Pipetune

Pipeline parallelism of hyper and system parameters tuning for deep learning clusters

Rocha, Isabelly; Morris, Nathaniel; Chen, Lydia Y.; Felber, Pascal; Birke, Robert; Schiavoni, Valerio

DOI
10.1145/3423211.3425692

Publication date
2020

Document Version
Final published version

Published in
Middleware 2020

PipeTune: Pipeline Parallelism of Hyper and System Parameters Tuning for Deep Learning Clusters

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

Nathaniel Morris
The Ohio State University
Columbus, Ohio
morris.743@buckeyemail.osu.edu

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

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

Robert Birke
ABB Research
Baden-Dättwil, Switzerland
robert.birke@ch.abb.com

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

Abstract

DNN learning jobs are common in today's clusters due to the advances in AI driven services such as machine translation and image recognition. The most critical phase of these jobs for model performance and learning cost is the tuning of hyperparameters. Existing approaches make use of techniques such as early stopping criteria to reduce the tuning impact on learning cost. However, these strategies do not consider the impact that certain hyperparameters and systems parameters have on training time. This paper presents PipeTune, a framework for DNN learning jobs that addresses the trade-offs between these two types of parameters. PipeTune takes advantage of the high parallelism and recurring characteristics of such jobs to minimize the learning cost via a pipelined simultaneous tuning of both hyper and system parameters. Our experimental evaluation using three different types of workloads indicates that PipeTune achieves up to 22.6% reduction and 1.7× speed up on tuning and training time, respectively. PipeTune not only improves performance but also lowers energy consumption up to 29%.

CCS Concepts • **Computing methodologies → Modeling methodologies**; *Cluster analysis*;

Keywords Parameter tuning, Deep Neural Networks training, accuracy time trade-off.

ACM Reference Format:
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 *21st International Middleware Conference (Middleware '20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3423211.3425692

1 Introduction

Deep Neural Networks (DNN) are becoming increasingly popular, both in academia and industry [16, 26]. They are being adopted across a variety of application domains, including speech [14, 37, 49] and image recognition [17], self-driving vehicles [23], face-recognition [54, 56], genetic sequence modeling [61], natural language processing [15], e-health [11] and more. Several public cloud providers offer native support to deploy, configure and run them, providing tools to automatically or semi-automatically drive the DNN processing pipeline. One important factor is 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.*). DNNs require careful tuning of the hyperparameters, and if done correctly, it can achieve impressive boosts in performance [4, 63]. However, misconfigurations can easily lead to wrong models and hence bad predictions [20, 53].

A naive approach to hyperparameter tuning is to perform a full exploration of the possible configuration variations. Such a tuning approach becomes quickly unpractical, costly and slow, as the number of variations grows exponentially [43]. We show this using 3 types of ML-optimized EC2 instances in Figure 1 for a small number of parameters. We take as example the tuning of a LeNet model on the MSNIT dataset and let it be tuned for different number of parameters (*i.e.*, varying from 1 to 6). In this case, each parameter was configured to take up to 3 different values. We measure the tuning time for each instance of this example and estimate the cost of doing so using a small, medium or large sized EC2 instance. We then observe that the cost of doing so grows

Middleware '20, December 7–11, 2020, Delft, Netherlands
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8153-6/20/12…$15.00
https://doi.org/10.1145/3423211.3425692

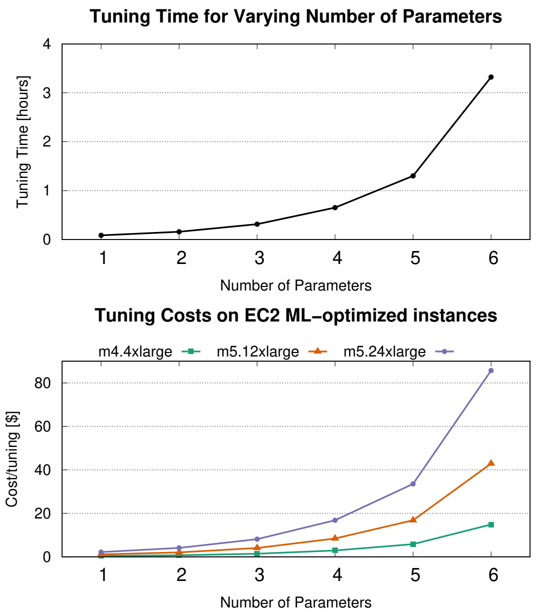


Figure 1. Clustering results grouped by workload type.

exponentially with the number of parameters being tuned, becoming impractical.

Commercial platforms (*i.e.*, Google Vizier [19], Amazon SageMaker [36]), as well as on-premises solutions (*i.e.*, Auto-Keras [25]) help deployers by offering tuning services to mitigate (possibly avoid) misconfiguration.

As a result of proper hyperparameters tuning, one should achieve fast convergence and high accuracy. Unfortunately, due to the tuning process length, this phase becomes expensive, and the situation exacerbates in cloud deployments [47]. Even using cheap cloud instances (*i.e.*, AWS EC2 Spot instances [6], as suggested for instance by AWS SageMaker [3]), the process can quickly lead to budget exhaustion.

We observe that some hyperparameters (*e.g.*, number of epochs, batch size, dropout) can drastically reduce training time. Importantly, training a DNN by using different system resources (*e.g.*, number of CPU cores, allocated memory, number of GPUs) lead to different results, as we also demonstrate later in Figure 3 for varying number of cores.

However, handling system parameters as one of the hyperparameters is very time consuming, requiring in-depth knowledge of the workload, and it is often an intuition-driven process. In addition, doing so would directly affect training and tuning time, and therefore state-of-the-art DNN tuning systems [8] simply ignore this opportunity. Instead, the majority of the existing tuning solutions restrict themselves to the sole hyperparameter tuning using a variety of techniques, including grid search [21], random search [9], hyperband [32], bayesian optimization [50, 52], evolutionary algorithms [55, 62], population-based training (PBT) [24], *etc.* While a possible yet naive approach to treat system parameters is to consider them as possible hyperparameters, this leads to longer training periods (see Table 2).

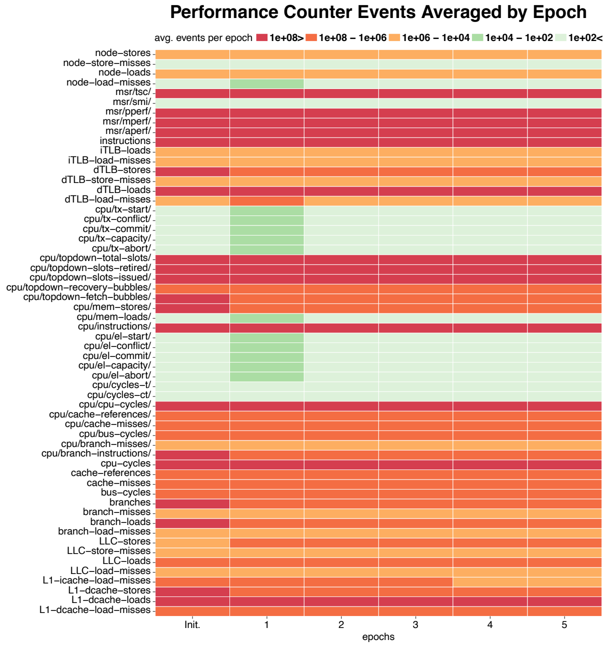


Figure 2. Profiling of training a CNN model on the NEWS20 dataset [1] during the initiation phase and the 5 following epochs with 16 cores and 32GB memory.

PIPETUNE strives to optimize both accuracy and training time of DNNs, while simultaneously tuning hyper and system parameters. The key observation of PIPETUNE is that the backbone of popular training algorithms for DNN is stochastic gradient decent [7], an iterative algorithm. PIPETUNE exploits such repetitive patterns as a unique opportunity to improve and achieve fast system parameter tuning. As an example, Figure 2 illustrates the typical repetitive behavior of a training process. We use a heatmap to show the hardware events happening through the training of a CNN model on the NEWS20 dataset [1] during 5 epochs. On the y-axis we show 58 different hardware events, on the y-axis initiation phase plus 5 training epochs. Each cell of the heatmap represents the average number of each event per single epoch. We see how certain events repeat throughput the epochs with the same occurrence.

Building on this observation, we design, implement and evaluate PIPETUNE, a middleware solution coordinating between the DNN training applications and systems. In a nutshell, PIPETUNE relies on low level metrics to profile the training trials on the epoch level and make quick decisions regarding the system parameters. The main research questions that PIPETUNE intends to answer, and the main contributions of this work are the following.

Table 1. State-of-the-art systems related to hyper and system parameter tuning.

