EdgeTune: Inference-Aware Multi-Parameter Tuning

Isabelly Rocha
University of Neuchâtel
Neuchâtel, Switzerland
isabelly.rocha@unine.ch

Pascal Felber
University of Neuchâtel
Neuchâtel, Switzerland
pascal.felber@unine.ch

Valerio Schiavoni
University of Neuchâtel
Neuchâtel, Switzerland
valerio.schiavoni@unine.ch

Lydia Y. Chen
TU Delft
Delft, Netherlands
y.chen-10@tudelft.nl

ABSTRACT

Deep Neural Networks (DNNs) have demonstrated impressive performance on many machine-learning tasks such as image recognition and language modeling, and are becoming prevalent even on mobile platforms. Despite so, designing neural architectures still remains a manual, time-consuming process that requires profound domain knowledge. Recently, *parameter tuning servers* have gathered the attention of industry and academia. Those systems allow users from all domains to automatically achieve the desired model accuracy for their applications. While the entire process of tuning and training models is performed solely to be deployed for inference, state-of-the-art approaches typically ignore system-oriented and inference-related objectives such as runtime, memory usage, and power consumption. This is a challenging problem: besides adding one more dimension to an already complex problem, the information about edge devices available to the user is rarely known or complete. To accommodate all these objectives together, it is crucial for tuning system to take a holistic approach to parameter tuning and consider all levels of parameters simultaneously into account. We present EdgeTune, a novel inference-aware parameter tuning server. It considers the tuning of parameters in all levels backed by an optimization function capturing multiple objectives. Our approach relies on inference estimated metrics collected from a dedicated emulation server running asynchronously from the main tuning process. The latter can then leverage the inference performance while still tuning the model. We propose a novel onefold tuning algorithm that employs the principle of multi-fidelity and simultaneously explores multiple tuning budgets, which the prior art can only handle as suboptimal case of single type of budget. EdgeTune outputs inference recommendations to the user while improving tuning time and energy by at least 18% and 53% when compared to state-of-the-art systems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Middleware '22, November 7–11, 2022, Quebec, QC, Canada
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9340-9/22/11…$15.00
https://doi.org/10.1145/3528535.3533273

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **General and reference** → **Experimentation**; *Performance*.

KEYWORDS

deep neural networks, tuning, training, inference

ACM Reference Format:
Isabelly Rocha, Pascal Felber, Valerio Schiavoni, and Lydia Y. Chen. 2022. EdgeTune: Inference-Aware Multi-Parameter Tuning. In *23rd ACM/IFIP International Middleware Conference (Middleware '22), November 7–11, 2022, Quebec, QC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3528535.3533273

1 INTRODUCTION

Deep Neural Networks (DNN) are increasingly popular both in academia and industry [20, 29], adopted across a variety of application domains, including speech and image recognition, self-driving vehicles, face-recognition, genetic sequence modeling, natural language processing, e-health,*etc.* Several public cloud providers offer native support to deploy, configure and run them, with tools to (semi)automatically drive the DNN processing pipeline. The choice of the DNN hyperparameters (*e.g.*, number of hidden layers, learning rate, dropout rate, momentum, batch size, weight-decay, epochs, pooling size, type of activation function, *etc.*) is critical. DNNs require careful tuning of the hyperparameters. If executed correctly, it offers major boosts in performance [17, 52]. However, misconfigurations can easily lead to wrong models and hence bad predictions [25, 44].

Commercial platforms (*i.e.*, Google Vizier [23], Amazon SageMaker [35]), as well as on-premises solutions (*i.e.*, Auto-Keras [28]) help deployers by offering tuning services to mitigate (possibly avoid) misconfiguration. Such tuning services assist users to achieve the target model accuracy. However, they lack input regarding the inference phase, the ultimate goal of the tuning process. Such tuning services typically output the identified optimal hyperparameters, paired with the model resulting from training using such parameters. As the trained model is already given as output, the information of which optimal parameters were identified during tuning is no longer useful for the users. If users plan on retraining the model with a different dataset, nothing guarantees that the optimal parameters remain the same across different datasets. In fact, the next step for the user after having a fully trained model is to deploy it for inference use.

The deployment of inference model is not straightforward: it requires a vast domain knowledge, as well as extra experimentation, in order to find the best environment for achieving the desired inference performance. Even if the user has only a specific edge device available for model deployment, it is still crucial to either *(1)* configure the system parameters of this device, or *(2)* take the model inference performance on that specific device into account on the tuning process. However, the most common case is that the tuned model might be deployed across different edge devices and having these configurations suggested can assist users to take the most out of their tuned models.

Regarding tuning trials, a common approach is to define a budget allowed for each trial.This is typically defined in terms of epochs, dataset, or time. Using a reduction factor will exclude unpromising trials and increase the budget of promising ones, resulting in a frugal usage of resources. This is also indeed the case when compared to fixed budged tuning or even algorithms using any budget [51]. However, the current existing budgets (*e.g.*, epoch or dataset based) only consider one dimension, which means we are still using either the entire dataset or the full number of epochs for each trial.

On top of choosing a given edge device or a specific configurations, a critical point is the number of samples in which inference should be applied to. Although single sampled inference is common in practice, there are scenarios (see §3.4) where multi-sample inference is beneficial. In those scenarios, the batch size must be tuned carefully, as too-large values can lead to saturation. For instance, in a server scenario where each inference query contains N samples arriving at fixed frequency, the user should know what is the optimal way to split these samples. A similar pattern happens in a multi-stream system, where single sample inference queries arrive randomly following a Poisson distribution. In this case, if the optimal batch size is identified, aggregating samples for multi-sampled inference could improve the overall mean response time of the system.

We propose EdgeTune, a novel holistic edge-based tuning system capable of taking inference objectives into account during the tuning of hyperparameters. EdgeTune produces more useful information such as the optimal configurations of edge device for inference. Users can then directly deploy their trained model for inference without further actions. As shown later, this approach reduces tuning runtime by 20% and energy by 50% if compared to Tune.

To summarize, the main contributions of this paper are:

(1) EdgeTune, a novel inference-aware tuning system, striking the right balance between model performance and inference latency.
(2) Compared to other approaches, EdgeTune covers multi-parameters tuning in a (non-hierarchical) onefold solution, combining multiple layers of optimization.
(3) We introduce a multi-budget approach for the training trials which reduces their runtime but maintains the level of accuracy necessary for efficient convergency.
(4) We demonstrate performance gains on runtime and energy measurements using state-of-the-art workloads.

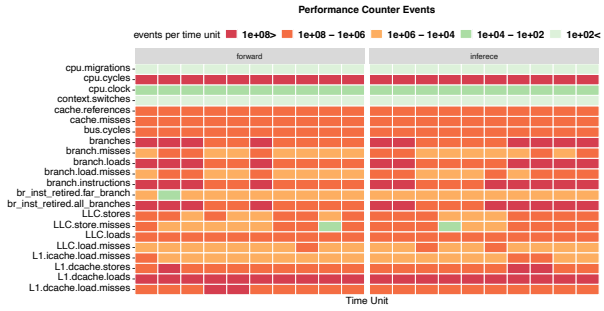


Figure 1: Performance counter events collected during the forward phase of training and inference.

2 PARAMETER TUNING PRIMER

This section discusses how a parameter tuning job operates, the different types of tuning budget typically considered by existing systems, and their limitations. Then, by means of an image classification motivating example, we show how the tuning server providers and the final users can both benefit from the tuning phase if the process considers different types of parameter and the inference phase.

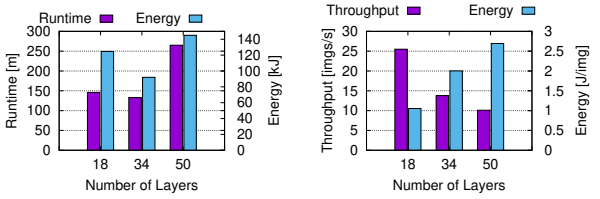
2.1 Tuning Deep Learning Job

A tuning deep learning job takes as input a given workload, a set of parameters to be tuned, and outputs the set of optimal parameters' values found together with the model which trained these values. In this context, we refer to workload as a tuple pairing a model and dataset. Typically, DNN workloads are used for training (*i.e.*, learning) or inference (*i.e.*, prediction). Training is the process of finding the model parameter's based on a given algorithm and objective. Once a model is trained, the inference phase deploys the model perform prediction of data with the same structure used in the training but with unseen values. Before training or inference, there is a set of hyperparameter and system parameters which have to be defined and have high impact on model performance both in terms of accuracy and runtime.

A tuning DL job consists of several trials, where each trial consists of the training/inference using a set of value for these parameters. At the end, the winning trial (*i.e.*, parameters leaving to best solution) is outputted to the user.

