PLiNIO: A User-Friendly Library of Gradient-based Methods for Complexity-aware DNN Optimization

PLiNIO: A User-Friendly Library of Gradient-based Methods for Complexity-aware DNN Optimization

Daniele Jahier Pagliari*, Matteo Risso*, Beatrice Alessandra Motetti*, Alessio Burrello*
*Politecnico di Torino, Turin, Italy.
Corresponding Email: daniele.jahier@polito.it

***Abstract*—Accurate yet efficient Deep Neural Networks (DNNs) are in high demand, especially for applications that require their execution on constrained edge devices. Finding such DNNs in a reasonable time for new applications requires automated optimization pipelines since the huge space of hyper-parameter combinations is impossible to explore extensively by hand. In this work, we propose PLiNIO, an open-source library implementing a comprehensive set of state-of-the-art DNN design automation techniques, all based on lightweight gradient-based optimization, under a unified and user-friendly interface. With experiments on several edge-relevant tasks, we show that combining the various optimizations available in PLiNIO leads to rich sets of solutions that Pareto-dominate the considered baselines in terms of accuracy vs model size. Noteworthy, PLiNIO achieves up to 94.34% memory reduction for a <1% accuracy drop compared to a baseline architecture.**

***Index Terms*—NAS, Pruning, Quantization, Deep Learning, PyTorch, Design Space Exploration, Domain-specific Computing**

I. Introduction

Deep Neural Networks (DNNs) reach state-of-the-art performance in many applications, ranging from computer vision to bio-signals processing, but are extremely expensive in terms of computation and memory [1]–[3]. This is currently considered somewhat of a secondary issue for cloud-hosted models, whose accuracy has improved in each new generation as an effect of sheer model upscaling, also thanks to the availability of gargantuous amounts of data. However, for tasks that require the execution of DNNs on mobile or edge devices, limiting computational complexity and memory footprint is fundamental [2], and even in the cloud, hardware/energy costs and sustainability issues will eventually mandate a careful consideration of complexity [4].

Unfortunately, DNNs have a very large set of hyper-parameters, i.e., configurations that are not (traditionally) trained by gradient descent together with the model weights, yet greatly influence results. At a high level, we can distinguish *training hyper-parameters* (e.g., the optimizer used for training, the initial learning rate and its schedule, etc) and *architectural hyper-parameters* (e.g., the number and type of layers, their configuration, the weights and activations bitwidth, etc). The former only affect the training process and, therefore, the accuracy of the resulting model. The latter, instead, strongly impact both predictive performance and inference complexity. Furthermore, they can be set in virtually infinite combinations, creating an immense optimization space [5]. Exploring such space by hand is prone to following conventional rules of thumb and results in suboptimal outcomes [6].

Accordingly, design exploration and automated optimization tools, generally referred to as AutoML [7] are becoming popular to design accurate yet compact and efficient DNNs for new applications, especially when targeting constrained edge hardware. More specifically, Neural Architecture Search (NAS) methods automate the search for optimal combinations of layers and their configurations [6], whereas Mixed-Precision Search (MPS) solutions look for the optimal data representation for each model tensor [8]. In both cases, early approaches resorted to time-consuming black-box optimization methods such as Reinforcement Learning (RL) and Evolutionary Algorithms (EA), which required tens of GPU-days for a single optimization [6]. More recently, gradient-based NAS and MPS have been proposed as lightweight alternatives to these solutions. These so-called *One-shot* or *Differentiable* methods utilize gradient-descent to simultaneously train a DNN and optimize its architecture, thus obtaining an optimized model in a time comparable to a single training [9].

One key limitation of gradient-based approaches, however, is the lack of user-friendly libraries that can be employed by ML practitioners without experience on NAS or MPS to optimize a DNN for their applications while ignoring implementation details. Such a library should also combine optimizations targeting multiple architectural hyper-parameters, at different granularity levels, in order to fully explore the design space. Similar tools have been recently released both commercially [10] and open-source [11], but mostly for resource-hungry iterative (i.e., RL, EA, etc.) AutoML methods.

In this paper, we present **PLiNIO**, a library for **P**lug-and-play **Li**ghtweight **N**eural **I**nference **O**ptimization, which tries to bridge this gap by providing a unified and user-friendly domain-specific language for a diverse set of gradient-based AutoML procedures. Namely, PLiNIO currently supports: i) a *coarse-grained NAS* for selecting among alternative layers [9]; ii) a *fine-grained NAS* for optimizing each layer's internal hyper-parameters (e.g., the number of channels in a Convolutional layer) [12]; iii) a *differentiable MPS* method for selecting both weights and activation bit-widths and quantization parameters, supporting common quantization formats [13]–[15]. Since the fine-grained NAS in ii) is analogous to structured pruning [12], PLiNIO supports three of the most common complexity-driven DNN optimizations in the state-of-the-art, i.e., **Quantization, Pruning and NAS** [2],

This work has received funding from the Key Digital Technologies Joint Undertaking (KDT-JU) under grant agreement No 101095947. The JU receives support from the European Union's Horizon Europe research and innovation programme.

Accepted at the 2023 Forum on Specification & Design Languages (FDL).

```
1 model = MyNN()
2 model = plinio.Method(model, {'cost': cost_fn}, ...)
3 for epoch in range(N_EPOCHS):
4   for sample, target in data:
5     output = model(sample)
6     loss = criterion(output, target)+model.get_cost('
       cost')
7     optimizer.zero_grad()
8     loss.backward()
9     optimizer.step()
10 exported_model = model.export()
```

Fig. 1. Standard PyTorch training loop turned into a PLiNIO optimization. The `Method()` call is a place-holder for `SuperNet()`, `PIT()` or `MPS()`.

[3] under a unified API thus enabling a complete Design Space Exploration (DSE) for DNN workloads. Furthermore, PLiNIO's internals are designed to support extensibility, and we plan to integrate the library with additional gradient-based optimizations in future releases. PLiNIO is available open-source at: `https://github.com/eml-eda/plinio`.

