Learning to infer: RL-based search for DNN primitive selection on Heterogeneous Embedded Systems

Miguel de Prado[1,2], Nuria Pazos[1] and Luca Benini[2]

[1] He-Arc Ingenierie, HES-SO. {miguel.deprado, nuria.pazos}@he-arc.ch

[2] Integrated Systems Laboratory, ETH Zurich. lbenini@iis.ee.ethz.ch

Abstract— Deep Learning is increasingly being adopted by industry for computer vision applications running on embedded devices. While Convolutional Neural Networks' accuracy has achieved a mature and remarkable state, inference latency and throughput are a major concern especially when targeting low-cost and low-power embedded platforms. CNNs' inference latency may become a bottleneck for Deep Learning adoption by industry, as it is a crucial specification for many real-time processes. Furthermore, deployment of CNNs across heterogeneous platforms presents major compatibility issues due to vendor-specific technology and acceleration libraries.

In this work, we present QS-DNN, a fully automatic search based on Reinforcement Learning which, combined with an inference engine optimizer, efficiently explores through the design space and empirically finds the optimal combinations of libraries and primitives to speed up the inference of CNNs on heterogeneous embedded devices. We show that, an optimized combination can achieve 45x speedup in inference latency on CPU compared to a dependency-free baseline and 2x on average on GPGPU compared to the best vendor library. Further, we demonstrate that, the quality of results and time "to-solution" is much better than with Random Search and achieves up to 15x better results for a short-time search.

I. INTRODUCTION

Artificial Intelligence (AI) is rapidly growing and will soon become ubiquitous and pervade our daily life. In particular, Deep Learning (DL) has rapidly grown in the last years achieving remarkable results in computer vision [3] and speech recognition [4]. Adoption of AI by major industrial partners, e.g. Google [1], Tesla [2], is already a reality and its wide-range applications are to bring on a new technological revolution.

Convolutional Neural Networks (CNN) are one of the most successful examples of Deep Learning due to their remarkable accuracy and flexibility to many applications. CNNs are capable of learning abstract features by stacking many layers in parallel and in depth, which turns them into complex architectures. Training of CNNs has drawn great attention in the last years towards building more and more competitive and accurate architectures and surpassing human capabilities, e.g. ImageNet competition [5].

Deployment of CNNs is not a trivial problem. However, it has not been on the focus until recently ago. The inference time, latency of the forward pass of a network, has become one of the main issues for the industrial stakeholders who would like to take up AI solutions for edge applications. Inference time represents a bottleneck in IoT or embedded devices due to the restricted resources they have and the large computational requirements [6].

Moreover, deployment of CNNs on embedded devices presents further difficulties due to the restrictions and dependencies that the wide variety of implementations may impose in terms of frameworks e.g. Caffe [7], Tensorflow [8], acceleration libraries, e.g. cuDNN [9], ArmCL [10] or heterogeneous embedded platform types, e.g. CPU [11], FPGA [12], GPU [13]. Each layer of a network may be executed by many possible libraries (and primitives from the library), or even in different processor, giving out quite a different performance. Hence, the space of approaches for CNN deployment becomes too large to test and obtain an optimal implementation [14], which usually results in the stakeholders selecting a single good-performing library.

The objective of this work is to ease the deployment of CNNs for industrial applications on a wide range of embedded platforms and automatically search for the best primitive combination to speed up the performance. To fulfill this objective, we present QS-DNN, an automatic exploration framework, which relies on a design space search based on Reinforcement Learning [30]. The RL-based search efficiently explores through the design space and finds an optimized combination of primitives that can be used to execute inference for a target CNN on a given platform. The search is combined with an inference engine optimizer which enables the production and deployment of CNN implementations for heterogeneous platforms. Thereby, we are able to obtain an optimized implementation by directly acquiring empirical measurements on embedded AI applications and notably boosting the performance of the process.