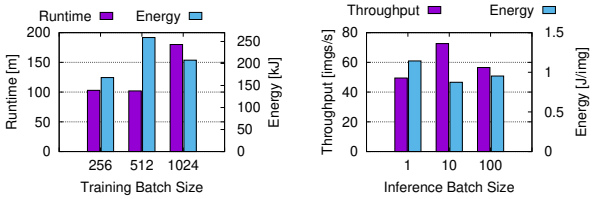
Before the training phase can even start, several other parameters must be configured, *e.g.*, the hyper- and system-parameters. Hyperparameters will influence the training process or the model architecture. The system parameters include the type of hardware used, number of CPU cores allocated, GPUs, etc.

Auto-tuning tools can find the optimal hyper/system configurations while also training the model. The training trials explore the search space of possible parameter's configurations. Hence, tuning a single workload consists of multiple training trials, each divided into epochs. One epoch consists of a full pass on the entire dataset, *i.e.*, forward and backward phase for all the batches. As training trials follow the stochastic gradient descent algorithm [8], each epoch involves one forward and one backward pass of the entire input dataset. For ease of processing, the dataset is split into smaller



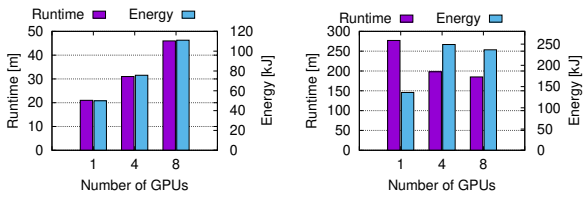
(a) Training **(b) Inference**

Figure 2: Model hyperparameters tuning.



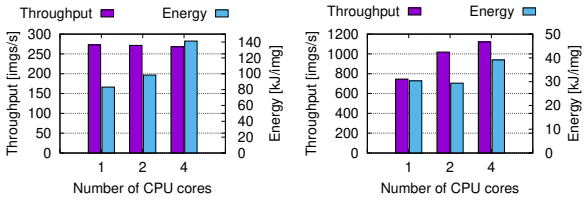
(a) Training **(b) Inference**

Figure 3: Training hyperparameters tuning.



(a) Batch = 32 **(b) Batch = 1024**

Figure 4: Training system parameters tuning.



(a) Batch = 1 **(b) Batch = 10**

Figure 5: Inference system parameters tuning.

batches, each one propagated forward and backward once during an epoch (*i.e.*, iteration). Once the model training is completed, it can be used for evaluation on an unseen chunk of the training data which is put aside at the beginning of training for this purpose (*i.e.*, 20% of the full dataset in our case). Finally, once the model reaches the expected accuracy, it is deployed in production with unknown data.

The following design options exist to consider inference while tuning: *(1)* rely on the forward phase of training, *(2)* offload the model to inference device and collects, or *(3)* simulate the inference devices in the tuning server. In theory, the inference is equivalent to the forward phase of training with the only difference being on the model weights. However, in practice this is not the case as the memory utilization during training is much higher than for the inference. In fact, while training, the weights are constantly being updated and kept in memory for faster access. Instead, during the inference phase the optimal weights are known and only consist of constant values. Finally, training is performed on relatively large batches of images, whereas inference is often done on a single images.

We show that relying on the forward phase to make predictions about the inference phase is not the best approach, by collecting hardware performance counters [3] during the different phases. We used three platforms: *(1)* an ARMv7 Processor rev 4 (v7l) with 4 cores and 4GB RAM, *(2)* a Raspberry Pi 3 Model B+ (v1.3), 4 cores, 1 GB RAM, and *(3)* an Intel(R) Core(TM) i7-7567U CPU with 16GB. Figure 1 shows CPU and memory related events for the AlexNet [27] model with CIFAR10 [31] dataset during the forward phase of training and inference performed with the trained model. We observe how CPU-bound events (*e.g.*, `cpu.*, context.switches`) are consistent between the forward phase and inference, while memory-bound events (*e.g.*, `cache-*, L-*, LLC-*`, branch mispredictions requiring pipeline flushes and fetching of new instructions) are not.

Hence, one is left with the options of offloading the model to actual edge devices, or simulating such devices in the tuning server. The first approach gives more precise results, at the cost of additional data transfers, and limits on the number of available devices. We settle to simulate the edge devices for inference. We show later (§ 5) how EdgeTune quickly evaluates a large search space without adding an overhead as the combination of optimizations proposed in our approach actually reduces tuning time. Moreover, the error of the simulation results on inference with respect to the actual measurement in edge devices is small (at most 20% in our experiments).

2.2 Tuning Budget

To optimize the otherwise costly search process of finding the set of best parameters, technics such as early-stop [32] or multi-fidelity [30] budgets exist. With multi-fidelity budget, the promising range of values is identified using a model approximation (subset of the training data, fewer epochs) which is cheap to evaluate by definition.

A training trial consists of a certain number of epochs runs applied on a given dataset. These runs explore the search space: the optimal configuration is from the winning trial. Hence, the majority of trials waste precious resources, and it is critical to quickly discard non-promising ones. Trials run on a given budget, defined in terms of *(1)* number of epochs, *(2)* portion of training dataset, and *(3)* duration. The tuning algorithm defines a min/max budget for the resources allowed for tuning, and a reduction factor which determines the fraction of configuration that continues in the next iteration.

Example. Consider the number of epochs. Assume the minimum number of epochs is 1, maximum 16, and the reduction factor is 2. We start tuning with 16 trials initialized on the minimal budget (*i.e.*, 1 epoch). The next iteration will consist of 8 trials with 2 epochs,

then 4 trials with 4 epochs, 2 trial with 8 epochs and a final iteration containing only one trial with 16 epochs. Similar reasoning applies for other budgetable parameters, *i.e.*, dataset fraction or time.

2.3 Tuning Space by Motivating Examples

The training phase of DNN workloads involves hyperparameters set before optimizing the model's parameters. Setting the values of hyperparameters can be seen as model selection, *i.e.*, choosing which model to use from the set of possible models. Hyperparameters are often set by hand, selected by some search algorithm, or optimized by some auto-tuner tool [23, 28, 35]. DNN models support many types of hyperparameters: related to the structure (model), as well as those for the training or the inference phases. System configurations (and related system parameters) greatly impact the tuning and inference performance, and must be taken into account We describe each of these parameters type and motivate their relevance by means of a practical example using the tuple ResNet and CIFAR10 as workload tuned to reach at least 80% model accuracy.

2.3.1 Model Hyperparameters. The structure of the neural network itself involves numerous hyperparameters in its design, including: size and nonlinearity of each layer, number of hidden layers, weight decay, activation sparsity, weight initialization, and preprocessing input data. The numeric properties of the weights can be constrained in some way, and their initializations have a strong effect on model performance. Finally, preprocessing of the input data can also be important for ensuring convergence [13, 13, 15].

We show how the number of layers impact the training (Figure 2a) and inference (Figure 2b) performance in terms of runtime and energy. The performance does not directly relate to the number of layers, nor is it straightforward to predict. For the inference phase we show the throughput (*i.e.*, images per second) and the energy consumption per single image inference in Joules (J). In this example, the throughput is inversely proportional to the number of layers, but the energy consumption is proportional to it. This trade-off between runtime performance and energy costs can be exploited during tuning if this process is inference-aware by simulating the inference phase during tuning and considering the inference-related results in the objective function.

2.3.2 Training Hyperparameters. When training a neural network, the resulting model depends not only on the chosen structure but also on the training method used to set the network's parameters. The training method itself can have many hyperparameters. Here we describe the hyperparameters of mini-batch gradient descent, which updates the network's parameters using gradient descent on a subset of the training data. Some examples of this type of hyperparameter are learning rate, loss function, mini-batch size, number of training iteration, and momentum. Figure 3a shows the impact of the batch size on training runtime and energy consumption. We observe how high batch sizes values (*i.e.*, 1024) result in high training times and energy consumption, while others (*i.e.*, 256 and 512) produce similar training times but different energy consumptions. This indicates that when energy is a concern, it should be explicitly taken into account while tuning. This observation supports the need of a multi-parameter tuning approach.

2.3.3 Inference Hyperparameters. In scenarios where multiple images are available, multi-image inference can be beneficial. In this case, the hyperparameter *batch size* must be defined for the inference phase and its value has a direct impact on performance. Figure 3b shows the throughput (*i.e.*, images per second) and the energy consumption per single image inference in Joules (J) when doing single-inference (*i.e.*, one image at a time) and multi-inference (*i.e.*, multiple images at a time). As show, both throughout and energy improve by performing multi-inference in comparison with single-inference. However, the choice of how many images to include in each batch to process at any given time is critical as the performance gains can quickly reach saturation and start decaying if the chosen batch size is too high.