Fig. 1 shows the modifications required to implement a PLiNIO optimization on top of a standard training loop in PyTorch, i.e., the DNN training framework on which our library is based. As shown, PLiNIO only requires three method invocations, highlighted in the figure, to convert a standard PyTorch DNN into an optimizable model (line 2), estimate its complexity according to one or more cost metrics during the optimization (line 6) and export the final optimized model at the end of the process (line 10). This interface applies to all supported gradient-based optimizations, making them very simple to integrate into existing codebases.

To the best of our knowledge, PLiNIO is the first library to support gradient-based NAS, pruning and MPS in a single unified framework. In order to demonstrate its usefulness for DNN workloads' DSE, we run experiments on three edge-relevant use-cases taken from the MLPerf Tiny benchmark suite [16], using corresponding state-of-the-art reference DNNs as starting point for the optimization. Our results show that by combining all three optimizations, PLiNIO can reduce the DNN storage size by up to 94.34% with a limited accuracy drop (-0.92%) with respect to the reference. Noteworthy, the final model has 78.88% fewer parameters compared to the best one obtained applying *each optimization individually*.

II. Background

A. Neural Architecture Search

NAS search algorithms can be broadly categorized into black-box methods, such as RL and EA, and Differentiable NAS (DNAS) or one-shot methods. The former requires three main steps: i) sampling one or more architectures from the search space, ii) evaluating the objective function and then iii) updating the sampling policy. These methods can optimize almost any function, with both accuracy- and complexity-related terms, on a discrete search space [6]. However, they are often intractable, mainly due to the evaluation step (ii). In fact, evaluating the accuracy of a sampled DNN ideally requires training it to convergence, whereas cost metrics should be obtained directly from deployment, both of which largely increase the search time. This is partially mitigated by the use of "proxies" [17], such as training on a subset of the dataset or for a few epochs [9] and using Look-up Tables [17] or other approximate models [18], [19] for cost metrics (e.g., memory occupation, latency or energy). Still, black-box methods remain extremely time-consuming [6], [20].

DNAS reduce the optimization time significantly by relaxing the search space from discrete to continuous, making the problem suitable for gradient-descent [6]. Namely, they define a set of *architectural parameters* (θ) which encode the selection of a DNN from the search space and train them together with the weights of the networks. When optimizing for both functional (accuracy) and non-functional metrics, a DNAS training loop typically uses a loss function in the form:

$$\min_{W,\theta} \mathcal{L}(W;\theta) + \lambda \mathcal{R}(\theta) \tag{1}$$

where W are the normal DNN weights, $\mathcal{L}$ is the task-dependent functional loss, $\mathcal{R}$ is a differentiable cost estimate, and λ is a strength hyper-parameter that controls the balance between the two. Typical expressions for $\mathcal{R}$ encode the model size (number of parameters) or inference operations (OPs) as a function of θ, but more complex latency approximations are also possible [17], [21] (Sec. IV-C). At the end of the search, the θ parameters are discretized to export the final DNN.

More specifically, *path-based* DNAS methods [9] define a DNN (the *supernet*) whose graph includes multiple alternative paths corresponding to the possible alternative operations in the search space. The optimization reduces to selecting one of these paths, as detailed in Sec. IV-A1. The main issue with this approach is that the supernet size grows quickly with the search space, limiting scalability. Advanced methods such as ProxylessNAS [17] and HardCoReNAS [22] solve this sampling few paths per training iteration.

Mask-based DNAS further reduce the optimization cost by searching only among the DNNs that can be obtained by shrinking an initial architecture, called *seed*, in a way similar to structured pruning. In particular, slices of the DNN weights or activations tensors are coupled with binary masks, whose continuous relaxation is trained with gradient descent. At the end of the search, the masked parts of the seed layers are eliminated. Thus, the search space of these methods is more restricted w.r.t. path-based DNAS (only subsets of the seed are explored). However, the search granularity can be much finer. Examples of *mask-mased* methods are MorphNet [23], FbNetV2 [24] and PIT [12].

B. Quantization and Mixed-precision Search

Integer quantization is a key DNN optimization, especially at the edge, consisting in the approximation of floating point weights and activations with low bitwidth integers, improving both model size and efficiency [25]. While the conversion can be done post-training [26], simulating the effect of quantization at training time (so-called Quantization-Aware Training or QAT) [25] can help the DNN adapt to the data approximation, reducing the drop in accuracy.

Standard *fixed-precision* quantization assigns the same integer bit-width to the whole DNN, thus neglecting the sensitivity of each layer to precision reductions. However, previous works [13], [15] show that some layers (e.g., those close to the input and to the output) tend to require higher precision.

Mixed-precision methods address this issue by quantizing various subsets of the DNN at different bitwidths. This creates a new and not trivial optimization problem, i.e., finding precision assignments that yield good trade-offs between accuracy and complexity, exploring a search space whose size increases exponentially with the number of considered bitwidths [27].

Various MPS approaches have been proposed to tackle this problem, which is orthogonal to NAS. Some exploit sensitivity metrics such as the layers' Hessian spectrum [27] or the Signal to Quantization Noise Ratio at different precisions [28], while others are based on RL [8]. More recently, the authors of [13], [15] proposed a gradient-based method similar to DNAS to assign bitwidths during training. This is done by quantizing data at every possible precision on-the-fly, and then learning to select a single precision during training, similarly to [9].

III. Related Works

The need for efficient and accurate DNNs has brought a plethora of techniques and algorithms for AutoML, including NAS, pruning and quantization [6]. Given the very quick innovation pace, the software ecosystem is naturally fragmented, with most techniques being shared as isolated and hardly usable "research scripts". Recently, comprehensive and engineered tools have started to emerge, both commercially and open-source, to tackle this problem.

Commercial tools for DNN optimization are usually inserted in larger AutoML pipelines, which also help users with data preprocessing and labeling, model deployment, etc, and are commonly offered as cloud services. Some examples include Azure Machine Learning, Google Cloud AutoML, etc. Other providers offer similar features but target specifically constrained edge devices. Qeexo [29] is an example of no-code end-to-end autoML platform for the edge, in which model selection is performed by selecting a specific instance from a zoo of predefined algorithms, which are then quantized at fixed precision. Edge Impulse's EONTuner [10] follows a similar approach but offers more flexibility for model selection, with the possibility to define a coarse search space (e.g., number of convolutional/linear layers, presence of pooling, etc), searched with hyperband or random-search.