System	CPU	GPU	Distributed	Training	Parameter Tuning		Supported DL Frameworks					Open Source
					Hyper	System	BigDL	TensorFlow	Keras	PyTorch	MXNet	
Astra [51]	✗	✓	✗	✓	✓	✗	✗	✓	✓	✗	✗	✗
AutoKeras [25]	✓	✓	✗	✓	✓	✓	✗	✓	✓	✗	✗	✓
ByteScheduler [46]	✓	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓
GRNN [45]	✓	✓	✗	✓	✗	✗	✗	✓	✗	✓	✗	✗
HyperDrive [48]	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗	✗
Hop [38]	✓	✗	✓	✓	✗	✗	✗	✓	✗	✗	✗	✗
Optimus [45]	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗
Orion [59]	✓	✗	✓	✓	✗	✗	✗	✓	✗	✗	✗	✓
Parallax [29]	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗	✗	✓
PipeDream [42]	✗	✓	✓	✓	✗	✗	✗	✓	✗	✗	✓	✓
SageMaker [36]	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗
STRADS [28]	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓
STRADS-AP [27]	✓	✗	✓	✓	✓	✗	✗	✓	✗	✗	✗	✗
Tune [35]	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗	✓
Vizier [19]	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗
PipeTune	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓

RQ1: *Why system parameters must be taken into account in the process of DNN tuning?*

We show (§ 3) that by taking into account the system parameters, the overall tuning runtime can be greatly reduced while at the same time improving the model performance. Moreover, the training time can at the same time benefit from this approach, especially if the underlying system resources and their usage is exposed to the tuning phase.

RQ2: *Can out-of-the-box hyperparameter optimization algorithms also take care of system parameters?*

We show that it is possible to include system parameters in the tuning process and ask the algorithm to optimize the ratio of accuracy to performance. However, our experimental evidences (§ 7) highlight the following drawbacks. First, tuning runtime significantly increases (up to ×1.5 in our experiments). Second, in doing so, the delicate equilibrium between performance and accuracy is negatively affected.

Roadmap. The reminder of this paper is organized as follows. We discuss related work and clarify how PipeTune positions in § 2. In § 3, we present a background of DNN tuning and outlines the basic features needed to support PipeTune. In § 4, we present an alternative approach relying on state-of-the-art solutions and show the need for our novel approach. We present the design of PipeTune in § 5 and describe its prototype implementation in § 6. In § 7, we present the results of our in-depth evaluation. Finally, we conclude in § 8.

2 Related Work

There is a large body of work behind machine learning in general, and parameter tuning more specifically. We survey the most prominent ones in Table 1. We distinguish between systems that support CPU or GPU processing nodes, if they can be deployed over a distributed cluster, if they support hyper or system parameters tuning. Finally, we identify what mainstream Deep Learning frameworks are natively supported by such systems. We distinguish between systems improving new techniques for training, others specifically optimized for hyperparameter tuning, and those focusing on system parameters tuning.

Improving training. GRNN [22] constructs a hybrid performance model that estimates the cost of a configuration according to the communication and computation needs. It ranks all the configurations and selects the first top-K to compile and run returning the fastest among them.

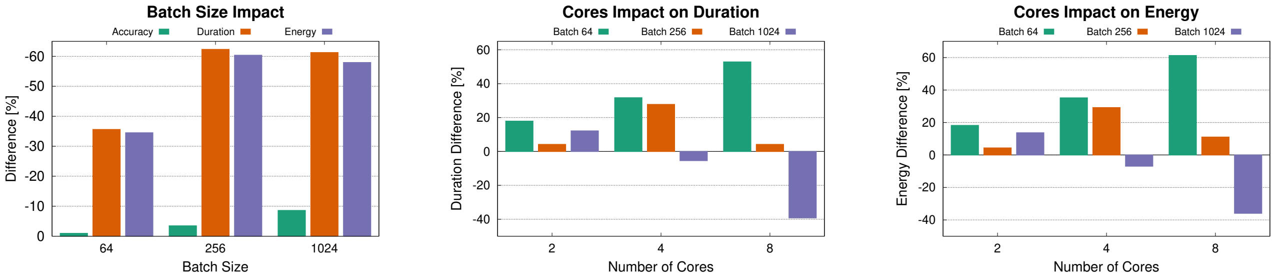
Hop [38] is a heterogeneity-aware decentralized training protocol. It relies on a queue-based synchronization mechanism that can implement backup workers and bounded staleness in a decentralized setting.

Optimus [45] uses online fitting to predict model convergence during training, and sets up performance models to estimate training speed as a function of allocated resources in each job. It estimates online how many more training epochs a job needs to run for convergence and how much time a job needs to complete one training epoch given a certain amount of resources. Speed model is computed based on a small sample set of training data, with possible combinations of parameter servers and workers.

Orion [59] performs static dependence analysis to determine when dependence-preserving parallelization is effective and map a loop computation to an optimized distributed computation schedule. It automatically parallelizes serial imperative ML programs on distributed shared memory.

Parallax [29] combines Hyperparameter Server [33] and AllReduce [39] architectures to optimize the amount of data transfers according to the data sparsity. It splits between a static phase for graphs with dense variables, and a sampling phase for fewer iterations.

PipeDream [42] combines inter-batch pipelining and intra-batch parallelism to improve parallel training throughput, helping to better overlap computation with communication and reduce when possible the amount of communication.



(a) Batch size impact on accuracy, runtime and energy. Baseline: batch size = 32.

(b) Cores impact on runtime per batch size. Baseline: sequential (*i.e.*, cores = 1).

(c) Cores impact on energy per batch size. Baseline: sequential (*i.e.*, cores = 1).

Figure 3. Impact of hyper and system parameters on accuracy, runtime and energy training of LeNet and MNIST workload.

These approaches focus on optimizing the training process, and can be combined with PipeTune to achieve further performance gains.

Hyperparameter tuning. As the process of tuning hyperparameters is, in most cases, crucial to find the best model performance of a given application, there are many proposed approaches and tools addressing this problem.

Astra [51] is a framework for online fine-grained exploration of the optimization state space in a work-conserving manner while making progress on the training trials.

STRADS [28] exposes parameter schedules and parameter updates as separate functions to be implemented by the user. A parameter schedule identifies a subset of parameters which a given worker should sequentially work on. STRADS-AP [27] extends STRADS to a distributed ML setting. These approaches leverage a runtime and API comprised of Distributed Data Structures (DDSs) and parallel loop operators.

AutoKeras [25] enables Bayesian optimization to guide the network morphism for efficient neural network architecture search. The framework develops a neural network kernel and a tree-structured acquisition function optimization algorithm to efficiently explore the search space. Similarly, Tune [35] provides a narrow-waist interface between training and search algorithms.

Finally, we mention two auto-tuning tools used in industry. HyperDrive [48] is a package part of Azure Machine Learning which supports hyperparameter tuning. It follows POP's scheduling algorithm which combines probabilistic model-based classification with dynamic scheduling and early stop techniques. Amazon SageMaker [36] is a fully managed machine learning service. It supports automatic model tuning component that finds the best version of a model by running many training trials on the dataset using the algorithm and ranges of hyperparameters specified by the user. Google Vizier [19] is an internal service used to optimize machine learning models and other systems. It also provides core capabilities to Google's Cloud Machine Learning HyperTune subsystem.

As our approach is an extension of pure hyperparameter tuning, the above mentioned systems and all others which focus on hyperparameter auto-tuning could profit from PipeTune.

System parameter tuning. ByteScheduler [46] is a Bayesian optimization approach. It specifically focuses on auto-tune tensor credit and partition size for different training models under various networking conditions. ByteScheduler uses auto-tune algorithms to find the optimal system related configurations. Instead, PipeTune allows the user to perform hyperparameter auto-tuning and finds the best system configurations independently of this process.

AutoKeras [25] supports a form of system parameter tuning, by means of an adaptive search strategy for different GPU memory limits. However, instead of adapting the system parameters to the workload, as we do in PipeTune, AutoKeras limits the size of the neural networks according to the GPU memory.

To the best of our knowledge, PipeTune is the first solution that efficiently combines hyper and system parameters in a holistic manner.

3 DNN Tuning: A primer

In this section, we discuss how hyperparameter tuning operates and explain why taking system parameters into account is beneficial. Then, §4 experimentally shows the benefits of our rationale.