We demonstrate the effectiveness of the method by applying it to several types of CNNs for image classification, face recognition and object detection on a heterogeneous platform. On average, we achieve 2x speedup in inference latency in the ImageNet benchmark, compared to the best vendor library on a GPGPU platform. The runtime of our RL-based optimized is also very reasonable: 5 minutes are sufficient to find solutions that consistently outperform those found by Random Search with the same time budget.

The paper is organized as follows: in Section 2, the State-of-the-Art is presented. Section 3 describes the inference of a DNN on heterogeneous devices and the inference engine optimizer. In Section 4, we address the problem of primitive selection and describe Reinforcement Learning. In Section 5, we introduce the RL-based search engine and the methodology of the experiments. Section 6 presents the results and discussion.

II. RELATED WORK

We find two main topics related to this work: Auto-tuning and Machine Learning for Design Space Exploration.

Auto-tuning. The massive computation that CNNs demand prompts for several optimization approaches for inference on embedded devices. We can categorize two main classes: *i)* computational graph engines and *ii)* acceleration libraries for specific layers. Computation graph engines reduce execution time and memory footprint by removing overhead dependencies, fusing pipelined operations and performing cross-layer optimizations [8]. Moskewicz et al. [24] use meta-programming and auto-tuning to provide portable implementations across different GPU-vendor platforms. However, their auto-tuning process is inefficiently done as they use brute force to search through the design space. Truong et al. [25] implemented Latte, a domain-specific language that abstracts the architecture and computation of a neural network. Latte's compiler is able to recognize dependencies and match patterns to perform cross-layer optimization as well as optimized-primitive calls.

In this work, we rather focus on the second approach: acceleration libraries for specific layers. We leverage primitives from acceleration libraries to speed up the performance of standard neural network layers. We draw inspiration from Anderson et al. [14] who use PBQP to optimize inference time by selecting suitable backends. In their work, they profile each implementation type and the transformation cost between different implementations. They make an optimization problem to select the best backend per layer which they solved with PBQP. However, they only profile convolutional layers and do not optimize any other layer type. In addition, we propose a totally different search method which is modeled as a learning problem and implements a sample-based approach. Our method drastically reduces the space exploration effort, while still obtaining an optimized solution.

Machine Learning (ML) for Design Space Exploration (DSE). General ML techniques have been applied as an automatic-search tool for large space exploration problems such as performance of processors [15] or high-level synthesis [16]. Lately, there has been an increasing trend of using Reinforcement Learning (RL) and Evolutionary Algorithms (EA) to build CNN architectures. EA works like [17], [18], [19] use Genetic Algorithms over a population of CNNs. By using mutation operators, the architectures of the population evolve towards new topologies. Baker et al. [29] used RL to sequentially choose CNN layers. They used Q-learning employing an ϵ-greedy strategy, which trades off exploration and exploitation. All these works share the assumption of fixed-sized number of parameters to select from (RL) or from which they can mutate (EA). However, they only take into account the accuracy of the CNN without any consideration for embedded deployment or inference time.

Recent works like [20], [21], [22], [23] use a multi-objective or joint reward function to reduce power consumption and/or inference time besides improving accuracy. NetAdapt [35] proposes an iterative process to compress a pre-trained CNN by reducing the number of channels and employing empirical measurements on a target platform. He et al. [36] employ AutoML for model compression by having an actor-critic agent learn the compression policy of a network with latency and accuracy as reward. Each agent's action represents the desired compression rate and structure.

Overall, all the works employing ML for DSE are bound to a specific platform and do not offer support for a wide range of heterogeneous platforms. Besides, they address the problem of improving DNN architectures or compression, but do not give any attention to primitive selection optimization. Further, our method is complementary to those implementing graph optimizations as a final-processing step for them. To the best of our knowledge, we are the first ones to apply an RL-based search for primitive selection optimization on multiple target platforms.