In the open-source landscape, DL frameworks such as Tensorflow and PyTorch support basic optimizations natively. Namely, the TensorFlow Model Optimization Toolkit (TFMOT) [30] implements fixed-precision QAT, pruning, and weight clustering, generating models compatible with the TFLite converter and interpreter for edge devices. Likewise, PyTorch supports different quantization and pruning techniques and it exposes APIs to implement new ones [31]. Similar optimizations are also targeted by the open-source AI Model Efficiency Toolkit (AIMET) [32].

Concerning NAS and hyperparameters optimization, one of the first attempts to realize a user-friendly library is represented by AutoKeras [11]. This tool lets users define search spaces or provides pre-defined ones for specific tasks such as Image or Text Classification. It then offers different search strategies, including hyperband and Bayesian optimization. Neural Network Intelligence (NNI) [33] by Microsoft implements several optimization algorithms under a unified framework with a common API. In particular, it supports Bayesian and heuristic-based hyper-parameters optimization, NAS with both iterative and gradient-based approaches, pruning, and fixed-precision quantization. Vega [34] similarly groups in the same codebase different NAS algorithms along with pruning and mixed-precision assignment based on EA. To the best of our knowledge, no existing single framework supports path-based DNAS, mask-based DNAS, and gradient-based MPS with a unified interface.

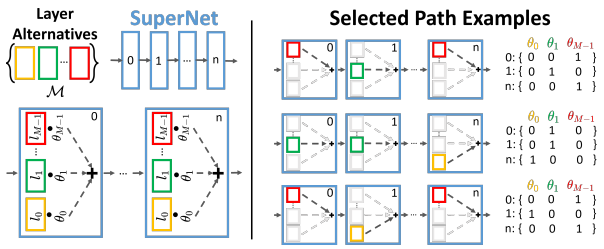

Fig. 2. SuperNet implementation in PLiNIO.

IV. PLiNIO

PLiNIO is, to our knowledge, the first open-source tool to support multiple gradient-based DNN optimizations, spanning the dimensions of: i) coarse architectural choices (path-based DNAS), ii) layer hyper-parameters optimization (mask-based DNAS) and iii) precision selection (MPS), through a simple, user-friendly interface. Its main scientific value is allowing to study the combination of DSE and optimizations at different levels, and their interactions, which may lead to superior results with respect to any single method (see Section V).

While there exist tools with similar flexibility leveraging at least in part black-box methods [34], an entirely gradient-based toolchain helps to democratize research in this field, since lightweight gradient-based methods might be the *only alternative* to being forced to use third-party cloud services for users that do not own large GPU clusters. One domain where this is particularly relevant is TinyML [3], i.e., systems that implement DNN inference directly on tightly constrained mobile or IoT edge devices. Although PLiNIO can in principle support the optimization of any DNN, regardless of its size and of the hardware target, TinyML is in fact its primary use case.

In the rest of this Section, we first describe the three optimization techniques currently supported by the library (Sec. IV-A). We then detail some of the DNN graph transformation passes required to implement the simple interface of Fig. 1, hiding most complexity from the users (Sec. IV-B). Lastly, we discuss the extensible DNN cost models supported by PLiNIO for complexity-aware optimization (Sec. IV-C).

A. Supported Optimization Techniques

1) SuperNet: As the first DNAS, PLiNIO implements a *path-based* method based on a *supernet*, schematized in Fig. 2. The method is inspired by DARTS [9], but it differs by using the Gumbel-Softmax sampling strategy, in accordance with more recent works [19], [24], instead of the standard SoftMax as done in [9]. The supernet is built by replacing each layer L of a standard DNN with an ensemble of M alternatives $\mathcal{M} = \{l_i\}_{m=0}^{M-1}$. All $l_i \in \mathcal{M}$ receive the same input, and their

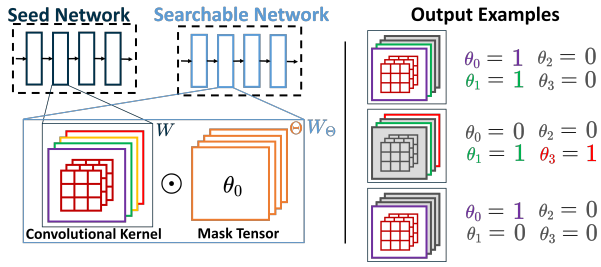

Fig. 3. PIT channel-masking implementation in PLiNIO.

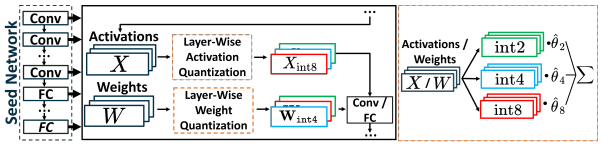

Fig. 4. MPS implementation in PLiNIO.

outputs are linearly combined using trainable parameters θ_i, passed through a Gumbel-Softmax (GS), i.e.,:

$$Y = \sum_i GS(\theta_i) \cdot l_i(X) \tag{2}$$

The selection of layers is reduced to training θ, jointly with the normal layer weights W, to minimize Eq. 1. At the end of the training, the optimized DNN is obtained by selecting, from each set, the alternative corresponding to the largest θ_i.

2) PIT: The second DNAS implemented in PLiNIO is a *mask-based approach* akin to structured pruning. Starting from a seed network, this method optimizes the main architectural hyper-parameters of Convolutional (Conv) and Fully-Connected (FC) layers at fine grain. It extends PIT [12], which was originally proposed to optimize the most important hyper-parameters of 1D Conv, (i.e., number of output channels, receptive-field, and dilation) to also support output channels optimization for 2D layers.

Its channel-search approach is summarized in Fig. 3. Starting from the seed, each Conv or FC weight tensor W, with C_{out} output channels, is masked as follows:

$$W_\Theta = W \odot \mathcal{H}(\theta) \tag{3}$$

where θ is a vector of C_{out} trainable mask parameters, $\odot$ is the Hadamard product, and $\mathcal{H}$ is a Heaviside step function to binarize θ. Each element θ_i masks an entire output channel of W, controlling whether it is kept ($\mathcal{H}(\theta_i) = 1$) or removed from the network ($\mathcal{H}(\theta_i) = 0$).

