TinderAI: Support System for Matching AI Algorithms and Embedded Devices

Matteo Francobaldi, Allegra De Filippo, Andrea Borghesi
DISI, University of Bologna

Nikola Pižurica, Igor Jovančević, Tim Llewellynn, Miguel de Prado
BCA, Faculty of Natural Sciences and Mathematics, University of Montenegro

Abstract

Artificial Intelligence (AI) is becoming increasingly important and pervasive in the modern world. The widespread adoption of AI algorithms is reflected in the extensive range of HW devices on which they can be deployed, from high-performance computing nodes to low-power embedded devices. Given the large set of heterogeneous resources where AI algorithms can be deployed, finding the most suitable device and its configuration is challenging, even for experts.

We propose a data-driven approach to assist AI adopters and developers in choosing the optimal HW resource. Our approach is based on three key elements: i) fair benchmarking of target AI algorithms on a set of heterogeneous platforms, ii) creation of ML models to learn the behaviour of these AI algorithms, and iii) support guidelines to help identify the best deployment option for a given AI algorithm. We demonstrate our approach on a specific (and relevant) use case: Deep Neural Network (DNN) inference on embedded devices.

Keywords: Vertical Matchmaking, Machine Learning, Benchmarking

Introduction

AI has made huge strides in recent years, and its influence over society appears poised to further growth in the future (Mariani, Machado, and Nambisan 2023). However, several challenges need to be addressed for this growth to continue. Among the most crucial ones is the problem of finding the most suitable hardware (HW) resources for AI algorithms in terms of HW type, deployment configuration, memory size, and latency (Talib et al. 2021). This problem is referred to as *vertical matching* (De Filippo et al. 2022).

In this paper, we focus explicitly on a class of AI algorithms whose usage has been steadily increasing in the last several years: DNNs inference. The deployment of DNNs for a specific problem occurs after the successful training of a NN and can take place on a variety of HW platforms. A significant concern for AI engineers during the deployment of AI Algorithms is maximizing the system's performance in terms of overall latency, quality of solution, power, and space. Since inference operations make up most of the DNNs' lifetime (Canziani, Paszke, and Culurciello 2016), finding the most efficient HW platform to be run on is of paramount importance.

Copyright © 2022 by the authors. All rights reserved.

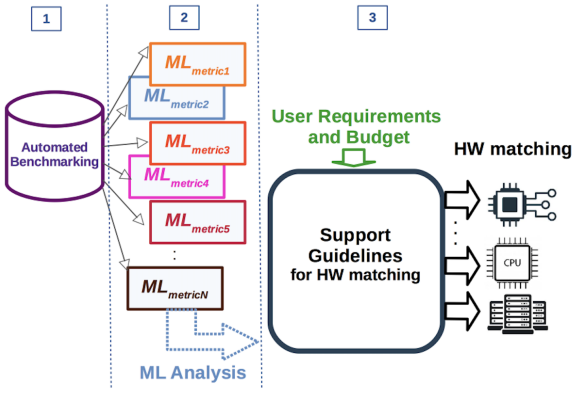

Figure 1: Schema of AI Matching Methodology

There exist multiple deployment frameworks such as TensorFlow Lite (Farhoodfar 2019), NVIDIA TensorRT (Shafi et al. 2021), Arm Compute Library (ArmCL) (Sun, Liu, and Gaudiot 2017), Intel OpenVINO (Zunin 2021) that target the optimisation of DNNs on edge devices. Each of these usually target the specific vendor's HW platform, limiting the portability across different systems. In this paper, we extend the original LPDNN, an enabling framework for deploying DNNs on low-power HW, by incorporating new inference engines and HW platforms for fair and systematic benchmarking. This allows for fast space exploration and a broader understanding of AI algorithm's behaviour across HW platforms.

Although some of these AI platforms offer AutoML methods to accommodate users' requirements, none offers a variety of HW devices and AI Applications to choose from as an open AI marketplace. With this work, we aim providing a support tool to guide the user through the variety of possible devices and help the choice of suitable deployment options. Thus, we aim to answer the following research questions: RQ1) how to effectively and efficiently perform benchmarks for AI applications on a variety of devices? and RQ2) is it possible to use data-driven insights to suggest the optimal HW device to end users, according to their requirements?

Our main contributions are:

- Systematic and fair benchmarking on heterogeneous platforms to profile AI algorithms.
- Training of ML models to learn the behaviour and requirement of these AI algorithms.
- Support guidelines to help identify the best deployment option for a given AI algorithm.

