THE UNIVERSITY OF CHICAGO

EFFICIENT RESOURCE SHARING FOR DATA-INTENSIVE COMPUTATION

A DISSERTATION PROPOSAL SUBMITTED TO

THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE

IN CANDIDACY FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

RUI LIU

CHICAGO, ILLINOIS

MAY 2022

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Acknowledgements.

ABSTRACT

Data-intensive computation has increasingly become prevalent and critical for a wide variety of applications that underpin the modern world. Such computation jobs are usually expensive in terms of time and resource consumption, thus it is beneficial but challenging to efficiently share the resource among them in a resource-constrained environment. Our studies show that by exploiting workload characteristics and leveraging resource traits, the constrained resource could be efficiently shared among multiple data-intensive computation jobs to achieve the same performance objective as they only achieve when competing for resources fiercely.

In our research, we explore efficient resource sharing for data-intensive computation and implement various prototype systems to realize them. We propose and implement Pack primitive for deep learning training to share common I/O and computing processes among models on the same device. We further propose and design a resource arbitration framework that can continuously prioritize the data-intensive computation jobs and determine if/when to reallocate and preempt the resources for them.

CHAPTER 1

INTRODUCTION

Data-intensive computation is vital for a wide variety of modern services and applications spanning from healthcare and market analytics to autonomous driving and natural language processing [11]. As best exemplified by two prevalent cases, approximate query processing (AQP) [10] and deep learning training (DLT) [18], most of the modern data-intensive computation jobs run in an iterative and progressive fashion where data are received in a batch mode and current results can be reused and merged with the latest results of processing the new batch of data. Such long-running data-intensive computation jobs are prohibitively expensive in terms of time and resource consumption, especially in a resource-shared and -constrained environment.

Yet, prior data-intensive computation systems are inefficient from the perspective of resource sharing. A key factor in the inefficiencies of prior data-intensive computation systems is that their design and implementation are mostly agnostic of the resources and workloads. They typically treat the modern data-intensive computation jobs as the classic data processing job without identifying the unique traits of the resources and jobs. For example, the deep learning training jobs rely on specialized hardware such as GPUs which makes fine-grained resource sharing (i.e., training multiple networks on the same device) significantly harder than the classic CPU-based computation job. Moreover, considering the efficiency of resource sharing is jointly described by the resource utilization and the progress per unit of resource, another common but unobtrusive inefficiency example is that some long-running data-intensive computation jobs can occupy the constrained resource for an extended period but only make subtle progress towards their performance objective.

To achieve efficient resource sharing for data-intensive computation, we argue that it is necessary to *leverage resource traits* and *exploit workload characteristics*. The modern resources bring new traits including multi-core processors, specialized computation capability, and heterogeneous data transmission and storage. Taking advantage of these traits requires extensive effort and great care because they are often indistinguishably exposed to different data-intensive computation jobs and integrated into a single infrastructure; using them naively or in a unified way could degrade performance. Applying optimization can become efficient under specific workloads with certain properties; interruption to long-running jobs is typically considered harmful because it may bring additional computation and context-switch overhead, but it cal also improves the performance of the entire workload when exploited properly.

1.1 Towards Efficient Resource Sharing for Data-Intensive Computation

Our efforts in designing and implementing efficient resource sharing systems touch two prevailing data-intensive computation scenarios as aforementioned, deep learning training (DLT) and approximate query processing (AQP). We provide a high level overview for each work as following.

Pack: As neural networks are increasingly employed in machine learning practice, how to efficiently share limited training resources among a diverse set of model training tasks becomes a crucial issue. To achieve better utilization of the shared resources, we explore the idea of jointly training multiple neural network models on a single GPU in this paper. We realize this idea by proposing a primitive, called `Pack`. We further present a comprehensive empirical study of `Pack` and end-to-end experiments that suggest significant improvements for hyperparameter tuning. The results suggest: (1) packing two models can bring up to 40% performance improvement over unpacked setups for a single training step and the improvement increases when packing more models; (2) the benefit of the `Pack` primitive largely depends on a number of factors including memory capacity, chip architecture, neural network structure, and batch size; (3) there exists a trade-off between packing and unpacking when training multiple neural network models on limited resources; (4) a `Pack`-aware Hyperband is up to 2.7× faster than the original Hyperband, with this improvement growing as memory size increases and subsequently the density of models packed.

Rotary: Increasingly modern computational applications employ progressive iterative analytics. In comparison to classic computation applications that only return the results after processing all the input data, progressive iterative analytics keep providing approximate or partial results to users by performing computations on a subset of the entire dataset until either the users are satisfied with the results, or the predefined completion criteria are achieved. Typically, progressive iterative analytic jobs have diverse completion criteria, produce diminishing returns, and process data at a different rate, which necessitates a novel *resource arbitration* that can continuously prioritize the progressive iterative analytic jobs and determine if/when to reallocate and preempt the resources for them. We propose and design a resource arbitration framework, Rotary, and we implement two resource arbitration systems, Rotary-AQP and Rotary-DLT, for approximate query processing and deep learning training. We build a TPC-H based AQP workload and a survey-based DLT workload to evaluate the two systems respectively. The evaluation results demonstrate that Rotary-AQP and Rotary-DLT outperform the state-of-the-art systems and heuristic baselines and confirm the generality and practicality of the proposed resource arbitration framework.

In the rest of this dissertation proposal, we first present the aforementioned systems in Chapter 2 and Chapter 3. Then we discuss the current progress and research plan of the next project in Chapter 4.

CHAPTER 2
UNDERSTANDING AND OPTIMIZING PACKED NEURAL NETWORK TRAINING FOR HYPER-PARAMETER TUNING

The successes of AI are in part due to the adoption of neural network models which can place immense demand on computing infrastructure. It is increasingly the case that a diverse set of model training tasks share limited training resources. The long-running nature of these tasks and the large variation in their size and complexity make efficient resource sharing a crucial concern. The concerns are compounded by an extensive trial-and-error development process where parameters are tuned and architectures have tweaked that result in a large number of trial models to train. Beyond the monetary and resource costs, there are long-term questions of economic and environmental sustainability [77, 70].

