HyperTune

Distributed Training and Hyperparameter Optimization for Deep Neural Networks

Joel Rorseth
David R. Cheriton School of Computer Science
University of Waterloo
joel.rorseth@uwaterloo.ca

Petar Basta
David R. Cheriton School of Computer Science
University of Waterloo
p3basta@uwaterloo.ca

ABSTRACT

Since their inception, the performance of deep neural networks has been highly dependent on their hyperparameters, motivating the employment of hyperparameter optimization tools. As the complexity and size of datasets and models increase, so too does the importance of efficiency in these tools. While advances in the literature have explored multiple dimensions upon which to distribute deep neural network training, hyperparameter optimization tools have been slow to adopt this new training paradigm. The few tools that do support distributed training are limited to the basic data parallel strategies, however recent advancements in model parallel, hybrid parallel, and pipeline parallel strategies have yet to be explored in the context of hyperparameter optimization. In this project, we implement our own hyperparameter optimization tool, HyperTune, which implements distributed deep neural network training using modular parallelization strategies. In our evaluation, we implement data parallel, model parallel, and GPipe [3] parallelization strategies, and examine performance against a leading data parallel distributed hyperparameter optimization tool.

ACM Reference format:

Joel Rorseth and Petar Basta. 2021. HyperTune: Distributed Training and Hyperparameter Optimization for Deep Neural Networks.

1 Introduction

In the modern age of deep learning (DL), deep neural networks (DNNs) have been implemented using a wide variety of architectures, and have been successfully applied to many real-world problems. However, while their capabilities have grown significantly, so too has their size and complexity. Recent DNNs such as BERT [2] have as many as 340 million parameters, and even more recently, GPT-3 [1] with 175 billion parameters. Considered alongside increasingly large datasets, these factors have significantly increase the demands of DNN training, requiring more time and memory resources. Recent research has reflected a renewed focus on the efficiency of DNN training, seeking to leverage recent hardware and software advancements such as those from the distributed computing literature. With the integration of distributed parallel computation, recent work has improved the efficiency of tractable DNN training, and provided a solution to train models that simply require more resources than available [6].

Continuing down the DNN pipeline, the process of hyperparameter optimization (HPO) is also directly affected by increasing DNN model size and complexity. Since any given HPO algorithm must repeatedly train DNNs with various hyperparameter configurations, the runtime of HPO scales proportionally with that of DNN training. Recent HPO tools have successfully integrated advancements proposed in the distributed DNN training literature, however recent improvements such as those within the scope of hybrid parallelism have not yet been explored. Moreover, recent work in applying parallel computation to HPO has often focused on the parallelization of DNN training jobs, referred to as *trials*. The intersection of these two dimensions for parallelization has remained relatively unexplored, with only a few works such as RubberBand [7] and Katib [10]. To formulate our problem in clear terms, we aim to implement and compare several recent parallel DNN training strategies within a distributed parallel HPO tool. To establish a fair comparison, we implement our own distributed parallel HPO tool which modularizes the DNN training strategy, providing an interface to support several different parallelization strategies for training DNNs.

To motivate this problem, we need only extrapolate upon the clear trend that DNNs are growing both in size and complexity, and realize that HPO tools will exhibit the same demands. Therefore, the continuous integration of new advances in distributed DNN training is equally crucial for HPO. These advancements benefit HPO in the same way as DNN training, by improving efficiency, better utilizing distributed hardware, reducing memory footprint, and reducing runtime. The problem is particularly interesting in the context of HPO, as the two possible dimensions of distributed parallelization greatly increases the search space for potential hardware allocations. It is surprising that current tools do not explore this search space, and instead settle on basic data parallel DNN training. Finding optimal ways to partition this search space certainly makes this problem challenging, though this is beyond the scope of our work. Instead,

we focus on the challenges introduced by the modularization of DNN training parallelism strategies, which is significant when combining with trial-level HPO parallelism. More importantly, the problem is particularly challenging from an implementation standpoint, as DNN training and HPO tools must be created and integrated from the ground up.

In this paper, we present our own simple distributed HPO tool HyperTune, which supports the integration of several recent parallel DNN training strategies within the distributed parallel execution trials. More specifically, HyperTune is a Python HPO tool that enables PyTorch DNN training with minimal training code modification, which allows the user to choose between data parallelism (DP), model parallelism (MP), GPipe (a recent pipeline parallel approach), and no parallelism. In summary, we make the following contributions:

- We develop a distributed HPO tool, HyperTune, with modular support for distributed DNN training
- We implement MP, DP, and GPipe distributed DNN training strategies for HyperTune, and provide interfaces to extend training for other potential strategies
- We compare the runtime and memory consumption of MP, DP, and GPipe, by running HPO on ImageNet and MNIST training tasks, with ResNet and AlexNet DNN models
- We compare the runtime and memory consumption for various configurations of HyperTune against Ray Tune [5] (with support for Horovod [9]), a leading full-distributed HPO tool

2 Background and Related Work

In this section, we review pertinent background information covering important concepts and terminology from distributed computing, HPO, and parallel DNN training. In addition, we discuss related works from these areas, along with any noteworthy limitations.

