True,
                                        "value_type": "int",
                                        "values": [1,2,4]})
for i in range(0, dense_layers+1):
        ax_parameters.append({"name": f"units_{i}",
                              "type": "choice",
                              "is_ordered": True,
                              "value_type": "int",
                              "values": [16,32,64]})
```

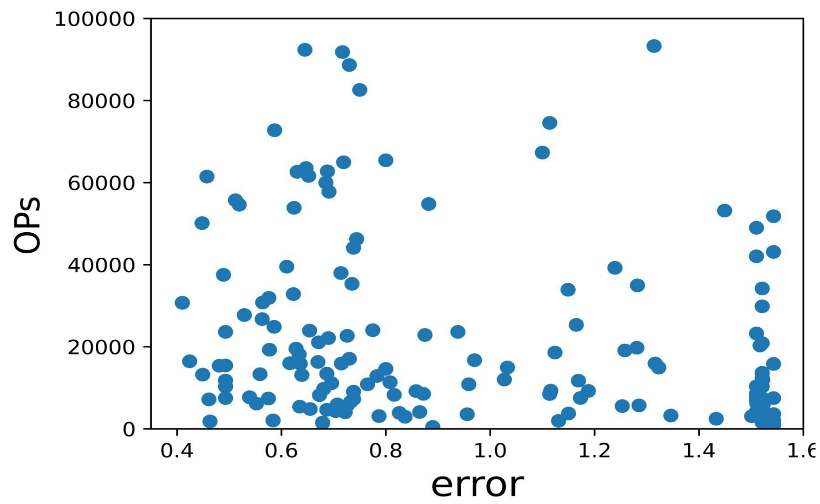
Ax - Results

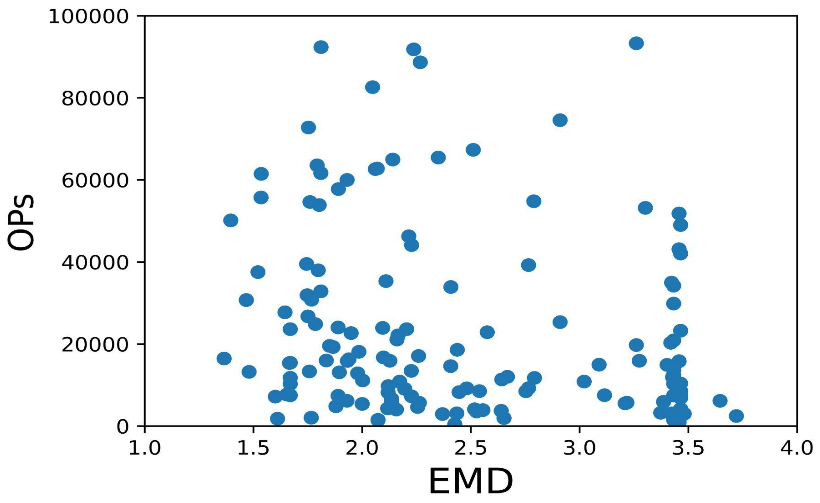
EMD	EMD_error	filters_1	kernel_1	pooling_1	stride_1	filters_2	kernel_2	pooling_2	stride_2	units_1	units_2	units_3	filters_3	kernel_3	pooling_3	stride_3
1.365	0.424	32	5	FALSE	4	16	3	FALSE	4							
1.467	0.41	8	5	TRUE	1											
1.467	0.41	8	5	TRUE	1											
1.467	0.41	8	5	TRUE	1											
1.467	0.41	8	5	TRUE	1											
1.467	0.41	8	5	TRUE	1											
1.467	0.41	8	5	TRUE	1											
1.48	0.449	8	5	FALSE	2	4	5	TRUE	1				0	5	TRUE	1
1.521	0.489	32	5	FALSE	2	32	1	TRUE	1				4	3	TRUE	2

Results from Ax using Bayesian optimization/Grid search hybrid approach. Each trial was trained for 20 epochs. There were 15 trials per grid tile.

