IOP Publishing Mach. Learn.: Sci. Technol. 6 (2025) 035009 https://doi.org/10.1088/2632-2153/adede1

MACHINE LEARNING
Science and Technology

OPEN ACCESS

RECEIVED
22 January 2025

REVISED
26 June 2025

ACCEPTED FOR PUBLICATION
9 July 2025

PUBLISHED
21 July 2025

PAPER

Neural architecture codesign for fast physics applications

Jason Weitz[1,4,*], **Dmitri Demler**[1,4], **Luke McDermott**[1], **Nhan Tran**[2,3] and **Javier Duarte**[1]

[1] Department of Physics, University of California San Diego, La Jolla, CA 92093, United States of America
[2] Fermi National Accelerator Laboratory, Batavia, IL 60510, United States of America
[3] McCormick School of Engineering, Northwestern University, Evanston, IL 60208, United States of America
[4] These authors contributed equally to this work.
[*] Author to whom any correspondence should be addressed.

E-mail: jdweitz@ucsd.edu

Keywords: neural architecture search, FPGA, pruning, quantization, high energy physics, materials science

Abstract

We develop a pipeline to streamline neural architecture codesign for physics applications to reduce the need for ML expertise when designing models for novel tasks. Our method employs neural architecture search and network compression in a two-stage approach to discover hardware efficient models. This approach consists of a global search stage that explores a wide range of architectures while considering hardware constraints, followed by a local search stage that fine-tunes and compresses the most promising candidates. We exceed performance on various tasks and show further speedup through model compression techniques such as quantization-aware-training and neural network pruning. We synthesize the optimal models to high level synthesis code for FPGA deployment with the `hls4ml` library. Additionally, our hierarchical search space provides greater flexibility in optimization, which can easily extend to other tasks and domains. We demonstrate this with two case studies: Bragg peak finding in materials science and jet classification in high energy physics, achieving models with improved accuracy, smaller latencies, or reduced resource utilization relative to the baseline models.

1. Introduction

Deep learning has proven to be a powerful tool for tackling complex problems across many scientific domains, from materials science to particle physics. However, designing an optimal neural network architecture given a given task, computational platform, and latency or resource constraints remains a significant challenge, often requiring extensive trial and error which is time-consuming, computationally expensive, and may not result in the best possible performance. Moreover, keeping pace with the rapid advancements in deep learning techniques can be daunting for researchers whose primary expertise lies outside the field of machine learning. To alleviate this problem, neural architecture search (NAS) [1] can greatly accelerate the design and deployment of deep learning models while also reducing the burden on domain experts to stay abreast of the latest machine learning developments.

NAS aims to automate the design of neural networks, enabling the discovery of high-performing architectures tailored to specific tasks and constraints. By systematically exploring a predefined search space of potential architectures, NAS algorithms can identify novel network designs that match or exceed the performance of hand-crafted models. In this paper, we introduce *neural architecture codesign (NAC)*, an extension of NAS that optimizes neural architectures for both task performance and hardware efficiency. We present a general-purpose NAC framework that can be adapted to a wide range of tasks and domains. Our approach consists of a two-stage optimization process based on the once-for-all methodology [2]—a global search stage that explores a diverse set of architectures while considering hardware constraints, followed by a local search stage that fine-tunes and compresses the most promising candidates. We employ a flexible and modular search space that allows for the incorporation of domain-specific knowledge and constraints. Additionally, we introduce techniques for efficient evaluation of candidate architectures during the search process, balancing the accuracy of performance estimates with computational cost. To make our NAC

pipeline accessible to researchers across scientific fields, we encapsulate state-of-the-art techniques within a user-friendly framework that integrates popular open-source packages. These include Brevitas [3] and QKeras [4] for quantization-aware training, Optuna [5] for hyperparameter optimization (HPO), and `hls4ml` [6] for automated deployment of optimized models on FPGAs.

We present the effectiveness of our NAC methodology in two distinct scientific tasks, showing its ability to discover models that outperform hand-crafted architectures and being more efficient when deployed on hardware.

The first study focuses on BraggNN [7], a deep learning model for fast x-ray Bragg peak analysis in high-energy diffraction microscopy (HEDM) experiments in materials physics. The second case study explores the use of the deep sets [8–10] neural architecture for the classification of high-energy particle jets in particle physics, known as *jet tagging* [11]. Our NAC framework software is publicly available at [12].

2. Related work

2.1. Neural architecture search

NAS aims to optimize the structure of neural networks for specific tasks and objectives. This is achieved through a combination of search strategies, such as evolutionary algorithms [13] and efficient architecture evaluation methods [14], like weight sharing or surrogate models. The key advantage of NAS is its ability to find optimal architectures for a given problem without relying on human expertise or trial-and-error, thus saving time and resources while potentially yielding better results. However, optimizing for multiple objectives, such as accuracy and computational complexity, to find the best models along a Pareto-optimal front is still an area of active research [15]. This includes searching over network sizes or even constructing completely different model classes.

There are three critical components: *search space*, *search strategy*, and *architecture evaluation*. The search space determines the potential architectures that can be sampled [16]. While a narrow search space can be heavily biased, a large one is extremely difficult to properly explore, necessitating a delicate balance. This space is explored by sampling architectures and evaluating them across our metrics. Instead of evaluating high-cost objectives, such as network performance, researchers often use a proxy for such objectives like partial training or even zero-cost methods [17]. After validating the candidate architecture, the search strategy will update its beliefs and sample again. Various strategies for this exist, such as Bayesian optimization (BO) [18], evolutionary algorithms [13], or reinforcement learning [19], with differing strengths and weaknesses. For example, BO methods struggle with many categorical hyperparameters due to the large number of combinatorial possibilities, prompting the use of genetic algorithms instead [2]. However, BO, specifically tree-structured Parzen estimators (TPEs) [20], performs exceptionally well on continuous hyperparameter optimization tasks where sample efficiency is important [5]. In this paper, we use the non-dominated sorting genetic algorithm (NSGA-II) [21]. NSGA-II maintains a population of candidate architectures and evolves them over multiple generations using genetic operations like mutation and crossover. The algorithm ranks the candidates based on their performance and diversity, promoting solutions that are both high-performing and diverse in their architectural choices.

2.2. Model compression

In addition to finding an efficient model configuration, models can be further optimized through neural network pruning and quantization [22]. Pruning aims to remove a model's superfluous parameters [23], while quantization reduces the number bits needed to represent them.

Structured pruning removes weights associated to larger structures in the network, such as neurons, channels, or attention heads. Removing large structures can be seen as reducing the dimensionality of the weight tensor, by decreasing the amount of rows or columns in the matrix. On the other hand, unstructured pruning removes individual weights with no specific structure requirements. This leads to sparse matrices of the original dimension, which can often limit gains in actual inference speed on GPUs despite the reduced number of parameters. However, on more versatile or flexible hardware like FPGAs and CPUs, unstructured pruning can provide significant speed up with a negligible drop in performance [24]. While structured pruning provides definite inference time improvements on general hardware, it can lead to a larger decline in performance. To get around this, newer hardware supports N:M or mixed sparsity [25], such as Nvidia's A100 that supports 2:4 sparsity, which alleviates the need to prune entire rows or columns as done with dropping neurons or filters. Therefore, the choice of pruning algorithm is closely tied to the target hardware for deployment.