2.1 Hyperparameter Optimization

Since the performance of a typical neural network is heavily influenced by its hyperparameters (such as its learning rate or batch size), it is beneficial to test many hyperparameter combinations in order to find the best one for the problem at hand. The power set consisting of all possible hyperparameter combinations (or more formally, *hyperparameter configurations*) is referred to as the *hyperparameter space*. Several algorithms exist to traverse the hyperparameter space, such as grid search or random search, and employ various heuristics to navigate the space in different ways.

Grid search performs an exhaustive search of the hyperparameter space, evaluating every possible hyperparameter configuration. When the hyperparameter space is too large to explore in practice, these algorithms may prune the search space dynamically. For example, random search will evaluate arbitrary subsets of the hyperparameter space, instead of evaluating all configurations exhaustively.

Each hyperparameter configuration will be used to train a model for a short period of time, and then the winner will be trained for the full duration and used as the final model. This process is called *hyperparameter optimization (HPO).* Similarly, the training (evaluation) of a DNN for a given hyperparameter configuration is referred to as a *trial*. In the context of a distributed hardware environment, HPO can elect to execute all trials on a single machine, or distribute them across machines and devices (GPUs or CPUs).

2.1.1 HPO Tools

Originally, popular HPO tools executed trials synchronously on a single machine. While vertically scaling the host machine could enable inter-machine trial parallelism, horizontal scaling is certainly more promising due to the immense size of many DNN models and datasets. Furthermore, each trial is independent of each other, and need only be coordinated by a central controller. Several recent HPO tools have added support for distributed parallel trial execution, with a small subset of these providing support for parallel DNN training.

Perhaps the most popular open-source tool for distributed HPO, Ray Tune builds upon the popular distributed computing framework Ray to enable trial-level parallelization across multiple machines. Tune supports many popular ML frameworks such as PyTorch, Keras, and TensorFlow, and allows users to choose from the latest HPO algorithms. Tune offers an API for direct integration with the popular distributed DNN training framework Horovod, which enables the integration of its parallel DNN training capabilities.

As a similar alternative for cloud deployments, RubberBand also provides fully distributed HPO at the trial and training level. RubberBand has not been open-sourced, therefore it lacks many of the features and optimizations that have been contributed to Tune. RubberBand's main contribution is its focus on minimizing cost in a cloud environment through dynamic resource allocation, being able to achieve a cost-reduction of up to 2x compared to static allocation methods.

2.2 Trial Parallelization

Ray Tune enables easy parallelization *across* trials, but has no functionality to parallelize *within* trials without the integration of Horovod. RubberBand is able to parallelize within trials as well, but is only able to use data parallelism (discussed below). Within the context of training a trial on a single machine, there exists many different approaches. The simplest method is to not implement any sort of parallelization within a given trial, but this has shown to perform rather poorly against advanced parallelization techniques.

2.2.1 Data Parallelism (DP). Data Parallelism is the most well-known of these techniques and involves replicating the entire DNN model to multiple GPUs. Each GPU trains its copy of the model with only a subset of the data that it is assigned, and periodically communicates with the other GPUs to synchronize model weights. Many implementations of DP exist, but we will use the popular Horovod tool as a baseline in our evaluation.

Horovod is a distributed machine learning training tool released by Uber and has support for PyTorch, Keras, TensorFlow, and Apache MXNet. The goal of Horovod is to create a framework which can easily modify a single-GPU training script and leverage a few optimizations on top of DP to train across multiple GPUs in parallel.

2.2.2 Model Parallelism (MP). Model parallelism is essentially the opposite of DP; each GPU gets its own copy of the training data, but only receives a subset of the model. Model Parallelism was initially brought to life for use with models that are too large to fit entirely into memory, in which case the model could be partitioned across several GPUs. By splitting models up, MP strategies have been able to increase the total size of models as well as their number of parameters. This has shown to lower the memory requirements necessary to train neural networks, as well as increase prediction accuracy, but unfortunately the effectiveness of this technique relies heavily on the partitioning of the model.

2.2.3 Hybrid Parallelism (HP). Hybrid parallelism involves a combination of both data & model parallel. Compared to traditional data parallelism, HP has been shown to decrease runtime as well as memory usage. Another big accomplishment of HP is its ability to decrease the amount of inter-process communication, and in some cases decrease the amount of data I/O.