Efficiently sharing the same infrastructure among multiple training tasks, or multi-tenant training, is proposed to address the issue [89, 40, 62, 57]. The role of a multi-tenancy framework is to stipulate policies and constraints on how contended resources are partitioned and tasks are placed on physical hardware. Most existing approaches divide resources at the granularity of full devices (e.g., an entire GPU) [34]. Such a policy can result in low resource utilization due to its coarse granularity. For example, models may greatly vary in size, where the largest computer vision models require multiple GBs of GPU memory [9] but mobile-optimized networks use a significantly smaller space [69]. Given that GPUs today have significantly more on-board memory than in the past (e.g., up to 32 GB in commercial offerings), if a training workload consists of a large number of small neural networks, allocating entire devices to these training tasks is wasteful and significantly delays any large model training.

Furthermore, the reliance on specialized hardware such as GPUs makes fine-grained resource sharing (i.e., training multiple networks on the same device) significantly harder than the typical examples in cloud systems. Unlike CPUs, the full virtualization of GPU resources is nascent [65]. While modern GPU libraries support running multiple execution kernels in parallel, sharing resources using isolated kernels is not a mature solution in this setting. Many deep learning workloads are highly redundant, for example, the typical parameter tuning process trains the same model on the same data with small tweaks in hyperparameters or network architectures. In this setting, those parallel kernels would transfer and store multiple copies of the same training data on the device. This is analogous to the redundancy problems faced with conventional hypervisors running many copies of the same operating system on a single server [86].

To avoid these pitfalls and provide efficient sharing, we need an approach that is aware of common I/O and computing processes among models that share a device. We consider a scheme, packing models, where multiple static neural network architectures (e.g., ones that are typically used in computer vision) can be rewritten as a single concatenated network that preserves the input, output, and backpropagation semantics through a `pack` primitive. Not only does such concatenations facilitate partitioning of a single device it also allows us to synchronize data processing on GPUs and collapse common variables in the computation graph. It is often the case during hyperparameter tuning that the same model with various hyperparameter configurations are trained, and `pack` can feed a single I/O stream of training features to all variants of the model. In contrast, an isolated sharing way (e.g., training models isolatedly in sequence) may lead to duplicated work and wasted resources.

One of the surprising conclusions of this paper is that packing models together is not strictly beneficial. Counter-intuitively, certain packing policies can perform significantly worse than whole-device baselines–in other words, training a packed model can be slower than the sum of its parts (i.e., training these "parts" one by one). This paper studies the range of possible improvements (and/or overheads) for using `pack`. Further, we deploy `pack` to hyperparameter tuning and demonstrates that it can greatly improve the performance of hyperparameter tuning in terms of the time needed to find the best or the most promising model.

Our experimental results suggest: (1) There is a range of performance impact, spanning from 40% faster execution to 10% slower execution on a single GPU for packing two models over unpacking them for a single training step, and the improvement is scalable when packing more models. (2) The benefits of the `pack` primitive largely depend on a number of factors including memory capacity, chip architecture, neural network structure, batch size, and data preprocessing overlap. (3) There exists a trade-off between packing and unpacking when training multiple neural network models on limited resources. This trade-off is further complicated by architectural properties that might make a single training step bounded by computation (e.g., backpropagation is expensive) or data transferring (e.g., transferring training batches to GPU memory). (4) The `pack` primitive can speedup hyperparameter tuning by up to 2.7×.

2.1 Background

2.1.1 Motivation

Figure 2.1 demonstrates the typical data flow in neural network model training with stochastic gradient descent (or related optimization algorithms). We use the term *host* to describe CPU/Main-Memory/Disk hierarchy and *device* to refer to the GPU/Device Memory. In this setup, all of the training data resides on the host. Considering the typical training setup on the left side, a batch of data is taken from the host and copied to the device. Additionally, it is common in machine learning (especially in Computer Vision) that this data is preprocessed before it is transferred. Then, on the device, the execution framework calculates a gradient using backpropagation. Finally, using the results from the backpropagation, the model is updated.

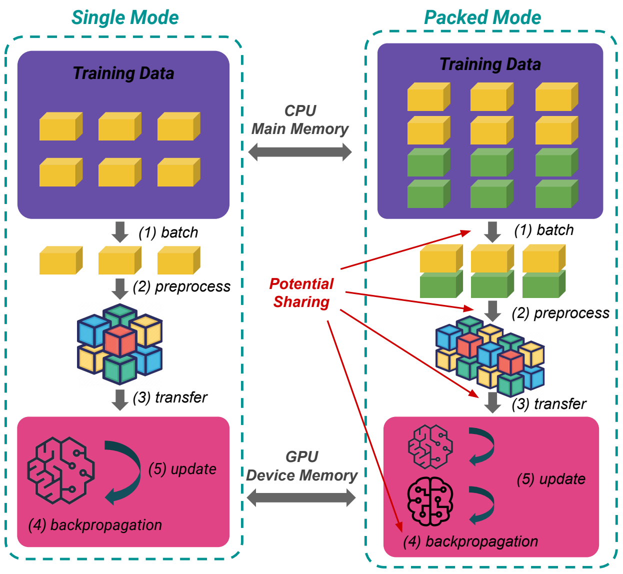


Figure 2.1: The dataflow of a training step in the single mode v.s. the packed mode. The training data resides in main-memory and is copied over to the device in batches during each training step resulting in a backprogation computation and then a parameter update. By synchronizing the dataflow, the packed mode can reuse work when possible.

In the typical ”multiple-tasks-single-device” mode, resource sharing is often temporal–where one training task uses the whole GPU first and then switches full control to another task. Resource sharing in single mode is wasteful if the models are small and there is sufficient GPU memory to fit both models on the device simultaneously. The right side of Figure 2.1 motivates a different solution. It allows multiple models to be placed on a single GPU. This packed mode could bring some potential benefits. Suppose, we are training two models on the same dataset to test if a small tweak in neural network architecture will improve performance. The same data would have to be copied and transferred twice for training. If the system could pack together models when compatible in size, then these redundant data streams can be fused together to improve performance.

2.1.2 Basic Framework API

We desire a framework that can pack models together and jointly optimize their respective computation graphs when possible to reduce redundant work. We assume that we have access to a full neural network description, as well as the weights of the network. Each training task is characterized by four key traits: (a) *Model.* A computation graph architecture of the model with pointers to the input and output, equipping with some training hyperparameters (e.g. learning rate, optimizer, etc.) and assigning to a logical name that is uniquely identified. (b) *Device.* The target device to be used for placing and training models. (c) *Batch Size.* The batch size used in the training process, where each batch refers to the size of input data used in a single training step. (d) *Training Step.* The number of steps to train the model, which is also relevant to the number of epochs since typically one epoch consists of numerous steps.