Quantization reduces the number of bits needed to represent weights or activations. Like pruning, quantization can be done post-training (PTQ) or during, with quantization-aware training (QAT) [4]. With QAT, the weights are quantized on the forward pass, but use full-precision gradients on the backward pass,

allowing for further fine-tuning of the low-bit representations. The effectiveness of quantization is heavily dependent on hardware support of low-bit data types. For the current study, we focus on deployment on FPGAs, which support sparse operations and a wide range of reduced precision data types; thus we use unstructured pruning with QAT.

This work targets fast applications in physics where neural networks are composed of fewer than a million parameters, where iterative pruning and different quantization sweeps are inexpensive. In contrast, large language models (LLMs) comprise billions of parameters, so retraining costs restrict model compression to methods such as one-shot pruning [26], dense-sparse decomposition [27], or PTQ. These scale differences are the reason behind the unique compression strategy compared to an LLM pipeline.

2.3. Hardware implementation and synthesis

Many experiments require fast inference times, and FPGAs are often used to achieve this goal. FPGAs offer several advantages over traditional computing architectures. They provide faster and more efficient processing compared to CPUs, while also allowing for more flexibility and customization than ASICs. This makes FPGAs well-suited for implementing machine learning models in scientific applications that demand high throughput and low latency.

However, deploying machine learning models on FPGAs can be challenging due to their fixed architecture and limited resources compared to GPUs. To address this, tools like `hls4ml` [6] have been developed to streamline the process of synthesizing ML models into FPGA firmware. `hls4ml` is an open-source library that translates models from common open-source machine learning frameworks, like TensorFlow and PyTorch, into high-level synthesis (HLS) code in C++. This HLS code can then be synthesized into FPGA firmware using commercial tools like Xilinx Vivado. By automating much of the process, `hls4ml` significantly lowers the barrier to entry for deploying ML on FPGAs.

`hls4ml` supports a variety of layer types and network architectures, and provides configuration options to tune the implementation for the specific use case and target FPGA. It also supports techniques like quantization and pruning to reduce the resource usage and latency of the synthesized model. While `hls4ml` was initially developed for particle physics applications, it is broadly applicable to other scientific domains that can benefit from fast ML inference on FPGAs [28].

3. Method

Our proposed NAC framework, illustrated in figure 1, consists of a two-stage optimization process: a *global search* stage and a *local search* stage. The global search stage explores a wide range of architectures within a predefined search space to identify promising candidate models. The local search stage fine-tunes the hyperparameters and compresses these candidate models to further improve their performance and optimize them for the specific task at hand. These stages are followed by a hardware implementation step for FPGA use, in which the resulting model is synthesized for hardware estimation and deployment.

3.1. Global search

3.1.1. Search space

The search space defines the set of possible architectures that can be explored during the optimization process. We design a flexible and modular search space that can be easily adapted to various tasks and domains.

The search space consists of a series of block(s), each representing a specific type of neural network layer or operation, that can be stacked in any order and replaced. These blocks can include convolutional layers, fully-connected layers, attention mechanisms, pooling operations, normalization, and activation functions. Each block has associated hyperparameters that control its behavior, such as the number of filters in a convolutional layer, the number of neurons in a fully-connected layer, or the type of activation function. The search spaces and block structures are defined in the appendix tables C1 and C2. The block structure for one case study is depicted in figure 3. While the attention mechanism cannot currently be synthesized with `hls4ml`, we include it within the search space to explore the upper bounds of performance and the possibility for other hardware applications other than FPGA deployment.

3.1.2. Search and evaluation

The search strategy is responsible for efficiently exploring the search space and identifying promising candidate architectures. In our framework, we employ a multi-objective optimization algorithm that considers both the performance of the models on the target task and their computational efficiency. This approach ensures that the resulting architectures not only achieve high accuracy but also meet the resource