2.2.4 Pipeline Parallelism (PP). Pipeline parallelism is a method which combines model parallelism with data pipelining to address the issue of GPU underutilization. By pipelining different subsequences of layers onto separate GPUs, PP is able to provide a great alternative to vanilla MP or DP. This method was first introduced in 2018 by GPipe and PipeDream [8], and further developed into a PyTorch library called torchgpipe [4].

GPipe allows for any network that can be expressed as a sequential model to be scaled up using pipeline parallelism. Thanks to its state-of-the-art batch-splitting pipelining algorithm, it can achieve near-linear decrease in runtime. GPipe handles all of this in a synchronous fashion.

PipeDream works on a similar concept, also avoiding the typical downfalls of DP methods when training large models. Through careful partitioning of DNN layers amongst GPUs, as well as a few other techniques, PipeDream achieves up to 5x faster time-to-accuracy results than typical DP training. It has also shown that in some cases, it is able to reduce communication overhead by almost 95%. The main difference between PipeDream and GPipe is that PipeDream performs its pipelining in an asynchronous manner.

These papers have made a massive impact on the distributed computing community but are difficult to reproduce and use in real-world scenarios. To solve this problem, torchgpipe implements GPipe's micro-batch pipeline parallelism technique in the form of a Python library, enabling PyTorch DNN training with GPipe. This is a big step forward as it allows practical integration of pipeline parallelism into real-world DNN training.

Some tools exist which can parallelize across trials, and some tools exist which can parallelize within trials, but there has yet to be an open-source project which both while allowing for multiple parallelization strategies to be utilized. Using a combination of Horovod with Ray Tune, it is possible to perform data parallel distributed hyperparameter optimization, but it is not easy to plug in MP, HP, or PP techniques. This is where the motivation for HyperTune arises, which is to create an easy-to-use framework which can parallelize across and within trials, while offering modular parallelization strategies.

3 HyperTune Architecture

From a high-level architectural view, HyperTune is comprised of two distinct components, namely the *controller* and the *training script*. The controller is responsible for the HPO-specific aspects of HyperTune, such as scheduling and executing trials on specified remote machines. Meanwhile, the training script is responsible for carrying out a specific DNN training job, and reporting the necessary metrics upon successful completion. By providing one of our training implementations to a single controller instance, HyperTune coordinates the execution of DNN training across remote machines and continues until determining the optimal hyperparameter configuration. For convenience and utility, we provide command line interfaces to both the controller and training scripts, and a Bash script to launch an entire HyperTune job.

3.1 Controller

Playing a central role in the HyperTune tool, the controller is a fully functional yet no-frills HPO tool that is similar to many popular open source HPO tools. Its design is highly modularized using abstract classes, providing flexibility to distribute and schedule trials in many different ways.

As a direct interface to the DNN training scripts, we define a Trial class to represent each trial. Intuitively, a trial encapsulates the execution and results of DNN training for a single hyperparameter configuration. Therefore, the Trial class runs the training script provided to the controller using specific hyperparameters, then stores the results in a TrialResult class. For our experiment, we record the specified HPO criteria value (such as test accuracy or loss), runtime, and various memory measurements. To support the distribution of trials across remote machines, each Trial establishes an SSH connection with an available machine, triggering the correct training execution and parsing its output.

The scheduler is responsible for scheduling the execution of trials, and is implemented as an abstract class. For our experiments, we implement a parallel round-robin scheduler subclass, which executes Trials in a parallel round-robin fashion. At any given time, each machine executes one trial, and draws another trial from a pool upon completion. Our round-robin scheduler is essentially a distributed implementation of grid search, a basic yet popular HPO search algorithm for navigating hyperparameter spaces. The scheduler abstraction allows for the implementation of other search algorithms and distribution strategies, which can be easily substituted in the controller.

Several other classes are defined to comprise the full controller, which cover tasks such iterating hyperparameter spaces (imported from a JSON config file) or calculating optimal hyperparameters across all results.

3.2 Training Scripts

To handle the details of distributed parallel DNN training, we delegate the entire training process to a training script. These scripts are triggered by trials in the controller, whose only expectation is that the training script print its results. In theory, most PyTorch DNN training scripts can be used directly with a minimal modification to print the expected results. To support several different types of distributed training, we add further configuration to enable DP, MP, and GPipe parallel training. Due to the inability to reuse training procedures within the base PyTorch library, we implement separate training scripts for ImageNet and MNIST tasks respectively. Being nearly identical in theory, these two scripts are modified identically to enable all parallel training strategies and compatibility with HyperTune controller.