The results reported in this paper can be found in `https://github.com/Francobaldi/TinderAI`.

AI Matching Methodology

Our methodology is divided into three blocks: i) automated benchmarking, ii) ML models to learn the AI algorithms' behaviour, and iii) support guidelines for HW matching.

Automated benchmarking framework

The performance of AI applications (compiled AI algorithms for deployment) is usually given by the vendors on limited benchmark problems and HW platforms, providing little understanding for a potential *Adopter* about the general behavior of an AI application on different systems. Thus, the availability of an automated benchmarking framework, i.e., Benchmark as a Service, decreases the barriers for *Adopters* to benchmark their AI solutions and study the feasibility of a given real-world problem. In this work, we extend the original LPDNN as our automated benchmarking framework incorporating new inference engines and HW platforms.

LPDNN: Deep Learning Inference Framework LPDNN (de Prado et al. 2018; 2020) is an enabling framework for deploying DNNs on low-power HW. It comprises a full flow for developing DNN models for edge devices, providing support for various platforms, AI model's catalog, optimization, compression, and benchmarking tools to ensure efficient and portable implementations. LPDNN offers a set of AI applications for AI tasks for image, audio, and signal processing that can be deployed and optimized across heterogeneous platforms, including CPUs, GPUs, FPGAs, and NPUs, allowing performance portability across a wide span of HW platforms.

Deployment on heterogeneous embedded devices During the deployment of an AI model on multiple HW platforms many different problems may arise. For example, third-party dependency issues, lack of support or acceleration for specific NN layers, and impractically large inference times. LPDNN addresses these challenges by featuring a compact C++ core minimizing dependencies on third-party libraries. Only needed dependencies are incorporated when required by the target platform. Furthermore, LPDNN includes cross-compilation and tailored tools to support many of heterogeneous HW platforms. LPDNN provides:

- Developer Platform Environments (DPEs) that include OS images, drivers, and cross-compilation tools
- Dockerized environments that increase the reliability and portability of AI applications
- Optimisation tools and computing libraries that serve to accelerate inference of DL models on embedded devices

Benchmarking LPDNN provides a benchmarking layer to analyze the execution of AI Applications on HW platforms. LPDNN integrates i) an analysis of static metrics during the offline compilation of the AI Applications and, ii) the on-device metrics during the execution of the AI Applications. Static metrics are typically GFLOPs, number of parameters and disk size of the model. Some of the dynamic parameters are latency, consumed CPU/GPU memory and power consumption. The totality of the metrics is represented in the Bonseyes Benchmark documentation[1]. Moreover, LPDNN's capability of supporting the integration of 3rd-party self-contained inference engines to perform DNN inference allows different SW vendors to be benchmarked and fairly compared across several systems with the same benchmarking code. This makes LPDNN a powerful tool for space exploration for AI *Adopters*.

ML predictor

In this section, we describe the ML approach adopted to produce a pool of predictive models to answer the following question: given an AI task ($task$), a DNN (dnn) designed for such $task$, and an HW platform (hw) supporting such dnn, what is the average computational demand required by dnn, running on hw, in processing an instance of $task$?

Features & Targets We frame the above question as a supervised regression problem: we estimate the inference performance ($inference$) for any combination $C = (task, dnn, hw)$. We measure the performance with 8 different metrics, each one used as a "target" for our regression task. For target T the label of C is denoted as $y_C^T \in \mathbb{R}$. We encode C as an input vector of 11 features, that condense the relevant pieces of information to estimate the DNN behaviour at inference time. The resulting input space is denoted by $\mathcal{F}$, and the feature vector of C by $x_C \in \mathcal{F}$. Table 1 reports the composition of our feature space and target set.

Features			**Targets**
$task$	dnn	hw	$inference$
AI_TASK	MODEL	PLATFORM	TIME
	VERSION	ENGINE	POWER
	INPUT_TILE	PROCESSOR	CPU_LOAD
	#PARAMS		GPU_LOAD
	STORAGE		CPU_MEMORY_MEAN
	GFLOP		GPU_MEMORY_MEAN
	PRECISION		CPU_MEMORY_PEAK
			GPU_MEMORY_PEAK

Table 1: The adopted features and targets.

Then, for each target T, our task is to build a model $M^T : \mathcal{F} \rightarrow \mathbb{R}$, which approximates the inference performance of any given combination C, that is, $M^T(x_C) \approx y_C^T$.