Our objective is the following isolation guarantee: given these four traits, our framework will train the models in a fine-grained way but preserve the accuracy as if the training tasks were trained isolatedly and sequentially on a dedicated device. No action that the framework takes should affect training accuracy. Such a framework requires three basic primitives `load`, `free`, and `pack`. Users should be able to interact with our framework without worrying about exactly how the resources are allocated and on which devices the models are placed.

The primitives `load` and `free` can ”copy in” and ”copy out” models. Given a device name and model, `load` places the model on the device:

```
load(model, device)
```

Given a device name and model, `free` retrieves the model and frees the resources taken by the model:

```
checkpt = free(model, device)
```

State-of-the-art neural network training algorithms have additional state as a part of the optimizer. This state is stored with the model (see our experiments on computer vision models with optimizers in Section 2.3). Then, the API provides the primitive `pack` for packing. Suppose, we have two neural network models:

```
output1 = nn1(input1)
output2 = nn2(input2)
```

The `pack` primitive combines both models into a new neural network by concatenating the output layers:

```
[output1 output2] = packed_nn([input1 input2])
```

This packing operation is fully differentiable and preserves the correctness semantics of the two original networks. Crucially this allows the execution layer to process inputs simultaneously.

Thus, the models can be jointly trained using `pack`. The training steps have to be synchronized in the sense that the models are differentiated and updated at the same time. This synchronization leads to a complex performance trade-off, if the models are too different the device may waste effort stalling on one model while either updating or differentiating on the other. This means that training a packed model may be significantly slower than sequentially training each constituent model in it. However, the overheads from stalling may be counteracted by the benefits of reducing redundant computation. Navigating this complex trade-off space is the motivation for this study, and we seek to understand under what conditions is `pack` beneficial.

2.2 Implementation

We build a prototype system on top of TensorFlow to implement the above framework APIs and take image classification as our motivating application in our implementation, but the idea of `pack` is generally compatible with other platforms and applications.

2.2.1 Packing

`Pack` is a lossless operation that concatenates the outputs of two or more neural network models. Since it is lossless, it preserves the forward and backward pass semantics of the model. The basic operation can be written as packing multiple output variables, as illustrated in the following example:

```
mlp_out = #reference to mlp output
resnet_out = #reference to resnet output
densenet_out = #reference to densenet output
packed_out = pack([mlp_out resnet_out densenet_out])
```

This `packed_out` can be thought of as a new neural network model that takes in all input streams (even possible different input data types) and outputs a joint prediction. Thus, we can do everything to a packed model that we could do to a single neural network. The packed model can be differentiated and the model parameters can be updated iteratively. The model can be placed on a device, such as GPU or TPU, as a single unit.

While this gives us scheduling flexibility, there is a major caveat. By packing the models together, we create an artificial synchronization barrier. If one of the models is significantly more complex than the others, it will block progress. Likewise, if one of the operations saturates the available compute cores, progress will stall as well. Naive packing leads to a further issue where the input batch has to be synchronized in dimension as well (each model is differentiated or evaluated the same number of times). Therefore, without further thought, the scope of `pack` is very narrow.

2.2.2 *Misaligned Batch Sizes*

Requiring that all packed models have the same batch size is highly restrictive, but we can relax this requirement. Our method is to rewrite the packed model to include a dummy operation that pads models with the smaller batch size to match the larger ones in dimension. The pad primitive is exploited for packing models with different batch sizes. The original models are packed and trained based on the batch with the largest size, but the batches for the models with smaller batch sizes will be padded. During training and inference, the padding is sliced.

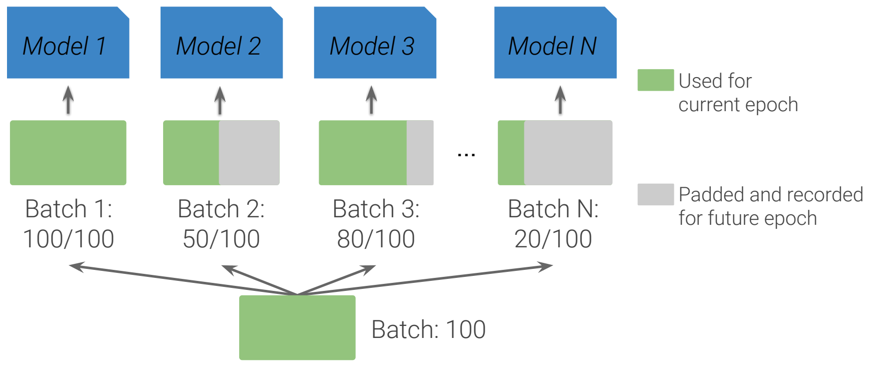


Figure 2.2: All the models share the batch input stream, each batch is padded and sliced for training the packed model.

As depicted in Figure 2.2, there are a set of original models with various batch sizes, the largest training batch size (i.e., 100) is selected and fed to the packed model for a single training step accordingly. Then, the batch will be replicated for n original models in the packed model. The model 1 takes the entire batch, whereas the replicated batches $2, \cdots, N$ are sliced to match the models' requirement. Thus, all the models can be trained together.

Simply slicing may result in statistical inefficiency since only a fraction of the entire dataset is used during each epoch for the models with smaller batch size. To address this issue, we track the progress of each model individually to ensure that there is no loss in training dataset. Assuming we train the packed model in Figure 2.2 for one epoch. When model 1 finishes training and is unloaded, model 2 achieves 50% progress and uses 50% of the training dataset, model 3 has been training using 80% dataset, and so do the other models (dataset usage is recorded for all models). Then, the packed model takes the current largest batch size (i.e. 80 from model 3) and uses the rest training dataset of model 3 to train the packed model.

Due to slicing, it is obvious that unused training dataset of model 3 are included in the unused dataset of other models. The process continues until all models are trained completely and thus no training data is missing.

2.2.3 Misaligned Step Counts

Another issue with synchronization is that different models may need to be trained for a different number of steps. Even if all of the models are the same, this can happen if the user is trying out different batch sizes.