Being the simplest of our chosen strategies by nature, DP is easily enabled using CUDA utilities provided through the PyTorch library. By wrapping the PyTorch DNN model in a special DistributedDataParallel class, PyTorch can replicate the model and therefore distribute training across available GPU devices. Since other parallelization strategies involve arbitrary partitioning of the arbitrarily complex DNN models, there is no such wrapper class for any other than DP. As explained in the following sections, the developer must manually devise arbitrary partitionings, or employ a heuristic partitioning search seen in recent works such as PipeDream. This is likely a primary factor in why HPO tools have limited themselves to DP, as MP-based strategies are subject to arbitrary partitionings that must be specified by the developer. However, libraries can certainly offer the ability to employ certain well-defined partitioning searches (as we attempt to demonstrate in this work), and incorporate this into a similar convenient model wrapper.

To explore the capabilities of a basic MP approach, we implement our own MP ResNet and AlexNet DNNs using intuitive (though not necessarily optimal) layer partitionings. As opposed to DP, the PyTorch DNN must be modified to assign its layers to GPU devices. However, the training script itself remains unchanged. To summarize the difference, implementing DP in PyTorch requires minimal modification of the training procedure only, but implementing MP requires minimal modification of the model itself.

By leveraging the torchgpipe library for PyTorch, HyperTune is able to train DNN models using the recent GPipe parallelization strategy. This library provides a familiar PyTorch model wrapper that determines a partitioning automatically, using a heuristic-based profiler to optimize for either runtime or accuracy. As discussed in their paper, torchgpipe only supports sequential DNN models (those that inherit from PyTorch's Sequential class), due to the inherent nature of pipeline parallelism (PP). The authors suggest that any neural network can be represented in sequential form, although this places the burden of adapting non-sequential DNN models on the developer. Moreover, the effects of this adaptation process on accuracy (or other metrics) are not discussed. For the purposes of this work, we successfully adapt a sequential ResNet DNN for our experiments, but leave the adaptation of AlexNet for potential future work. Further, we use the runtime-optimized heuristic for all evaluations in this paper, and fix the degree of GPipe partitioning to the value of 8.

4 Evaluation

In this section, we describe and quantify the performance of HyperTune in terms of runtime and memory usage, especially to highlight the performance implications of MP, DP, and GPipe training. Moreover, we compare these results against those of the popular distributed HPO tool Ray Tune, which supports both dimensions of parallelism when run using the Horovod tool. As discussed, AlexNet is not compatible with torchgpipe without modification, therefore this combination is excluded from the corresponding experiments.

4.1 Experimental Setup

Machines: For all experiments, we use 3 high-performance *worker* machines located on-premise at the University of Waterloo, with 1 extra machine (the *head*) to coordinate the worker machines. All machines have Intel Xeon Silver 4114 processors running at 2.20 GHz, and run Ubuntu 18.04.5. Each worker machine has 2 Nvidia Tesla P40 GPUs, as well as 192GB RAM. It is important to note that these are shared machines, so it is possible the resources were being used by others while our experiments were being run.

Datasets: The main dataset we used in our experiments is the dataset for the Large Scale Visual Recognition Challenge (ILSVRC), better known as the ImageNet dataset. This dataset contains just under 1.3 million training images, 50,000 validation images, and 100,000 test images, all containing 3 colour channels. This dataset was chosen to test the inherent memory efficiency of MP, as this particularly large dataset would provide a greater challenge with limited resources. Since the ImageNet dataset took a long amount of time to train even a single epoch, another dataset was necessary to be able to traverse a large hyperparameter space. The Modified National Institute of Standards and Technology dataset, or MNIST, contains only 60,000 training images, and 10,000 test images, all of which are greyscale (1 channel).

Hyperparameter space: Due to the significant size of the ImageNet dataset and the proportional effect on training runtime, we use a small hyperparameter space with six configurations. We specify two discrete values for learning rate, three discrete values for batch size, yielding six trials in total.

```
{
    "lr": [0.1, 0.01],
    "batch-size": [64, 128, 256]
}
```

For the MNIST task, we specify a larger hyperparameter space to take advantage of its reduced runtime. Three discrete values are specified for batch size, learning rate, and gamma hyperparameters, yielding 27 trials in total.

```
{
    "batch-size": [64, 128, 256],
    "lr": [0.5, 0.9, 0.99],
    "gamma": [0.5, 0.7, 0.9]
}
```

Models: The first model used is ResNet-50, a 50 layer DNN which won 1st place on the ILSVRC 2015 classification task. The second model used is AlexNet, which placed first on the ILSVRC 2012 classification task. Again, these large models were chosen to test the claims that model parallelism can achieve memory usage reductions on large models.