Similarly to the supernet of Sec. IV-A1, the DNN with masked weights is inserted in a normal training loop, where W and θ are trained together to minimize Eq. 1. During forward training passes, the use of $\mathcal{H}$ has the effect of sampling of one architecture from the search space, as shown on the right of Fig. 3. Instead, in backward passes, a Straight-Through Estimator (STE) [12] technique is used to ensure that gradients flow through the non-differentiable $\mathcal{H}$.

3) MPS: Fig. 4 summarizes the MPS method implemented in PLiNIO to assign independent precision to weights W and activations X in Conv and FC layers. The method is inspired by [13], extended with additional quantization formats. Given the set of supported bit-widths $p \in P$, e.g., $P = \{2, 4, 8\}$,

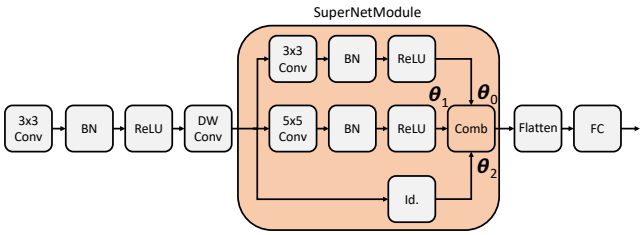

Fig. 5. SuperNet module example. Id. = Identity operation.

each tensor T (either W or X) is fake-quantized [25] at all bit-widths. The differently quantized "variants" are linearly combined by trainable parameter vectors θ of length $\|P\|$, normalized by means of a SoftMax function (SM). In practice, an *effective* tensor is obtained as:

$$\hat{T} = \sum_p SM(\theta_p) \cdot T_p \tag{4}$$

where T_p is the p-bit version of T. Therefore, increasing the value of θ_p causes the output tensor $\hat{T}$ to resemble more the result of p-bit quantization. Importantly, all fake-quantized versions are derived from a *single* float tensor, thus minimizing the method's memory overhead at training time.

The effective tensors $\hat{W}$ and $\hat{X}$ are then used to compute the layer's output, e.g.,: $Y = \text{Conv}(\hat{X}, \hat{W})$. As for the other PLiNIO methods, the DNN, thus modified, is inserted in a DNAS-like training loop to jointly optimize W and θ according to Eq. 1.

B. Automatic Model Conversion and Export

PLiNIO lets users define the optimization input DNN as a standard `nn.Module` sub-class, as in vanilla PyTorch. The only special DNN definition construct is a new type of "layer", through which users can explicitly define the alternative paths that form the search space of the method in Sec. IV-A1. The constructor of this class, called `SuperNetModule`, takes as input a list of `nn.Module` instances, each corresponding to a possible optimization alternative, i.e., either a single layer or a more complex sub-network. Fig. 5 shows an example with three inputs, where the additional Comb node, added automatically during the conversion, combines the various branches through Eq. 2. This gives maximum freedom to users, allowing them to easily consider different alternatives for each layer rather than a fixed set of operations for the whole DNN. Besides this, all other transformations required to make a standard PyTorch DNN optimizable by PLiNIO occur *transparently*, when the model is passed to a method's constructor (line 2 of Fig. 1). Namely, a series of conversion passes are performed, which make extensive use of the `torch.fx` toolkit, as detailed below. Fig. 6 shows an example of this conversion for PIT on a portion of a plausible DNN graph.

1) Layer Auto-conversion: PIT and MPS are commonly applied to *all* Conv and FC layers of a DNN. Thus, in this case, PLiNIO does not require users to explicitly define optimizable layers. Rather, it identifies and converts `nn.Conv` and `nn.Linear` layers automatically (orange boxes in Fig. 6), adding architectural masks/parameters (θ) as needed. Optional user-specified rules, by name or type, can exclude parts of the model from the optimization.

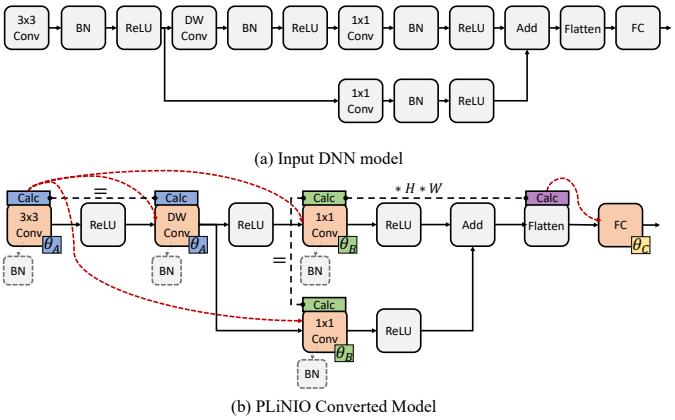

Fig. 6. PLiNIO auto-conversion example.

One key analysis pass performed during layer conversion is *mask sharing*. In fact, for both PIT and MPS, the architectural parameters of different layers shall not always be optimized independently of each other. For instance, each output channel of the DepthWise (DW) Conv layer in Fig. 6b processes a different input channel. Therefore, if the j-th output channel is masked by PIT based on Eq. 3, the j-th input activation map also becomes useless, and so do the weights and computations of the preceding 3x3 Conv layer that produced it. This is addressed *sharing the channels mask* for the two layers (θ_A in the figure). Similar reasoning also applies to the two 1x1 Conv layers, whose outputs converge into an element-wise Add. MPS auto-conversion includes an analogous pass for layers that need to share the same output activations bit-width and quantizer parameters, such as the two Add inputs. Tools such as EdMIPS [13] and the NAS API of NNI [33] do not apply mask sharing automatically, producing outputs that require further post-hoc transformations to become deployable, possibly affecting both their accuracy and their cost.