We use `load` and `free` to address this issue. As demonstrated in Figure 2.3, we train three models with batch size 20, 50, and 100 for one epoch using $10,000$ images and labels, and they require 500 steps, 200 steps, and 100 steps. We pack them for training, and when the model with 100 steps is finished, it is freed and checkpointed. Then, a new model can be loaded and packed with the incomplete models to continue training. This mechanism may bring an overhead of loading models but can support training models with a different number of steps.

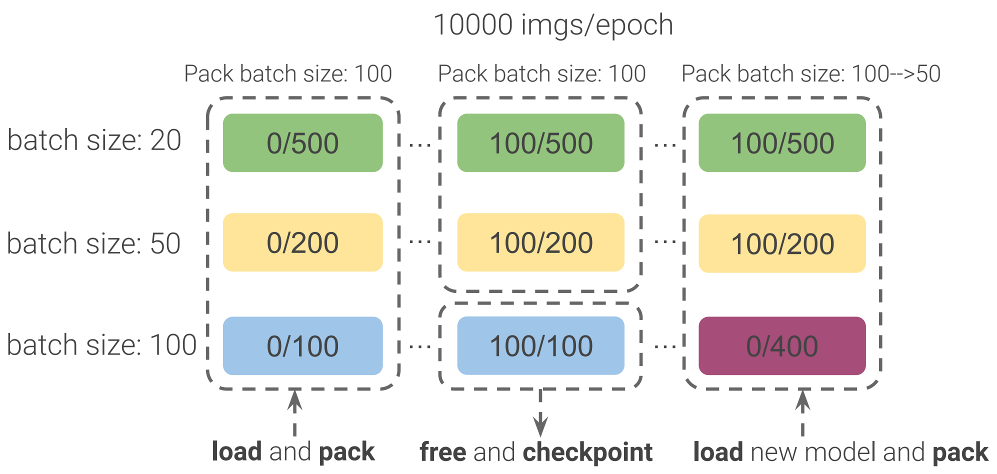


Figure 2.3: Early finished model is freed and checkpointed, new model is packed with the others for further training.

2.2.4 Eliminating Redundancy

Pack forces synchronization, which means that dimensional differences between the models or training differences between the models can lead to wasted work. However, Pack can allow the system to eliminate redundant computations and data transfers. Consider a hyperparameter tuning use case where we are training the same network with a small configuration tweak on the same dataset:

```
nn_conf1_out = nn_conf1(input1)
nn_conf2_out = nn_conf2(input2)
```

In this case, input1 and input2 refer to the same dataset. We can avoid transferring the batch multiple times by symbolically rewriting the network description to refer to the same input:

```
nn_packed_out = pack([nn_conf1_out nn_conf2_out])
[output1 output2] = nn_packed_out(input)
```

The potential upside is significant as it reduces the amount of data transferred along a slower I/O bus. Furthermore, eliminating redundant computation goes beyond identifying common inputs. Preprocessing is a common practice for machine learning training tasks. The preprocessing operations (e.g., data augmentation, image decoding) happen before training and can actually dominate the total execution time of some models. When packing models that take the same preprocessing, the pack primitive can fuse the steam processing and eliminate redundant tasks. This idea can be extended if multiple models have fixed featurization techniques or leverage the same pretrained building blocks.

2.3 Profiling Model Packing

As it stands, model packing leads to the following trade-offs. Potential performance improvements include: (1) eliminating redundant data transfers when models are trained on the same dataset, (2) combining redundant computations including preprocessing, (3) performing computations (forward and back propagation) in parallel if and when possible. On the other hand, the potential overheads include (a) models that dominate the device resources and block the progress of the others, (b) overhead due to misaligned batch sizes, and (c) overhead due to loading and unloading models with a differing number of training steps.

This section describes a series of experiments that illustrate when (what architectures and settings) packing is most beneficial.

2.3.1 Profiling Setup

Our server is 48-core Intel Xeon Silver 4116@2.10GHz with 192GB RAM, running Ubuntu 18.04. The GPU is NVIDIA Quadro P5000. Our evaluation uses 4 models: Multilayer Perceptron with 3 hidden layers (MLP-3), MobileNet [69], ResNet-50 [26], and DenseNet-121 [31] – with all models implemented in TensorFlow 1.15. The default training dataset is $10,000$ images from ImageNet [68] and the required input image size of each batch is 224×224 which is commonly used. Batch sizes start from 32 and goes up to 100 in the experiments [6].

In our experimental methodology, the first training step is always omitted for measurement due to the CUDA warm-up issue, and the measurement of the single step excluded loading time. We only measure the loading cost for investigating whether it dominates the performance (middle column in Figure 2.6). So, this measurement is orthogonal to any pipelining that might happen at a different level of abstraction. The results in the paper are averaged over 5 independent runs.

2.3.2 Profiling Metrics

We evaluate the `pack` primitive against three performance metrics defined as follows.

Improvement: We measure the time of a single training step of the packed model. Since one training epoch can be treated as a series of repeating training steps and a complete training process is made with multiple epochs, the single training step measurement can be used to estimate the overall training time. We denote the step time as T_s and assume that there are n models (model $1, \cdots, n$), and we compare the time of a single training step in packed and sequential mode. We first train models $1, \cdots, n$ isolatedly and sequentially, and measure the time of a single training step:

$$T_s(Seq) = T_s(\text{Model 1}) + \cdots + T_s(\text{Model } n) \tag{2.1}$$

Then, we pack these models for training and measure the single training step, which is defined as follows:

$$T_s(Pack) = T_s(Pack(\text{Model 1}, \cdots, n)) \tag{2.2}$$

Thus, we define the improvement metric as follows:

$$IMPV = \frac{T_s(Seq) - T_s(Pack)}{T_s(Seq)} \tag{2.3}$$

The improvement metric can quantify the benefits brought by `pack` primitive, and comparing $IMPV$ of various training setups can identify performance bottlenecks.

Memory: Fine-grained resource sharing (e.g., training multiple models together on a single device) requires sufficient device memory, thus measuring the memory usage of the packed model can provide insights for scheduling different models given a specific device memory capacity. We evaluate the peak of memory usage over the training epoch. This is because if the usage peak is over the GPU memory capacity, the training process will be terminated due to a GPU memory error. We measure the allocated memory and not the active memory used.