3.1 Hyperparameters

A hyperparameter is a configuration external to the model. Its value cannot be estimated from data, it is set before the training starts, and does not change afterwards. Choosing the right hyperparameters during the *tuning* phase is key, as the output accuracy of the trained models can vary significantly. This phase is typically based on trial-and-error with model selection criteria. The complexity of this phase sparked several research efforts towards its automation and

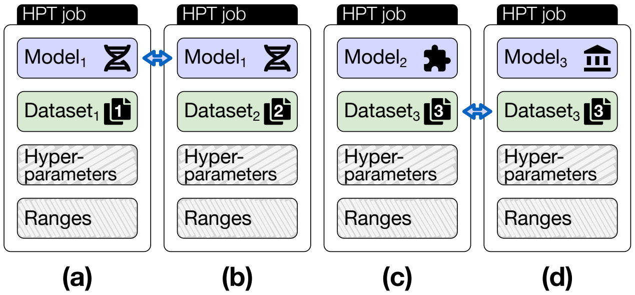


Figure 4. Workloads similarity of HPT jobs on the model and dataset level. Jobs (a) and (b) consist of the same model. Jobs (c) and (d) consist of the same dataset.

autotune frameworks [19, 25, 35]. As a result, hyperparameter optimization outputs a tuple of hyperparameters that yields an optimal model which minimizes a predefined loss function on given independent data [12].

Typically the selection criterion considered is model accuracy. However, the hyperparameters values will impact model accuracy, its training time and the energy footprint. The former is typically related to the utility of the trained model, the latter two to its costs. Figure 3a shows this behavior by reporting the impact of varying one hyperparameter (*i.e.*, batch size) for the training of a LeNet model [31] on the MNIST [30] dataset. On the y-axis we show the measured differences for accuracy, duration and energy observed for 3 possible batch size values (*i.e.*, 64, 256, and 1024), against a default value of 32.

We can observe how larger values of batch size achieve worse accuracy, but shorter training time and lower energy footprints. However, these observed trends might present considerable variations for different applications as it strongly depends on the workload and the values of the other hyperparameters. Therefore, these trade-offs are not trivially predicted, making it challenging to handle multidimensional selection criteria.

3.2 System Parameters

We define system parameters the configurable resources of the underlying computing infrastructure where the training will execute (*e.g.*, memory, CPU cores, CPU frequency). Typically, the hyperparameter optimization fixes the same system parameters for each trial, although they might benefit from different configurations. To highlight this, we train again a LeNet model on the MNIST dataset. We vary the number of CPU cores used with different batch sizes. Figure 3b and Figure 3c depict our findings. We observe how the number of cores is beneficial for larger batch size values (*e.g.*, 1024), but not for smaller ones. In fact, for smaller values (*e.g.*, 64) the runtime increases as the number of cores increases. This behavior is explained by the synchronous mini-batch stochastic gradient descent (SGD) algorithm used to train the neural network model. Each *N* iterations, SGD first computes the gradients using the current mini-batch, and then makes a single update to the weights of the neural network model. The *batch size* hyperparameter is divided by *N* to form these mini-batches, where *N* is the number of cores. When this value is too small, the overhead of model parameters synchronization is too high and ends up slowing down the training itself. This overhead can be amortized by using techniques such as the ones implemented by Drizzle [57] which schedules multiple iterations of computations at once, greatly reducing scheduling overheads even if there are a large number of tasks in each iteration [13].

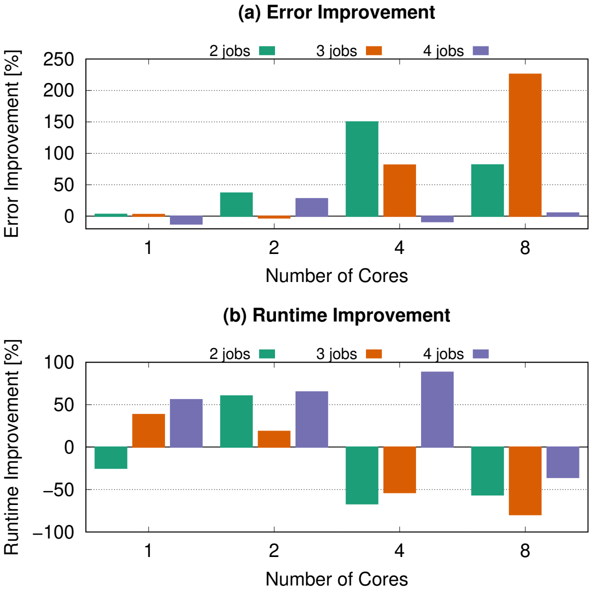


Figure 5. Characterizing Tune's performance under various system conditions (*i.e.*, system load, number of cores, and hyperparameters) during tuning.

Regarding the energy observations, we estimate the overall energy consumption of the cluster by calculating the trapezoidal integral of the power values collected every second during training. We observe a clear correlation between the energy variations (Figure 3c) and training runtime's gains (Figure 3b). These observations might however vary when the tuning is applied to different set of system parameters, *e.g.*, CPU frequency, or for different workloads.

In summary, these preliminary results show the delicate trade-offs between hyper and system parameters. One needs to balance them all towards optimal values, such that the underlying system achieves the best training performance without compromising the model accuracy.

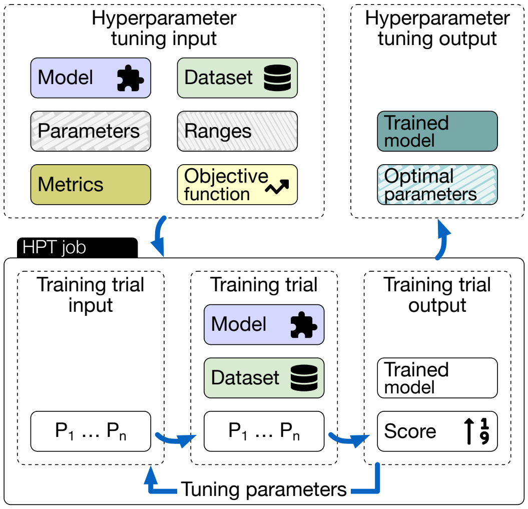


Figure 6. Hyperparameter tuning flow.

3.3 Workload

A workload is a tuple pairing a model and dataset. Typically, DNN workloads are used for training (*i.e.*, learning) or inference (*i.e.*, prediction). In this work, we only consider the training phase of DNN workloads. Moreover, we assume that this training phase includes parameters tuning on top of learning the weights of the model. Hence, tuning a single workload consists of multiple training trials, each divided into epochs. Each epoch involves one forward and one backward pass of the entire input dataset. For ease of processing, the dataset is split into smaller batches, and each batch is propagated forward and backward once during an epoch (*i.e.*, iteration). These mechanisms apply generally to all DNNs. It is a common practice to train the same model with different datasets, as well as different models using the same dataset. Figure 4 depicts this practice. Our approach leverages the similarity existent among such jobs to improve the tuning performance.

4 The "System *as* Hyperparameters" Case

The idea to consider system parameters as an additional set of hyperparameters is appealing. To verify its viability, we consider a state-of-the-art hyperparameter auto-tuning system, Tune [35], an open-source library implemented in Python supporting an extensive list of hyperparameters optimization algorithms. Note that the ideas shown next are nevertheless independent of the underlying tool used for the auto-tuning process of hyperparameters.

First, we consider two versions of Tune. In V1, it is used out-of-the-box to perform hyperparameters tuning with the objective of maximizing accuracy, without taking the system parameters into account. In this version all trials run with the same default system parameters. Then, in V2, the system parameters are included in the list of parameters to be tuned.

Table 2. Accuracy, training and tuning time taken by each considered approach for LeNet model on MNIST dataset.

Approach	Accuracy [%]	Training Time [s]	Tuning Time [s]
Arbitrary	84.47	445	-
Tune V1	91.54	272	4575
Tune V2	81.76	187	4817
PipeTune	92.70	188	3415

This second version requires the resources used by each trial to be manually controlled. Also, the objective function must be adapted to maximize the ratio accuracy to duration, rather than restricting it to accuracy only.

Figure 5 shows the results of Tune's performance characterization under various system conditions (*i.e.*, the number of cores assigned to the tuning job and the number of jobs assigned to the same logical cores). We used the V2 version of Tune to perform hyperparameter tuning. The tuning process was pinned to the same set of cores as the background jobs. For example, a configuration of 2 cores and 3 jobs meant a tuning job and 2 background jobs used the same 2 cores for execution. Figure 5 (a) illustrates the improvement in error relative to a single Tune V1 job. Figure 5 (b) is similar but shows training time improvement. Tuning under different system conditions significantly impacts the performance of the model being trained. There are only a few system configurations that yielded improvements over the baseline for error and training time. Some system configurations caused the tuning to trade better accuracy for faster training.

Hyperparameter tuning without system conditions can produce less efficient models. Table 2 shows the accuracy, training and tuning time achieved by different approaches for a LeNet model on MNIST dataset. These results show us the following. First, arbitrary values, if not correctly chosen, lead to both worse accuracy and training time. Second, if the user's focus is accuracy only, then PipeTune's accuracy results are comparable to Tune V1 however achieve in a lower tuning time. Third, if the user's focus is both accuracy and training time, then PipeTune's training time results are comparable to Tune V2 but with better accuracy and lower tuning time as well.