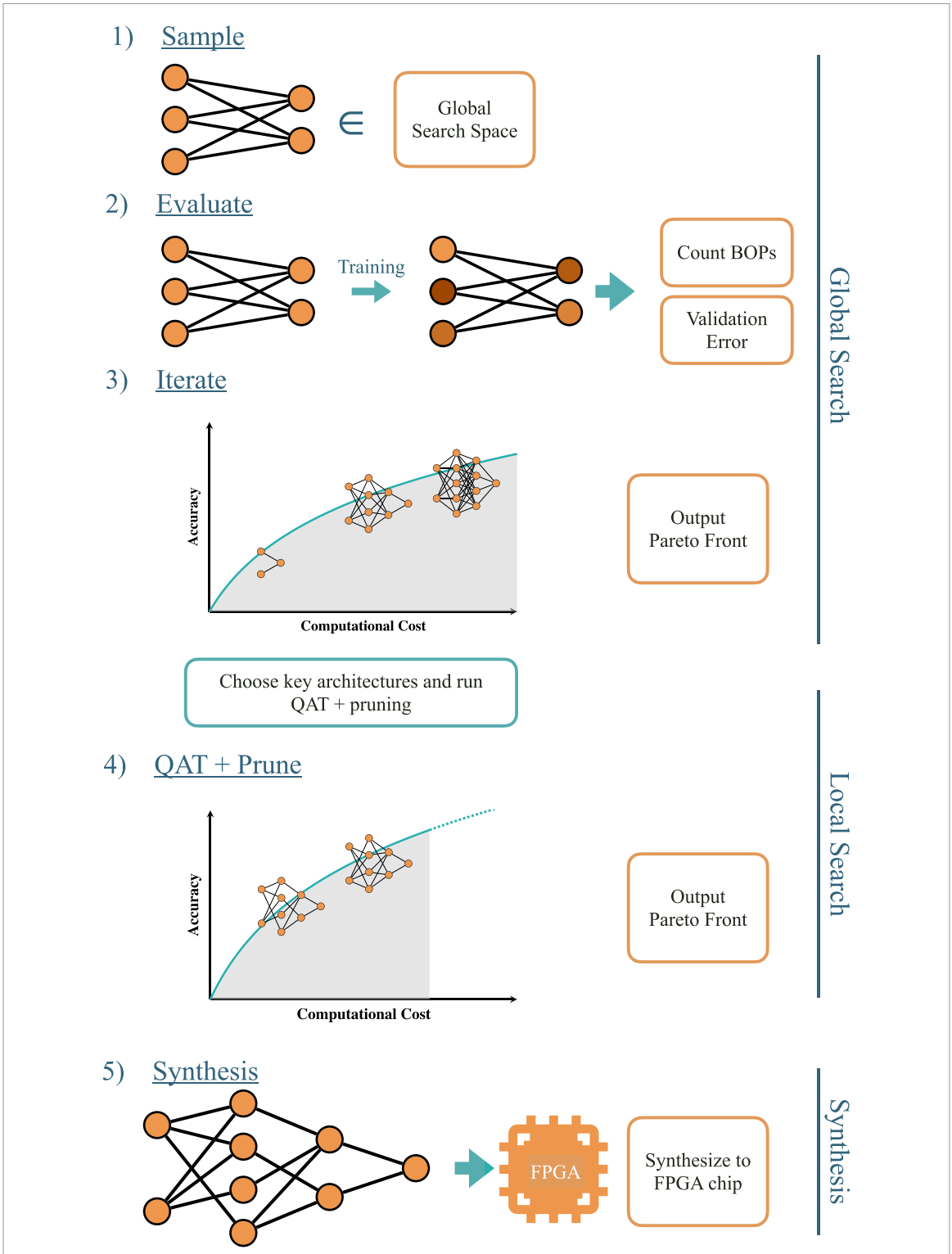

Figure 1. Full pipeline methodology of neural architecture codesign containing the two stages: global and local search. Global search stage explores a wide range of architectures and local search further fine-tunes hyperparameters and applies compression techniques. We then synthesize the optimal models to high level synthesis code for FPGA deployment.

constraints of the target platform, such as inference time or memory usage. We implement NSGA-II for this multi-objective optimization in the global search.

In addition to the performance metrics, we also evaluate the computational efficiency of the candidate architectures using bit operations (BOPs) or the actual inference time on the target hardware platform[5]. By

[5] More information on the BOPs calculation can be found in appendix A.

incorporating these metrics into the optimization process, we can guide the search towards architectures that strike a balance between performance and efficiency.

Throughout the search process, we maintain an archive of the most promising architectures discovered so far. This serves as a valuable resource for the subsequent local search stage, providing a diverse set of high-quality candidate solutions that can be further refined and optimized. Models are selected based on their respective performance and categorized according to their BOPs. This selection process is determined by a BOP ceiling and models under this threshold with the highest accuracy are selected.

3.2. Local search

To improve the efficiency even further, QAT can be paired with neural network pruning. The bit precisions used are 4, 8, 16, and 32. For each bit precision, iterative magnitude-based unstructured pruning is performed with QAT using Brevitas [3] inside the inner loop, removing 20% of the parameters each iteration, for a total of 20 iterations. This ultimately produces models ranging from 0% to 99% sparsity. The models are chosen with the goal of lowest bit precision, highest sparsity, and best performance. Generally, a more compressed model (sparse and quantized) comes at the cost of accuracy. While unstructured pruning does not decrease latency on the FPGAs we used in the case studies, it does show speedup on other hardware. Therefore, we include pruning steps to showcase the pipeline for other chips.

3.3. Model FPGA synthesis

With the optimal models found after the two-stage optimization process, the architectures are now synthesized for hardware deployment. In this process, key configurability options including precision, strategy, and reuse factor are chosen for optimization. The reuse factor indicates the number of multipliers used for multiplication operations within the network's computation of values. Increasing the reuse factor reduces resource utilization while increasing latency and decreasing throughput. The precision option configures the bit precision of the weights in the model, which can be altered at the layer level. One of two strategies, 'latency' or 'resource,' can be selected, which optimize for lower latency or lower resource utilization, respectively. Using `hls4ml` version 0.8.0 [29] and AMD Vivado 2020.1, the respective model is translated from a machine learning framework to HLS code for deployment.

4. Bragg peak case study

BraggNN is a materials science deep learning model developed for fast x-ray Bragg peak analysis in HEDM experiments [7]. HEDM is a technique used to characterize the microstructure and micromechanical state of polycrystalline materials [30]. A key step in the data analysis pipeline is fitting Bragg peaks to a pseudo-Voigt profile, which is computationally expensive and can be a bottleneck for real-time analysis. BraggNN aims to accelerate this process by using a convolutional neural network instead to directly predict the peak center positions.

The input data to BraggNN are small 11×11 pixel patches cropped around each Bragg peak, depicted in figure 2. The dataset contains approximately 70 000 such patches from a scan of a gold sample. The small size and sparsity of the input patches make the dataset well-suited for deployment on edge devices like FPGAs that have limited on-chip memory and benefit from low-precision computation. OpenHLS [31] further optimized BraggNN for deployment on FPGAs demonstrating significant speed-ups for real-time analysis.

4.1. Method adaptations

We used the same general two-stage neural architecture search strategy outlined in section 3. In the global search stage, we utilized a similar search space composed of convolutional, attention, and fully-connected blocks as seen in figure 3. However, we restricted the space to small architectures to ensure compatibility with the Xilinx Virtex UltraScale+ U250 (`xcu250-figd2104-2 L-e`) target FPGA. For the local search stage, we perform model compression to fine-tune the selected models.

4.2. Bragg peak results

We evaluated our optimized models on the gold diffraction dataset. Tables 1 and 2 provide a comprehensive comparison of our optimized models to the original BraggNN implementation in terms of performance, BOPs, parameters, FPGA resources, and latency, respectively. The resuse factor is a hyperparameter that we fix but can be varied in future studies. The size of the model is determined by its BOP count. The 'Tiny' model has the fewest BOPs and the 'Large' model has the most BOPs.

In the BraggNN case study, our large model slightly improved the mean distance while achieving a $5.9\times$ reduction in BOPs, albeit with a $2\times$ increase in parameters. This showcases the imperfection of efficiency metrics that aim to approximate the actual latency and resource utilization when synthesized to the FPGA.

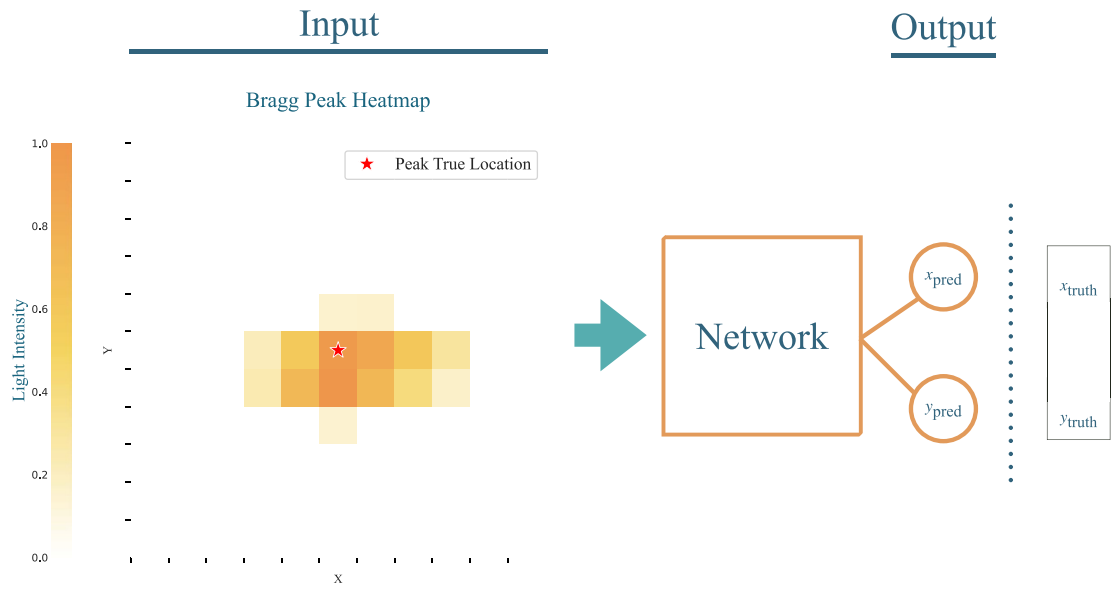

Figure 2. Bragg peak NAC visualization. The input consists of 11×11 pixel patches centered on Bragg peaks from x-ray diffraction patterns. The neural network predicts the x and y coordinates of the peak center within each patch.