2.3.4 System Parameters. The configurable resources of the underlying computing infrastructure executing the training and inference (*e.g.*, memory, CPU cores, CPU frequency, number of GPUs, *etc.*) are the system parameters. Typically, the hyperparameter optimization fixes the same system parameters for each trial, although they might benefit from different configurations. Moreover, while model and parameters are tuned for model accuracy, its impact on the inference performance is not considered. Figure 4 and Figure 5 show the impact of system parameters on the training and inference phases by varying the number of GPUs and the number of CPU cores.

Figure 4a and Figure 4b show the training results for batch 32 and 1024, respectively. We note that for smaller batch sizes, neither runtime nor energy are improved by increasing the number of GPUs. Actually, we observe the opposite, as the performance considerably decreases by up to 120%. With larger batch sizes, results become even harder to predict. The running performance improves but not proportional to the number of GPUs, while the energy consumption actually increases even in the cases where runtime is lower. These results highlight the trade-offs to consider while tuning, specially if energy consumption is a concern for the user.

Regarding inference, Figure 5a and Figure 5b show similar results. For single image inference, increasing the number of cores does not increase throughput (as expected) and increases energy consumption. For multi-image inference, the throughout grows proportionally to the number of cores but the energy consumption of 4 cores is 33% higher than for 2 cores although the throughput is only 9% higher. Therefore, when energy savings are more important than inference performance, it becomes harder to find the sweet spot solution.[1]

3 EDGETUNE: SYSTEM DESIGN

In this section we present the overall architecture of EdgeTune as well as a description of its main components, how they work individually to take care of specific aspects covered by the system, and how they communicate and interact with each other.

3.1 Overview

EdgeTune consists of two main components: *Model Tuning Server* and *Inference Tuning Server*. The user gives as input 1) the workload to be tuned (*i.e.*, dataset and model), 2) the set of hyperparameters,

[1] In the shown example, the most energy-saving solution requires 2 CPU cores, which is however not the one with highest throughput.

Algorithm 1: EdgeTune complete algorithm.

```
1  Function model_tuning_server:
2      inf_results = {};
3      train_metrics = [];
4      for p in model_train_params.search_space() do
5          if not inf_results[p] then
6              inf_results[p] = async inference_server(model);
7          p_result = trial(model, data, p);
8          train_metrics.add(p_result.metrics());
9      model_params =
         train_obj_function(train_metrics).params();
10     return model_params, inf_results[model_params];
11 Function inference_server:
12     inf_model = model.random_init();
13     inf_metrics = [];
14     for p in inf_params.search_space() do
15         set(p);
16         p_result = inf_model.predict(data);
17         inf_metrics.add(p_result.metrics());
18     return inf_obj_function(inf_metrics).params();
```

3) the set of system training parameters, 4) the set of inference training parameters, 5) the tuning objective (*e.g.*, tuning duration or energy, model accuracy), 6) the inference objective (*e.g.*, throughput, energy), and 7) the choices tuning algorithms. Each set of parameters to be tuned comes together with another set containing the corresponding range of values which each parameter can assume. As for outputs, the users receive the optimal trained model with respect to the tuning objective as well as the optimal system configurations to be used for inference deployment with respect to the inference objective.

Algorithm 1 shows the overall logic of this system as well as how the components interact with each other. Different from the hierarchical tuning, both *Model* and *Inference* servers jointly explore model parameters and inference system parameters in a parallel and pipelining fashion – so called onefolding approach. For any configuration, the *Model Server* first uses a tuning algorithm to explore the accuracy related parameters which are then nested with the inference system parameters. The *Inference server* then takes over this nested part of tuning, which can use a different algorithm from the model server. As there are multiple configurations to be tried out, the model server can parallelize its tuning process. To avoid (frequent) stalling from the dependency to the inference server, the inference server pipelines its tuning process. The asynchronous communication among the model and inference server is thus the key for EdgeTune. Figure 6 illustrates this process for 3 values of model and inference parameters.

In more details, model tuning server chooses the architecture structure as input to the *Inference Tuning Server*, which first checks if the required information is already available for the defined architecture and model hyperparameters. This step consists of a table look-up based on historical data stored from previously processed trials. If results are already available, the *Inference Tuning Server* immediately returns it and no further action is required on this component. Otherwise, the inference-based tuning starts on the given architecture, model hyperparameters, and the set up inference parameters. In this case, the optimal configuration is stored and given back to the *Model Tuning Server* which takes it into account to update its metric of interest. This process is repeated for all the trials until the final result is reached and given as output to the user.

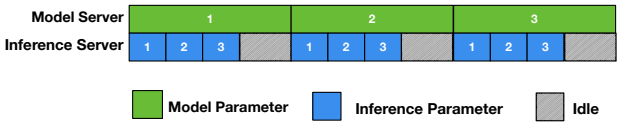


Figure 6: Example of model and inference tuning servers with 3 values of parameters each.

Objective. We assume that the objective function given to the *Inference Tuning Server* component does not relate to model accuracy as the *Model Tuning Server* already takes care of this aspect (*e.g.*, maximize model accuracy as the objective of the tuning server and minimize inference energy for the inference server). Therefore, as soon as the values of a given trial are defined, the *Inference Tuning Server* can asynchronously be started with arbitrary weight values, and the inference tuning process can take place in parallel to model tuning. Moreover, some types of parameters such as training batch size and number of epochs do not affect the inference phase. Considering this, the *Inference Tuning Server* results can be reused for different parameters as long as they do not affect the architecture structure. Since we explicitly divide the hyperparameters into these two types (*i.e.*, model and training), we can easily identify for which parameters the results can be reused.

Tuning algorithm. The *Model Tuning Server* and *Inference Tuning Server* components are implemented such that the user can also individually specify which tuning algorithm to be used by each of them (*e.g.*, Random Search [7], HyperBand [32], BOHB [22]). For instance, a user could choose to run the *Model Tuning Server* following the HyperBand approach while *Inference Tuning Server* following GridSearch. One setup where this configuration could make sense is where the range of inference parameters is not too large. In this case, trying all the parameters for inference would give more accurate results without necessarily affecting the overall tuning duration.

3.2 Architecture

Figure 7 depicts the architecture components of EdgeTune design and its main workflow. EdgeTune consists of two main components which we describe bellow in details (*i.e.*, model and inference tuning servers). The model tuning server can be executed using both CPUs or GPUs, while the inference server is only CPU based. The reason for the later is that, first, the inference server simulates edge devices which typically do not contain any GPU card, and, second, the inference tuning is straightforward and therefore does not require any accelerator. Regarding the model tuning server on the other hand, although both scenarios are supported, it performs significantly better when used with GPUs. We also support multi GPUs training and tuning, which can improve the performance in some case but

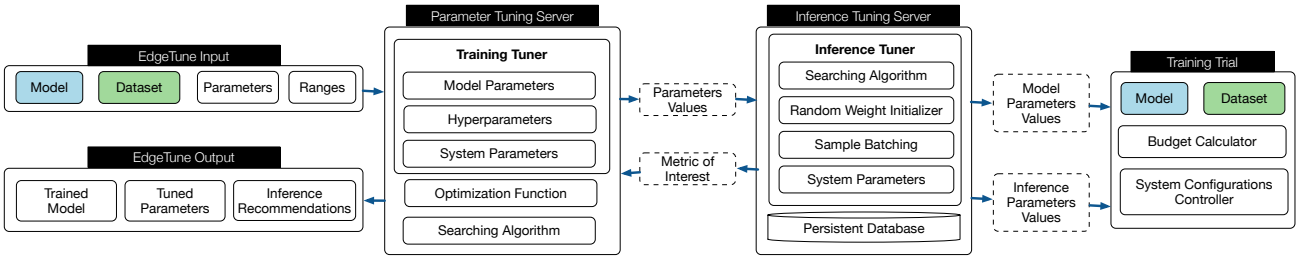


Figure 7: EdgeTune architecture.

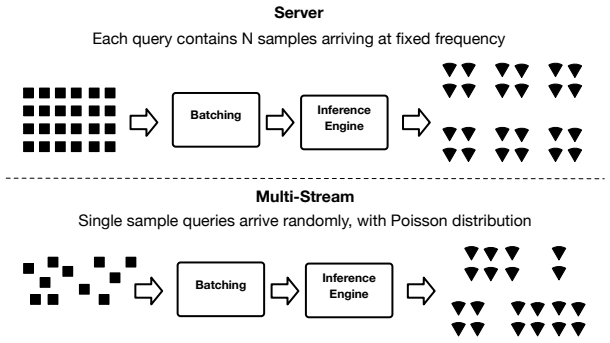


Figure 8: Illustration of scenarios where multi-image tuning is required.

for others it does not scale as one might expect. As already shown in Figure 4, increasing the number of GPUs used for tuning does not positively affects runtime in every case. Therefore, the system parameters tuning takes care of finding the optimal architecture configurations for each tuple of workload and parameter's values.