2) Batch Normalization Folding: For PIT and MPS, another pass folds Batch Normalization (BN) with the preceding Conv/FC layers. For MPS, this is needed to closely mimic a full-integer inference, which normally does not support BN, thus improving the consistency between the fake-quantized layers and the final integer model. For PIT, folding is required because the zero-mean output of BN would back-propagate a very small-magnitude gradient from the task loss term $\mathcal{L}$ of Eq. 1 to the channel mask parameters of Eq. 3. Thus, the masks gradients would be dominated by the cost term $\mathcal{R}$. In other words, the optimization would prune channels only based on their cost and not on their impact on accuracy. PLiNIO saves the original BN parameters in a special Conv/FC field (small dashed squares in Fig. 6b), permitting to optionally *unfold* the BN at the end of the optimization.

3) Effective Input Shape/Bitwidth Calculation: A layer's input tensor shape and precision greatly influence its cost. For instance, pruning some channels from the 1x1 Conv layers in Fig. 6b with PIT not only reduces their size/OPs, but also affects the cost of the following FC layer, which has to process a smaller number of inputs. As for architectural parameters sharing, many other libraries do not account for these cost dependencies during gradient-based optimization.

PLiNIO does so by first performing a DNN graph traversal that associates each layer to the one(s) that determine its input tensor shape/bitwidth. An example of the result for the *channels* dimension is shown by red dashed lines in Fig. 6b. The map is created for all nodes but shown only for the orange ones for simplicity. In practice, the static analysis pass identifies layers that may alter the number of channels (e.g., 3x3 or 1x1 Conv) or not (e.g., DW Conv or ReLU), hierarchically traversing user-defined `nn.Modules`. It then associates each layer with its closest channel-defining predecessor. Special cases are also dealt with, e.g., concat operations. A similar association is also done for MPS, linking each layer with the one that determines its input activations bitwidth.

Then, a second pass associates *effective shape/bitwidth calculator* objects to each layer. The effective shape differs from the static size of PyTorch tensors because, for example, PIT does not actually *eliminate* parts of the layer but only sets them to zero. Thus, the static shape remains unchanged until the final DNN export, and using it to estimate a layer's computational cost, e.g., in terms of parameters or OPs would lead to gross over-estimations. Effective shape calculators (Calc), shown as small coloured rectangles in Fig. 6b for the channels dimension, solve this issue by estimating the shape that would be obtained by exporting the currently sampled model as a function of the θ parameters. For example, if a binarized θ array for a layer with $C_{out} = 32$ has 20 zeroes, then $C_{out,eff} = 12$. Clearly, layers that share the same masks also share the calculator. Additionally, more complex relationships are also inferred. For instance, the number of output channels of the Flatten operation in Fig. 6b depends on the preceding Conv through a multiplicative factor. A similar mechanism estimates the *effective input bitwidth* for MPS, since the input activations precision is relevant for estimating the time/energy cost of a layer.

4) Final Model Export: At the end of a PLiNIO search loop, the `export()` method (line 10 of Fig. 1) triggers an opposite conversion process to output the final optimized model as a vanilla `nn.Module`, that users can further train or deploy using their existing infrastructure.

To this end, the model is cleaned up from all the support structures added by PLiNIO, and the target layers are converted back to the corresponding standard PyTorch classes. `SuperNetModule` instances are replaced by the selected branches, and the combiner node is removed from the DNN graph. PIT target layers are replaced with a new instance of the same type which does not include the pruned portions, and the weights which have been preserved by the optimization are copied to the new layer. BN is optionally unfolded. For MPS, all layers are converted to fake-quantized versions at the selected precision. This serves as an intermediate step that allows us to possibly fine-tune the model before the final integerization, which is hardware-specific [25].

C. DNN Cost Specification

PLiNIO is flexible with respect to the definition of the DNN complexity model used for the optimization ($\mathcal{R}$ in Eq. 1). Default cost metrics such as model size or number of OPs are provided with the library, as well as more detailed

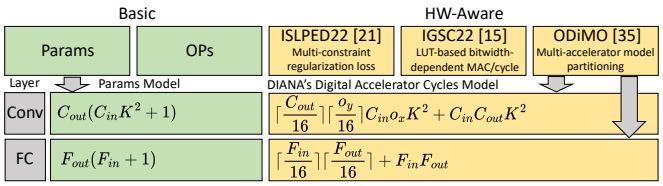

Fig. 7. PLiNIO cost specification examples.

models for specific HW targets. Some examples, reported in Fig. 7, include the LUT-based bitwidth-dependent MAC/cycle model for the MPIC RISC-V processor's vector unit of [15] and the dataflow-aware cycles model for the accelerators of [35]. A dictionary of cost specifications is passed to the PLiNIO constructors (line 2 of Fig. 1), and the corresponding cost values as a function of the optimization parameters θ can then be retrieved using the `get_cost` method of the optimizable model (line 6). We support more than one cost specification, which the user can freely combine through any differentiable function when computing the total loss, to allow DNN optimization over multiple complexity dimensions, for instance, trying to balance accuracy and latency under a size constraint, as discussed in [21].

Cost specifications are key-value maps, associating DNN graph patterns to differentiable PyTorch functions that estimate the corresponding cost as a scalar. A simple example is shown in Fig. 7 for the "params" model, and for the digital accelerator model of [35]. The parameters passed to the cost function include all geometrical shapes of the matched layers for PIT and SuperNet (e.g., for a Conv., input/output channels, kernel size, dilation, etc.), and the bit-width of all involved tensors (inputs, weights, biases and outputs) for MPS.

User-defined cost specifications can use all or a subset of these inputs, depending on their level of detail. Inputs use the default naming of PyTorch (e.g., C_{out} is `out_channels` for a Conv), to make the definition of cost metrics as orthogonal as possible to the optimization method. It is then PLiNIO's responsibility to internally invoke the function with the correct input values. For example, PIT will substitute the static C_{out} with $C_{out,eff}(\theta)$ calculated as discussed in Sec. IV-B3. Cost specifications also provide a default behaviour for unmatched DNN graph portions, which is often to assume 0 cost (e.g., for negligible operations) or to trigger an exception.