5 The PipeTune System

This section presents the system design of PipeTune. We begin clarifying the problem addressed by our system (§ 5.1). Then, we showcase its workflow (§ 5.2), the role of PipeTune's profiling phase (§ 5.3), the ground-truth phase (§ 5.4) and finally probing (§ 5.6).

5.1 Problem statement

One of the first challenges of applying deep learning algorithms in practice is to find the appropriated hyperparameter values for a given workload. We assume that most DNN

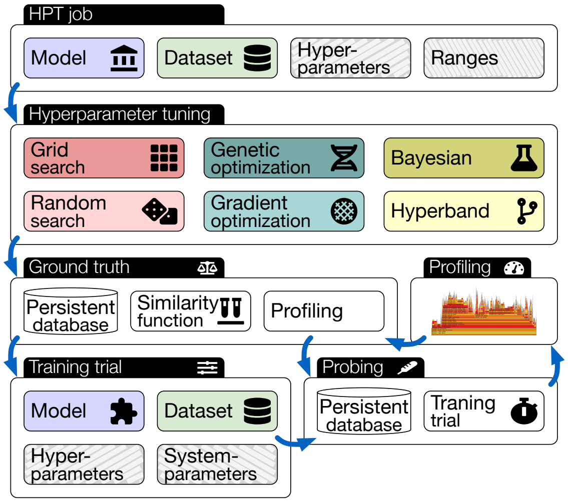


Figure 7. PipeTune architecture.

tuning jobs make use of some existing hyperparameter optimization solution. In the following we refer to these types of jobs as HPT Jobs (*i.e.*, Hyperparameters Tuning Jobs).

A given HPT Job takes as input a given workload, a set of parameters, its respective set of range values, an objective function and the metric of interest (*e.g.*, accuracy, performance, energy). This job spawns a collection of *training trials* based on the possible values of the parameters, following a given search algorithm (*e.g.*, GridSearch, HyperBand). Each *training trial* takes as input the workload and a set of fixed values for the parameters of interest, where these values belong to their respective given ranges. These trials can run either sequentially or in parallel depending on the setup. They produce a trained model and a score for the given parameters values. Scores correspond to the metric of interest defined by the user. The optimal set of parameters values is chosen by applying the objective function to the scores. Figure 6 illustrates this process.

We consider a deep learning cluster consisting of N nodes, each containing C cores and M GB of memory. Note that despite a common trend to include GPUs in DNN clusters, we explicitly put aside this option. We do this given the (rather small) nature of jobs on which we focus, for which commodity machines are sufficient for training. HPT Jobs are scheduled in a FIFO manner. We categorize these jobs in the following two main types: Type-I: tuning the same model for different datasets (*e.g.*, recommendation engines), and Type-II: tuning different models for the same dataset (*e.g.*, computer vision).

Both types of tuning jobs can still be divided into two sub-types: *(a)* same set of hyperparameters and ranges, and *(b)* same set of hyperparameters but different ranges. Each job, independent of its category, performs the earlier described tuning process from scratch. **A key observation is that these jobs could benefit from previously computed results for other jobs in the same category to converge faster**. Moreover, training trials spawned by the same HPT Job run all with the same system parameters even though they might require different resources configuration.

Algorithm 1: PipeTune algorithm.

```
Function train(model, data, hyperparameters):
    job = async model.train(data, hyperparameters);
    async tuneSystem(job);
    job.wait();
    return model;
Function tuneSystem(model, data):
    profile = getProfile(job);
    (score, config) = getSimilarity(profile);
    if score > threshold then
        setSystemParameters(config);
    else
        foreach $sp_v \in systemParameters$ do
            setSystemParameters($sp_v$);
            wait until epoch finishes;
            add collected metrics to $m$;
        bestConfig = find best config in $m$;
        setSystemParameters(bestConfig);
```

Another major limitation of the currently available approaches to hyperparameter auto-tuning is that only a single objective metric can be specified. This means that for a given HPT Job, one could choose to optimize either accuracy or performance, but not both simultaneously.

In summary, our problem's input consists of an HPT Job with the objective of achieving either maximum accuracy, or maximum accuracy with minimum training time. The former must output the best possible hyperparameters leading to the highest accuracy, independent of training time. For the latter, a combination of optimal hyper and system parameters is expected which leads to the highest accuracy and lowest training time. Note that for both scenarios, a shorter tuning times is beneficial, as allowed by our approach.

5.2 PipeTune Workflow

Figure 7 depicts the architecture components of PipeTune design and the main workflow. While training hyperparameters, a *trial* is a single training run with a fixed initial hyperparameter configuration. In order to find the best values for a given set of hyperparameters, the system executes a collection of trials, supervised by a given tuning library (*e.g.*, Vizier, Tune) and using one of the supported trial scheduling algorithms (*e.g.*, GridSearch, HyperBand).

PipeTune enhances the tuning of system parameters following a pipelined parallelism approach. That is, within each trial, a collection of sub-trials is executed, with the goal of defining the best system configurations for a given optimization function and metric of interest. This sub-trial consists of varying the system configuration on the epoch level and monitoring the system itself as well as the metrics of interest. The execution of sub-trials is controlled by PipeTune, which may also rely on different underlying scheduling algorithms.

Algorithm 1 details the pipelined approach. Function *train* (lines 1-5) is executed during a trial for a given workload (*i.e.*, model and dataset). After initiating the model training using the hyperparameter configuration given for that trial, *tuneSystem* (line 3) is invoked asynchronously.

The *profiling* phase (lines 7) is initiated for this given trial with the objective of characterizing the workload properties and its systems requirements. This process is done at the granularity of epochs for the currently running trial. We rely on kernel performance counters (*e.g.*, cpu cycles memory stores, instructions) to gather hardware events corresponding to low-level metrics of the underlying system.

Once this *profiling* phase is over, its outcome is used as input to a *ground truth* phase. This process consists of applying a similarity function (line 8) on the job's profile. This is done to reuse optimal configurations known by the system for other jobs with similar characteristics. If the score of this similarity function is within a specific confidence level (line 9), then the optimal known configurations are applied (line 10) and no further system metric trials are required. However, if the score does not cross the threshold, a new *probing* phase starts, searching the optimal system configurations for that trial.

The probing requires each system configuration to be applied for a different epoch, following a given scheduling algorithm. We collect several meaningful metrics (*e.g.*, runtime, energy) plus low-level metrics (*e.g.*, hardware events). Then the optimization function is applied over these metrics (line 16) to identify the overall best system configuration. This process consists of iterating over the collected values for each tuple of system parameters, looking for the one which best fits the optimization function (*e.g.*, shortest runtime, lowest energy consumption). The complexity of this search is $O(n)$, where n is the number of distinct system parameters considered. Finally, the configuration identified as optimal is applied for the remaining iterations (line 17) and saved for further improving of the *ground truth* phase.

5.3 Profiling

The profiling component leverages hardware performance counters to collect low-level events of the system during the applications execution time. After an initial experiment campaign, we gathered a comprehensive list of such events. As the number of events collected per time unit is limited by the number of actual hardware counters of the CPU, we filter out highly correlated as well as unsupported events. As result, our prototype deployed on x86 architectures current considers 58 measurable events, most of them being Performance Monitoring Unit (PMU) hardware events (*e.g.*, branch-instructions, cache-misses, cpu-cycles, mem-loads), reported by Linux's *perf* (v4.15.18). Although we have filtered the list of possible events to be collected, common Intel processors have only 2 generic and 3 fixed counters. Generic counters can measure any events while fixed counters can only measure one event.

When there are more events than counters (as it is in our case), then the kernel uses time multiplexing to give each event a chance to access the monitoring hardware. When this happens, an event might miss a measurement. If this happens, its occurrences are recomputed once the run ends, based on total time enabled *vs* time running [2], with:

$$final\_count = raw\_count * time\_enabled/time\_running.$$

This provides an estimate of what the count would have been, had the event been measured during the entire run.

Considering that the output value is not an actual count, depending on the workload, there might be blind spots which can introduce errors during scaling. Although we profile workloads at the epochs granularity, each epoch runs for at least a few minutes and we measure the events of interest every second. To mitigate the potential profiling errors, we store the average of results during each epoch's time window.

5.4 Ground Truth

During this phase, new incoming HPT Jobs exploit the ground truth results from historical data collected during the previously completed jobs with similar system characteristics, to accelerate their system-parameter tuning phases. Our design allows the similarity function to be pluggable, and while we do settle on k-means [58] in the current implementation, PipeTune allows to easily switch to alternative techniques.

The implementation of *ground truth* is done as a separate module which is used by PipeTune. In this module, the user can point to a pre-trained similarity function for a warm start or let the system build a new one from scratch. For this, our currently implementation relies on the *scikit-learn* machine learning library for Python [44] which already supports several clustering algorithms (*e.g.*, affinity propagation, mean-shift, DBSCAN, OPTICS, Birch). The exhaustive list of supported models are then inherited by PipeTune and could be easily used as alternative similarity functions.