Data Preprocessing The data from the benchmarking framework has to be transformed to make it suitable for training an ML model. First, irrelevant information is filtered out. Then, feature transformation routines i) encode the

[1] https://gitlab.com/bonseyes/bonseyes-benchmarks

categorical features [2] in a one-hot fashion, and ii) re-scale the numerical features [3] within the range $[0, 1]$ to normalize their magnitudes across the dataset. We split our dataset by adopting an 80%-20% *stratified random split* based on `AI_TASK`: for each task, 80% of the corresponding observations are randomly assigned to the train set, the remaining 20% are used in the test set.

We consider many different ML models grouped into 3 classes (in order of increasing complexity): linear models, tree-based models, and NNs. The first class consists of three linear regressors, respectively fitted through an ordinary Least Squares (LS), a Ridge (RIDGE), and a Support Vector Regression method with linear kernel (SVR). The second class includes Decision Trees (DT-n) and Random Forests (RF-n), where n denotes the maximum depth at which the tree is grown during training for DTs, and the number of individual predictors within the ensemble for RFs. Finally, the last class consists of NNs of different architectures (NN-$n_1 \cdot n_2 \cdot \ldots \cdot n_d$), where n_i and d denote the size of the i-th layer (assuming that the input layer is indexed by 0), and the depth of the network, respectively.

Support Guidelines

Based on the benchmarking results and the data-driven ML models built on top of these, this third step aims at matching target AI applications with the HW platforms, by suggesting the set of most suitable HW architectures (and their configuration parameters). The proposed matching depends on the users' requirements, ranging from required maximum inference time, deployment constraints[4], to available financial budget. The matching process is guided by the trained ML models that encode the knowledge extracted from the benchmarking results. This knowledge is used to predict the targets of a given DNN and rank/identify the best deployment option capable of providing the desired performance (e.g., in terms of budget consumption) under the constraints imposed by the available HW resources.

Experimental Results

In this section, we report the experimental evaluation results of our approach. First, we describe the benchmark results from LPDNN across multiple platforms. Then, we assess the ML models' performance to learn the behaviour of DNN inference on different HW devices. Finally, we provide a real use-case example where an Adopter SME chooses a model following the support guidelines.

Benchmarking Results

This section describes how LPDNN has been employed to produce a benchmark dataset for multiple AI Applications across different SW configurations and HW devices.

[2] `AI_TASK`, `MODEL`, `VERSION`, `PRECISION`, `PLATFORM`, `ENGINE`, `PROCESSOR`

[3] `INPUT_TILE`, `#PARAMS`, `STORAGE`, `GFLOP`

[4] For instance, if the DNN must be used on low-power embedded system, computational constraints are to be assumed

Experimental Setup We employed a set of pre-trained networks (AI Applications) for various tasks such as Face Landmarks Detection, Bodypose Estimation, Age and Gender Estimation, Emotion Classification, Eye Gaze Detection, Headpose Estimation, and more general research tasks like Imagenet, Coco for Image Classification and Object Detection. LPDNN supports the integration of 3rd-party libraries or self-contained inference engines to execute DNNs with multiple deployment configurations. In this work, we have benchmarked the previous AI Applications with the following inference engines on the following HW platforms:

HW platforms	Inference Engines
Raspberry Pi 3b+	LNE
Raspberry Pi 4b	TensorRT
NVIDIA Jetson Nano	ONNX runtime
NVIDIA Jetson Xavier	NCNN
Intel NUC	
iMX8m Nano	
STM32 MP1	

Table 2: LPDNN's inference engines and HW platforms.

Benchmarking dataset The execution of AI applications across all inference engines and configurations on the HW platforms produced a benchmark dataset of just under 3000 rows. Each experiment accounts for a different AI Application, SW, or HW configuration. Each benchmark consisted of the deployment and execution of the AI application over 20 runs with a previous warm-up run.

For a detailed description of the inference engines, HW platforms, benchmarking process and collected dataset using LPDNN, refer to the open-source repository[5].

ML Predictor Results

Linear and tree-based models are trained with Scikit-Learn, while NNs with Keras, and evaluated through the *Mean Absolute Percentage Error* (*MAPE*). However, since MAPE is undefined for predictions close to zero (as for our targets), we adopt a slightly modified version of this metric: given a predictive model M and a test set of observations $\mathcal{T}$, we compute the *MAPE* of M over $\mathcal{T}$ as $\frac{1}{|\mathcal{T}|}\sum_{(x,y)\in\mathcal{T}} APE\,(y, M(x))$, where *APE* is defined as $\frac{|y-M(x)|}{|y|}$ if $y \neq 0$, otherwise as y. To guarantee unbiased results, we always train and test our models exclusively on the train and test set, respectively. Moreover, to mitigate the impact of the randomness occurring in most of the adopted algorithms, we repeat any training/testing procedure for 3 different random seeds, after that we aggregate the results.