V. Results

A. Experimental Setup

We test PLiNIO on three benchmarks taken from the MLPerf Tiny suite [16]. Namely, image classification (ICL), visual wake word (VWW), and keyword spotting (KWS). For each task, the suite defines a reference DNN, which we use as seed for PIT or as blueprint to construct the SuperNet. We create the SuperNet replacing all Conv layers of the reference DNN with a `SuperNetModule` that selects between: i) a Conv with 3×3 filter, ii) a Conv with 5×5 filter, iii) a DW-separable convolution, which consists of a 3×3 DW Conv followed by a 1×1 pointwise Conv [16] and iv) an identity operation, to possibly skip the layer.

The ICL task is based on the CIFAR-10 dataset and the reference DNN is a ResNet-like architecture with 8 Conv layers. For VWW, the goal is to classify whether an input image contains at least one person. The dataset is MSCOCO 2014 with a reference model based on MobileNetV1 with a width multiplier of 0.25. Lastly, KWS uses the Speech Commands v2 dataset. The reference architecture is a simple DW Separable CNN (DS-CNN) [16]. For sake of space, we omitted the least interesting MLPerf Tiny task, Anomaly Detection, whose reference architecture is a fully-connected autoencoder which does not offer the possibility to explore different layer alternatives with SuperNet. PLiNIO is implemented using Python 3.9 and PyTorch 1.13.1. We compare the results of PLiNIO optimizations with the reference DNNs for each task.

Moreover, we compared the SuperNet and PIT results on ICL with two similar methods taken from NNI [33] 2.10.1. All experiments are executed on a machine with 32 GBs of RAM, a Intel(R) Core(TM) i3-9100F CPU running at 3.60GHz, and a Quadro P2200 GPU. We use the model size as PLiNIO's cost model for complexity-aware optimization (see Sec. IV-C).

B. Single NAS: PIT vs SuperNet

The results of applying PIT and SuperNet individually to the three reference DNNs are shown in Fig. 8. Each plot shows the reference (black square) and the optimized architectures (coloured dots) in the accuracy vs model size. The λ in Eq. 1 was varied in the range between 1e-2 and 1e-10.

The left plot shows that, on ICL, SuperNet tends to outperform PIT in terms of accuracy for a given storage footprint budget. Table I reports some of the most interesting architectures found by PLiNIO, whose memory spans from 31.6 kB to 405.08 kB and accuracy ranges from 72.9% to 88.05%. SuperNet achieves the highest accuracy (4.02% higher than the seed) with a memory overhead of 34.04%. Additionally, SuperNet also achieves the greatest reduction in memory (73.23%) without accuracy loss (+0.91%).

The middle graph shows that, on the contrary, PIT greatly outperforms SuperNet on VWW. This is due to the large number of channels in seed layers, which creates a lot of memory-saving opportunities for mask-based DNAS, and demonstrates the importance of having both types of optimization in the library. Many of the discovered architectures Pareto-dominate the seed: PIT finds DNNs that achieve between 78.73% and 85.88% accuracy with memory between 6.24 kB and 83.55 kB. At Iso-Accuracy, we achieve a striking 97.60% memory reduction. The SuperNet approach, while being outperformed by PIT, is still capable of extracting architectures that are smaller yet equally accurate than the seed.

The right plot shows the results on KWS, where PLiNIO finds many Pareto-optimal solutions with the PIT algorithm, spanning between 83.74% and 92.82% accuracy. Conversely, SuperNet never outperforms the seed, a result that testifies the goodness of the hand-tuned layer selection for this particular reference DNN. Similar to VWW, the networks found with SuperNet are outperformed by those found with PIT, indicating that changing the layer types or removing some of them is not beneficial for all tasks.

The time to complete one search epoch with SuperNet and PIT is comparable to one training epoch of the reference DNN. For instance, on ICL, one PIT epoch is 1.8× slower than the

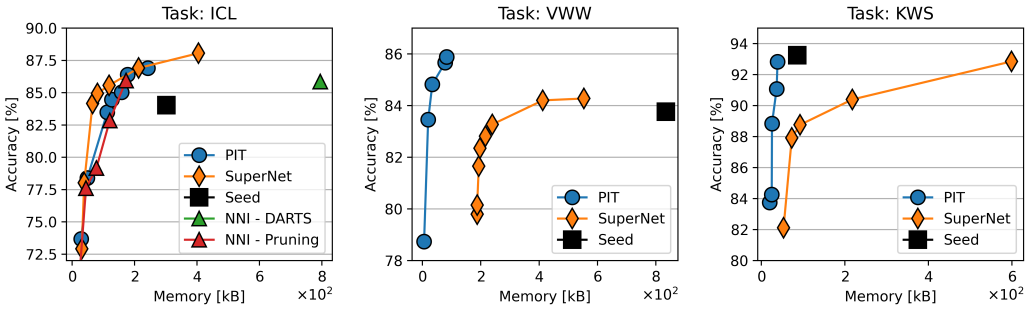

Fig. 8. Application of PIT and SuperNet algorithms from the PliNIO library to three benchmarks of the MLPerf Tiny Suite.

TABLE I
BEST ARCHITECTURES OBTAINED AT ISO-ACCURACY OR MAXIMIZING ACCURACY WITH ONE BETWEEN PIT OR SUPERNET.

Task	Model	Algorithm	Memory	MMACs	Accuracy	Memory-Reduction	Accuracy-improvement
ICL	Seed	None	302.19 kB	12.5M	84.03 %	n.a.	n.a.
	Iso-Accuracy	PIT	127.35 kB	7.17M	84.45 %	- 56.57 %	+ 0.42 %
		SuperNet	80.90 kB	5.28M	84.94 %	- 73.23 %	+ 0.91 %
	Max-Accuracy	PIT	242.58 kB	9.96M	86.89 %	- 19.72 %	+ 2.86 %
		SuperNet	405.08 kB	18.38M	88.05 %	+ 34.04 %	+ 4.02 %
VWW	Seed	None	843.33 kB	7.49M	83.76 %	n.a.	n.a.
	Iso-Accuracy	PIT	20.04 kB	1.74M	83.45 %	- 97.60 %	- 0.31 %
		SuperNet	239.69 kB	11.38M	83.28 %	-71.27 %	- 0.48 %
	Max-Accuracy	PIT	83.55 kB	3.3M	85. 88%	- 89. 99 %	+ 2.12 %
		SuperNet	553.52 kB	17.16M	84.27 %	- 33.66 %	+ 0.51 %
KWS	Seed	None	86.02 kB	2.66M	93.25 %	n.a.	n.a.
	Max-Accuracy	PIT	38.91 kB	1.03M	92.82 %	- 54.76 %	- 0.43 %
		SuperNet	598.79 kB	21.0M	92.84 %	+ 596 %	- 0.41 %

reference, whereas SuperNet takes 5× longer, due to replacing each layer with multiple alternatives.