Regarding the currently used model (*i.e.*, k-means), it is trained over the low-level system metrics collected during the profiling phase. The datasets are then partitioned into $k = 2$ groups (*i.e.*, model and dataset). Extensions to other values of k, as well as to other similarities dimensions (*e.g.*, hyperparameters, ranges) are left for future work.

Figure 8 shows clustering results using k-means grouped by model and dataset labeled with their respective cluster's

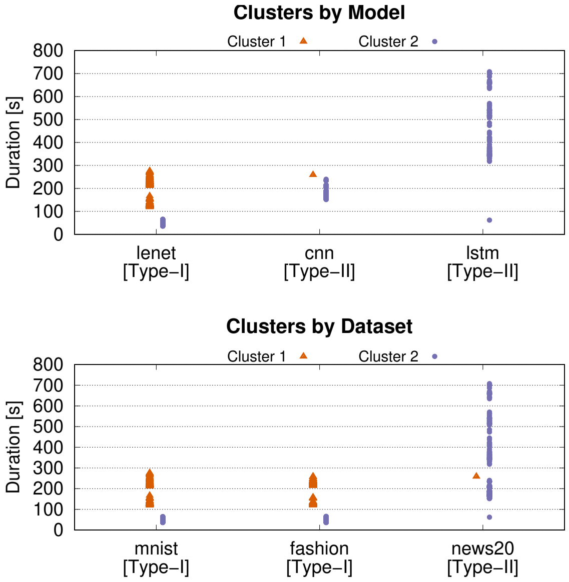


Figure 8. Clustering results grouped by workload type.

Table 3. Workloads used for experiments.

	Model	Dataset	Datasize	Train Files	Test Files
Type-I	LeNet5	MNIST	12 MB	60 000	10 000
	LeNet5	Fashion-MNIST	31 MB	60 000	10 000
Type-II	CNN	News20	15 MB	11 307	7538
	LSTN	News20	15 MB	11 307	7538
Type-III	Jacobi	Rodinia	26 MB	1650	7538
	SPK-means	Rodinia	26 MB	1650	7538
	BFS	Rodinia	26 MB	1650	7538

labels. We can observe that the majority of data fits into Type-I and Type-II are labeled as *cluster1* and *cluster2*, respectively. This result supports our assumptions regarding workloads similarities and shows that the chosen profiling technique can also capture the implicit characteristics of each workloads. Finally, it shows that the clustering algorithm utilized can identify the similarities present in those characteristics and efficiently cluster them.

5.5 Privacy concerns

Although the *ground truth* component of PipeTune makes use of historical data, it does not require any information regarding the users' workloads (*i.e.*, model or dataset). Instead, this process relies entirely on system events collected using the hardware performance counters. This profiling based on low level metrics allows PipeTune to characterize the applications while preserving user data privacy (*e.g.*, user parameters like model and dataset are not revealed). We assume that potential data, model and parameters similarities between workloads will affect the collected metrics in the same ways and therefore also be reflected in the similarity function. The results observed in Figure 8 supports this assumption.

5.6 Probing

The probing phase profiles a given set of workloads in different system conditions, in order to collect sufficient data for a warm start of the *ground truth* component. In practice, the *ground truth* model is refined as the similarity of the incoming jobs with the historical data of the system starts to decrease. When this happens, we launch a grid search on the system-parameters at the epoch granularity, yet other search strategies are possible. In this case, the tuning of system parameters for the current job is performed directly on the analytical data collected. Moreover, this collected data is saved to be taken into account once re-clustering is applied.

We decide upon the necessity to launch a new probing or not for a given workload based on the similarity score outputted from the ground truth phase. When using k-means, the threshold matches the distance from the new set of data points to their current cluster's centroid. The distance is compared against the models' inertia, to measure the reliability of the prediction, or else if a re-clustering is needed.

6 Implementation

PipeTune is implemented in Python (v3.5.2) and it consists of 947 LOC. We leverage two open-source projects, namely Tune and BigDL. Tune [35] is a Python library for hyperparameter search, optimized for deep learning and deep reinforcement learning [34]. Tune provides several trial schedulers based on different optimization algorithms. While we select HyperBand for the reminder of this work, Tune allows to switch among the available ones, as well as to implement new ones. As a consequence, PipeTune indirectly supports all its hyperparameter optimization algorithms.

The training applications are executed by BigDL [13], a distributed deep learning framework on top of Apache Spark. BigDL supports TensorFlow and Keras, hence PipeTune supports models defined using such frameworks. The Ground Truth module is based on a battle-tested k-means implementation openly available in the *scikit-learn* machine learning library for Python [44].

Finally, as storage backend, we leverage `InfluxDB` (v1.7.4), an open-source time series database. It offers a convenient `InfluxDB-Python` client for interacting with InfluxDB which we use to query information regarding the collected system metrics. PipeTune is released as open-source [1].

7 Evaluation

This section presents our in-depth evaluation of PipeTune using real-world datasets. Our main findings are:

1. PipeTune achieves significant tuning speedups without affecting model performance (*i.e.*, accuracy);

[1] https://github.com/isabellyrocha/pipetune

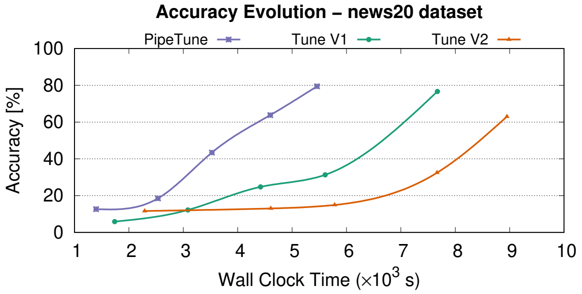


Figure 9. Accuracy convergence.

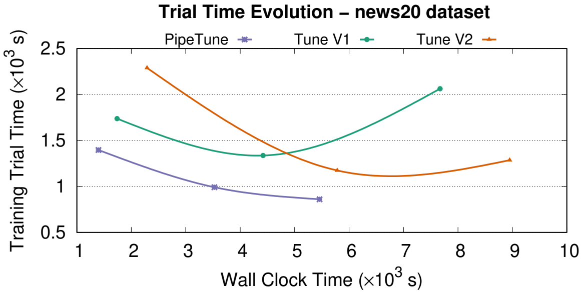


Figure 10. Training trial time convergence.

2. By speeding up the tuning process, we also have a more energy efficient approach, not only due to the runtime reduction but also because of the more efficient utilization of system resources;
3. The proposed approach is sensitive to varying system loads as this is also reflected on the events used to profile and our system adapts on a fine granularity (*i.e.*, epochs level).

7.1 Experimental Setup

7.1.1 Testbed

We deploy our experiments using Type-I and Type-II workloads on a cluster of 4 quad-socket Intel E3-1275 CPU processors with 8 cores per CPU, 64 GiB of RAM and 480 GB SSD drives. Experiments involving Type-III workloads are deploy on a single node containing an Intel E5-2620 with 8 cores, 24 GB of RAM and a 1 TB HDD. All machines run Ubuntu Linux 16.04.1 LTS on a switched 1 Gbps network. Power consumptions are reported by a network connected LINDY iPower Control 2x6M Power Distribution Unit (PDU), which we query up to every second over an HTTP interface to fetch up-to-date measurements for the active power at a resolution of 1W and 1.5% precision.

7.1.2 Workloads

We consider 7 state-of-the-art deep learning workloads for image classification, LLC-Cache computational sprinting and natural language processing. Table 3 summarizes their details.

LeNet5 [31] is a convolutional network for handwritten and machine-printed character recognition. Convolutional Neural Networks (CNNs) [40] are a special kind of multi-layer neural networks, trained via back-propagation. CNNs can recognize visual patterns directly from pixel images with minimal preprocessing. Long Short-Term Memory (LSTMs) [18] are artificial Recurrent Neural Networks (RNNs) architectures used to process single data points (such as images, connected handwriting recognition and speech recognition), as well as sequences of data (*i.e.*, speech, videos). Finally, Jacobi is a differential numerical solver, BFS is breath-first-search and spk-means is k-means implemented on top of Spark framework.

The MNIST dataset [30] of handwritten digits has a training set of 60 000 examples, and a test set of 10 000 examples. The digits have been size-normalized and centered in a fixed-size image. Fashion-MNIST dataset [60] is a dataset of article images consisting of a training set of 60 000 examples and a test set of 10 000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. Fashion-MNIST shares the same image size and structure of training and testing splits as the original MNIST dataset. The News20 dataset [1] is a collection of 20 000 messages collected from 20 different netnews newsgroups. We sample uniformly at random 1000 messages from each newsgroup, and we partition them by name. The Rodinia Benchmark Suite [10] is a collection of profiling short-term resource allocation (*i.e.*, computational sprinting) policies which targets heterogeneous computing platforms with both multicore CPUs and GPUs. These workloads have the objective to classify or predict the original data reserved for testing purposes.