III. BACKGROUND: INFERENCE OF DNN ON HETEROGENEOUS EMBEDDED DEVICES

Deep Neural Networks (DNN) are composed of a set of layers in cascade, e.g. convolution, pooling, activation and fully connected, that transform an input into a set of features maps which can be classified, detected or recognized based on a score or probability function. Training of DNN involves both a forward pass to compute the final score function and a backward pass to learn the weights according to a loss function. In this work, we address the problem of improving inference efficiency, that is, the forward pass latency of a DNN after training, and its deployment for industrial applications on heterogeneous embedded devices.

A. INFERENCE ENGINE OPTIMIZER

We form part of an European collaboration to bring Deep Learning to any party who would like to take up DL solutions in an industrial environment [38]. One of the main goals of the project is to reduce development time and to optimize deployment of DNN on embedded systems. In this context, a neural network framework has been developed to produce efficient and tunable code which enables and maximizes the portability among heterogeneous platforms [37]. The core of the inference engine optimizer comprises a set of CPU dependency-free functions which can be complemented by specific-platform acceleration libraries to generate optimized implementation for the system. In this work, we address the integration of the inference engine optimizer into our search environment to tightly couple empirical measurements of a heterogeneous platform to a learning-based search.

B. ACCELERATION LIBRARIES

We present the set of libraries and primitives available in the inference engine optimizer for DNNs:

- **Vanilla:** This group embraces the set of CPU dependency-free and direct functions implemented in ANSI C with the objective of maximizing portability. It does not rely on any acceleration library.

- **Basic Linear Algebra Subprograms (BLAS):** This group includes ATLAS and openBLAS libraries which implement GEMM and GEMV routines [28] on CPU cores. Any of these libraries can use the following lowering methods: im2col, im2row and kn2row.
- **NNPACK:** It is an open-source acceleration library which provides low-level performance primitives on CPU cores for specific DL layers [26].
- **ArmCL:** Set of high-performance routines for Arm processors. We have used Winograd transformation and BLAS routines for convolutional layers and specific-optimized code for Depth-Wise convolutions [10].
- **Sparse:** It includes multiple implementations which can be used to compress the model representation in memory for convolutional and FC layers.
- **cuDNN:** Highly optimized primitives for Nvidia GPUs which implement several DNN routines [9]. It is important to remark that this library does not include a specific implementation for FC layer.
- **cuBLAS:** BLAS routines for Nvidia GPUs [27]. We have only used the GEMV routine for FC layer.

IV. LEARNING-BASED SEARCH ENGINE

In this section, we address the problem of primitive selection and we propose Reinforcement Learning as a solution.

A. PROBLEM FORMULATION

Given a DNN, each layer may be executed by different acceleration libraries which, in turn, might provide several primitives to yield an optimal implementation. The problem is not as trivial as to benchmark all primitives individually and select the fastest for each layer to make up the optimal network implementation. Each primitive may have a different input or output tensor layout which might not correspond to those layouts of previous and following layers, e.g. NCWH and WHNC. Therefore, incompatibilities arise and a layout conversion layer is needed which incurs in a penalty. Likewise, in an heterogeneous environment, layers can be executed in different processor types which involves a costly (slow) memory transfer, see Fig. 1.

The number of combinations within a network, which is the design space to explore, grows exponentially with the number of layers, N_L, having as base the number of different implementations for such layer, N_I. Hence, the design space size for a network would be $N_I^{N_L}$ as the worst case. It becomes a non trivial problem and therefore, a careful search must be carried out to select the right set of primitives that, combined among them and assuming the conversion penalties, yields the fastest inference.

B. REINFORCEMENT LEARNING

Reinforcement Learning (RL) lends itself perfectly to exploring large design spaces due to its sample-based approach and far-sighted accumulative reward [30], [31]. Consider the network space exploration as a Markov Decision Process (MDP) containing an agent. We are interested in learning a function that optimizes the agent's behavior i.e. mapping
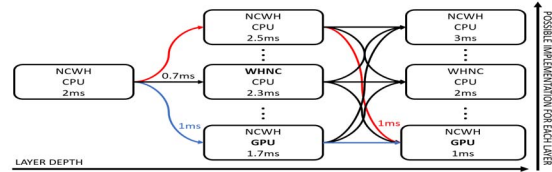