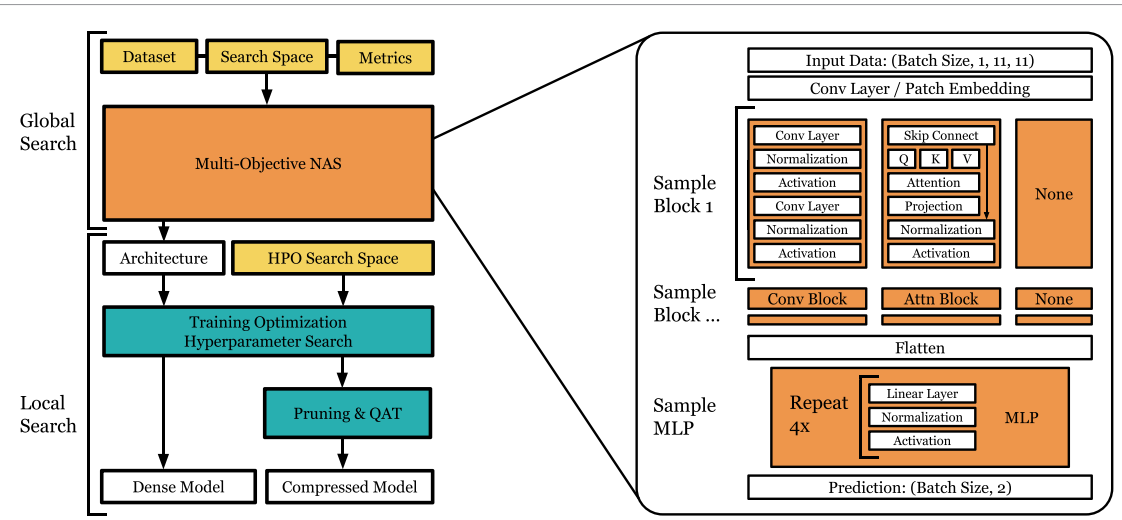

Figure 3. Our automated pipeline for neural architecture search for the Bragg peak dataset. Yellow components are human inputs, white are outputs, and orange are search processes. The right side demonstrates the template of each candidate architecture in our search space. Each sub-component of the blocks also contains the hyperparameters to optimize. 'Q,' 'K,' and 'V' correspond to 'Query,' 'Key,' and 'Value' respectively, in the attention mechanism.

Table 1. Bragg Peak Model comparison with respect to distance, MegaBOPs, and parameters. The best values are bolded.

Model	Mean Distance [pixels]	MegaBOPs	Parameters
BraggNN	0.202	34 540	45 274
Large	**0.201**	5861	92 438
Medium	0.208	1447	26 402
Small	0.211	881	23 788
Tiny	0.227	**800**	**23 094**

The small model has a better balance: with only a 3% larger mean distance but with a 39.2× decrease in BOPs and a 2.90× decrease in parameters. Full model architectures can be accessed in appendix table B1 and the configuration is detailed in appendix table B3. All models can be pruned to 80% sparsity (8 iterations) with less than a 10% drop in performance, depicted in figure 4. BraggNN uses a non-local attention block which we find is unnecessary for performance.

Table 2. Comparison of different Bragg peak models' latency, initiation interval (II) based on clock cycles (cc), and hardware resource utilization. DSP and BRAM are not listed as they are not utilized (0%). All models are quantized to 8 bits. A reuse factor of 4 is used. The baseline BraggNN model is not listed, as its attention block cannot be synthesized because `hls4ml` does not currently support it. The best values are bolded.

Model	Latency (μs) (cc)	II (μs) (cc)	DSP	LUT (%)	FF
Large	8.56 (1711)	1.82–8.48 (365–1696)	6433 (52.35%)	181 298 (10.49%)	124 887 (3.61%)
Medium	4.94 (987)	**1.82–4.85 (365–970)**	2697 (21.95%)	47 903 (2.77%)	54 937 (1.59%)
Small	4.92 (984)	**1.82–4.85 (365–970)**	**290 (2.36%)**	121 941 (7.06%)	**34 765 (0.70%)**
Tiny	**4.88 (975)**	**1.82–4.85 (365–970)**	3148 (25.62%)	**40 801 (2.36%)**	36 360 (1.05%)

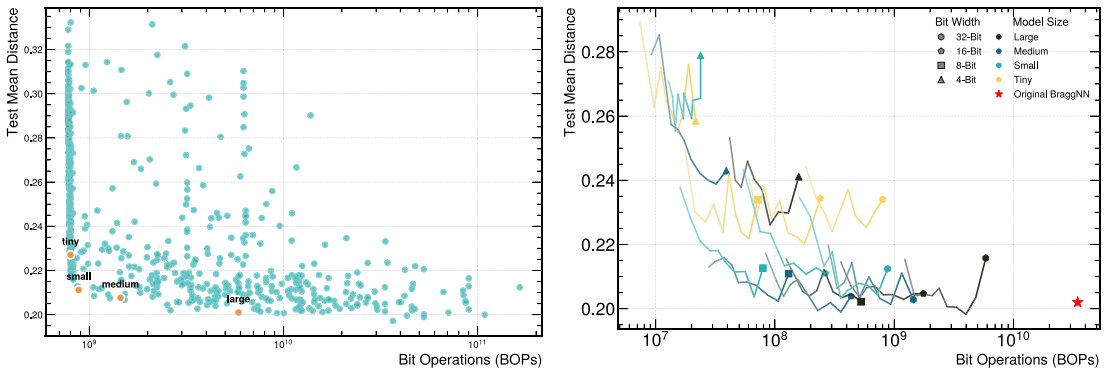

Figure 4. Bragg peak dataset Pareto-optimal front for global (left) and local search (right). The global search contains 1000 trials with the selected models in orange. For the local search, each chosen model is quantized to various bit precisions, and then pruned by 20% for 20 iterations indicated by increasing opacity. The first 10 iterations are displayed for visualization purposes.