The left-most graph of Fig. 8 also compares SuperNet and PIT with two similar approaches from NNI [33]. The comparisons are presented only on ICL for the sake of space. In particular, SuperNet is compared with NNI's *GumbelDARTS* method (green triangle). Despite exploring the same search-space, GumbelDARTS obtains a single optimized DNN, since it does not support complexity-driven search. The obtained point is strongly outperformed by PLiNIO's SuperNet, with 4× size reduction at Iso-Accuracy. From a training-time perspective, PLiNIO is also 11% faster than NNI. PIT, instead, is compared with NNI's *L1NormPruner*, which again supports the same search-space. The two obtained Pareto-curves are very similar, with PIT slightly outperforming NNI by up to +0.64% accuracy at Iso-Memory. Further, one key drawback of the NNI pruning is that it poorly supports complexity awareness, only allowing users to set the sparsity of individual layers (a knob difficult to control to achieve, for instance, a target model size) while not supporting other cost models.

C. Combination of NAS: SuperNet → PIT

Fig. 9 depicts the results obtained by sequentially applying the two NAS algorithms in PLiNIO. The rationale is to first select the optimal number and type of layers with SuperNet, then optimize each layer's hyper-parameters at fine-grain with PIT. In this case, the total cost is the sum of the cost of the two optimizations. We test this on the two benchmarks for which the Supernet approach had identified networks that either outperformed the seed in accuracy, or achieved Iso-Accuracy with a smaller model size, i.e., ICL and VWW. We obtain the green curves exporting the solutions found by SuperNet (line 10 of Fig. 1) to standard PyTorch networks and, then, applying PIT to the SuperNet results reported in

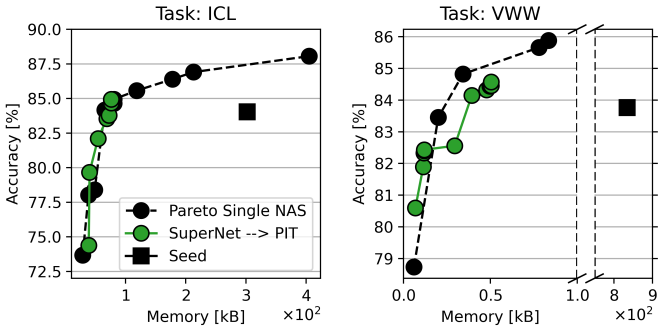

Fig. 9. Comparison between the application of a single NAS and the concatenation of SuperNet and PIT.

Table I. The black baseline curve is the combination of the two Pareto fronts of Fig. 8 obtained by applying PIT (blue curve) and SuperNet (orange curve) independently.

On both benchmarks, applying the two techniques in sequence results in small yet not negligible memory reductions. For instance, on ICL, we further reduce the memory usage by 4.5 kB for a solution matching the accuracy of the seed. Similarly, on VWW, we add four new models to the Pareto frontier in the 80% - 83% accuracy range. These limited improvements are primarily due to the already optimized nature of the seed models from MLPerf Tiny [16]. Starting from a non-optimized DNN would make the initial SuperNet step essential prior to the application of PIT, in order to avoid utilizing sub-optimal layers.

D. Full Pipeline: SuperNet → PIT → MPS

Fig. 10 illustrates the significant improvement achieved by applying MPS on top of the DNNs obtained in the previous sections by first exporting them to standard PyTorch models and then converting them with the `plinio.MPS()` auto-

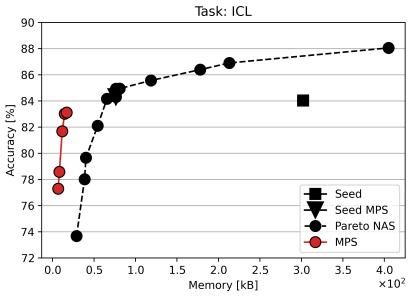

Fig. 10. Comparison between MPS networks and floating point ones.

TABLE II
OPTIMIZATION OF A SEED NEURAL NETWORK WITH THE THREE DESCRIBED NEURAL ARCHITECTURE SEARCH APPLIED IN SEQUENCE.

Model	Memory	MMACs	Accuracy
Seed	302.19 kB	12.5M (fp32)	84.03%
SuperNet	80.90 kB (- 73.23%)	5.28M (fp32)	84.94%
PIT	76.5 kB (- 74.68%)	5.03M (fp32)	84.93%
MPS	17.09 kB (- 94.34%)	5.03M (mixed-prec)	83.11%

conversion feature. We search between 8-bit, 4-bit, and 2-bit integer precision and use symmetric min-max quantization and PaCT for weights and activations respectively, as in [15]. We show results on ICL only, due to space constraints, although MPS could be applied to all other benchmarks as well. In the graph, the dashed black line is the global Pareto curve obtained using SuperNet, PIT, or their combination. The red curve contains the new points obtained by applying MPS to the smallest architecture that matches the seed accuracy (shown as a black triangle). Applying MPS to other Pareto-optimal architectures would be feasible too, albeit requiring more trainings. However, obtaining the complete best Pareto curve is not the focus of this work, which is rather to demonstrate the significant optimization potential unlocked by sequentially applying all PLiNIO optimizations.

The most accurate quantized architecture found with MPS reduces memory by 94.34% compared to the seed model, with a marginal accuracy drop of -0.92% (83.11% vs 84.03%). In comparison to the input of the MPS, it reduces memory usage by 77.66% while sacrificing 1.82% in accuracy. This DNN uses 8-bit quantization for all activation tensors and either 4-bit or 8-bit for weight tensors. Additionally, by further reducing the weights/activation precision of some layers, MPS identifies several additional Pareto points. One optimization epoch with MPS takes on average 4.3× longer compared to one reference DNN training epoch on this benchmark.