Fig. 1: 3-layer network. Arrows with time express incompatibility penalty. The agent is able to avoid local minimum, e.g. red path, which contains the fastest intermediate implementation. Instead, it selects the blue path: fastest overall.

from state s_t to actions a_t without modeling the environment and only relying on the reward function. Q-learning [32] fits well this description as it is a model-free and value-based implementation, having the policy implicit in the value function. The action-value function is the expected return in a state s_t taking an action a_t:

$$q_\pi(s, a) = E_\pi[G_t|s_t = s, a_t = a] \tag{1}$$

The objective of Q-learning is to maximize the total reward: $R_T = \sum_{t=0}^{\infty} \gamma^t r_t$ where r_t is an individual reward and γ is the discounted factor for successive states. Besides, Q-learning is an off-policy implementation, that is, it may follow a behavior policy $\mathcal{A}$ while targeting a greedy policy $\mathcal{B}$. Following Bellman's equation, we can iteratively update the action-value function as follows:

$$Q(s_t, a_t) = Q_{s_t, a_t}(1-\alpha) + \alpha \left[r_t + \gamma \max_a Q(s_{t+1}, a) \right] \tag{2}$$

C. SEARCH ENGINE

We consider an agent whose aim is to learn the optimal path among a large but finite set of states $\mathcal{S}$ i.e. layer representations, employing a set of actions $\mathcal{A}$ i.e. layer implementations. RL suits well the specifications of the problem that we address in this work. Inference time represents a clear reward function given by the environment that we aim to explore: a Deep Neural Network.

The agent samples sequentially a new set of primitives for the network, layer by layer. The state space is defined as a tuple of the parameters that specify the execution of a layer with a certain primitive on a target platform, see Table I. All primitives are defined by an algorithm, its implementation format and a BLAS library. The agent chooses one primitive from the set of acceleration libraries given the current layer type. Based on the action, the agent moves to another state and the process is repeated until the end of the network.

State Parameters	Definition
Layer type	Any layer e.g. convolution, pooling
Layer depth	Position of the layer in the network
Acceleration Library	Name of the library
Algorithm	Routine type
Algorithm impl	Sub-routine or lowering method
Hardware processor	CPU, GPU, FPGA.
BLAS library	Library name

TABLE I: State Space. Parameters define the execution of a layer with a specific primitive on a target platform.

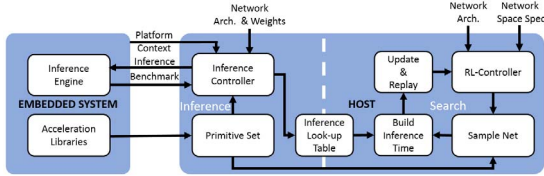*Design, Automation And Test in Europe (DATE 2019)*

Fig. 2: Architecture of QS-DNN. Complete flow: Inference on an embedded on the left, RL-based learning on the right.
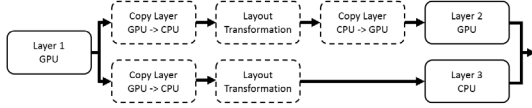

Fig. 3: Profiling of compatibility layers between all consecutive layers. Exception and branches are handled.

Similar to Baker et al. [29], we have implemented an ϵ-greedy strategy [33] which trades off between exploitation and exploration. The agent starts mainly exploring the design space (random actions) to sample the diverse possibilities ($\epsilon = 1$). We slowly decrease ϵ over the episodes for the agent to select the best actions and finally learn an optimal path: full exploitation ($\epsilon = 0$). In addition, we have added an experience replay after each episode which helps the action-value function converge faster [34]. We have set the experience replay's buffer size to 128 following [29].

Although initially we used the network inference time as unique reward signal, we have applied Reward Shaping for better convergence. The objective is to maximize the total reward, in this case, minimize the inference time. Hence, each state receives as reward its own layer inference time but reversing the sign, e.g. 0.01ms $\Rightarrow$ -0.01ms. Thanks to the Q-learning update rule, each layer also receives Q-knowledge from the best following state. Therefore, the agent is able to combine both sources of knowledge, look ahead and avoid local minima due to penalties introduced by incompatibility between layers, see Fig. 1.