When synthesized for FPGA deployment, our optimized models demonstrate significant improvements in latency and resource utilization. Our small model achieves a latency of 4.920 μs with a 1.825 μs initialization interval, using 2.36% of the available DSPs, 7.06% of the LUTs, and 1.05% of the flip-flops on the target FPGA (table 2).

The performance of the BraggNN models during the global and local searches can be seen in figure 4. Each point indicates a unique trained model architecture, demonstrating the exploration and refinement of the search process. The global search stage evaluated 1000 model architectures, utilizing 1 NVIDIA 4090 GPU for 96 hours. The local search stage trained 4 models at 4 precisions, with iterative magnitude pruning for 20 iterations, with 100 epochs per iteration. 4 NVIDIA 3090 GPUs were used in parallel for 29 hours for this stage.

5. Jet classification case study

In particle physics, jets are collimated sprays of particles that originate from the decay of heavy particles like top quarks, W bosons, or Z bosons. Identifying the type of particle that initiated a jet, a task known as jet tagging, is crucial for many particle physics analyses as it provides valuable information about the underlying physics processes. Jet tagging can be formulated as a set-based problem, where each jet is represented as a set of its constituent particles, and the goal is to classify the jet based on the properties of these particles, as depicted in figure 5.

Deep sets is a neural network architecture designed to operate on sets of objects, where the order of the elements in the set does not matter [8]. This permutation-invariant property makes deep sets particularly suitable for tasks involving unordered sets of variable size. The key idea behind deep sets is to apply a shared neural network, ϕ, to each element of the input set independently. ϕ maps each input element to a high-dimensional representation. These individual representations are then pooled using a permutation-invariant function such as sum, mean, or max operations, resulting in a fixed-length embedding that captures the relevant information from the entire set. Next, another neural network, ρ, is applied to the pooled embedding to produce the desired classification output.

Deep sets are well-suited for this task because they can handle variable-length sets of particles and are invariant to the order in which the particles are presented, which is a desirable property since the order of particles within a jet is not physically meaningful.

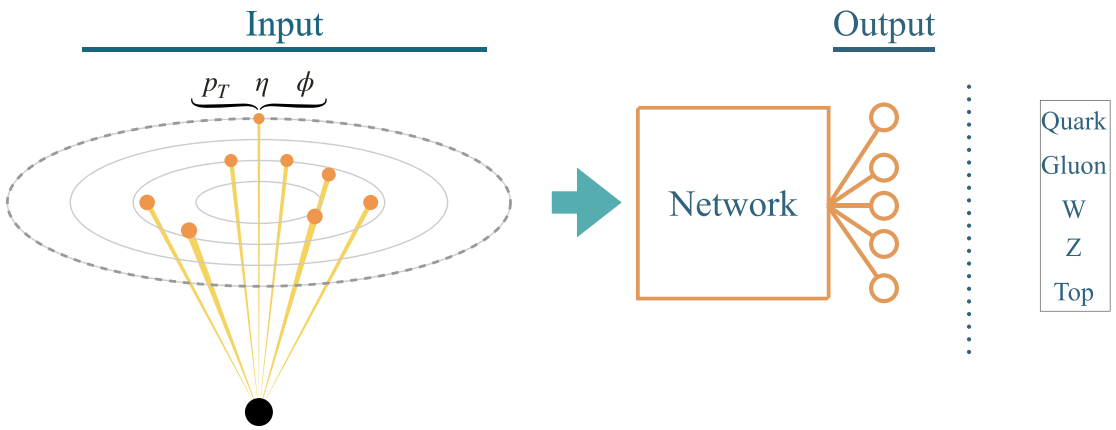

Figure 5. Particle jet NAC visualization. The input consists of sets of up to 8 constituent particles from a jet, with the transverse momentum (p_T), pseudorapidity (η), and azimuthal angle (ϕ) known for each particle. The neural network predicts the origin of the jet (light quark, gluon, W boson, Z boson, or top quark).

Table 3. Comparison of the particle jet models' accuracy, MegaBOPs, and parameters. The best values are bolded.

Model	Accuracy (%)	MegaBOPs	Parameters
Baseline	64.0	12.10	3461
Large	**66.55**	5.23	7535
Medium	65.06	1.68	2813
Small	63.06	0.75	815
Tiny	61.16	**0.40**	**573**

Table 4. Comparison of the particle jet models' latency, initiation interval (II), and hardware utilization, including arithmetic DSPs, flip flops (FF)s, and block RAM (BRAM). All models are quantized to 8 bits. A reuse factor of 2 is used. The best values are bolded.

Model	Lat. (ns) (cc)	II (ns) (cc)	DSP	LUT	FF	BRAM
Baseline	95 (19)	15 (3)	626 (5.1%)	386 294 (22.3%)	121 424 (3.5%)	4 (0.1%)
Large	135 (27)	15 (3)	2458 (20.0%)	337 172 (19.51%)	139 905 (4.05%)	4 (0.1%)
Medium	110 (22)	15 (3)	548 (4.46%)	130 426 (7.55%)	49 326 (1.43%)	4 (0.1%)
Small	105 (21)	**10 (2)**	**302 (2.46%)**	53 398 (3.09%)	28 057 (0.81%)	4 (0.1%)
Tiny	**70 (14)**	**10 (2)**	321 (2.61%)	**32 884 (1.90%)**	**15 918 (0.46%)**	4 (0.1%)

5.1. Method adaptation

In the global search stage, we explored a wide range of hyperparameters to find the optimal configuration for the deep sets architecture. These hyperparameters included the dimensionality of the latent space after applying the permutation-invariant pooling operation, the choice of pooling operation (sum, mean, or max), and the architectures of the ϕ and ρ MLPs. The ϕ MLP is applied independently to each element of the input set, while the ρ MLP operates on the pooled embeddings.

In the local search stage, we used HPO to fine-tune the training hyperparameters, such as learning rate, batch size, and regularization strength. Since inference speed is critical for jet tagging in real-time particle physics experiments, we also used model compression techniques like quantization and pruning during this stage to reduce the model size and computational cost while maintaining high classification accuracy.

For the FPGA implementation, we used a custom branch of `hls4ml` [32] that allows for more flexible reuse and parallelization factors for the deep sets architecture. This enabled us to better optimize the model for the specific target FPGA, balancing resource utilization and inference speed. The target FPGA is the Xilinx Virtex UltraScale+ VU13P (`xcvu13p-flga2577-2-e`).

5.2. Particle jet results

Tables 3 and 4 provide a comprehensive comparison of our optimized models with the original deep sets implementation [8] as the baseline for the particle jet classification task.