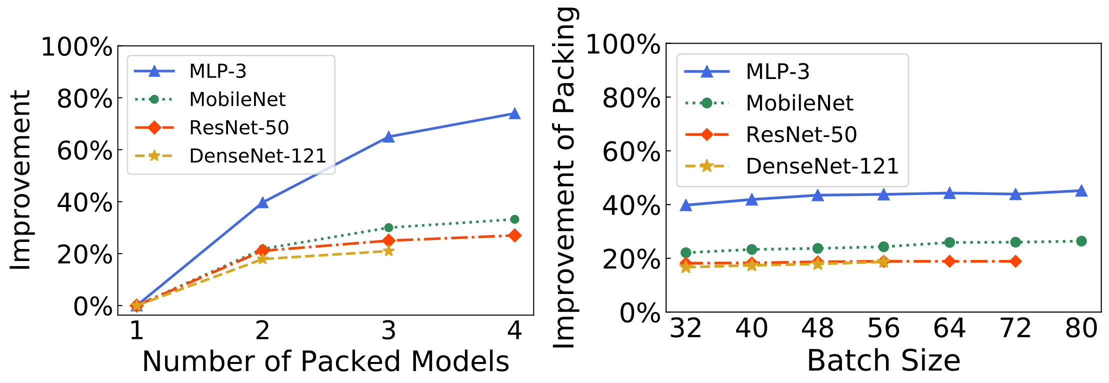
Switching Overhead: Training the models isolatedly and sequentially on a single device can bring an additional switching overhead. For example, the GPU has to unload the old model and the associated context and then load the new models and prepare the context. `Pack` significantly reduces such overhead since packing models suffer from model switching less often (multiple models can be trained together given enough GPU memory so that loading and unloading operations can be avoided). The switching overhead is measured through the following method: We train n models isolatedly and sequentially for one epoch and capture the training time, which is denoted as $T_e(Seq)$. Then we train n models individually and denote $T_e(Model)$ as the training time of one epoch for each model. Thus, the switching overhead of training n models is defined as:

$$SwOH(n) = T_e(Seq) - T_e(\text{Model 1}) \cdots - T_e(\text{Model n}) \tag{2.4}$$

However, we hypothesize that the overhead amortizes over an entire training procedure. This is because $SwOH$ depends on the number of models instead of the number of training steps and epochs. Since training a model usually involves numerous training steps and many training epochs, compared with much longer training time, the switching overhead is minor (section 2.3.5).

2.3.3 Improvement

We evaluate packing performance as a function of batch size and the number of models. Figure 2.4a shows that as the number of packed models increases so do the relative benefits until the resources are saturated. The line of DenseNet-121 ends early because that packing four DenseNet-121 takes too much GPU memory and results in an Out-Of-Memory (OOM) issue. However, the potential for resource savings is significant. If one is training multiple MLP models, there can be up to an 80% reduction in training time. In short, *it is wasteful to allocate entire devices to small models.*



(a) Improvement on various numbers of packed models

(b) Improvement on various batch sizes

Figure 2.4: *Improvement* of packing models when increasing number of models and batch size on GPU. Y-axis indicates the reduction in training time compared to sequential execution (Eq. 2.3).

Figure 2.4b illustrates the relationship between batch size and relative improvement when packing two models. The lines of ResNet-50 and DenseNet-121 both end early because the OOM issue emerges when the batch size goes to 80 and 64 respectively. These models are mostly GPU-compute bound. Increasing the batch size has a negligible improvement in time even if the packing setup can combine the data transfer. We will see that this story gets more complicated when considering preprocessing.

2.3.4 *Memory Usage*

We track the GPU memory usage of training individual models and packed models with different batch sizes for one epoch. We particularly care about the memory peak and whether it is beyond the memory capacity.

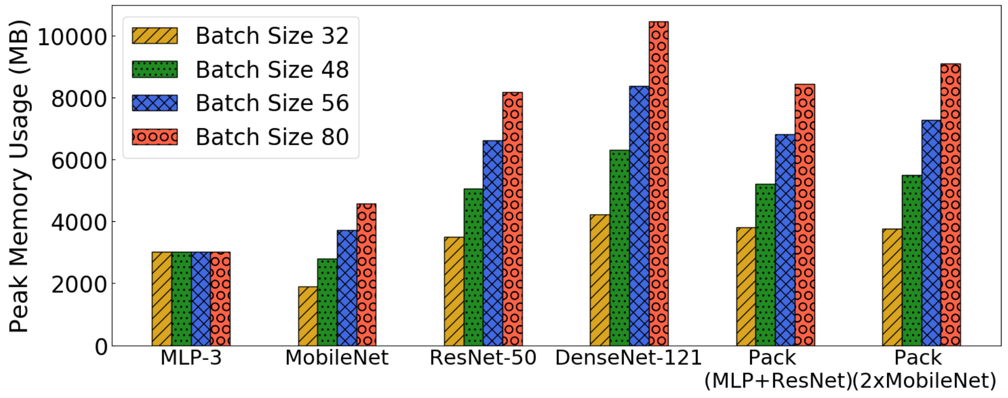


Figure 2.5: GPU memory peak of different models

As depicted in Figure 2.5, for convolutional neural networks like ResNet, MobileNet, and DenseNet, the GPU memory usage is proportional to the batch size as more intermediate results will be stored as batch size increases. Similarly, when packing two models the GPU memory usage is the sum of memory usage. However, the GPU memory usage peak of MLP-3 model maintains the same as the batch size goes up. This is mainly due to two reasons: (1) we find that for simple models TensorFlow's greedy memory allocation policy over-allocates more GPU's memory when the actual usage is lower than a specific threshold; (2) the majority of computations for MLP-3 are dot products and are placed on CPU by TensorFlow and do not occupy much GPU memory. More specifically, without any annotations, TF automatically decides whether to use the GPU or CPU for an operation [19] (we also used the TF profiler to trace the training process and found the majority of operations in MLP-3 are placed on the CPU). Thus, GPU memory usage of single MLP-3 remains the same due to the pre-allocation.

2.3.5 Switching Overheads

We profile the switching overhead of two models to illustrate how much the overhead can accumulate as more models are trained (the time `pack` can save).