V. Q-BASED SEARCH FOR DEEP NEURAL NETWORKS (QS-DNN)

The aim of QS-DNN is to automatically optimize the inference of any DNN and boost its performance on an embedded system. The process is composed of two phases: *1)* inference of the DNN on the embedded system to obtain empirical measurements, *2)* automatic RL-based search over a reduced number of episodes to explore the design space. We have separated the phases to avoid inferring on the embedded system each possible solution of the space search, which would slow down remarkably the process, see Fig. 2.

A. INFERENCE

We employ the inference engine optimizer described in Section III and its acceleration libraries to obtain real measurements although the search could be also applied to any other inference framework. We consider Vanilla library as

---

Algorithm 1 QS-DNN - Search
1: $\epsilon \leftarrow \epsilon_{new}$
2: **while** Learned Episodes < Episodes(ϵ) **do**
3:     Reset Path
4:     **while** Layer $\neq$ End Layer(ϵ) **do**
5:         **if** Generate Random < ϵ **then**
6:             Action $\leftarrow$ Q-values(Random)
7:         **else**
8:             Action $\leftarrow$ Q-values(Max)
9:         Layer $\leftarrow$ Next Layer
10:    Check for Incompatibility
11:    Compute Inference Time
12:    Experience Replay & Update (eq. 2)

the base implementations for all measurements since it is the most simple, direct, dependency-free and it contains all layers that a DNN may use.

Having set the base, the inference controller benchmarks each primitive type[1], one at a time, by substituting Vanilla for the chosen primitive type in all those layers where the acceleration library is able to implement such primitive. Therefore, we only need to infer the whole network on the embedded platform as many times as different global implementations there exists. In each inference, the execution time for each layer is measured and retrieved.

Once all primitive types have been benchmarked, we profile the compatibility layers for layout transformation and data transfers between different processor. A single inference is performed to benchmark all possible compatibility layers between each consecutive layer of the neural network, see Fig.3. After all inference measurements have been retrieved, a look-up table is built.

B. SEARCH

The search space and the conditions of the search can be defined for each network. They specify the behavior of the agent: number of episodes for each ϵ, learning rate, discounted factor and replay buffer's size. We have set the learning rate to 0.05 and discounted factor to 0.9 to give slightly more importance to short-term rewards. Once the inference phase has finished, the Q-learning -based search begins and proceeds as shown in Algorithm 1.

First, ϵ is retrieved from the specifications as well as the number of episodes for such ϵ. In all experiments, 50% of the total episodes correspond to full exploration and 5% to any other ϵ from 0.9 to 0.1. By these means, the agent obtains enough knowledge from the environment before starting exploitation, see Fig. 4.

For each episode, the agent samples sequentially a new set of primitives based on the $\epsilon-$strategy. Once the network's configuration is set, the engine automatically looks for incompatibilities between layers due to layout and processor type. At last, the total network inference time is computed by looking up each implementation in the inference table