4.2 Trial-Level Runtime

Across many experiments, we compare the runtime of HyperTune and Ray Tune at the granularity of each trial. In particular, we consider the runtime across ResNet and AlexNet, on ImageNet and MNIST, using HyperTune (MP, DP, and GPipe variants) and Ray Tune. While all ImageNet figures include all six trials, MNIST figures illustrate only a common subset of nine trials for brevity and clarity.

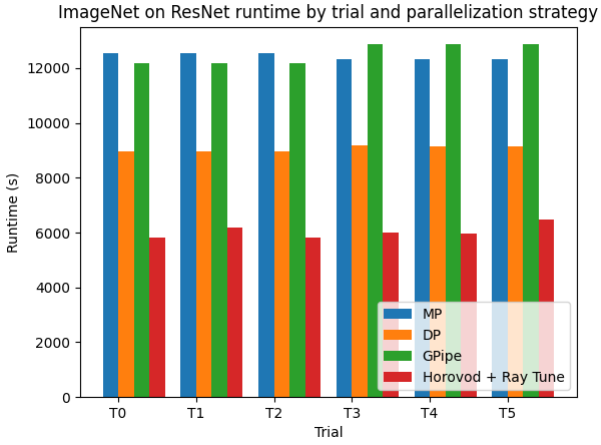

Figure 1: Trial-level runtime comparison of HyperTune (with MP, DP, and GPipe parallelism) and Ray Tune (with Horovod support) for ResNet on ImageNet.

Considering the ImageNet task first, HyperTune with MP and GPipe yield trial runtimes approaching 3.5 hours with ResNet. However, HyperTune with DP shows significant improvement for ResNet in Figure 1, with an average runtime around 2.5 hours (almost 40% faster). Ray Tune with Horovod significantly outperforms all HyperTune variants with an average runtime of 1.7 hours, almost 110% faster than HyperTune with MP or GPipe.

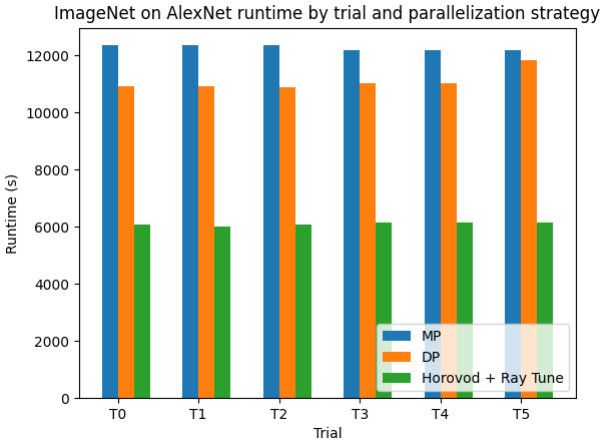

Figure 2: Trial-level runtime comparison of HyperTune (with MP, DP, and GPipe parallelism) and Ray Tune (with Horovod support) for AlexNet on ImageNet.

As a testament to the influence of the dataset itself, AlexNet runtime in Figure 2 exhibits the same performance difference between DP and MP. For the ImageNet problem, DP is clearly better suited to handle the extremely large dataset and resulting model. Furthermore, the DP-based Ray Tune with Horovod has integrated many significant optimizations over many years of open-source development, which explains the disparity between our less-mature HyperTune DP tool.

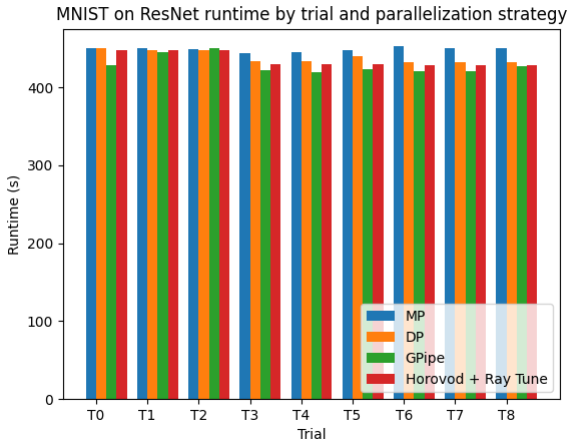

Figure 3: Trial-level runtime comparison of HyperTune (with MP, DP, and GPipe parallelism) and Ray Tune (with Horovod support) for ResNet on MNIST.

Clearly illustrating how neither MP nor DP are ideal for all problems, we see clear advantages with MP-based DNN training

for the MNIST dataset in Figure 3. Exhibiting low variance across all nine trials, all methods tested finish with 420-450 seconds with a ResNet DNN. In this case, HyperTune with GPipe ran the fastest with an average runtime of 428 second per trial. It ran 1.6% faster than Horovod + Ray Tune (435 seconds/trial), 2.8% faster than DP (440 seconds/trial), and 4.7% faster than MP (448 seconds/trial).