7.1.3 Hyperparameters

There are several potential hyperparameters to tune. For practical reasons, in our evaluation we select the 5 described below. Note that their recommended range is typically application-driven, and we settle on specific values without however generalizing for any workload.

1. **Batch size.** Number of samples to work through before updating the internal model parameters. Large values for batch size have a negative effect on the accuracy of network during training, since it reduces the stochasticity of the gradient descent. Range: [32 - 1024].
2. **Dropout rate.** Dropout randomly selects neurons to be ignored during training. Dropout layers are used in the model for regularization (*i.e.*, modifications intended to reduce the model's generalization error without affecting the training error). The dropout rate value defines the fraction of input to drop to prevent overfitting [41]. Range: [0.0 – 0.5].

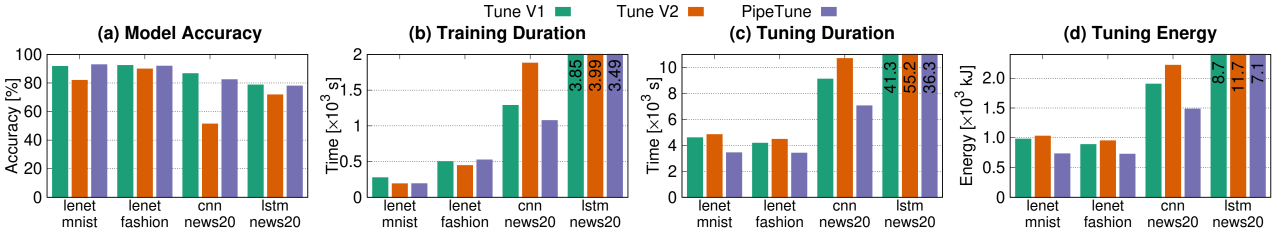


Figure 11. Evaluation of PipeTune's accuracy, performance and energy consumption for Type-I and Type-II Jobs.

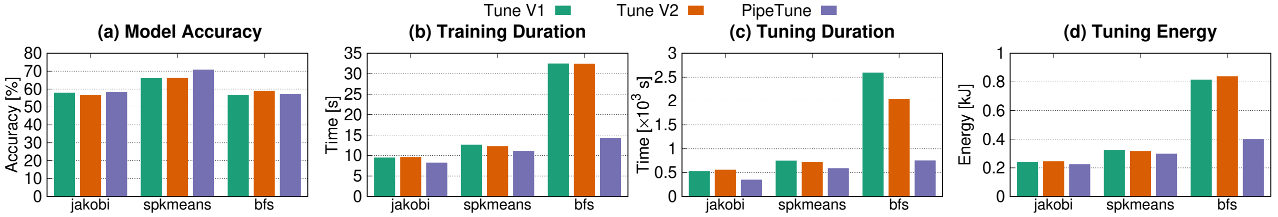


Figure 12. Evaluation of PipeTune's accuracy, performance and energy consumption for Type-III Jobs.

3. **Embedding dimensions.** Word embeddings provide a mean of transfer learning. This mechanism can be controlled by having word vectors fine-tuned throughout the training process. Depending on the dataset size on which word embeddings are being refined, updating them might improve accuracy [5]. Range: [50 – 300].
4. **Learning rate.** Rate at which the neural network weights change between iterations. A large learning rate may cause large swings in the weights, making impossible to find their optimal values. Low learning rates requires more iterations to converge. Range: [0.001 - 0.1].
5. **Number of epochs** Number times that the learning algorithm will work through the entire training dataset. Typically, larger number of epochs yields in longer runtimes but also higher training accuracy. However, the number of epochs required to achieve a given minimum desired accuracy depends on the workload. Range: [10 - 100].

7.1.4 System Parameters

For the purpose of this evaluation, we restrict the list of parameters to number of cores and memory. However, the same mechanisms can be applied to any other parameter of interest (*e.g.*, CPU frequency, CPU voltage). In our cluster, the ranges of valid values for system parameter tuning are [4 - 16] and [4 - 32] (GB) for for number of cores and memory, respectively.

7.1.5 Baselines

Baseline I: hyperparameters tuning. Our first baseline system (*i.e.*, Tune V1) uses the tuning of hyperparameters ignoring any system parameter. We rely on HyperBand for the parameter optimization with the objective function set to maximize accuracy.

Baseline II: system and hyper parameters tuning. We further compare against Tune V2, where we include the list of system parameters to be considered in the list of parameters to be tuned by the HyperBand algorithm. We also include the training duration as part of the optimization function which in this baseline is set to maximize the ratio accuracy to duration (details in § 4).

7.2 Convergence Evolution

In order to build our initial similarity model we rely on profiling data of the workloads described in Table 3. For each workload, we vary the system configurations as follows. Memory allocation can be 4GB, 8GB, 16GB, and 32GB. The total number of cores that could be allocated were 4, 8, or 16. Finally, batch size could take the values 32, 64, 512, or 1024. In total, this sums up to 48 different configurations for each workload. There is no reason to expect variations in the data collected from different training instances using the exact same parameters. However, we repeat this process twice for each configuration to make sure that the achieved model is not affected by potential unseen variations.

We begin our evaluation by analyzing the convergence trajectory of PipeTune compared to Tune V1 and Tune V2. Figure 9 illustrates the accuracy evolution of the training trials over the tuning time of a CNN model on the News20 dataset. We observe that PipeTune converges to an accuracy value comparable to Tune V1 but at a much faster rate. For

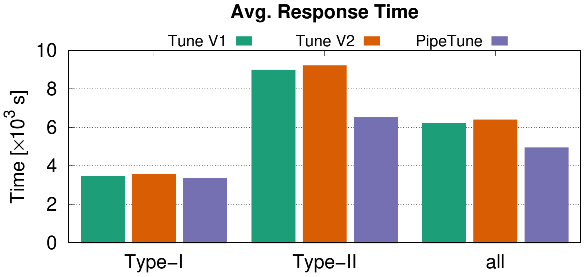


Figure 13. Average response time for Type-I and Type-II Jobs considered independently and all together.

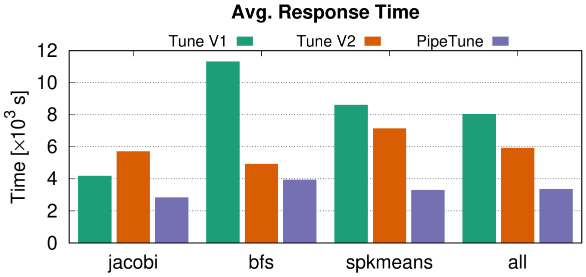


Figure 14. Average response time for Type-III Jobs.

instance, PipeTune reaches a 60% accuracy after approximately 4500 seconds. On average our approach is 1.5× and 2× faster than Tune V1 and Tune V2, respectively.

The training time achieved shows similar behavior (see Figure 10). Interestingly, Tune V1 performs worse than Tune V2. Since Tune V1 optimizes only for accuracy, the most accurate model not necessarily achieves the shortest training time. On the other hand, as Tune V2 optimizes for the ratio accuracy to performance, the accuracy achieved might not be the highest possible. However, the training time in the given configurations might be lower (which is exactly what happens in this instance of the problem). Finally, we observe that PipeTune consistently presents shorter trial times than the other two approaches during the entire tuning process.

7.3 Single-Tenancy

We now consider a single-tenancy scenario, and assume each HPT Job runs in a dedicated cluster, where the required resources demanded by the system parameters are available and exclusive for a given tenant. This prevents interference caused by other jobs co-located on the same cluster. However, as a given HPT Job spawns several *training trials* asynchronously, the cluster still remains shared among these sub jobs. We evaluate how PipeTune performs in such stable setting, comparing it against Tune V1 and Tune V2, for all the workloads.

Comparison with baseline. Figure 11 presents the results of model accuracy, training and tuning runtime, and overall cluster energy consumption of offline HPT Jobs for the different workloads described in Table 3.

Figure 11 (a) presents the accuracy results. We can observe that the accuracy of PipeTune is not affected by the performance optimization. In fact, results are on par with Tune V1, where hyperparameters tuning is done with the only objective of maximizing accuracy. As expected, Tune V2 decreases accuracy up to 43%, since the objective function no longer tries to optimize accuracy but also takes the runtime into account.