Our medium model achieved a 1.06% increase in accuracy while reducing the number of BOPs by 7.2× and parameters by 23% compared to the original deep sets model. For applications where inference speed is critical, our tiny model has a compelling trade-off, with a 2.8% decrease in accuracy from the original model but a substantial 30.25× reduction in BOPs and a 6× reduction in parameters. Full model architectures can

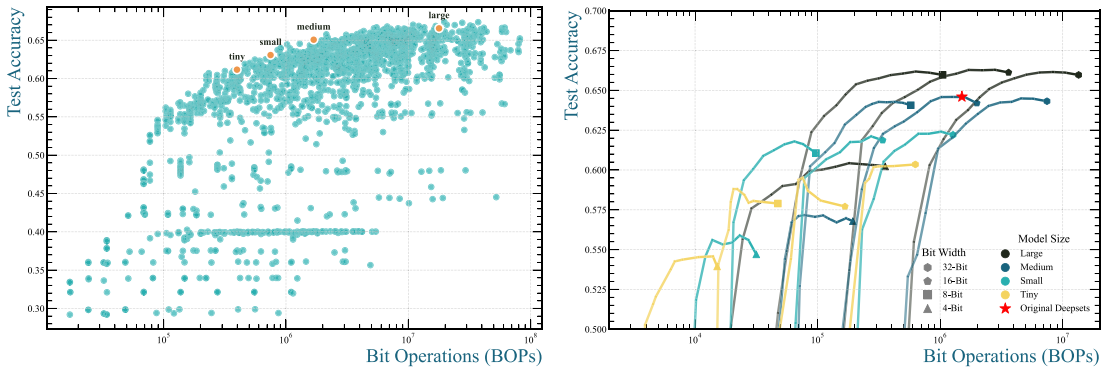

Figure 6. Particle jet dataset Pareto-optimal front for the global (left) and local (right) searches. The global search contains 2500 trials with the selected models in orange. For the local search, each chosen model is quantized to various bit precisions, and then pruned by 20% for 20 iterations indicated by increasing opacity. A bit width of 8 is chosen for its balance of BOPs and accuracy.

be accessed in appendix table B2 and the hyperparameters are depicted in appendix table B3. Additionally, all models can be pruned to over 80% sparsity (8 iterations) with less than a 10% drop in performance, depicted in figure 6(right).

When synthesized for FPGA deployment, our tiny model achieves a latency of only 70 ns with a 10 ns initialization interval, using less than 3% of the available DSP slices, LUTs, and flip-flops on the target FPGA (table 4).

The global search stage evaluated 1000 model architectures, utilizing 5 NVIDIA 3090 GPUs in parallel for a total of 61 hours. The subsequent local search stage trained 4 models with 4 quantization precisions each for 20 iterative magnitude pruning steps using 4 NVIDIA 3090 GPUs running in parallel for an additional 26 hours.

The performance of the deep sets models during global and local search can be seen in figure 6. As expected, accuracy drops down with increased pruning but the larger models can handle more sparisty before the performance drastically decreases. Furthermore, the large model achieves a 2.5% higher accuracy and a 0.87% relative improvement in LUT utilization, although the inference time is slightly longer.

6. Conclusion

In this work, we have developed a pipeline for neural architecture codesign that streamlines the process of designing and optimizing deep learning models for physics applications. Our method employs a two-stage search strategy, consisting of a global search to explore a wide range of architectures and a local search to fine-tune the most promising candidates. We have demonstrated the effectiveness of our approach through two case studies: Bragg peak analysis for materials science and jet classification for particle physics.

For the BraggNN case study, our optimized models achieved comparable performance to the original model while significantly reducing computational complexity. Our large model improved the mean distance metric by 0.5% while reducing BOPs by 5.9×, and our small model increased mean distance by only 3% while achieving a substantial 39.2× reduction in BOPs and 2.90× fewer parameters. When synthesized for FPGA deployment, our small model attained a latency of only 4.92 μs with less than 10% utilization of DSPs, LUTs, and FFs.

In the particle jet classification task using the deep sets architecture, our optimized models again demonstrated significant improvements in efficiency with minimal impact on accuracy. The medium model increased accuracy by 1.06% with a 7.2× reduction in BOPs and 23% fewer parameters compared to the original. For latency-critical applications, our tiny model trades a 2.8% decrease in accuracy for a 30.25× reduction in BOPs and 6× fewer parameters. Synthesized for target FPGA, it achieves an inference latency of only 70 ns while using less than 3% of the FPGA resources.

The results from these case studies highlight the power of NAC to discover highly efficient ML architectures tailored for physics applications and resource-constrained edge devices. This lowers the barrier to entry for domain experts looking to incorporate deep learning into their research. By automating the neural architecture design and optimization process, we allow users to take advantage of state-of-the-art techniques without requiring extensive machine learning expertise.

While we exhibit a variety of architectures that either improve performance, latency, or resource utilization, the models found are limited in that none exceed in all criteria. This limitation is primarily due to

our BOPs metric that does not translate directly to latency, as it is only a general predictor for resources. We plan for future work to cover more hardware-aware metrics, which can be done with proposed surrogate models [33, 34] to predict inference time. OpenHLS [31] and BraggHLS [35] are examples of another high level synthesis framework that synthesized the BraggNN model, achieving a latency of 4.8 μs. However, there are challenges in reproducing this work.

Future work includes introducing more integrated parallelization techniques for each stage that will help dramatically decrease search time. Additionally, our framework can be improved by expanding the search space to incorporate more diverse layer types, such as multi-branch convolutional blocks [36], and additional hyperparameters to allow for the discovery of more creative network architectures. Investigating alternative search strategies tailored to specific physics tasks or FPGA synthesis should lead to further improvements.

Data availability statement

The data that support the findings of this study are openly available at the following URL/DOI: https://doi.org/10.5281/zenodo.14618369.

Acknowledgments

N T and J D are supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research under the 'Real-time Data Reduction Codesign at the Extreme Edge for Science' Project (DE-FOA-0002501). J D is also supported by the DOE, Office of Science, Office of High Energy Physics Early Career Research program under Grant No. DE-SC0021187, and the U.S. National Science Foundation (NSF) Harnessing the Data Revolution (HDR) Institute for Accelerating AI Algorithms for Data Driven Discovery (A3D3) under Cooperative Agreement No. PHY-2117997. N T is also supported by the DOE Early Career Research program under Award No. DE-FOA-0002019.

Appendix A. Bit operations calculation

To evaluate the computational efficiency of our models, we calculate the BOPs for each layer. The BOPs for linear and convolutional layers are calculated based on [37], using the following equations. For a linear layer with m output features, n input features, weight bit precision b_w, activation bit precision b_a, and sparsity $1-p$,

$$\mathrm{BOPs}\left(\mathrm{linear}\right) = mn\left(pb_ab_w + b_a + b_w + \log_2\left(n\right)\right). \tag{A.1}$$

For a 2D convolutional layer with m output features, n input features, weight bit precision b_w, activation bit precision b_a, sparsity $1-p$, and kernel size k,

$$\mathrm{BOPs}\left(\mathrm{conv2d}\right) = mnk^2\left(pb_ab_w + b_a + b_w + \log_2\left(nk^2\right)\right). \tag{A.2}$$

To calculate the BOPs for the attention mechanism used in the BraggNN model, we first derive the BOPs for the softmax operation and matrix multiplication. For softmax with input tensor of shape (b,h,w,h,w) and bit precision b_w, the BOPs are

$$\mathrm{BOPs}\left(\mathrm{softmax}\right) = (1.5)\,b\left(hw\right)^2\left(b_w - 1\right) + bhw\left(hw - 1\right) + b\left(hw\right)^2. \tag{A.3}$$

This accounts for the exponential function, summation, and division operations in softmax.