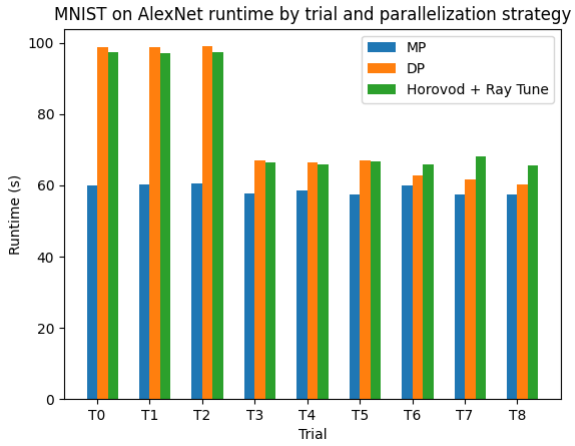

Figure 4: Trial-level runtime comparison of HyperTune (with MP, DP, and GPipe parallelism) and Ray Tune (with Horovod support) for AlexNet on MNIST.

Further illustrating the suitability of MP training for MNIST, HyperTune MP demonstrates superior runtime for the AlexNet DNN, with an average runtime around 60 seconds. More importantly, despite evaluating hyperparameter configurations that significantly increase DP training runtime, MP training runtime exhibits almost no variance. As evident in the first three trials in Figure 4, hyperparameters such as batch size and learning rate can significantly influence training runtime (and memory consumption). Due to the efficiency gained by partitioning the large DNN, MP can maintain typical runtime speeds for this DNN and dataset. Along with HyperTune with DP, Ray Tune with Horovod exhibits higher runtimes around 95 seconds for T0 through T2 (batch size 64) and 65 seconds for T3 through T8 (batch size either 128 or 256). In contrast to substantial disparity in our ImageNet experiments, the performance of HyperTune with DP is nearly identical to Ray Tune with Horovod. This suggests that the additional enhancements and capabilities of Ray Tune are exploited more effectively when applied to larger datasets.

4.2 Trial-Level Peak DNN Memory Usage

Using the same experiment setup, we compare the peak memory usage of the DNNs trained by HyperTune and Ray Tune at the same trial-level. Applied to the ImageNet dataset, we see improvements in memory usage with GPipe that correspond to the specified degree of parallelization (8, which fixed across all experiments). In their pipeline parallel approach, this means that 8 micro-batches will be used for partitioning. As such, GPipe peak memory utilization reflects the expected 8x reduction visible in our experiments (such as those in Figure 5 and Figure 7). In our peak DNN memory usage experiments, the reported value reflects the peak usage per distributed *model* partition within the respective parallelization strategy.

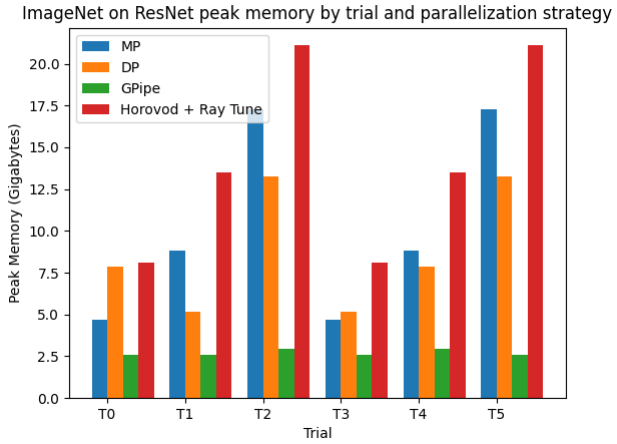

Figure 5: Trial-level peak memory usage comparison of HyperTune (with MP, DP, and GPipe parallelism) and Ray Tune (with Horovod support) for ResNet on ImageNet.

Although Ray Tune with Horovod often outperforms HyperTune in runtime, it uses significantly more memory in doing so, exceeding 20 gigabytes at peak in some trials. Significantly improving upon all other models, HyperTune with GPipe maintains near-constant peak memory utilization of 2.5 gigabytes across all trials, as shown in Figure 5. To rationalize the variance between trials, we note that trials T5 / T2 use batch size 256, T4 / T2 use batch size 128, and trials T3 / T1 use batch size 64.

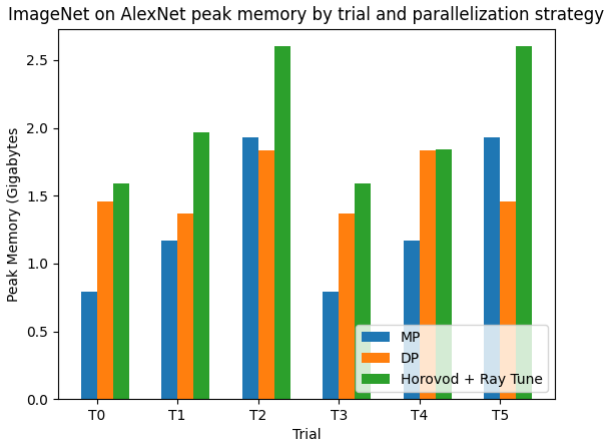

Figure 6: Trial-level peak memory usage comparison of HyperTune (with MP, DP, and GPipe parallelism) and Ray Tune (with Horovod support) for AlexNet on ImageNet.