---

[1]Each primitive is inferred for 50 images and the mean is calculated

Processor	Method	LeNet-5	AlexNet	SqueezeNet	MobileNet v1	MobileNet v2	GoogleNet	Resnet32	VGG19	Mobile-FaceNet	MobileNet v1 SSD
CPU	Vanilla					x1					
	OpenBLAS	7.69x	13.45x	18.55x	16.32x	9.54x	14.79x	24.95x	26.79x	11.38x	21.14x
	NNPACK	8.41x	8.82x	17.53x	12.61x	8.30x	15.63x	19.76x	41.36x	9.66x	15.81x
	ArmCL	6.27x	13.38x	18.50x	17.31x	10.58x	14.37x	25.01x	25.45x	12.61x	21.58x
	RS	11.31x	12.84x	12.94x	9.12x	5.26x	8.92x	13.71x	33.14x	8.43x	18.72x
	QS-DNN	13.02x	17.46x	21.48x	17.89x	11.25x	17.53x	31.06x	55.15x	13.33x	22.34x
	QS-DNN VS BSL	1.55x	1.30x	1.16x	1.03x	1.06x	1.12x	1.24x	1.33x	1.06x	1.04x
GPGPU	cuDNN	4.06x	18.88x	95.14x	26.39x	12.23x	138.61x	98.36x	236.96x	14.43x	40.03x
	RS	11.89x	22.33x	19.11x	18.51x	6.74x	11.19x	33.10x	54.60x	11.38x	26.73x
	QS-DNN	13.02x	154.7x	95.14x	37.59x	16.55x	166.33x	133.07	714.46x	20.59x	48.46x
	QS-DNN VS BSL	3.21x	8.19x	1.00x	1.42x	1.35x	1.20x	1.35x	3.02x	1.43x	1.21x

TABLE II: Inference time speedup of CPU- and GPGPU-based implementations respect to Vanilla (dependency-free implementation). Results correspond to most performing libraries employing their fastest primitive for single-thread and 32-bit floating-point operations. QS-DNN VS BSL shows the improvement of the search over the Best Single Library (BSL) and clearly outperforms RS (Random Search) for 1000 episodes.
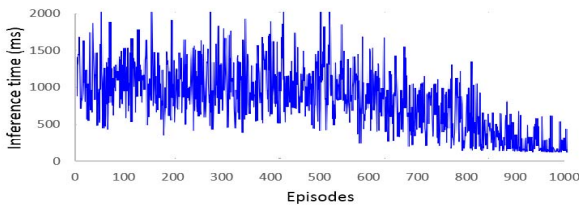

Fig. 4: RL search for 1000 episodes where the 500 first episodes are fully exploration. From there on, ϵ is decreased by 0.1 towards exploitation after every 50 episodes.

and summing up the execution time of all layers. If any incompatibility has been found between two layers, the extra penalty is added to the inference time of the latter layer. Finally, the action-value function is updated with the current reward and stored for experience replay. When the number of episodes for a given ϵ has been met, ϵ is decreased towards exploitation phase. By the end of the search, the engine gives out the best inference configuration and the learning curve that the agent has followed, see Fig 4.

VI. RESULTS AND DISCUSSION

In this section, we show the results from applying QS-DNN to several DNNs for image classification, face recognition and object detection tasks.

A. INFERENCE OPTIMIZATION

All inference experiments have been conducted on the heterogeneous platform Nvidia Jetson TX-2 using single-precision floating-point operations. All CPU inferences correspond to using a single-thread on an ARM Cortex A-57 core, while GPGPU inferences correspond to using either the single-thread CPU or the Nvidia Pascal GPU which features 256 cores. The design space search is carried out in a standard Intel CPU and takes less than 10 min. to converge.

Given the acceleration libraries from Section III, the maximum number of different primitive for a layer, taking all the variants, is 13. Table II summarizes the results of the most performing implementations. It is possible to observe that,

QS-DNN outperforms all single-library implementations and achieves considerable speedups compared to the Best Single Library (BSL) for CPU and GPGPU modes.

It is interesting to note that the fastest implementation for Lenet-5 in GPGPU mode is actually a pure CPU implementation. In this case, the agent learns that, despite GPU implementation is faster for some layers, data transfers between CPU and GPU diminish the speedup that GPU yields. It is also possible to note a great improvement of QS-DNN (GPGPU) over cuDNN in VGG19 or AlexNet as cuDNN does not implement the costly FC layer of these networks. In particular, QS-DNN (GPGPU) achieves a notable speedup for MobileNets (over 1.4x) where it learns to combine the optimized Depth-Wise code from ArmCL, convolutions from cuDNN and certain ReLU and B-Norm layers from Vanilla to avoid costly extra copies to GPU.