3.3 Model Tuning Server

The *Model Tuning Server* receives as input the hyper (*e.g.*, number of layers, batch size, number of epochs) and system (*e.g.*, number of cores, memory, CPU frequency) parameters to be tuned regarding model training, together with the workload of interest (*i.e.*, dataset and model) and the metric of interest (*e.g.*, runtime, energy) and the optimization function (*e.g.*, min, max, threshold). Following a given search algorithm, the tuning server defines the search space of the given parameters and starts performing training trials on these values. For each trial, the *Inference Tuning Server* is asynchronous called which in parallel computes the information of the optimal inference system parameters together with its runtime and energy consumption. It is important that the *Inference Tuning Server* result comes back before the end of the ongoing trial, as it contains crucial information for the overall optimization function of the *Model Tuning Server*. This constraint is guaranteed as the entire *Inference Tuning Server* duration is contained in the duration of a *Model Tuning Server* training trial and, therefore, also does not add any overhead to the main process. Once the result is received and the training trial is finished, the resulting metrics from both cases can be combined and considered together on the decision of promising configurations.

3.4 Inference Tuning Server

The *Inference Tuning Server* receives as input the model structure and model hyperparameter values defined by the trial which called it, together with the hyper (*i.e.*, batch size) and system (*i.e.*, number of cores, memory, frequency) parameters to be tuned regarding the inference phase. In this case, before starting the parameter search, the system verifies whether the optimal configurations are already known for the given model structure based on historical data. The feature of looking into historical data allows us to improve performance since it avoids retuning architectures and parameters twice, with the cost of a small storage overhead [42]. If this is the case, then the found parameters together with their metric of interest (*e.g.*, runtime and energy consumption) are directly returned and no further action is required. Otherwise, a process similar to the one described for *Model Tuning Server* takes place but this time with focus on inference instead of training. Following a given search algorithm (*e.g.*, BOHB [22]), the tuning server defines the search space of the given parameters and start performing inference trials on these values. The optimization function defined by the user is applied to the performed trials and the optimal set of parameters is then identified, saved for later usage and returned together with the metrics of interest.

As this server takes as input the network structure from the *Model Tuning Server*, the definition of model hyperparameters are already taken care of in this step. Therefore, the focus of this component is on the inference hyperparameters and system parameters for the inference phase. Each workload might have particular hyperparameters to be tuned but batch size is a hyperparameter of common interest and therefore considered for all workloads by a subcomponent named *Batching*. Figure 8 illustrates two scenarios where the *Batching* subcomponent is crucial for inference performance. The first scenario is a server where each query contains N samples arriving at fixed frequency. In this case, it is important to define how many samples should be processed at a time to achieve the expected performance. The second scenario is a multi-stream where single sample queries arrive randomly, following a Poisson distribution. In this case, aggregating the individual samples to perform batch inference can also optimize the overall mean response time and therefore *Batching* is also relevant.

Finally, the systems parameters for inference are the second main aspect playing a role in the inference throughput and therefore has to be carefully considered. As we have seen earlier describing the motivating examples, different batch sizes might require different system configurations. Therefore, in order to reach optimal

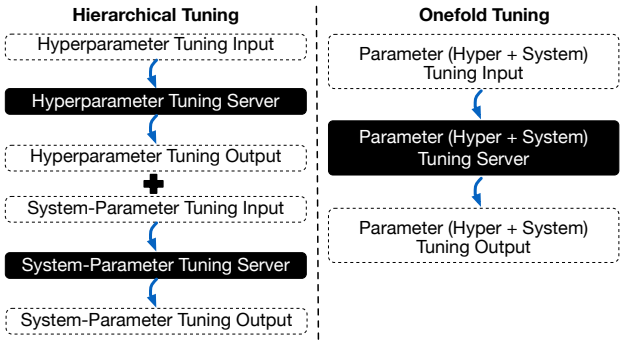


Figure 9: Difference on the execution flow between hierarchical and onefold tuning approaches.

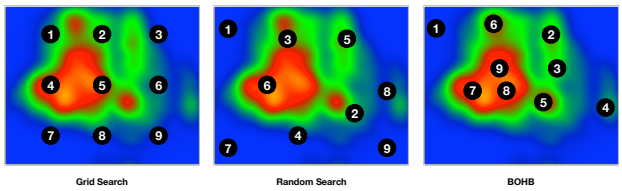


Figure 10: Flow of training trials during parameter tuning following 3 different searchings algorithms.

inference tuning performance, the batch size and the system configurations are tuned in combination. Although this adds an extra step to the main tuning process, this component runs in parallel to the training trials which in principle takes much longer than the inference trials. Moreover, in the worse case, the inference trial is performed for the first training trial of a given configuration and all the further training trials will reuse these results.

4 ONEFOLD TUNING ALGORITHM

We describe here the proposed onefold tuning algorithm and its novel features, including search algorithm, training budget and objective functions.

4.1 Hierarchical *vs.* Onefold approach

The process of tuning hyper and system parameters can be solved in two different ways: 1) non-hierarchically (*i.e.*, onefold), where both parameters are tuned together in an one-tier manner, and 2) hierarchically, initially tuning the hyperparameters, and then the system parameters are tuned only for the set of optimal hyperparameters values found by the first tuning step. The main drawback of non-hierarchical tuning approaches lies in the search space size, as it increases significantly when two sets of parameters are considered together. However, this approaches allows to take the system parameters impact on model training and inference performance into account during the process of finding hyperparameters. Opposed to that, hierarchical approaches treat these two types of parameters independently and tune the hyperparameters in a manner which does not consider the strong dependency of hyper and system parameters. Figure 9 depicts the execution flow difference between

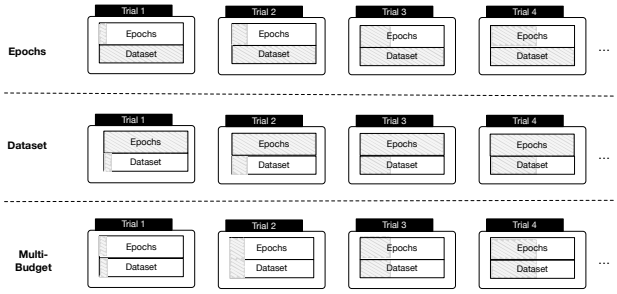


Figure 11: Flow of trials for 3 budget approaches: epochs, dataset, and multi-budget.

hierarchical and non-hierarchical (*i.e.*, onefold) tuning approaches. We implement a prototype for each strategy, and compared the results to support our assumption.

4.2 Search Algorithm

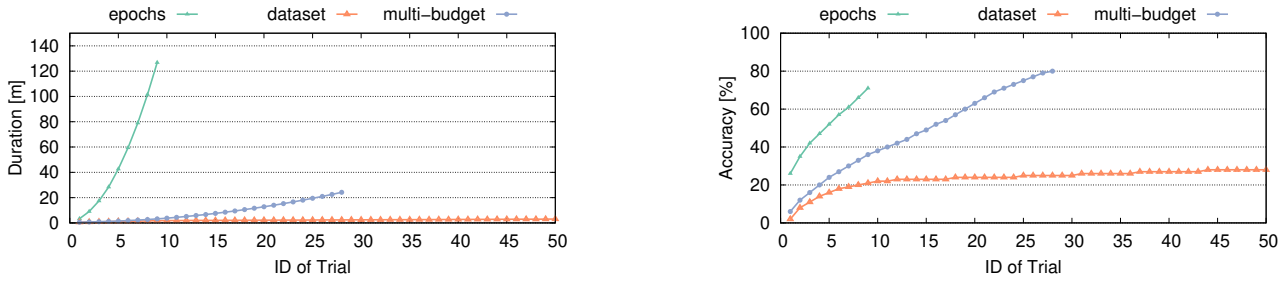
The core of parameters tuning is the search algorithm (*e.g.*, Random search, HEBO, BOHB, HyperBand). Each of these algorithms follow a specific strategy to define the search space, the most basic ones for parameter search being via random and grid search. Random search is backed up by a variant generator which randomly picks a value in the given interval, while grid search exhaustively tries all the possible values. Optimized algorithms (*i.e.*, Bayesian Optimization HyperBand - BOHB-) implement early termination of bad trials and uses Bayesian Optimization to improve the parameter search. In practice, each type of parameters to be tuned may be specified to follow its own searching algorithm. For instance, a user can choose to tune the number of cores following random search and the batch size following BOHB in the same tuning run. In our context, while user can freely choose the strategy, the default behavior of the current prototype picks the BOHB strategy for all parameters. We focus on BOHB as our novel strategies (*i.e.*, multi-budget) can easily be integrated.

Figure 10 shows the parameter tuning problem with three different searching algorithms: grid search, random search, and BOHB. Each point represents a specific parameter configuration. Warmer colors indicate a performance following a given metric of interest. The circled numbers from 1 to 9 indicate the training trials performed in each case. We observe that the trials following BOHB concentrate on the most promising regions of the search space, differently than grid and random search. Given these results, the default behavior in our current implementation is that all parameters are tuned following the BOHB strategy.