For matrix multiplication of matrices **A** and **B** with shapes (b,m,n) and (b,n,p), respectively,

$$\mathrm{BOPs}\left(\mathrm{matmul}\right) = bmn\left(pb_w^2 + b_w\left(\log_2\left(n\right) + 1\right)\right). \tag{A.4}$$

Combining these with the BOPs for 2D convolution, the total BOPs for our ConvAttention block with input shape (b,c,h,w), hidden channels d, and kernel size k is

$$\begin{aligned}\mathrm{BOPs}\left(\mathrm{ConvAttn}\right) = &\sum_i \mathrm{BOPs}\left(\mathrm{conv2d}_i\right) + \mathrm{BOPs}\left(\mathrm{softmax}\right)\\ &+ \mathrm{BOPs}\left(\mathrm{matmul}_{\mathrm{QK}}\right) + \mathrm{BOPs}\left(\mathrm{matmul}_{\mathrm{SV}}\right),\end{aligned} \tag{A.5}$$

where i indexes over the W_q, W_k, W_v, and projection convolutions, QK indicates the query-key matrix multiplication with output shape (b,hw,hw), and SV denotes the softmax-value matrix multiplication with output shape (b,hw,d).

Appendix B. Model architectures

Table B1. Architectures of optimized BraggNN models. Conv(a, b, c, d) denotes a 2D convolutional layer with a input channels, b output channels, kernel size c, and stride d. BN denotes batch normalization layers. LeakyReLU layers have a negative slope of 1/128.

Model	Layers
Tiny	Conv(1, 8, 3, 1), Flatten, Linear($8 \cdot 9 \cdot 9$, 32), LeakyReLU, Linear(32, 32), LeakyReLU, Linear(32, 32), BN, LeakyReLU, Linear(32, 2), BN
Small	Conv(1, 8, 3, 1), Conv(8,2,3,1), LeakyReLU, Conv(2,4,3,1), BN,LeakyReLU, Flatten, Linear($4 \cdot 5 \cdot 5$, 64), BN, LeakyReLU, Linear(64,32), BN, ReLU, Linear(32,16), BN, LeakyReLU, Linear(16, 2), BN
Medium	Conv(1, 8, 3, 1), Conv(8, 16, 3, 1), BN, ReLU, Conv(16, 4, 3, 1), BN,LeakyReLU, Flatten, Linear($4 \cdot 5 \cdot 5$, 64), LeakyReLU, Linear(64, 64), LeakyReLU, Linear(64, 16), BN, LeakyReLU, Linear(16, 2), BN
Large	Conv(1, 8, 3, 1), Conv(8, 64, 3, 1), BN, Conv(64, 32, 3, 1), BN,Flatten, Linear($5 \cdot 5 \cdot 32$, 32), LeakyReLU, Linear(32, 64), ReLU, Linear(64, 64), BN, LeakyReLU, Linear(64, 2), BN

Table B2. Architectures of optimized particle jet models. BN denotes batch normalization layers. LeakyReLU layers have a negative slope of 1/128.

Model	Component	Layers
Tiny	ϕ	Linear(3, 8), BN, ReLU, Linear(8, 8)
	ρ	Linear(8, 32), ReLU, Linear(32, 5)
Small	ϕ	Linear(3, 16), LeakyReLU, Linear(16, 8), BN
	ρ	Linear(8, 8), BN, ReLU, Linear(8, 16), LeakyReLU, Linear(16, 16), LeakyReLU, Linear(16, 5), BN
Medium	ϕ	Linear(3, 32), ReLU, Linear(32, 8)
	ρ	Linear(8, 32), BN, LeakyReLU, Linear(32, 16), LeakyReLU, Linear(16, 64), BN, LeakyReLU, Linear(64, 5)
Large	ϕ	Linear(3, 64), BN, ReLU, Linear(64, 16), BN
	ρ	Linear(16, 128), BN, ReLU, Linear(128, 16), BN, LeakyReLU, Linear(16, 64), BN, ReLU, Linear(64, 5), BN
Original DeepSets	ϕ	Linear(3, 32), ReLU, Linear(32, 32), ReLU
	ρ	Linear(32, 16), ReLU, Linear(16, 5)

Table B3. Hyperparameters and configurations implemented.

Task	Learning rate	Batch size	Population size
Bragg peak	0.0015	4096	20
Jet classification	0.0032	32	20

The model architectures found for the Bragg peak and particle jet classification task can be found in tables B1 and B2. The hyperparameters and configurations implemented are described in table B3.

Appendix C. Search spaces

The search spaces for the BraggNN and particle jet models can be found in tables C1 and C2.

Table C1. Comprehensive parameter space for BraggNN.

Parameter	Space or description
General parameter space	
Block	{Conv, Attention, None}
Channel dimension	{1, 2, 4, 8, 16, 32, 64}
Kernel size	{1, 3}
Normalization method	{Batch, None}
Activation function	{ReLU, LeakyReLU, None}
Linear layer dimension	{4, 8, 16, 32, 64}
Conv block parameters	
Conv1 in channels	Previous dimension
Conv1 out channels	Sample channel dimension
Conv1 kernel	Sample kernel size
Norm1	Sample normalization method
Act1	Sample activation function
Conv2 in channels	Sample channel dimension
Conv2 out channels	Sample channel dimension
Conv2 kernel	Sample kernel size
Norm2	Sample normalization Method
Act2	Sample activation function
Attention block parameters	
QKV Dimension	Sample channel dimension
Skip Activation	Sample activation function
MLP classifier parameters (all 4 layers)	
FC1 in dimension	Previous dimension
FC1 out dimension	Sample linear space
Norm1	Sample normalization method
Act1	Sample activation function
FC2 In Dimension	Previous dimension
FC2 Out Dimension	Sample linear space
Norm2	Sample normalization method
Act2	Sample activation function

Table C2. Comprehensive parameter space for Deep Sets.