Considering the obvious effect of batch size on memory usage, it is not surprising that T5 / T2 have the largest memory usage, followed by T4 / T2, and then T3 / T0. Independent of the influence of such hyperparameters, which are proportionally reflected by all tested models, HyperTune with MP and DP consistently utilize far less memory than Ray Tune with Horovod. Despite the omission of a GPipe HyperTune implementation, the AlexNet evaluation in Figure 6 largely reiterates this relative ranking, albeit in slightly smaller proportions.

The near-zero variance (in peak memory usage) provided by GPipe promises true horizontal scalability for a future likely subject to increasingly massive datasets and DNN models. Further exemplified in our MNIST experiments (Figure 7), GPipe appears to maintain extremely low peak-memory usage across many datasets. These results indicate that GPipe-based HPO can efficiently leverage much larger distributed environments with fewer resources, such as large clusters with lower-quality commodity hardware, which is a significant improvement to the state of the art.

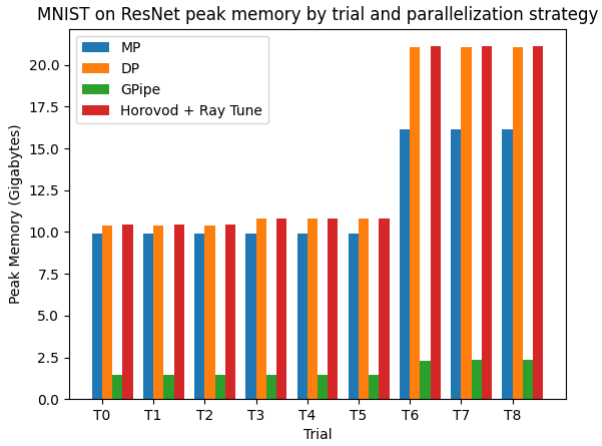

Figure 7: Trial-level peak memory usage comparison of HyperTune (with MP, DP, and GPipe parallelism) and Ray Tune (with Horovod support) for ResNet on MNIST.

GPipe aside, our MNIST evaluations of HyperTune and Ray Tune with both DNN models suggest that MP-based partitioning is much more optimal in terms of peak memory usage. Ray Tune with Horovod used ~4.7% more memory than MP for trials T0 through T5, which correspond to batch sizes of 64 and 128. For T6 through T8, which has batch sizes of 256, Horovod + Ray Tune used 30% more memory than MP. As established in the literature, MP is difficult to judge directly, since its effectiveness is heavily reliant on the optimality of the chosen DNN partitioning.

While both HyperTune DP and Ray Tune with Horovod (DP) share very similar performance for MNIST, HyperTune with DP consistently requires less memory (and with less variance) for the larger ImageNet dataset. This indicates that the combined overheads of Ray and Horovod may be unnecessary and counter-intuitive for certain hyperparameters and datasets, since HyperTune's basic DP implementation was much more efficient on ImageNet (Figure 5). Considering Ray Tune's runtime speedup in the corresponding test (Figure 1), HyperTune with DP seems to offer a solution which better prioritizes memory efficiency in the apparent runtime vs. memory tradeoff.

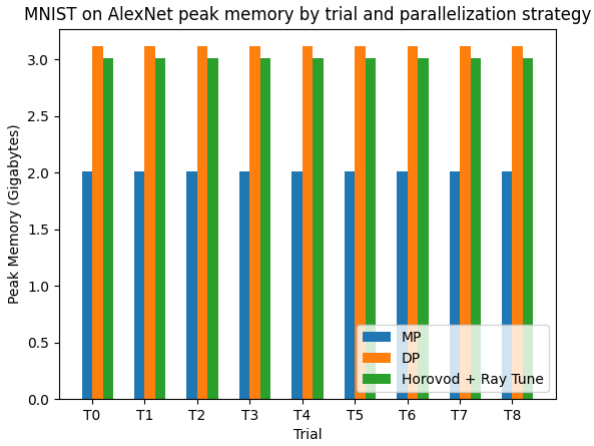

Figure 8: Trial-level peak memory usage comparison of HyperTune (with MP, DP, and GPipe parallelism) and Ray Tune (with Horovod support) for AlexNet on MNIST.