To summarize the results, Table II shows the gains obtained applying each PLiNIO optimization (SuperNet, PIT, and MPS) in sequence on ICL, considering the smallest network that outperforms the seed model (if any) or the one achieving the highest accuracy after the optimization.

VI. CONCLUSIONS

We presented PLiNIO, an open-source library for DNN inference optimization based on lightweight gradient-based complexity-aware techniques, including coarse- and fine-grained NAS, and MPS. PLiNIO exposes an extendable and user-friendly interface that allows users to rapidly apply and combine these optimizations to their specific use cases. With results on different benchmarks and DNN architectures, we have shown that PLiNIO's optimizations, combined, can generate rich Pareto-fronts in the accuracy vs memory-footprint space, reducing the size of a DNN by up to 94.34% at almost Iso-Accuracy (-0.92%) with respect to the baseline.

REFERENCES

[1] A. Burrello *et al.*, "Bioformers: Embedding Transformers for Ultra-Low Power sEMG-based Gesture Recognition," in *2022 DATE*.
[2] D. Liu *et al.*, "Bringing AI to edge: From deep learning's perspective," *Neurocomputing*, vol. 485, pp. 297–320, May 2022.
[3] F. Daghero *et al.*, "Energy-efficient deep learning inference on edge devices," in *Hardware Accelerator Systems for Artificial Intelligence and Machine Learning*. Elsevier, 2021, vol. 122, ch. 8, pp. 247–301.
[4] J. An *et al.*, "Chatgpt: tackle the growing carbon footprint of generative ai," *Nature*, vol. 615, no. 7953, pp. 586–586, 2023.
[5] X. Dong and Y. Yang, "Nas-bench-201: Extending the scope of reproducible neural architecture search," in *ICLR 2020*.
[6] C. White *et al.*, "Neural architecture search: Insights from 1000 papers," *arXiv preprint, arXiv:2301.08727*, 2023.
[7] X. He *et al.*, "Automl: A survey of the state-of-the-art," *Knowledge-Based Systems*, vol. 212, p. 106622, 2021.
[8] K. Wang *et al.*, "HAQ: Hardware-Aware Automated Quantization With Mixed Precision," in *Proc. IEEE/CVF CVPR*, 2019.
[9] H. Liu *et al.*, "Darts: Differentiable architecture search," in *ICLR*, 2018.
[10] Edge Impulse, "EON Tuner," https://docs.edgeimpulse.com/docs/edge-impulse-studio/eon-tuner, April 26th, 2023.
[11] H. Jin *et al.*, "Autokeras: An automl library for deep learning," *JMLR*, vol. 24, no. 6, pp. 1–6, 2023.
[12] M. Risso *et al.*, "Lightweight neural architecture search for temporal convolutional networks at the edge," *IEEE Trans. Comp.*, 2023.
[13] Z. Cai and N. Vasconcelos, "Rethinking differentiable search for mixed-precision neural networks," in *CVPR*, 2020.
[14] J. Choi *et al.*, "PACT: Parameterized Clipping Activation for Quantized Neural Networks," *CoRR*, vol. abs/1805.0, 2018.
[15] M. Risso *et al.*, "Channel-wise Mixed-precision Assignment for DNN Inference on Constrained Edge Nodes," in *IEEE IGSC*, 2022.
[16] C. Banbury *et al.*, "Mlperf tiny benchmark," in *NeurIPS*, 2021.
[17] H. Cai *et al.*, "ProxylessNAS: Direct neural architecture search on target task and hardware," in *ICLR*, 2019.
[18] E. Liberis *et al.*, "μNAS: Constrained Neural Architecture Search for Microcontrollers," in *Proc. Workshop on ML and Sys*. ACM, 2021.
[19] S. Xie *et al.*, "Snas: stochastic neural architecture search," in *ICLR*, 2018.
[20] H. Cai *et al.*, "Enable deep learning on mobile devices: Methods, systems, and applications," *ACM TODAES*, vol. 27, no. 3, mar 2022.
[21] M. Risso *et al.*, "Multi-complexity-loss dnas for energy-efficient and memory-constrained deep neural networks," in *ISLPED*, 2022.
[22] N. Nayman *et al.*, "HardCoRe-NAS: Hard Constrained diffeRentiable Neural Architecture Search," in *ICML*, Jul. 2021, pp. 7979–7990.
[23] A. Gordon *et al.*, "Morphnet: Fast & simple resource-constrained structure learning of deep networks," in *Proc. IEEE CVPR*, 2018.
[24] A. Wan *et al.*, "Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions," in *Proc. IEEE/CVF CVPR*, 2020.
[25] B. Jacob *et al.*, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," in *CVPR*, 2018.
[26] R. Banner *et al.*, "Post training 4-bit quantization of convolutional networks for rapid-deployment," *NIPS*, vol. 32, 2019.
[27] Z. Dong *et al.*, "HAWQ: Hessian AWare Quantization of Neural Networks With Mixed-Precision," in *IEEE/CVF ICCV*, 2019.
[28] N. P. Pandey *et al.*, "A Practical Mixed Precision Algorithm for Post-Training Quantization," Feb. 2023, arXiv:2302.05397 [cs].
[29] TDK Qeexo, "Qeexo AutoML," https://qeexo.tdk.com, May 08th, 2023.
[30] TensorFlow, "TensorFlow Model Optimization Toolkit," https://tensorflow.org/model_optimization, May 08th, 2023.
[31] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," 2019.
[32] Qualcomm, "AI Model Efficiency Toolkit," https://developer.qualcomm.com/software/ai-model-efficiency-toolkit, May 08th, 2023.
[33] Microsoft, "Neural Network Intelligence," 1 2021. [Online]. Available: https://github.com/microsoft/nni
[34] B. Wang *et al.*, "Vega: Towards an end-to-end configurable automl pipeline," 2020.
[35] M. Risso *et al.*, "Precision-aware latency and energy balancing on multi-accelerator platforms for dnn inference," *arXiv:2306.05060*, 2023.