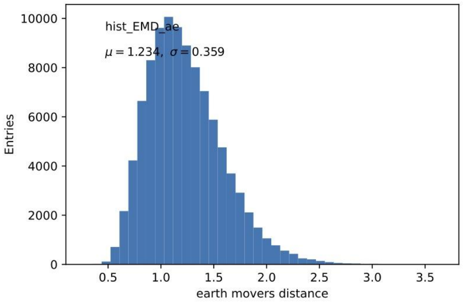
Pareto Fronts

- Pareto front is line on which optimal parameterizations exist
- Parameterizations on Pareto front trade off between EMD and OPs
- An increase in OPs corresponds to increase in size of model on chip

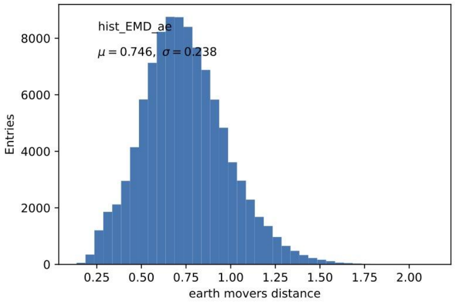




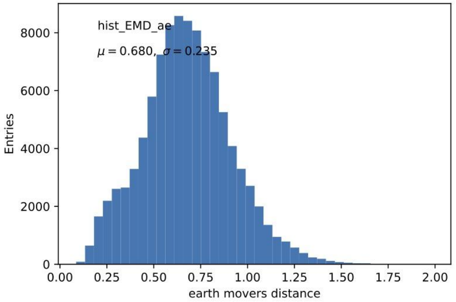
Ax - Results



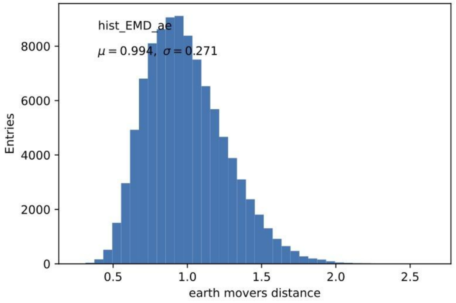
8x8_c8_k5_pTrue_S1_tele



8x8_c32_k3_pFalse_S1_tele



8x8_c64_k3_pFalse_S1_tele

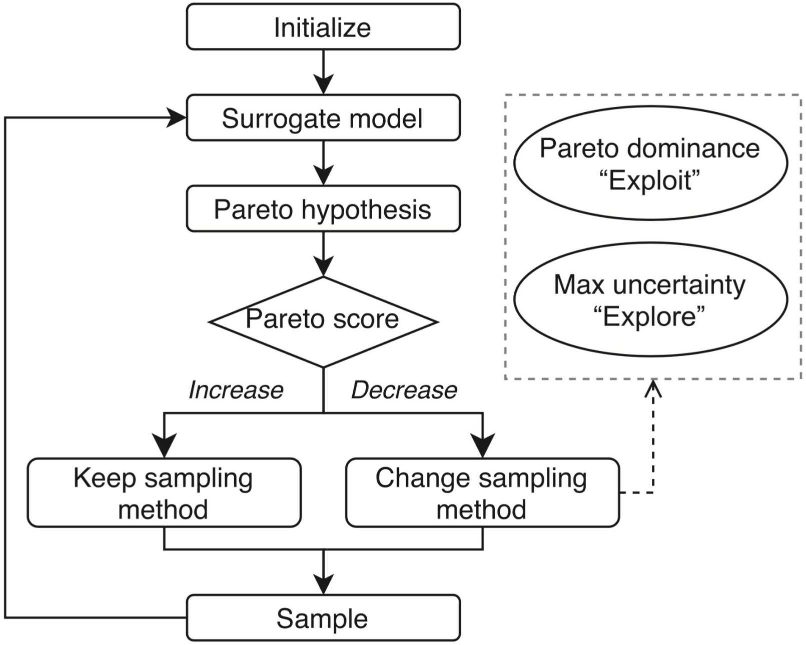


8x8_c16_k3_pTrue_S1_tele

- 8x8_c64_S1_tele EMD: 0.680
  - 1 CNN layer, filters 64, kernel size (3,3), stride 1
  - No Max pooling layer
- 8x8_c8_S2_tele EMD of 1.067
  - 1 CNN layer, filters 8, kernel size (3,3), stride 1
  - No Max pooling layer
- 8x8_c8_S2_tele FLOPS: 6544
- 8x8_c64_S1_tele FLOPS: 208912
- 36% reduction in EMD
- 309% increase in FLOPS

Sherlock

- Designed specifically for parameter optimization in FPGA synthesis
- Uses active learning to sample parameterization and model pareto front
- Samples from gaussian process, random forest, and radial bias function as the surrogate model



Future Work

- Hyperparameter optimization via Determined.ai
- Model and hardware parameter optimization via Sherlock
- Use hls4ml to generate code to transpile to FPGA firmware via High-Level Synthesis libraries
- Test performance on FPGA