Figure 11 (b) shows the training time of the achieved model. In this case, PipeTune presents comparable results to the baseline. In fact, we observe up to 1.7× speed-up in comparison with Tune V2 which focuses exactly in reducing training runtime. We observe that Tune V2 increases tuning duration by up to 18% when compared to Tune V1. This happens for the following two reasons. First, the search space of Tune V2 is larger than of Tune V1, as it includes the system-parameters. Second, the optimization function consists of accuracy and runtime together. These two reasons make it harder for the search algorithm to find the optimal set of configurations, hence longer tuning times are observed.

On the other hand, PipeTune reduces tuning runtime by at least 18% when compared against Tune V1, as shown in Figure 11 (c). This performance gain is obtained because the search space and optimization function remains the same, and at the same time PipeTune finds and applies during runtime the optimal system configurations for each trial. Moreover, all the additional steps introduced by PipeTune are done in parallel, without impacting the hyperparameters tuning process.

Figure 11 (d) reports the energy results. The overall energy consumption of the cluster is directly affected both by the performance decays and gains. Compared against Tune V1, we observe up to 22% energy increase for Tune V2 and up to 29% energy decrease for PipeTune.

Figure 12 compares Tune V1, Tune V2 and PipeTune on a single node. The Type-III workloads used in these experiments have shorter epochs and each a different CNN model. Previous experiments deploy PipeTune on workloads with epochs lasting minutes. Long epochs work in favor of PipeTune since low-overhead profiling is performed across the first couple of epochs to classify new workloads. Therefore, next we perform an extra analysis on Type-III Jobs which present this more challenging setup for PipeTune to observe how it behaves.

Figure 12 (a-d) plots the same metrics as seen in Figure 11. The goal is to test how well PipeTune can improve tuning for workloads with short but many epochs per trial. Here we can observe that PipeTune also achieves the expected results in this more challenging scenario and reduces both training and tuning time when compared to the baseline systems. Regarding model accuracy, we can also see that our approach achieves comparable or better results than the baseline. Finally, the energy results reflects the performance gains resulting in a more energy efficient approach as well.

To summarize, for these single-tenancy scenarios, PipeTune presents better performance with up to 23% reduction on tuning time, is more energy efficient reducing up to 29% the overall energy consumption of the utilized cluster, and does not affect model accuracy as the observed differences in this aspect are negligible.

Profiling overhead. Profiling is a fundamental part of our system design and essential for the decision making process. During the profiling of a given epoch, the extra computation introduce additional load, depending on the system configuration. However, as this profiling overhead only occurs in the epoch granularity and does not apply for all the epochs, the performance benefits resulting from tuning the system-parameters overtake the measured overhead. The experimental results presented above also support these assumptions as, otherwise, we would not observe performance gains when compared with the approaches Tune V1 and Tune V2 which do not perform any profiling.

7.4 Multi-Tenancy

Next, we evaluate PipeTune in a multi-tenancy scenario (*i.e.*, a shared cluster handling multiple HPT Jobs). In this case, we show the average response time of jobs as an indicator of performance. We consider that jobs arrive randomly with the interarrival times being exponentially distributed. For the case where two workload types are considered together, each of them corresponds to 50% of the overall jobs (i.e., equally balanced). In all cases, within a given workload type, the workloads are chosen following a round-robin strategy. The portion of overall unseen jobs corresponds to 20%.

Figure 13 shows the results for the multi-tenancy scenario considering workloads of Type-I and Type-II grouped by type as well as the overall results. As in Section 7.3, this evaluation has been performed in a distributed environment. In this experiment we observe improvements similar to the ones in the single-tenancy scenario. Regarding response time, PipeTune results in up to 30% reduction when compared with Tune V1 and Tune V2.

Figure 14 shows the same results described above but considering workloads of Type-III. This trace was executed in a single node in contrast with the distributed environment of the previously described results. In this specific scenario we observe that the performance gain trends earlier observed becomes even more evident in such environment and workload type. In this case, PipeTune results in up to 65% reduction on the average response time in comparison with Tune V1 and Tune V2. This indicates that the overhead of computation added for the unseen jobs is compensated by the gain of future similar incoming ones.

8 Conclusion

The combination of hyper and system parameter for Deep Neural Network tuning is an overlooked opportunity that many state-of-the-art tuning solutions ignore. This paper presented PipeTune, an open-source system that leverages the repetitive behaviour of DNN tuning jobs to quickly find the best set of parameters. Our approach is modular which makes it easy to swap between similarity functions and underlying search algorithms. We evaluated 7 different real-world datasets from different domains, including text classification and image recognition. When compared against state-of-the-art DNN tuning systems, PipeTune shows experimental evidence that the approach greatly reduces tuning and training time while achieving on-par accuracy.

Acknowledgments

We would like to thank our shepherd, Prof. Indranil Gupta, for his helpful feedback. The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under the LEGaTO Project (legato-project.eu), grant agreement No 780681. This research was partly funded by the SNSF NRP75 project Dapprox 407540_167266.

References

[1] 2020. 20 Newsgroups. http://qwone.com/~jason/20Newsgroups. Accessed: 2020-14-09.

[2] 2020. Linux kernel profiling with perf. https://perf.wiki.kernel.org/index.php/Tutorial. Accessed: 2020-14-09.

[3] 2020. Perform Automatic Model Tuning. https://docs.aws.amazon.com/sagemaker/latest/dg/automatic-model-tuning.html. Accessed: 2020-14-09.

[4] 2020. TensorFlow ConvNets on a Budget with Bayesian Optimization. https://sigopt.com/blog/tensorflow-convnets-on-a-budget-with-bayesian-optimization/.

[5] Ahmad Aghaebrahimian and Mark Cieliebak. 2019. Hyperparameter Tuning for Deep Learning in Natural Language Processing. In *Proceedings of the 4th edition of the Swiss Text Analytics Conference, SwissText 2019, Winterthur, Switzerland, June 18-19, 2019 (CEUR Workshop Proceedings)*, Mark Cieliebak, Don Tuggener, and Fernando Benites (Eds.), Vol. 2458. CEUR-WS.org. http://ceur-ws.org/Vol-2458/paper5.pdf

[6] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. 2013. Deconstructing Amazon EC2 Spot Instance Pricing. *ACM Trans. Economics and Comput.* 1, 3 (2013), 16:1–16:20. https://doi.org/10.1145/2509413.2509416

[7] Yoshua Bengio. 2000. Gradient-Based Optimization of Hyperparameters. *Neural Computation* 12, 8 (2000), 1889–1900. https://doi.org/10.1162/089976600300015187

[8] 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. http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization

[9] 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

[10] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*. IEEE Computer Society, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[11] Dan C. Ciresan, Alessandro Giusti, Luca Maria Gambardella, and Jürgen Schmidhuber. 2013. Mitosis Detection in Breast Cancer Histology Images with Deep Neural Networks. In *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2013 - 16th International Conference, Nagoya, Japan, September 22-26, 2013, Proceedings, Part II (Lecture Notes in Computer Science)*, Kensaku Mori, Ichiro Sakuma, Yoshinobu Sato, Christian Barillot, and Nassir Navab (Eds.), Vol. 8150. Springer, 411–418. https://doi.org/10.1007/978-3-642-40763-5_51

[12] Marc Claesen and Bart De Moor. 2015. Hyperparameter Search in Machine Learning. *CoRR* abs/1502.02127 (2015). arXiv:1502.02127 http://arxiv.org/abs/1502.02127

[13] Jason Jinquan Dai, Yiheng Wang, Xin Qiu, Ding Ding, Yao Zhang, Yanzhang Wang, Xianyan Jia, Cherry Li Zhang, Yan Wan, Zhichao Li, Jiao Wang, Shengsheng Huang, Zhongyuan Wu, Yang Wang, Yuhao Yang, Bowen She, Dongjie Shi, Qi Lu, Kai Huang, and Guoqiong Song. 2019. BigDL: A Distributed Deep Learning Framework for Big Data. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 50–60. https://doi.org/10.1145/3357223.3362707

[14] Li Deng, Geoffrey E. Hinton, and Brian Kingsbury. 2013. New types of deep neural network learning for speech recognition and related applications: an overview. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*. IEEE, 8599–8603. https://doi.org/10.1109/ICASSP.2013.6639344

[15] Li Deng and Yang Liu. 2018. *Deep learning in natural language processing*. Springer.

[16] 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).