4.3 Training Trial Budget

We design a novel multi-budget approach and compare it against an epoch based and a dataset based budget.

The state of the art tuning algorithms are based on the concept of multi-fidelity [30], *e.g.*, successive halving, using low-fidelity data to explore the entire system performance and then high-fidelity data to exploit the optimal configuration. Specifically, the small amount of *budget* is used to explore a large number of configurations and then big amount of budget is spent on a small number of



(a) Training trial time **(b) Accuracy**

Figure 12: Model accuracy and training time convergency over the trials execution with the 3 budget approaches. Workload: Resnet18 on CIFAR 10.

Algorithm 2: Trial multi-budget algorithm.

```
1 Function trial(model, data, hyperparameters, it):
2     epochs = min(min_epochs*it, max_epochs);
3     data_frac =min(min_data*it, 1);
4     data = data.subset(data_frac);
5     for epoch in epochs do
6         model.train(data);
7     return model;
```

promising configurations. Typically, budgets are defined through training iterations, *i.e.*, epoch, or training data size. While most of the prior art overlooks the impact of different budgets, [16] shows that an appropriate choice of budget type can improve the tuning efficiency significantly. In epoch based approaches, the number of epochs performed in each training trial is proportional to the current iteration (*i.e.*, epochs is equal to two times the iteration level). In a dataset based approach, only one epoch is performed per training trial but the amount of data used in each trial is proportional to the iteration level (*i.e.*, percentage of dataset used is equals to *min(1, iteration_id*0.1)*).

Finally, the multi-budget approach combines both strategies, *i.e.*, each training trial uses a number of epochs and a chunk of dataset which is proportional to its iteration. By combining both approaches, we have a trial which does not take as long as of we would use the entire or run for a fixed number of epochs. At the same time, we run it for long enough or with enough number of samples to make the accuracy of a trial representative. Figure 13 depicts the trial execution flow of the 3 above described budget approaches.

In the multi-budget approach (see Algorithm 2), we start with the minimum number of epochs and minimum fraction of the dataset. Then, in every new iteration, the budget for number of epochs and portion of the dataset grows simultaneously and proportionally to the current iteration. For instance, if we start with minimum epochs equals to 2 and minimum dataset fraction 10%, the next iteration will consist of 4 epochs on 20% of the dataset, the third iteration will take 6 epochs on 30% of the dataset, and so on. Although both dimensions grow simultaneously, their maximum limits are set independently. Once the maximum limit for one of the dimension is reached, the maximum value is used further and the other one continues to grow until both reaches their limits. In our example, if the maximum number of epochs is 10, then from the 5^{th} iteration onward, the number of epochs is fixed to 10 and the dataset keeps growing until the 10th iteration when both dimensions reach their maximum values. In summary, the budget for both epochs and dataset is given by *min(max, iteration_id*min_fraction)*).

Figure 12 shows the model accuracy convergency over the trials for each of the three approaches described above. In this example we can see that the epoch based approach reaches the target accuracy (*i.e.*, 80%) within few trials but the execution time per trial is extremely high. On the other hand, the dataset based approach has a very low execution time in comparison to the others but the model accuracy does not go higher than 40% even after 100 trials. Finally, the multi-budget approach presents the perfect balance between these two approaches, reaching the target accuracy with more trails than the epoch based approach but with significantly lower trials execution times. Hence, we show how the multi-budget approach is the most appropriated for tuning servers as it finds the sweet-spot between trial duration and accuracy. Having this balance is crucial: the trial duration is critical for the overall performance of the process, but the model accuracy given by a trial needs to be representative enough to avoid advancing the training on non-promising trials.

Now, combining this multi-budget approach with the halving process of algorithms such as BOBH would mean that, besides of the specifications above, the algorithm defines a reduction factor represented by η. This reduction factor defines the fraction of hyperparameters which continues to the next iteration at the end of each cycle. For instance, if $\eta = 2$, then half of the configurations being considered continues to the next iteration at each cycle. This, combined with the multi-budget strategy allows the tuning to try many configuration at low cost and, more importantly, spend more time and resources in the most promising configurations.

4.4 Objective Functions and Metrics

The *ModelTuning* objective function is to maximize model accuracy while still considering tuning and inference performance.

Table 1: Workloads used for experiments.

Type	ID	Model	Dataset	Datasize	Train Files	Test Files
Image Classification	IC	ResNet	CIFAR10	163 MB	50.000	10.000
Speech Recognition	SR	M5	Speech Commands	8.17 GiB	85.511	4.890
Natural Language Processing	NLP	RNN	AG News	60.10 MB	120.000	7.600
Object Detection	OD	YOLO	COCO	19 GB	164.000	41.000

As tuning is composed by several training trails, optimizing training performance directly impacts the tuning performance. In current implementation, performance can be defined either in terms of runtime or energy consumption. Therefore, the final optimization function of this server to minimize the metric of interest (time or energy) defined as the ratio performance to accuracy:

$$(1)\ ratio = \frac{training\_time * inference\_time}{accuracy}$$

$$(2)\ ratio = \frac{training\_energy * inference\_energy}{accuracy}$$

In the case of the *Inference Tuning Server*, the objective function is defined only in terms of inference performance (*i.e.*, inference runtime or energy consumption). This mean that the objective function is set to minimize the metric of interest defined either as runtime or energy consumption of the inference phase alone.

Those objective function and metrics are used out-of-the-box, as our implementation already takes care of collecting and configuring all the necessary metrics. However, both the objective function and the metric of interest are configurable and can be easily set up. For instance, if the user would like to focus on the inference performance only then another metric of interest could be defined where the training performance is not included.

4.5 Implementation Details

The EdgeTune prototype is implemented in Python (v3.6.8), and it consists of 817 LOC. We release its code to the research community, available from https://github.com/isabellyrocha/edgetune. We leverage the Tune [34] Python library from Ray (v1.4.0) to reuse existing implementations of state-of-the-art search algorithms. Tune is used for experiment execution and hyperparameter tuning, in particular for tuning schedulers based on different optimization algorithms (*i.e.*, BOHB).

For the training and inference of workloads we rely on PyTorch [41] (v1.9.0). This is an imperative-style high-performance deep learning Python library, supporting data structures for multi-dimensional tensors and mathematical operations over these tensors. We rely on torchaudio (v0.7.0) for speech recognition workloads and on torchtext (v0.10.0) for natural language processing ones. Finally, for image classification and object detection workloads, we use torchvision (v0.8.1+cu101).

CUDA support for PyTorch [12] allows these tensors to run on an NVIDIA GPU [1]. We exploit this option, using CUDA (v11.3) combined with NVIDIA driver (v465.19.01) to accelerate the training part of our framework and therefore further optimize overall tuning duration.

5 EVALUATION

This section presents our extensive experimental evaluation of the EdgeTune prototype, including different considerations from 3 state-of-the-art application domains.

5.1 Experimental Setup

We begin by detailing our experimental setup, including testbed, baseline, workloads, and as well as the considered parameters and their ranges.

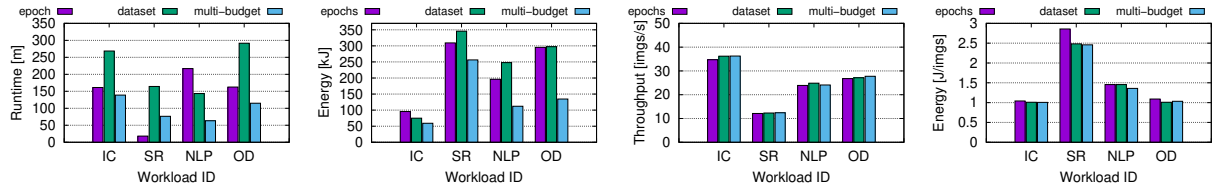
Testbed. We use Titan RTX GPU, Turing architecture with 24GB of Memory and 7.5 compute capability running CentOS Linux (v7.9). Power metrics are collected using the library PyRAPL (v0.2.3.1) [2] which is a software toolkit to measure the energy footprint of a host machine along the execution of a piece of Python code.

Baseline. Our baseline system (*i.e.*, Tune) uses the tuning of hyperparameters ignoring all system parameters and the inference phase. For a fair comparison, we configure Tune to use the same searching algorithm as EdgeTune (*i.e.*, BOHB).

Workloads. Table 1 summarizes the properties of the selected workloads for evaluation including image classification, speech recognition, and natural language processing problems. The CIFAR10 [31] dataset consists of 32x32 colour images labeled in 10 mutually exclusive classes. Speech Commands [47] is an audio dataset of spoken words designed to help train and evaluate keyword spotting systems. AG News [24] is a collection of news articles gathered from more than 2000 news sources spanning 1 year of activity. Common Objects in Context (COCO) [37] is a large-scale object detection, segmentation, and captioning dataset featuring 5 captions per image and 80 object classes.