	Model	$T_e(Seq)$	$T_e(Model)$	$SwOH(2)$
GPU	MLP-3	133s	61s	11s
	MobileNet	227s	107s	13s
	ResNet-50	274s	130s	14s
	DenseNet-121	305s	144s	17s

Table 2.1: $SwOH$ of training two models sequentially

As shown in the Table 2.1, albeit the accumulation, the switching overhead (using Eq. 2.4) is minor compared to the overall training time and it is even negligible when more epochs are involved in a training process. This also confirms our hypothesis of the switching overhead.

2.3.6 `Pack` vs CUDA Parallelism

Current NVIDIA GPUs support executing multiple CUDA kernels in parallel at application level. Thus, we conduct an experiment under the same environment as we used in the paper to train models in parallel at the CUDA GPU kernel. We run multiple simultaneous training processes on TensorFlow. We evaluate this method in the experiments where two processes are boosted at the same time to train the same models (MLP, MobileNet, ResNet, DenseNet) with the same optimizer and same batch size (ranging from 32 to 100).

Although CUDA supports it, our results show that it is not an efficient technique. When the models train on the same data, parallel training in isolated kernels leads to duplicated I/O and duplicated data in memory. In the image processing tasks that we consider, the training data batch takes up a substantial amount of memory. We find that in all but the simplest cases lead to an OOM error: "failed to allocate XXX from device: `CUDA_ERROR_OUT_OF_MEMORY`". We also find similar results when the models train on *different* data—as there is duplicated TensorFlow context information in each of the execution kernels. This error happens all the above experiments except packing the MLP model (due to its lightweight size).

Even with the MLP model, the `pack` primitive shows benefits at scale. For instance, the training time of a single step based on CUDA parallelism is 184ms for both two processes and the packing method takes 200ms. However, as the batch size is increased to 100, the former one takes 1660ms, while the latter one costs 1500ms. We interpret these numbers as an indication that the `pack` primitive incurs smaller context overhead over the native CUDA parallelism at application level.

2.3.7 Ablation Study

Factor	Config	Description
Model	Same	Packing two same models
	Different	Packing two different models. To figure out more configurations, we evaluated MLP-3 vs. MobileNet, MobileNet vs. DenseNet-121, ResNet-50 vs. MobileNet, and DenseNet-121 vs. ResNet-50.
Training Data	Same	All packing models take the same training batch data
	Different	All packing models take the different training batch data.
Preprocess	Yes	Preprocessing is included in each training step. Training batch are raw image (e.g., JPEG), transferring from disk to GPU.
	No	Preprocessing is excluded in each training step. Training batch are preprocessed and formatted before transferring to GPU.
Optimizer	Same	Two models use the same optimizer for single training step, e.g., both of them use Momentum optimizer.
	Different	Two models use the different optimizer for single training step, e.g., one uses Momentum, the other uses SGD.
Batch Size	Same	Two models take the same batch size for single training step, e.g., both of them take 32 batch size.
	Different	Two models take the different batch sizes for single training step, e.g., one is 32 batch size, the other is 50 batch size.

Table 2.2: Model configurations for ablation study

To further evaluate the performance of packing models on GPU, we test more cases based on the five factors: (1) whether the models have the same architecture; (2) whether the models share the same training data; (3) whether the models take the preprocessed data or raw data for training, i.e., if the preprocessing is included in training; (4) whether the models use the same optimizer; and (5) whether the models have the same training batch size. In this ablation study, we follow the configurations illustrated in Table 2.2 and evaluate the `pack` primitive. Without loss of generality, we focus on packing two models to understand the relationship between the training time and the above factors, packing more models follows the trends as demonstrated in Figure 2.4.

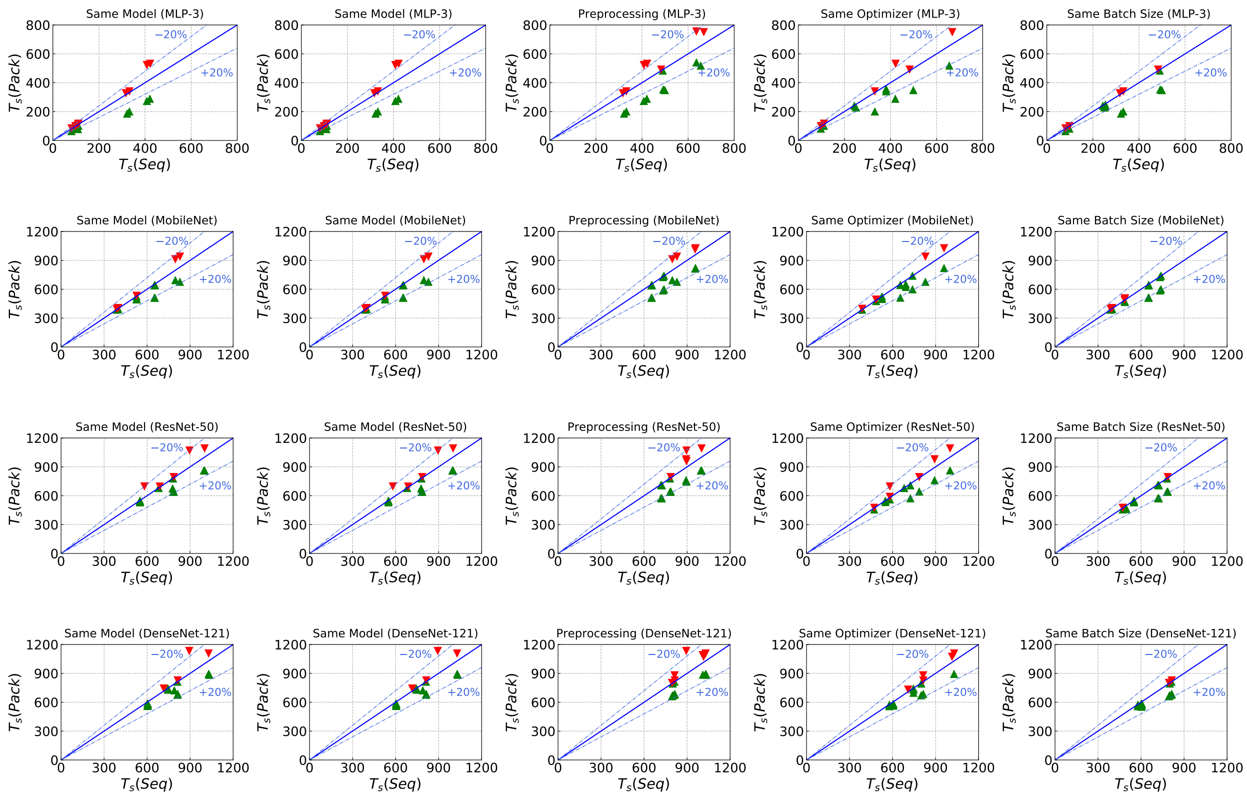


Figure 2.6: $T_s(Seq)$ vs. $T_s(Pack)$ (milliseconds) when packing two models on a GPU (NVIDIA Quadro P5000)