B. REINFORCEMENT LEARNING VS RANDOM SEARCH

In this section, we address the learning process of RL and compare it to Random Search (RS). RL outperforms RS in all networks and achieves speedups of up to x15 over RS for larger design spaces, e.g. GoogleNet or VGG19, on GPGPU mode, see Table II.
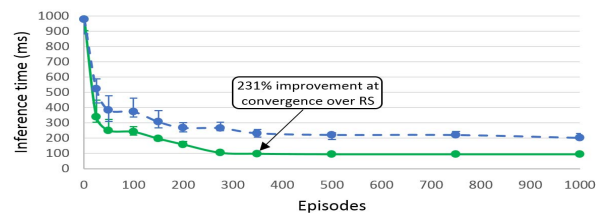

Fig. 5: RL VS RS for Mobilenet. Each point indicates the average result for a complete search for the given episodes. Variance reduces towards the end as the search converges.

Fig. 5 gives an example of RL VS RS for MobileNet-v1 where each point represents the mean inference time from 5 full experiments for a reduced budget: number of episodes. With a budget of a few episodes, the variance of

both implementation is high as they do not obtain much knowledge about the environment. RL's solutions quickly decrease inference time as the agent observes more episodes and it falls near convergence after only 350. On the other hand, RS fails to find implementations as optimized as RL's since it does not implement any learning method. RS's solutions are already 50% worse than RL's with only 25 episodes and twice as worse after 350 episodes. RS's implementations decrease inference time after seeing more options as it discards naive implementations, but it only converges towards the infinite.

VII. CONCLUSIONS AND FUTURE WORK

We have presented an automatic exploration framework which relies on a design space search based on Reinforcement Learning (RL). The RL-based search efficiently learns an optimized combination of primitives to tune and boost the inference of DNNs. The search is tightly coupled with an inference engine optimizer which facilitates the deployment and optimization of DNNs on heterogeneous embedded platforms. We have shown that, the search, together with the inference engine optimizer, is able to achieve 2x speedup on average in inference latency compared to the best single vendor library in a GPGPU platform. Further, the RL-based search quickly converges and outperforms Random Search achieving up to 15x better results in large design spaces. In addition, our approach is very modular and can be applied to other optimization methods as a post-processing step.

We aim to extend this work to other heterogeneous target platforms, e.g FPGA, VPU or ASIC[2]. In addition, we envision to extend exploration to e.g. different reward choices or having multi-objective search, for problems related to inference of DNNs on constrained environments. Further, we also aim to look into Deep RL to approximate the value function for better scalability towards larger networks and more dimensions in the search space.