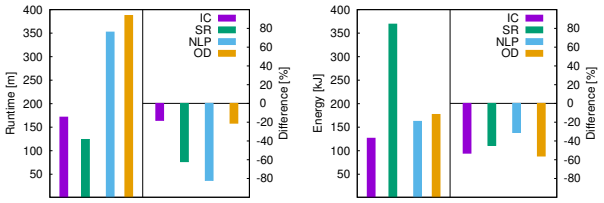
Training Hyperparameters. We study the impact of the training batch size, *i.e.*, the number of samples are aggregated to perform the training. The range of values considered in our setup vary from 32 to 512, across all the workloads.

Model Hyperparameters. The model hyperparameter considered for ResNet was *number of layers*, with the possible values being 50, 34, and 18. In the case of M5 (*i.e.*, the speech recognition case), the model hyperparameter considered is the *embedded dimension*, assuming values 32, 64, or 128. For the RNN model (*i.e.*, for natural language processing), we tune the *stride* parameter, as it maps to the amount of movement over the image or video. For example, if a neural network's stride is 1, the filter moves one pixel, or unit, at a time. In our setup, the stride value vary from 1 to 32. Finally, for the YOLO model we tune the *dropout rate* with values varying from 0.1 up to 0.5. Dropout is a regularization technique that randomly sets activations in a neural network to 0, preventing models from overfitting.



(a) Tuning duration (b) Tuning energy (c) Throughput (d) Inference energy

Figure 13: Results of the 3 budget approaches, namely, epoch, dataset, and multi-budget, for 4 different workloads.



(a) Tuning duration (b) Tuning energy

Figure 14: Tuning duration and energy overhead of EdgeTune in comparison with *Tune* baseline which does not make use of the inference tuning server component.

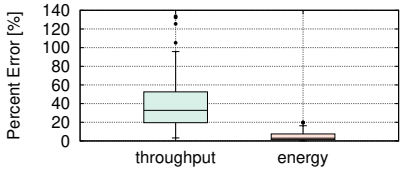


Figure 15: Error of throughput and energy consumption estimated by the *Inference Tuning Server* component in comparison with actual values collected on edge device.

System-parameters. Regarding training system parameters, we consider the number of GPUs to be used by the trials which can vary from 1 to 8. For the inference system parameter, we consider number of CPU cores and inference batch size. The inference batch size, from 1 to 100 in our experiments, represents the number of samples used during inference.

Overall, following the convention of parameter tuning literature, we choose a wide range of values for the parameters to be tuned aiming at covering the most common cases. To keep our evaluation consistent, we apply the same ranges for each workload presented. The optimal parameters might vary from workload to workload. In practice, these ranges could be adapted to more specific requirements without affecting the general concepts here demonstrated. When the domain knowledge is applied, such ranges can be narrower but lead to unfair comparison. One can thus see our ranges as a kind of worse case scenario.

5.2 Tuning Budget Choice

Given the three possible budget options mentioned earlier (*i.e.*, epoch, dataset, or multi-fidelity), we evaluate our novel multi-fidelity approach against the former two. Figure 13 compares these approaches under the tuning and inference perspectives, respectively. For workload IC, for instance, we notice that the inference configuration of these 3 approaches are very similar.This is expected: there are different possible optimal solutions, and we run enough trials such that each approach can converge to one of these. However, there are significant differences regarding tuning runtime and energy consumption. Overall, we observe that the epoch based approach is better than the dataset approach in terms of runtime but worse for energy consumption. Instead, the multi-budget approach performs consistently better (*i.e.*, shorter runtime and smaller energy consumptions) than the other two in both aspects. Taking OD workload as an example, multi-budget results into a signifiant reduction in turning run time and energy (roughly 50%), compared to the budget of epoch and dataset. This suggests our novel approach can reach comparable inference results to state-of-the-art approaches while being more efficient.

5.3 Inference Tuning Server

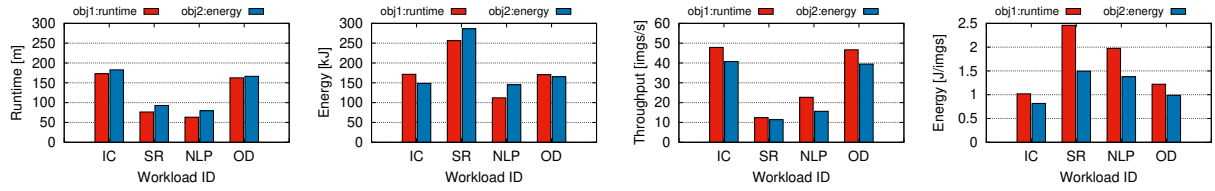
Next, we study the overhead and precision of the *Inference Tuning Server*, one of EdgeTune's key contributions.

Overhead. Figure 14 shows the overhead of the *Inference Tuning Server* component of EdgeTune with respect to Tune which does not have such a component integrated. For instance, we can observe that for both the workload IC and OD, the tuning duration and energy are reduced by 18% and 53%, respectively. This indicates that the performance gains achieved by considering a multi-objective optimization function compensates the minimal overhead of the *Inference Tuning Server* component. One of the objectives of the optimization function used in our implementation is to minimize the duration on training trials which indirectly also minimizes the overall tuning time.

Precision. Figure 15 uses a box-and-whiskers representation to show the percent error between the runtime and energy estimated by the *Inference Tuning Server* component and the actual metrics collected on an edge device of the simulated configurations. The percent error (*i.e.*, PE) is:

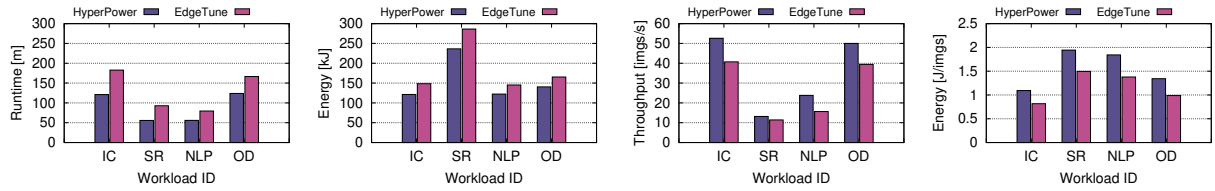
$$PE = \left(\frac{|empirical\_value - estimated\_value|}{empirical\_value} \right) x100,$$

where *empirical_value* is the metric collected on the edge device and *estimated_value* is the metric collected in simulation mode.



(a) Tuning duration (b) Tuning energy (c) Throughput (d) Inference energy

Figure 16: Impact of runtime v.s. energy based objective functions on the tuning efficiency and inference performance.



(a) Tuning duration (b) Tuning energy (c) Throughput (d) Inference energy

Figure 17: EdgeTune v.s. HyperPower: tuning efficiency and inference performance.

5.4 Objective Function: runtime *vs.* energy

Beyond model performance, runtime and energy performance are EdgeTune's main concerns. hence, we have implemented EdgeTune using two different versions of objective function, one optimized for runtime and one for energy, respectively. Figure 16 shows a comparison between the utilization of these two functions under the tuning and inference perspectives, respectively. Zooming into the workload IC and OD, we note that tuning time is slightly lower and the energy is higher for the function based on runtime than for the energy based function.

Moreover, when comparing the inference results of the configurations suggested in each case, both throughput and energy are higher for the runtime function than for the energy one. Although the differences are not significantly large (at most 20% for runtime and 29% for energy), we can notice the focus of the functions affecting the direction of the achieved results. We explain this by the fact that energy is often strongly correlated with runtime and, by optimizing runtime we are also indirectly optimizing energy consumption.

5.5 Other Tuning Systems

There are several state-of-the-art systems related to the approaches we propose in this paper. Here, we discuss in more details how they differ from each other from a conceptual point of view. Among those systems, we specifically chose *HyperPower* to perform a quantitative comparison with EdgeTune. We perform this evaluation in terms of tuning duration and energy as well as inference throughput and energy consumption of model resulting from the tuning phase. Please note that, different than EdgeTune, *HyperPower* does not output to the user any parameter regarding the inference phase. To make the inference comparison fair, we use the same parameters outputted by our approach in both cases. Figure 17 depicts these results. We observe that the tuning duration and energy of *HyperPower* are up to 39% and 33% lower than EdgeTune, respectively. The additional tuning duration and energy consumption stem from additional space of the inference configuration explored in EdgeTune. However, the inference results of EdgeTune are at least 12% higher in terms of throughput and 29% lower when it comes to energy consumption. The inference awareness characteristic of EdgeTune is reflected in these results, coming at a given tuning cost but achieving the overall objective which *HyperPower* is not designed to.

6 RELATED WORK

There is an exhaustive number of state-of-the art systems [14, 19, 48] covering the parameter tuning problem for machine learning and deep learning. Table 2 lists the most relevant ones to EdgeTune. We distinguish between systems that support CPU or GPU processing nodes, if they support hyper, system, or architecture parameters, if their objective focus is on tuning, training, or inference, and whether they exploit the multi-image inference aspect.