For AlexNet trained with MNIST data, all three HyperTune parallelization strategies resulted in near-constant peak memory usage, but MP maintained a low 2 gigabytes peak memory aross all trials.

4.3 Experiment Performance Metrics

Dataset	Model	Parallel Training Strategy	Parameter Memory (GB)	Buffer Memory (GB)	Overall Runtime (s)
ImageNet	AlexNet	DP	0.227618	0	22,740
		MP	0.227618	0	24,600
		GPipe	N/A	N/A	N/A
		Ray Tune	0.227618	0	12,240
	ResNet	DP	0.095207	0.000198 28	18,120
		MP	0.095207	0.000198 28	21,720
		GPipe	0.095207	0.000198 28	25,020
		Ray Tune	0.095207	0.000198 28	12,840
MNIST	AlexNet	DP	0.227561	0	712
		MP	0.227561	0	563
		GPipe	N/A	N/A	N/A
		Ray Tune	0.227561	0	937
	ResNet	DP	0.095207	0.000198 28	3,960
		MP	0.095207	0.000198 28	4,080
		GPipe	0.095184	0.000198 28	3,900
		Ray Tune	0.095207	0.000198 28	4,140

Table 1: Experiment-level memory usage and runtime comparison of HyperTune (with MP, DP, and GPipe parallelism) and Ray Tune (with Horovod support) for ResNet and AlexNet on ImageNet and MNIST.

At the experiment-level, we observe mixed results for overall runtime that echo the mixed results of the constituent trials. Specifically, Ray Tune with Horovod executes all ImageNet HPO experiments nearly 100% faster than the next best HyperTune variant. However for AlexNet on MNIST, HyperTune with MP executes 66% faster than Tune, with improved runtime for other variants as well. ResNet on MNIST executes fastest using HyperTune with GPipe, although marginally. We reiterate the fact that certain datasets and models may be better suited for DP or MP, which has been apparent across our tests. These statistics validate our original motivation to create a modular HPO tool, one which allows different training strategies to be substituted in different situations. Future work should explore the use of profiling runs and other heuristics to dynamically determine these partitionings (or partitioning strategies), as has been explored in recent distributed DNN training research.

5 Conclusion

In this work, we presented our fully distributed HPO tool HyperTune, the first (to the best of our knowledge) to modularize its parallel DNN training strategy to support recent MP and PP approaches. We show that for some datasets and models, our DP-based HyperTune variant utilizes less memory than the popular DP-based Ray Tune with Horovod. Moreover, we demonstrate the utility of the GPipe parallel DNN training strategy for HPO, proving that it efficiently partitions training across more devices while maintaining or reducing runtime. In summary, our work justifies the integration of recent hybrid parallel DNN training strategies into future HPO tools and gives rise to new directions of future research in this field. In future work, our tool could be improved to support more advances in parallel DNN training, such as the promising PipeDream approach. Further testing using models and datasets from different domains would identify more strengths and weaknesses. Additionally, partitioning exploration algorithms could be incorporated to explore the large search space formed by trial-level and training-level partitionings.

ACKNOWLEDGMENTS

We would like to thank Lori Paniak from the University of Waterloo IT department, for his help in configuring the hardware for our evaluation. We also extend our gratitude to Dr. Khuzaima Daudjee at the University of Waterloo for his guidance during the completion of this project.

REFERENCES

[1] Tom B. Brown, et al. 2020. Language models are few-shot learners. arXiv:2005.14165

[2] Jacob Devlin et al. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. axXiv: 1810.04805

[3] Yanping Huang et al. 2019. GPipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems 32*, 103-112.

[4] Chiheon Kim et al. 2020. torchgpipe: On-the-fly pipeline parallelism for training giant models. arXiv:2004.09910

[5] Richard Liaw et al. 2018. Tune: A research platform for distributed model selection and training. arXiv:1807.05118

[6] Ruben Mayer and Hans-Arno Jacobsen. 2020. Scalable Deep Learning on Distributed Infrastructures. *ACM Computing Surveys 53*, 1 (2020), 1–37. DOI:https://doi.org/10.1145/3363554

[7] Ujval Misra et al. 2021. RubberBand: Cloud-based Hyperparameter Tuning. *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–28, 2021, Online, United Kingdom*. ACM, New York, NY, USA, 16 pages. DOI:https://doi.org/10.1145/3447786.3456245

[8] Deepak Narayanan et al. 2019. PipeDream: generalized pipeline parallelism for DNN training. *Proceedings of the 27th ACM Symposium on Operating System Principles*. arXiv:1806.03377

[9] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. arXiv:1802.05799

[10] Jinan Zhou et al. 2019. Katib: A distributed general automl platform on Kubernetes. *Conference on Operation Machine Learning (OpML 19)*