Table 3 reports the evaluation results. The tree-based models achieve the best results across all the targets, except for `CPU_MEMORY_PEAK`, on which the NNs perform slightly better. This reveals that our learning problem is too complex to be addressed by a linear regressor. On the other side, our dataset is not large enough to properly train data-hungry NNs. However, since large datasets are very costly tree-based models might be more suited to our end.

[5] https://gitlab.com/bonseyes/bonseyes-benchmarks/-/blob/master/README.md

MODEL	TIME	POWER	LOAD		MEM. MEAN		MEM. PEAK	
			CPU	GPU	CPU	GPU	CPU	GPU
LS	25.09	0.13	0.13	4.17	0.28	0.90	0.50	0.17
SVR	1.42	0.06	0.12	0.77	0.22	0.38	0.32	0.15
RIDGE	24.17	0.13	0.13	4.14	0.27	0.91	0.49	0.16
DT-5	5.36	0.02	0.05	0.18	0.14	0.05	0.24	0.05
DT-10	0.51	**0.01**	0.03	0.12	0.06	**0.04**	0.13	**0.04**
DT-15	0.21	**0.01**	0.03	**0.10**	0.06	**0.04**	0.14	**0.04**
DT-20	0.20	**0.01**	0.03	0.11	0.06	**0.04**	0.14	**0.04**
DT-25	0.20	**0.01**	0.03	0.11	0.06	**0.04**	0.14	**0.04**
DT-30	**0.19**	**0.01**	0.03	0.11	0.06	**0.04**	0.14	**0.04**
RF-10	0.24	**0.01**	**0.02**	0.11	**0.05**	**0.04**	0.13	**0.04**
RF-25	0.24	**0.01**	**0.02**	0.11	**0.05**	**0.04**	0.13	**0.04**
RF-50	0.23	**0.01**	**0.02**	0.11	**0.05**	**0.04**	0.13	**0.04**
RF-100	0.22	**0.01**	**0.02**	0.11	**0.05**	**0.04**	0.13	**0.04**
RF-150	0.23	**0.01**	**0.02**	0.11	**0.05**	**0.04**	0.13	**0.04**
RF-200	0.23	**0.01**	**0.02**	0.11	**0.05**	**0.04**	0.13	**0.04**
RF-300	0.23	**0.01**	**0.02**	0.11	**0.05**	**0.04**	0.13	**0.04**
NN-5	0.72	0.43	0.19	0.17	0.31	0.50	0.21	0.27
NN-10	0.69	0.27	0.13	0.17	0.20	0.40	0.15	0.13
NN-20	0.67	0.32	0.11	0.18	0.17	0.44	0.14	0.16
NN-5·5	0.68	0.50	0.13	0.17	0.22	0.54	0.15	0.43
NN-10·5	0.64	0.39	0.10	0.17	0.14	0.50	0.13	0.22
NN-10·10	0.60	0.38	0.10	0.17	0.13	0.48	0.12	0.18
NN-20·10	0.55	0.23	0.06	0.17	0.11	0.38	0.11	0.11
NN-20·10·5	0.49	0.31	0.04	0.17	0.08	0.43	**0.10**	0.13
NN-10·10·5·5	0.50	0.49	0.07	0.17	0.12	0.53	0.11	0.37
NN-20·10·10·5	0.44	0.37	0.06	0.17	0.09	0.47	**0.10**	0.29
NN-20·20·20·20	0.38	0.43	0.04	0.17	0.07	0.49	**0.10**	0.34

Table 3: Models performance across all targets.

DTs and RFs provide excellent results on more than half of the considered targets, with an average error that remains under 5% MAPE on `CPU_MEMORY_MEAN` and 4% on both `GPU_MEMORY_MEAN` and `GPU_MEMORY_PEAK` with an astonishing result of 2% and 1% MAPE on `CPU_LOAD` and `POWER`, respectively. On the other side, DTs and RFs are not able to achieve the same performance on `GPU_LOAD` and `CPU_MEM_PEAK`, on which the error rises up to 10% and 13% MAPE, respectively. The most critical target, however, is represented by `TIME`, where we are only able to achieve a not very satisfactory 19% error. Motivated by this, in the following section we investigate further the most promising tree-based models: DT-20, DT-25, DT-30, RF-50, RF-100, RF-150 (i.e., 3 DTs and 3 RFs), on the target `TIME`.