Parameters Tuning. In this context, tuning is the process of choosing optimal parameters for a given workload (*i.e.*, model and workload). The process of tuning parameters is crucial to find the best model performance of a given application. However, the best model performance can significantly vary depending on the performance definition, typically based on the interests of users. Moreover, the tuning process itself can take different perspectives into account such as searching algorithms [5–7], supported tools [39], and type of processing nodes (*e.g.*, GPUs, CPUs, FPGAs). Therefore, there are many proposed approaches and tools addressing this problem.

Inference-Aware Tuning. In the process of tuning, the users define their objective function which is often to maximize model performance in terms of accuracy [18, 40]. However, some users are also interested in model performance from other perspective such as the inference throughput. In this case, the tuning servers must

Table 2: State-of-the-art systems related to hyper and system parameter tuning.

System	CPU	GPU	Parameter			Objective			Multi-Sample Inference
			Hyper	System	Architecture	Tuning	Training	Inference	
ChamNet [14]	✓	✓	✗	✗	✓	✗	✓	✓	✗
DPP-Net [19]	✓	✓	✗	✗	✓	✗	✓	✓	✗
FBNet [48]	✓	✓	✗	✗	✓	✗	✓	✓	✗
HyperPower [45]	✗	✓	✓	✗	✓	✓	✓	✗	✗
MnasNet [46]	✓	✗	✗	✗	✓	✗	✓	✓	✗
NeuralPower [9]	✗	✓	✗	✗	✓	✓	✓	✗	✗
ProxylessNAS [10]	✓	✓	✗	✗	✓	✗	✓	✓	✗
EdgeTune	✓	✓	✓	✓	✓	✓	✓	✓	✓

be inference-aware, meaning that either empirical or estimated measurements of the inference runtime is included in the objective optimization function under consideration.

Multi-Objective Tuning. Multi-Parameter Tuning consists of tuning servers which simultaneously consider more than one dimension in their objective optimization functions [11, 26]. For instance, when both the model accuracy and inference throughput are equally relevant, then the optimization must follow a multiple criteria decision making strategy [36]. Sometimes these objectives are conflicting, which makes the decision making process not trivial and leading to multiple possible Pareto optimal solutions [43]. HyperPower leverages Bayesian optimization combined with some enhancements such as early termination of the training at the objective evaluation. Similarly to our approach, HyperPower also focus on optimizing the power consumption of the tuning process. However, no inference objective is taken into account for this approach as quantitatively demonstrated in our evaluation section.

Neural Architecture Search (NAS). Neural architecture search (NAS) [21] is a technique for automating the design of artificial neural networks (ANN), a widely used model in the field of machine learning. NAS has been used to design networks that are on par or outperform hand-designed architectures. Methods for NAS can be categorized according to the search space [38], search strategy [4, 33] and performance estimation strategy used [50]. EdgeTune also includes NAS since model hyperparameters are included in the tuning process which can also define the structure of the architecture. ChamNet, for instance, proposes Chameleon, an efficient neural network architecture design methodology. These systems differs from EdgeTune as they focus specifically in designing the model's architecture. The output also targets a specific type of network, performing very well in contrast to other state-of-the-art networks but lacking the generality proposed by our approach, specially in terms of inference awareness.

Multi-Image Inference. When the model has all parameters tuned, it is deployed to be used for inference on unseen data. Data arrives in batch or streaming form [49]. Although the most common approach for the inference phase is to apply the model for one data point at a time, this phase considers the hyperparameter *batch size* which determines how many data points can be aggregated before performing model inference. When the inference batch size is set to a value higher than one, then multi-image inference is being performed.

To the best of our knowledge, EdgeTune is the only system supporting CPUs, GPUs, parameter tuning, multi-objective tuning as well as multi-sample inference.

7 CONCLUSION

This paper presents the design, implementation and evaluation of EdgeTune, a novel edge-based parameter tuning framework that simultaneously considers multiple parameters and is inference-aware. To achieve the onefold tuning, EdgeTune is composed of tuning and inference servers which can asynchronously and in parallel explore large space of model and inference system parameters. By tuning multiple interlaced parameters (*i.e.*, model, hyper and system parameters), EdgeTune achieves higher model accuracy and energy-efficiency inference at a lower tuning cost, compared to the existing solutions, *e.g.*, non-inference aware tuning and hierarchical tuning. We design a novel multi-budget tuning algorithm that flexibly exploits the pros and cons of different budget types, overcoming the limitations of state of the art multi-fidelity tuning algorithms. We extensively evaluate EdgeTune on four popular AI workloads and three edge systems, against different combinations tuning solutions. The evaluation shows that EdgeTune can achieve 20% tuning improvement on runtime and 50% on energy. EdgeTune is available to the research community at https://github.com/isabellyrocha/edgetune.

REFERENCES

[1] 2022. *NVIDIA Graphic Cards.* Retrieved May 08, 2022 from https://www.nvidia.com/en-gb/graphics-cards/

[2] 2022. *PyRAPL.* Retrieved May 08, 2022 from https://github.com/powerapi-ng/pyRAPL

[3] Glenn Ammons, Thomas Ball, and James R. Larus. 1997. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI), Las Vegas, Nevada, USA, June 15-18, 1997*, Marina C. Chen, Ron K. Cytron, and A. Michael Berman (Eds.). ACM, 85–96. https://doi.org/10.1145/258915.258924

[4] Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. 2018. Accelerating Neural Architecture Search using Performance Prediction. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings.* OpenReview.net. https://openreview.net/forum?id=HJqk3N1vG

[5] Thomas Bartz-Beielstein and Sandor Markon. 2004. Tuning search algorithms for real-world applications: a regression tree based approach. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2004, 19-23 June 2004, Portland, OR, USA.* IEEE, 1111–1118. https://doi.org/10.1109/CEC.2004.1330986

[6] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for Hyper-Parameter Optimization. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems*

2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain, John Shawe-Taylor, Richard S. Zemel, Peter L. Bartlett, Fernando C. N. Pereira, and Kilian Q. Weinberger (Eds.). 2546–2554. https://proceedings.neurips.cc/paper/2011/hash/86e8f7ab32cfd12577bc2619bc635690-Abstract.html

[7] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-Parameter Optimization. *J. Mach. Learn. Res.* 13 (2012), 281–305. http://dl.acm.org/citation.cfm?id=2188395

[8] Léon Bottou. 2010. Large-Scale Machine Learning with Stochastic Gradient Descent. In *19th International Conference on Computational Statistics, COMPSTAT 2010, Paris, France, August 22-27, 2010 - Keynote, Invited and Contributed Papers*, Yves Lechevallier and Gilbert Saporta (Eds.). Physica-Verlag, 177–186. https://doi.org/10.1007/978-3-7908-2604-3_16

[9] Ermao Cai, Da-Cheng Juan, Dimitrios Stamoulis, and Diana Marculescu. 2017. NeuralPower: Predict and Deploy Energy-Efficient Convolutional Neural Networks. *CoRR* abs/1710.05420 (2017). arXiv:1710.05420 http://arxiv.org/abs/1710.05420

[10] Han Cai, Ligeng Zhu, and Song Han. 2018. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. *CoRR* abs/1812.00332 (2018). arXiv:1812.00332 http://arxiv.org/abs/1812.00332

[11] Maria G. Baldeon Calisto and Susana K. Lai-Yuen. 2020. AdaResU-Net: Multiobjective adaptive convolutional neural network for medical image segmentation. *Neurocomputing* 392 (2020), 325–340. https://doi.org/10.1016/j.neucom.2019.01.110

[12] Jie Cheng. 2010. CUDA by Example: An Introduction to General-Purpose GPU Programming. *Scalable Comput. Pract. Exp.* 11, 4 (2010). http://www.scpe.org/index.php/scpe/article/view/663

[13] Sven F. Crone, Stefan Lessmann, and Robert Stahlbock. 2006. The impact of preprocessing on data mining: An evaluation of classifier sensitivity in direct marketing. *Eur. J. Oper. Res.* 173, 3 (2006), 781–800. https://doi.org/10.1016/j.ejor.2005.07.023

[14] Xiaoliang Dai, Peizhao Zhang, Bichen Wu, Hongxu Yin, Fei Sun, Yanghan Wang, Marat Dukhan, Yunqing Hu, Yiming Wu, Yangqing Jia, Peter Vajda, Matt Uyttendaele, and Niraj K. Jha. 2019. ChamNet: Towards Efficient Network Design Through Platform-Aware Model Adaptation. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 11398–11407. https://doi.org/10.1109/CVPR.2019.01166

[15] Jonathan J. Davis and Andrew J. Clark. 2011. Data preprocessing for anomaly based network intrusion detection: A review. *Comput. Secur.* 30, 6-7 (2011), 353–375. https://doi.org/10.1016/j.cose.2011.05.008

[16] Shikhar Dev. 2020. MultiTune: Dynamic budget allocation for hyperparameter tuning. (2020).