Figure 2.6 presents the results of the ablation study. In the figure, the data points in each subfigure represent the $T_s(Seq)$ and $T_s(Pack)$ of various configurations with fixing one configuration (e.g., same batch size or same model). The red point (triangle pointed down) indicates that packing two models brings more overhead compared with training them sequentially with this configuration, i.e., $T_s(Pack) > T_s(Seq)$, while the green point (triangle pointed up) means the opposite. The further from the line, the more significant the performance difference.

As we can see from Figure 2.6, the best scenarios are where the same training data and the same batch size are used. Over all the configurations, the `pack` primitive always brings benefits when we train models with the same data since it will reduce the data transfer. Similar benefits happen with the same batch size configuration. This is important to note because even when the same models are trained but with different data inputs and batch sizes, there can be significant downsides to packing. It is not simply a matter of looking at the neural network architecture, but the actual training procedure factors into the decision of packing.

2.4 `Pack`-Aware Hyperparameter Tuning

We demonstrate how `pack` benefits hyperparameter tuning in this section. As we show in the previous section, `pack` brings the biggest improvement when the models trained are similar and train on the same input data. Such a scenario naturally arises in hyperparameter tuning. Developers have to search over adjustable parameters such as batch sizes, learning rates, optimizers, etc. Tuning such hyperparameters is crucial to finding models that generalize to unseen data and achieve promising accuracy.

There are a number of real-world scenarios where multiple models are trained on the same data, we demonstrate hyperparameter tuning as a representative application. Furthermore, `pack` is a *simple but practical* mechanism that can be implemented at application level, which allows for a wide variety of deployment scenarios.

2.4.1 Hyperband

We explore how we can extend a state-of-the-art hyperparameter tuning algorithm, Hyperband [50], to better share GPU resources. Hyperband works by repeatedly sampling random parameter configurations, partially training models with those configurations and discarding those configurations that do not seem promising. Prior work suggests that Hyperband is effective for parallel hyperparameter search in comparison to sequential algorithms such as Bayesian Optimization [48].

Hyperband poses the search as an online resource allocation problem. Given N discrete model configurations to test, it partially trains each configuration and discards those that do not seem promising based on a technique called successive halving. The search routine follows the structure of Algorithm 1.

```
Algorithm 1: Hyperband
  input  : $R, \eta$
  output: Conf with the smallest intermediate loss so far
  for $r \leftarrow 0$ to $\lfloor log_\eta(R) \rfloor$ do
      Randomly sample T from N confs without replacement;
      for $i \leftarrow 0$ to $r$ do
          Train conf $i$ for multiple epochs;
          Calculate the intermediate loss of confs $i$;
          Keep a fraction of the best confs for the next iteration;
      end
  end
```

Intuitively, Hyperband only allocates resources to the most promising configurations. At the maximum iteration, the most promising configurations are trained for the longest. This basic loop can be trivially distributed a random partition of N configurations. Although Hyperband is able to optimize the process of hyperparameter tuning, the algorithm is long-running since it consists of a large number of trial hyperparameter configurations to run and each of them usually occupies the entire GPU resource when running.

2.4.2 `pack`-aware Hyperband

Our `pack` primitive allows Hyperband to jointly train configurations when possible thereby reducing the overall training time. We propose a `pack`-aware Hyperband that leverages model packing to improve its performance when there are more models to evaluate than available GPU devices. The challenge is to determine which configurations to train jointly and which to train sequentially.

For each iteration, our `pack`-aware Hyperband will partition sampled T models to multiple packed groups that can fit on a single device (the size of packed group do not exceed the amount of memory of the GPU). Then, the optimization problem is to search over all packable groups to find the best possible configuration (one that maximizes the overall run time). Note that the singleton partitioning (every single model forms a group) is always a viable solution and potentially even an optimal solution in some cases. We call this primitive `pack_opt`, which solves the search problem by producing feasible packing groups and identifying the most promising configuration. Accordingly, we can run a modified Hyperband loop that packs models when beneficial, as shown in Algorithm 2.

There are two challenges in `pack_opt`: *(C1)* developing an accurate cost model to evaluate the cost of a packed plan, and *(C2)* a search algorithm that can effectively scale with T. Of course, the combinatorial nature of this problem makes both *(C1)* and *(C2)* hard to accomplish optimally and we need a heuristic to address this problem. Recognizing that similar models could be packed well together, we design a nearest-neighbor based heuristic.

The method randomly selects a single configuration (out of T for each round) as the centroid and packs all other configurations similar to it until the device runs out of memory. This process is repeated until all models are packed or determined that the best choice is to run them sequentially. For calculating the similarity, we map hyperparameter configurations to multi-dimensional feature space and measure the pairwise Euclid distance among all the configurations. A user-tuned similarity threshold decides how aggressively the system will pack models. For example, considering the sampled hyperparameter configurations is shown in the Table 2.3, we take standard distance unit as 1, and compute the distance between any two configurations. For categorical hyperparameters like optimizer and activation, the distance is 0 if same and 1 if different, for numeric hyperparameters, we use the index to compute distance. So, the distance between configuration *A* [batch size:20, optimizer: SGD, learning rate:0.01, activation: ReLu] and configuration *B* [batch size:40, optimizer: Adagrad, learning rate:0.01, activation: ReLu] is 5.