[17] Dumitru Erhan, Christian Szegedy, Alexander Toshev, and Dragomir Anguelov. 2014. Scalable Object Detection Using Deep Neural Networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*. IEEE Computer Society, 2155–2162. https://doi.org/10.1109/CVPR.2014.276

[18] Felix A. Gers, Jürgen Schmidhuber, and Fred A. Cummins. 2000. Learning to Forget: Continual Prediction with LSTM. *Neural Comput.* 12, 10 (2000), 2451–2471. https://doi.org/10.1162/089976600300015015

[19] 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

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

[21] Geoffrey E. Hinton. 2012. A Practical Guide to Training Restricted Boltzmann Machines. In *Neural Networks: Tricks of the Trade - Second Edition*, Grégoire Montavon, Genevieve B. Orr, and Klaus-Robert Müller (Eds.). Lecture Notes in Computer Science, Vol. 7700. Springer, 599–619. https://doi.org/10.1007/978-3-642-35289-8_32

[22] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. 2019. GRNN: Low-Latency and Scalable RNN Inference on GPUs. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, George Candea, Robbert van Renesse, and Christof Fetzer (Eds.). ACM, 41:1–41:16. https://doi.org/10.1145/3302424.3303949

[23] Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue, Fernando A. Mujica, Adam Coates, and Andrew Y. Ng. 2015. An Empirical Evaluation of Deep Learning on Highway Driving. *CoRR* abs/1504.01716 (2015). arXiv:1504.01716 http://arxiv.org/abs/1504.01716

[24] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. 2017. Population Based Training of Neural Networks. *CoRR* abs/1711.09846 (2017). arXiv:1711.09846 http://arxiv.org/abs/1711.09846

[25] 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

[26] 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

[27] Jin Kyu Kim, Abutalib Aghayev, Garth A. Gibson, and Eric P. Xing. 2019. STRADS-AP: Simplifying Distributed Machine Learning Programming without Introducing a New Programming Model. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 207–222. https://www.usenix.org/conference/atc19/presentation/kim-jin

[28] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A. Gibson, and Eric P. Xing. 2016. STRADS: a distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, Cristian Cadar, Peter R. Pietzuch, Kimberly Keeton, and Rodrigo Rodrigues (Eds.). ACM, 5:1–5:16. https://doi.org/10.1145/2901318.2901331

[29] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. 2019. Parallax: Sparsity-aware Data Parallel Training of Deep Neural Networks. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, George Candea, Robbert van Renesse, and Christof Fetzer (Eds.). ACM, 43:1–43:15. https://doi.org/10.1145/3302424.3303957

[30] Yann LeCun. 1998. The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/* (1998).

[31] Yann LeCun et al. 2015. LeNet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet* 20 (2015), 5.

[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] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 583–598. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu

[34] Yuxi Li. 2017. Deep Reinforcement Learning: An Overview. *CoRR* abs/1701.07274 (2017). arXiv:1701.07274 http://arxiv.org/abs/1701.07274

[35] 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

[36] 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

[37] Alicia Lozano-Diez, Ruben Zazo, Doroteo T Toledano, and Joaquin Gonzalez-Rodriguez. 2017. An analysis of the influence of deep neural network (DNN) topology in bottleneck feature based language recognition. *PloS one* 12, 8 (2017).

[38] Qinyi Luo, Jinkun Lin, Youwei Zhuo, and Xuehai Qian. 2019. Hop: Heterogeneity-aware Decentralized Training. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 893–907. https://doi.org/10.1145/3297858.3304009

[39] Amith R. Mamidala, Jiuxing Liu, and Dhabaleswar K. Panda. 2004. Efficient Barrier and Allreduce on Infiniband clusters using multicast and adaptive algorithms. In *2004 IEEE International Conference on Cluster Computing (CLUSTER 2004), September 20-23 2004, San Diego, California, USA*. IEEE Computer Society, 135–144. https://doi.org/10.1109/CLUSTR.2004.1392611

[40] Tomas Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*, Takao Kobayashi, Keikichi Hirose, and Satoshi Nakamura (Eds.). ISCA, 1045–1048. http://www.isca-speech.org/archive/interspeech_2010/i10_1045.html

[41] Dmitry Molchanov, Arsenii Ashukha, and Dmitry P. Vetrov. 2017. Variational Dropout Sparsifies Deep Neural Networks. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017 (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, 2498–2507. http://proceedings.mlr.press/v70/molchanov17a.html

[42] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 1–15. https://doi.org/10.1145/3341301.3359646

[43] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013 (JMLR Workshop and Conference Proceedings)*, Vol. 28. JMLR.org, 1310–1318.

[44] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (2011), 2825–2830. http://dl.acm.org/citation.cfm?id=2078195

[45] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, Rui Oliveira, Pascal Felber, and Y. Charlie Hu (Eds.). ACM, 3:1–3:14. https://doi.org/10.1145/3190508.3190517

[46] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed DNN training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 16–29. https://doi.org/10.1145/3341301.3359642

[47] Mercy Prasanna Ranjit, Gopinath Ganapathy, Kalaivani Sridhar, and Vikram Arumugham. 2019. Efficient Deep Learning Hyperparameter Tuning Using Cloud Infrastructure: Intelligent Distributed Hyperparameter Tuning with Bayesian Optimization in the Cloud. In *12th IEEE International Conference on Cloud Computing, CLOUD 2019, Milan, Italy, July 8-13, 2019*, Elisa Bertino, Carl K. Chang, Peter Chen, Ernesto Damiani, Michael Goul, and Katsunori Oyama (Eds.). IEEE, 520–522. https://doi.org/10.1109/CLOUD.2019.00097

[48] Jeff Rasley, Yuxiong He, Feng Yan, Olatunji Ruwase, and Rodrigo Fonseca. 2017. HyperDrive: exploring hyperparameters with POP scheduling. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Las Vegas, NV, USA, December 11 - 15, 2017*, K. R. Jayaram, Anshul Gandhi, Bettina Kemme, and Peter R. Pietzuch (Eds.). ACM, 1–13. https://doi.org/10.1145/3135974.3135994

[49] Fred Richardson, Douglas A. Reynolds, and Najim Dehak. 2015. Deep Neural Network Approaches to Speaker and Language Recognition. *IEEE Signal Processing Letters* 22, 10 (2015), 1671–1675. https://doi.org/10.1109/LSP.2015.2420092

[50] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. 2016. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proc. IEEE* 104, 1 (2016), 148–175. https://doi.org/10.1109/JPROC.2015.2494218

[51] Muthian Sivathanu, Tapan Chugh, Sanjay S. Singapuram, and Lidong Zhou. 2019. Astra: Exploiting Predictability to Optimize Deep Learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 909–923. https://doi.org/10.1145/3297858.3304072

[52] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger (Eds.). 2960–2968. http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms

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

[54] Daniel Strigl, Klaus Kofler, and Stefan Podlipnig. 2010. Performance and Scalability of GPU-Based Convolutional Neural Networks. In *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP 2010, Pisa, Italy, February 17-19, 2010*, Marco Danelutto, Julien Bourgeois, and Tom Gross (Eds.). IEEE Computer Society, 317–324. https://doi.org/10.1109/PDP.2010.43

[55] Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. 2018. A Genetic Programming Approach to Designing Convolutional Neural

Network Architectures. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, Jérôme Lang (Ed.). ijcai.org, 5369–5373. https://doi.org/10.24963/ijcai.2018/755

[56] Yi Sun, Ding Liang, Xiaogang Wang, and Xiaoou Tang. 2015. DeepID3: Face Recognition with Very Deep Neural Networks. *CoRR* abs/1502.00873 (2015). arXiv:1502.00873 http://arxiv.org/abs/1502.00873

[57] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 374–389. https://doi.org/10.1145/3132747.3132750

[58] Kiri Wagstaff, Claire Cardie, Seth Rogers, and Stefan Schrödl. 2001. Constrained K-means Clustering with Background Knowledge. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML 2001), Williams College, Williamstown, MA, USA, June 28 - July 1, 2001*, Carla E. Brodley and Andrea Pohoreckyj Danyluk (Eds.). Morgan Kaufmann, 577–584.

[59] Jinliang Wei, Garth A. Gibson, Phillip B. Gibbons, and Eric P. Xing. 2019. Automating Dependence-Aware Parallelization of Machine Learning Training on Distributed Shared Memory. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, George Candea, Robbert van Renesse, and Christof Fetzer (Eds.). ACM, 42:1–42:17. https://doi.org/10.1145/3302424.3303954

[60] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *CoRR* abs/1708.07747 (2017). arXiv:1708.07747 http://arxiv.org/abs/1708.07747

[61] Hui Y Xiong, Babak Alipanahi, Leo J Lee, Hannes Bretschneider, Daniele Merico, Ryan KC Yuen, Yimin Hua, Serge Gueroussov, Hamed S Najafabadi, Timothy R Hughes, et al. 2015. The human splicing code reveals new insights into the genetic determinants of disease. *Science* 347, 6218 (2015), 1254806.

[62] Steven R. Young, Derek C. Rose, Thomas P. Karnowski, Seung-Hwan Lim, and Robert M. Patton. 2015. Optimizing deep learning hyperparameters through an evolutionary algorithm. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments, MLHPC 2015, Austin, Texas, USA, November 15, 2015*. ACM, 4:1–4:5. https://doi.org/10.1145/2834892.2834896

[63] 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