Parameter	Space or description
ϕ	
Width	{4, 8, 16, 32, 64}
Normalization method	{Batch, None}
Activation function	{ReLU, LeakyReLU, None}
Bottleneck dimension	{1, 2, 4, 8, 16, 32, 64}
Aggregator	{mean, maximum}
ρ	
Width	{4, 8, 16, 32, 64}
Normalization method	{Batch, None}
Activation function	{ReLU, LeakyReLU, None}

References

[1] Elsken T, Metzen J H and Hutter F 2019 Neural architecture search: a survey *J. Mach. Learn. Res* **20** 1–21 (available at: http://jmlr.org/papers/v20/18-598.html)
[2] Cai H *et al* 2020 Once for all: train one network and specialize it for efficient deployment *Int. Conf. on Learning Representations* (arXiv:1908.09791)
[3] Pappalardo A 2023 Xilinx/brevitas: v0.11.0 *Zenodo* (https://doi.org/10.5281/zenodo.3333552) (available at: https://github.com/Xilinx/brevitas)
[4] Coelho C N Jr *et al* 2021 Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors *Nat. Mach. Intell.* **3** 675
[5] Akiba T *et al* 2019 Optuna: a next-generation hyperparameter optimization framework *Proc. 25th ACM SIGKDD Int. Conf. on Knowledge Discovery & Data Mining* p 2623
[6] Duarte J *et al* 2018 Fast inference of deep neural networks in FPGAs for particle physics *JINST* **13** 07027
[7] Liu Z Sharma H, Park J-S, Kenesei P, Miceli A, Almer J, Kettimuthu R and Foster I 2022 BraggNN: fast x-ray Bragg peak analysis using deep learning *IUCrJ* **9** 104–13
[8] Zaheer M *et al* 2017 Deep sets *Advances in Neural Information Processing Systems* vol 30, ed I Guyon (Curran Associates, Inc) (arXiv:1703.06114)
[9] Komiske P T, Metodiev E M and Thaler J 2019 Energy flow networks: deep sets for particle jets *J. High Energy Phys.* JHEP01(2019)121
[10] Odagiu P *et al* 2024 Ultrafast jet classification at the hl-lhc *Mach. Learn. Sci. Technol.* **5** 035017
[11] Mondal S and Mastrolorenzo L 2024 Machine learning in high energy physics: a review of heavy-flavor jet tagging at the LHC *Eur. Phys. J. ST* **233** 2657
[12] Demler D *et al* fastmachinelearning/nac-opt: v0.2.0 2025 *Zenodo* (available at: https://github.com/fastmachinelearning/nac-opt) (https://doi.org/10.5281/zenodo.14618369)
[13] Liu Y Sun Y, Xue B, Zhang M, Yen G G and Tan K C 2021 A survey on evolutionary neural architecture search *IEEE Trans. Neural Netw. Learn. Syst.* **34** 550
[14] Liu S, Zhang H and Jin Y 2022 A survey on computationally efficient neural architecture search *J. Autom. Intell.* **1** 100002
[15] Shariatzadeh S M, Fathy M, Berangi R and Shahverdy M 2023 A survey on multi-objective neural architecture search (arXiv:2307.09099)
[16] Ren P *et al* 2021 A comprehensive survey of neural architecture search: challenges and solutions *ACM Computing Surveys (CSUR)* **54** 1
[17] Abdelfattah M S, Mehrotra A, Dudziak Ł and Lane N D 2021 Zero-cost proxies for lightweight NAS *Int. Conf. on Learning Representations* (arXiv:2101.08134)
[18] Kandasamy K *et al* 2018 Neural architecture search with bayesian optimisation and optimal transport *Advances in Neural Information Processing Systems* vol 31
[19] Zoph B and Le Q V 2016 Neural architecture search with reinforcement learning (arXiv:1611.01578)
[20] Bergstra J, Bardenet R, Bengio Y and Kégl B 2011 Algorithms for hyper-parameter optimization *Advances in Neural Information Processing Systems* vol 24, ed J Shawe-Taylor (Curran Associates, Inc)
[21] Deb K, Pratap A, Agarwal S and Meyarivan T 2002 A fast and elitist multiobjective genetic algorithm: Nsga-ii *IEEE Trans. Evol. Comput.* **6** 182
[22] Liang T Glossner J, Wang L, Shi S and Zhang X 2021 Pruning and quantization for deep neural network acceleration: a survey *Neurocomputing* **461** 370
[23] Blalock D, Ortiz J J G, Frankle J and Guttag J 2020 What is the state of neural network pruning? *Proc. Mach. Learn. Syst.* **2** 129
[24] Sui X , Lv Q, Zhi L, Zhu B, Yang Y, Zhang Y and Tan Z 2023 A hardware-friendly high-precision CNN pruning method and its FPGA implementation *Sensors* **23** 824
[25] Zhang Y *et al* 2022 Learning best combination for efficient N:M sparsity *Advances in Neural Information Processing Systems* vol 35 p 941 (arXiv:2206.06662)
[26] Sun M, Liu Z, Bair A and Kolter J Z 2024 A simple and effective pruning approach for large language models *The 12th Int. Conf. on Learning Representations* (arXiv:2306.11695)
[27] Kim S *et al* 2024 SqueezeLLM: dense and sparse quantization *Proc. 41st Int. Conf. on Machine Learning* (JMLR.org) (arXiv:2306.07629)
[28] Deiana A M *et al* 2022 Applications and techniques for fast machine learning in science *Front. Big Data* **5** 787421
[29] FastML Team 2023 fastmachinelearning/hls4ml: v0.8.0 *Zendo* (available at: https://github.com/fastmachinelearning/hls4ml) (https://doi.org/10.5281/zenodo.10140081)
[30] Park J-S Zhang X, Kenesei P, Wong S L, Li M and Almer J 2017 Far-field high-energy diffraction microscopy: a non-destructive tool for characterizing the microstructure and micromechanical state of polycrystalline materials *Microsc. Today* **25** 36
[31] Levental M *et al* 2023 OpenHLS: high-Level synthesis for low-latency deep neural networks for experimental science (arXiv:2302.06751)
[32] Que Z W, Duarte J, Odagiu P and Sznajder A 2024 fastmachinelearning/l1-jet-id: v0.2.0 *Zenodo* (available at: https://github.com/fastmachinelearning/l1-jet-id) (https://doi.org/10.5281/zenodo.14531416)
[33] Gautier Q, Althoff A, Crutchfield C and Kastner R 2022 Sherlock: a multi-objective design space exploration framework *ACM Trans. Des. Autom. Electron. Syst.* **27** 1–20
[34] Rahimifar M M, Rahali H E and Therrien A C 2025 rule4ml: an open-source tool for resource utilization and latency estimation for ML models on FPGA *Mach. Learn.: Sci. Technol.* **6** 015009
[35] Levental M *et al* 2024 Bragghls: High-level synthesis for low-latency deep neural networks for experimental science *Proc. 14th Int. Symp. on Highly Efficient Accelerators and Reconfigurable Technologies*, HEART'24 (Association for Computing Machinery) pp 10–17 (https://doi.org/10.1145/3665283.3665284)
[36] Szegedy C, Ioffe S, Vanhoucke V and Alemi A 2016 Inception-v4, inception-resnet and the impact of residual connections on learning (arXiv:1602.07261)
[37] Campos J , Mitrevski J, Tran N, Dong Z, Gholaminejad A, Mahoney M W and Duarte J 2024 End-to-end codesign of hessian-aware quantized neural networks for fpgas *ACM Trans. Reconfigurable Technol. Syst.* **17** 1