Task Reduction The following experiment is motivated by the fact that, across our dataset, the distribution of tasks (i.e., of `AI_TASK`) is not uniform, i.e., some tasks are substantially under-represented when compared against others. These tasks might jeopardize the accuracy of the predictions, given that our models need more data to learn properly. Hence, now we progressively reduce the variability in our data by removing some tasks from the train/test sets. More precisely, for the considered target, we compare the following 4 cases: 1) task-generic, trained/tested on *all* represented tasks (11); 2) reduced-task-generic, trained/tested only on *abundantly represented* tasks (6), precisely the ones described by more than 100 observations; 3) task-specific, trained/tested only on the *second most represented* task (`face-landmarks-detection`); 4) task-specific, trained/tested only on *the most represented* task (`bodypose-estimation`).

Figure 2 shows that, overall, the task reduction approach positively impacts our models' performance. DTs models trained on *task-specific* subset outperform their task-generic counterparts by roughly 20% (relative increase). Moreover,

Figure 2: *MAPE* performance (Y-axis) of the task-reduced cases of the tree-based models in predicting `TIME`.

an even more pronounced error decrease can be observed between the task-generic RFs and those specifically dedicated to `face-landmarks-detection`, for which we achieve a relative improvement of 30%.

Support Guidelines

In this section, we provide a real use-case example to find the best AI algorithm - HW platform match. Image a Small-Medium Enterprise (SME) expert in developing object detection algorithms for camera security systems. The company has an object detection DNN but it needs help in deploying it on different HW platforms for various clients. The inference time for the DNN varies greatly depending on the device and configuration, making it difficult to provide accurate performance guarantees to its clients.

The predictive ML models presented above allows the company to predict the performance of its DNN on different devices before deploying it. Based on the client's requirements, e.g., accuracy, min latency, and budget, the best deployment configuration and platform for their object detection algorithm are identified and selected. This allows the company to provide accurate performance guarantees and confidently deploy its DNN on various HW platforms.

Conclusion

We addressed the problem of finding the most suitable matching between HW platforms and SW deployment configurations to maximize AI applications' performance at inference time. We tested our approach on a widespread scenario, the deployment of DNNs on various HW devices.

We proposed a 3-stage sequential methodology for AI deployment optimization. First, different SW-HW configurations are benchmarked. Then, we used the benchmark data to train ML models that could predict the performance of DNNs on specific HW platforms, which are finally used to suggest most suited resources.

References

Canziani, A.; Paszke, A.; and Culurciello, E. 2016. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*.

De Filippo, A.; Borghesi, A.; Boscarino, A.; and Milano, M. 2022. Hada: an automated tool for hardware dimensioning of ai applications. *Knowledge-Based Systems*.

de Prado, M.; Denna, M.; Benini, L.; and Pazos, N. 2018. Quenn: Quantization engine for low-power neural networks. *Proceedings of the 15th ACM International Conference on Computing Frontiers*.

de Prado, M.; Mundy, A.; Saeed, R.; Denna, M.; Pazos, N.; and Benini, L. 2020. Automated design space exploration for optimized deployment of dnn on arm cortex-a cpus. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40(11):2293–2305.

Farhoodfar, A. 2019. Machine learning for mobile developers: Tensorflow lite framework.

Mariani, M. M.; Machado, I.; and Nambisan, S. 2023. Types of innovation and artificial intelligence: A systematic quantitative literature review and research agenda. *Journal of Business Research* 155:113364.

Shafi, O.; Rai, C.; Sen, R.; and Ananthanarayanan, G. 2021. Demystifying tensorrt: Characterizing neural network inference engine on nvidia edge devices. *2021 IEEE International Symposium on Workload Characterization (IISWC)*.

Sun, D.; Liu, S.; and Gaudiot, J.-L. 2017. Enabling embedded inference engine with arm compute library: A case study.

Talib, M. A.; Majzoub, S.; Nasir, Q.; and Jamal, D. 2021. A systematic literature review on hardware implementation of artificial intelligence algorithms. *The Journal of Supercomputing* 77:1897–1938.

Zunin, V. V. 2021. Intel openvino toolkit for computer vision: Object detection and semantic segmentation. *2021 International Russian Automation Conference (RusAutoCon)*.

Acknowledgments

This work has been partially supported by the EU Horizon 2020 Project StairwAI (g.a. 101017142).