[17] Ian Dewancker. 2016. *TensorFlow ConvNets on a Budget with Bayesian Optimization.* Retrieved May 08, 2022 from https://sigopt.com/blog/tensorflow-convnets-on-a-budget-with-bayesian-optimization/

[18] Alfonso Rojas Domínguez, Luis Carlos Padierna, Juan Martín Carpio Valadez, Héctor José Puga Soberanes, and Héctor J. Fraire H. 2018. Optimal Hyper-Parameter Tuning of SVM Classifiers With Application to Medical Diagnosis. *IEEE Access* 6 (2018), 7164–7176. https://doi.org/10.1109/ACCESS.2017.2779794

[19] Jin-Dong Dong, An-Chieh Cheng, Da-Cheng Juan, Wei Wei, and Min Sun. 2018. DPP-Net: Device-aware Progressive Search for Pareto-optimal Neural Architectures. *CoRR* abs/1806.08198 (2018). arXiv:1806.08198 http://arxiv.org/abs/1806.08198

[20] Sourav Dutta. 2018. An overview on the evolution and adoption of deep learning applications used in the industry. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8, 4 (2018).

[21] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural Architecture Search: A Survey. *J. Mach. Learn. Res.* 20 (2019), 55:1–55:21. http://jmlr.org/papers/v20/18-598.html

[22] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer G. Dy and Andreas Krause (Eds.). PMLR, 1436–1445. http://proceedings.mlr.press/v80/falkner18a.html

[23] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. 2017. Google Vizier: A Service for Black-Box Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*. ACM, 1487–1495. https://doi.org/10.1145/3097983.3098043

[24] Antonio Gulli. 2018. *AG's corpus of news articles.* Retrieved May 08, 2022 from http://groups.di.unipi.it/~gulli/AG_corpus_of_news_articles.html

[25] Karen Hao. 2019. Police across the US are training crimepredicting AIs on falsified data. *MIT Technology Review.* 13 (2019).

[26] Daniel Horn and Bernd Bischl. 2016. Multi-objective parameter configuration of machine learning algorithms using model-based optimization. In *2016 IEEE Symposium Series on Computational Intelligence, SSCI 2016, Athens, Greece, December 6-9, 2016*. IEEE, 1–8. https://doi.org/10.1109/SSCI.2016.7850221

[27] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[28] Haifeng Jin, Qingquan Song, and Xia Hu. 2019. Auto-Keras: An Efficient Neural Architecture Search System. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis (Eds.). ACM, 1946–1956. https://doi.org/10.1145/3292500.3330648

[29] Minsuk Kahng, Pierre Y. Andrews, Aditya Kalro, and Duen Horng (Polo) Chau. 2018. ActiVis: Visual Exploration of Industry-Scale Deep Neural Network Models. *IEEE Trans. Vis. Comput. Graph.* 24, 1 (2018), 88–97. https://doi.org/10.1109/TVCG.2017.2744718

[30] Kirthevasan Kandasamy, Gautam Dasarathy, Jeff G. Schneider, and Barnabás Póczos. 2017. Multi-fidelity Bayesian Optimisation with Continuous Approximations. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017 (Proceedings of Machine Learning Research, Vol. 70)*, Doina Precup and Yee Whye Teh (Eds.). PMLR, 1799–1808. http://proceedings.mlr.press/v70/kandasamy17a.html

[31] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[32] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *J. Mach. Learn. Res.* 18 (2017), 185:1–185:52. http://jmlr.org/papers/v18/16-558.html

[33] Petro Liashchynskyi and Pavlo Liashchynskyi. 2019. Grid Search, Random Search, Genetic Algorithm: A Big Comparison for NAS. *CoRR* abs/1912.06059 (2019). arXiv:1912.06059 http://arxiv.org/abs/1912.06059

[34] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. 2018. Tune: A Research Platform for Distributed Model Selection and Training. *CoRR* abs/1807.05118 (2018). arXiv:1807.05118 http://arxiv.org/abs/1807.05118

[35] Edo Liberty, Zohar S. Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, Can Balioglu, Saswata Chakravarty, Madhav Jha, Philip Gautier, David Arpin, Tim Januschowski, Valentin Flunkert, Yuyang Wang, Jan Gasthaus, Lorenzo Stella, Syama Sundar Rangapuram, David Salinas, Sebastian Schelter, and Alex Smola. 2020. Elastic Machine Learning Algorithms in Amazon SageMaker. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 731–737. https://doi.org/10.1145/3318464.3386126

[36] Jiguan Lin. 1976. Multiple-objective problems: Pareto-optimal solutions by method of proper equality constraints. *IEEE Trans. Automat. Control* 21, 5 (1976), 641–650.

[37] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2014. Microsoft COCO: Common Objects in Context. In *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V (Lecture Notes in Computer Science, Vol. 8693)*, David J. Fleet, Tomás Pajdla, Bernt Schiele, and Tinne Tuytelaars (Eds.). Springer, 740–755. https://doi.org/10.1007/978-3-319-10602-1_48

[38] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. DARTS: Differentiable Architecture Search. *CoRR* abs/1806.09055 (2018). http://arxiv.org/abs/1806.09055

[39] Francisco Madrigal, Camille Maurice, and Frédéric Lerasle. 2019. Hyper-parameter optimization tools comparison for multiple object tracking applications. *Mach. Vis. Appl.* 30, 2 (2019), 269–289. https://doi.org/10.1007/s00138-018-0984-1

[40] Alessandro Moschitti. 2003. A Study on Optimal Parameter Tuning for Rocchio Text Classifier. In *Advances in Information Retrieval, 25th European Conference on IR Research, ECIR 2003, Pisa, Italy, April 14-16, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2633)*, Fabrizio Sebastiani (Ed.). Springer, 420–435. https://doi.org/10.1007/3-540-36618-0_30

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[42] Isabelly Rocha, Nathaniel Morris, Lydia Y. Chen, Pascal Felber, Robert Birke, and Valerio Schiavoni. 2020. PipeTune: Pipeline Parallelism of Hyper and System Parameters Tuning for Deep Learning Clusters. In *Middleware '20:*

21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020, Dilma Da Silva and Rüdiger Kapitza (Eds.). ACM, 89–104. https://doi.org/10.1145/3423211.3425692

[43] Pradyumn Kumar Shukla and Kalyanmoy Deb. 2007. On finding multiple Pareto-optimal solutions using classical and evolutionary generating methods. *Eur. J. Oper. Res.* 181, 3 (2007), 1630–1652. https://doi.org/10.1016/j.ejor.2006.08.002

[44] Jacob Snow. 2018. Amazon's face recognition falsely matched 28 members of Congress with mugshots. *American Civil Liberties Union* 28 (2018).

[45] Dimitrios Stamoulis, Ermao Cai, Da-Cheng Juan, and Diana Marculescu. 2017. HyperPower: Power- and Memory-Constrained Hyper-Parameter Optimization for Neural Networks. *CoRR* abs/1712.02446 (2017). arXiv:1712.02446 http://arxiv.org/abs/1712.02446

[46] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V. Le. 2018. MnasNet: Platform-Aware Neural Architecture Search for Mobile. *CoRR* abs/1807.11626 (2018). arXiv:1807.11626 http://arxiv.org/abs/1807.11626

[47] Pete Warden. 2018. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *CoRR* abs/1804.03209 (2018). arXiv:1804.03209 http://arxiv.org/abs/1804.03209

[48] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. 2019. FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 10734–10742. https://doi.org/10.1109/CVPR.2019.01099

[49] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim M. Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. 2019. Machine Learning at Facebook: Understanding Inference at the Edge. In *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*. IEEE, 331–344. https://doi.org/10.1109/HPCA.2019.00048

[50] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. 2019. NAS-Bench-101: Towards Reproducible Neural Architecture Search. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 7105–7114. http://proceedings.mlr.press/v97/ying19a.html

[51] Guanghui Zhu and Ruancheng Zhu. 2020. Accelerating Hyperparameter Optimization of Deep Neural Network via Progressive Multi-Fidelity Evaluation. In *Advances in Knowledge Discovery and Data Mining - 24th Pacific-Asia Conference, PAKDD 2020, Singapore, May 11-14, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12084)*, Hady W. Lauw, Raymond Chi-Wing Wong, Alexandros Ntoulas, Ee-Peng Lim, See-Kiong Ng, and Sinno Jialin Pan (Eds.). Springer, 752–763. https://doi.org/10.1007/978-3-030-47426-3_58

[52] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2018. Learning Transferable Architectures for Scalable Image Recognition. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, 8697–8710. https://doi.org/10.1109/CVPR.2018.00907