ACKNOWLEDGMENT

REFERENCES

[1] Google AI. URL: https://ai.google/
[2] Tesla. URL: https://www.forbes.com/sites/bernardmarr/2018/01/08/the-amazing-ways-tesla-is-using-artificial-intelligence-and-big-data
[3] Rowley, Henry A., Shumeet Baluja, and Takeo Kanade. "Neural network-based face detection." IEEE Transactions on pattern analysis and machine intelligence 20.1 (1998): 23-38.
[4] Graves, Alex, Abdel-rahman Mohamed, and Geoffrey Hinton. "Speech recognition with deep recurrent neural networks." Acoustics, speech and signal processing (icassp), 2013 ieee international conference on. IEEE, 2013.
[5] Imagenet. URL: http://www.image-net.org.
[6] Shafique, Muhammad, et al. "An overview of next-generation architectures for machine learning: Roadmap, opportunities and challenges in the IoT era." Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018. IEEE, 2018.
[7] Jia, Yangqing, et al. "Caffe: Convolutional architecture for fast feature embedding." Proceedings of the 22nd ACM international conference on Multimedia. ACM, 2014.
[8] Abadi, Martn, et al. "Tensorflow: a system for large-scale machine learning." OSDI. Vol. 16. 2016.
[9] Chetlur, Sharan, et al. "cudnn: Efficient primitives for deep learning." arXiv preprint arXiv:1410.0759 (2014).
[10] Arm Compute Library. URL: https://developer.arm.com-technologies/compute-library
[11] Qualcomm. URL: https://www.qualcomm.com/snapdragon
[12] Xilinx. URL: https://www.xilinx.com/
[13] https://www.nvidia.com/en-us/
[14] Anderson, Andrew, and David Gregg. "Optimal DNN primitive selection with partitioned boolean quadratic programming." arXiv preprint arXiv:1710.01079 (2017).
[15] Ozisikyilmaz, Berkin, Gokhan Memik, and Alok Choudhary. "Efficient system design space exploration using machine learning techniques." Proceedings of the 45th annual design automation conference. ACM.
[16] Liu, Hung-Yi, and Luca P. Carloni. "On learning-based methods for design-space exploration with high-level synthesis." Proceedings of the 50th annual design automation conference. ACM, 2013.
[17] Real, Esteban, et al. "Regularized evolution for image classifier architecture search." arXiv preprint arXiv:1802.01548 (2018).
[18] Cortes, Corinna, et al. "Adanet: Adaptive structural learning of artificial neural networks." arXiv preprint arXiv:1607.01097 (2016).
[19] Al-Hyari, Abeer, and Shawki Areibi. "Design space exploration of Convolutional Neural Networks based on Evolutionary Algorithms." Journal of Computational Vision and Imaging Systems 3.1 (2017).
[20] Hsu, Chi-Hung, et al. "MONAS: Multi-Objective Neural Architecture Search using Reinforcement Learning." arXiv preprint arXiv:1806.10332 (2018).
[21] Tan, Mingxing, et al. "MnasNet: Platform-Aware Neural Architecture Search for Mobile." arXiv preprint arXiv:1807.11626 (2018).
[22] Dong, Jin-Dong, et al. "PPP-Net: Platform-aware Progressive Search for Pareto-optimal Neural Architectures." (2018).
[23] Kim, Ye-Hoon, et al. "Nemo: Neuro-evolution with multiobjective optimization of deep neural network for speed and accuracy." ICML.
[24] Moskewicz, Matthew W., Ali Jannesari, and Kurt Keutzer. "Boda: A Holistic Approach for Implementing Neural Network Computations." Proceedings of the Computing Frontiers Conference. ACM, 2017.
[25] Truong, Leonard, et al. "Latte: a language, compiler, and runtime for elegant and efficient deep neural networks." ACM SIGPLAN Notices 51.6 (2016): 209-223.
[26] NNPACK. URL: https://github.com/Maratyszcza/NNPACK
[27] cuBLAS. URL: https://docs.nvidia.com/cuda/cublas/index.html
[28] BLAS: URL: http://www.netlib.org/blas/
[29] Baker, Bowen, et al. "Designing neural network architectures using reinforcement learning." arXiv preprint arXiv:1611.02167 (2016).
[30] Li, Yuxi. "Deep reinforcement learning: An overview." arXiv preprint arXiv:1701.07274 (2017).
[31] Sutton, Richard S., and Andrew G. Barto. Introduction to reinforcement learning. Vol. 135. Cambridge: MIT press, 1998.
[32] Watkins, Christopher John Cornish Hellaby. Learning from delayed rewards. Diss. King's College, Cambridge, 1989.
[33] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529.
[34] Lin, Long-Ji. "Self-improving reactive agents based on reinforcement learning, planning and teaching." Machine learning 8.3-4 (1992).
[35] Yang, Tien-Ju, et al. "NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications." arXiv preprint arXiv:1804.03230
[36] He, Yihui, and Song Han. "ADC: Automated Deep Compression and Acceleration with Reinforcement Learning." arXiv preprint arXiv:1802.03494 (2018).
[37] de Prado, Miguel, et al. "QUENN: QUantization engine for low-power neural networks." Proceedings of the 15th ACM International Conference on Computing Frontiers. ACM, 2018.
[38] Llewellyn, Tim, et al. "BONSEYES: platform for open development of systems of artificial intelligence." Proceedings of the Computing Frontiers Conference. ACM, 2017.

[2]If API at network-layer level is provided e.g. Convolution