Algorithm 2: Pack-aware Hyperband

```
input  : $R, \eta$
output: Conf with the smallest intermediate loss so far
for $s \leftarrow 0$ to $\lfloor log_\eta(R) \rfloor$ do
    Randomly sample T from $N$ confs without replacement;
    packed_group $\leftarrow$ pack_opt(T);
    for $g \leftarrow 0$ to packed_group do
        Train packed_conf $g$ for multiple epochs;
        Calculate the intermediate loss of packed_conf $g$;
        Keep a fraction of the best confs for the next iteration;
    end
end
```

Despite being imperfect, Euclid distance has been proven to be a practical metric. We also applied a pairwise Training Time-based distance that reflects the importance of all features using the training time metric. Specifically, we train two configurations in a packed way and a sequential way for a single step respectively, measuring the training time, and calculating the difference with normalization. We take the difference as the distance and deploy it to the pack-ware hyperband. Our empirical experiments show that the Euclid distance method is still faster than Training Time-based distance method up to 18%.

Note that since the main benefit of `pack` comes from sharing and padding the input, packing different models can still improve the performance. So, `pack` is performant in the exploration phase of various hyperparameter tuning methods. Taking Bayesian Optimization as an example, in its exploration phase, the hyperparameter configurations are sampled and evaluated for some predefined objective functions. Thus, the sampled configurations can be packed during the exploration for accelerating.

2.4.3 Evaluation for Hyperparameter Tuning

The goals of our evaluation are two-fold: first to demonstrate that `pack` can significantly improve hyperparameter tuning performance and second to evaluate our `pack`-aware Hyperband. We conduct the experiments based on the same hardware environment as illustrated in section 2.3.1. We examine the Hyperband variants on CIFAR-10 [42] which consists of 60000 color 32×32 images in 10 classes (50000 for training dataset, 10000 for testing dataset). The system's goal is to find the best configuration of those described in Table 2.3, thus all hyperparameter configurations are from the combination of all hyperparameters which has 1056 configurations in total. The input, R and η, are set to 81 and 3, according to the original Hyperband paper [50].

Hyperparameter	Value
Batch size	20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70
Optimizer	Adam, SGD, Adagrad, Momentum
Learning Rate	0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.1
Activation	Sigmoid, Leaky ReLu, Tanh, ReLu

Table 2.3: Hyperparameter configurations for evaluation

We also compare our `pack`-aware Hyperband against two other heuristics:

Random Pack Hyperband: After sampling hyperparameter configurations, the method randomly selects m configurations to pack and evaluates them together, then it keeps the best n configurations and discards the rest as the original Hyperband does.

Batch-size Pack Hyperband: Rather than randomly selecting, *Batch-size Pack Hyperband* only packs the models with the same batch size. Although the number of packed models is confined by GPU memory size, greedy method is employed (i.e. packing as many models as possible until full usage of GPU memory).

We evaluate the overall running time of Hyperband with the different `pack_opt` algorithms. As presented in the Table 2.4, all the `pack`-aware Hyperband variants can reduce the running time w.r.t the original Hyperband algorithm for all scenarios. Our proposal, *kNN Pack Hyperband*, achieves the best performance since it takes advantage of our findings from the previous section where packing the most similar models leads to the biggest improvements. The conclusion is that such an approach can save time (and consequently money) in real end-to-end tasks. A simpler heuristic, *Batch-size Pack Hyperband*, is not as effective because it under-utilizes the available GPU resources by missing packing opportunities with models with slightly different batch sizes. To emphasize this point, a *Random Pack Hyperband* can save more time than *Batch-size Pack Hyperband* since it achieves a better GPU resource utilization. Our kNN strategy gets the best of both worlds: it finds the most beneficial packing opportunities while completely utilizing the available resources, and benefits are scalable when deployed in an environment with a larger GPU resource.

	Original	Batch-size	Random	kNN	Speedup
MLP-3	9236s	5260s	3682s	3491s	$\sim$ **2.7**$\times$
MobileNet	52092s	45787s	36973s	30182s	$\sim$ **1.7**$\times$
ResNet-50	98067s	89162s	75436s	70047s	$\sim$ **1.4**$\times$
DenseNet-121	131494s	126437s	117405s	108673s	$\sim$ **1.2**$\times$

Table 2.4: Performance of `pack`-aware Hyperband

2.5 Related Work

There are number of systems that attempt to control resource usage in machine learning, specifically memory optimization [64, 87, 98, 44, 35, 71, 94, 7], but we see this problem as complementary. For example, `pack` is similar in mechanism to a recent proposal, HiveMind [64], where multiple models are fused into a single computational graph during training. However, we additionally contribute: (1) a cost-model and optimizer that decides when this fusion is most beneficial, (2) integration with a hyperparameter tuning algorithm to demonstrate end-to-end improvements over a training workload, and (3) a data batching scheme that allows packing models with different batch sizes without hurting statistical efficiency. These contributions are noted as existing limitations in HiveMind.

We also discuss the related works that study hyperparameter tuning systems and multi-tenancy systems in machine learning.

2.5.1 Systems for Hyperparameter Tuning

Since hyperparameter tuning is a crucial part of the machine learning development process, a number of systems have been proposed to scale up such search routines. For example, Google Vizier [17] exposes hyperparameter searching as a service to its organization's data scientists. Aggressive "scale-out" has been the main design principle of Vizier and similar systems [53, 47, 50].

Recently, there has been a trend toward more controlled resource usage during hyperparameter tuning. Cerebro borrows the idea of multi-query optimization in database system to raise resource efficiency [61]. HyperSched proposes a scheduling framework for hyperparameter tuning tasks when there are contended specialized resources [52]. And, some work has been done on resource management [75] and pipeline re-use [49] in the non-deep learning setting. We believe that `pack` and `pack_opt` are two primitives that are useful in hyperparameter tuning when specialized hardware such as GPUs and TPUs are limited in usage. Also, although our experiments focus on hyperparameter tuning, `pack` and `pack_opt` primitives can be easily extended to other scenarios.

2.5.2 Systems for Multi-tenancy

Most current projects about building multi-tenant systems for machine learning deployment is based on device-level placement, i.e., dividing resources at the granularity of full devices (e.g., an entire server or GPU). Here, the scheduler partitions a cluster of servers where each server has one or more GPUs for various model training tasks and seeks to reduce the overall training time by intelligent placement. Other scheduling methods have followed, such as Tiresias [21] and Optimus [66]. Several extensions have been proposed to this basic line of work including fairness [57], preemption [93], and performance prediction [101]. Gandiva is a cluster scheduling framework for deep learning jobs that provides primitives such as time-slicing and migration to schedule different jobs. CROSSBOW is a system that enables users to select a small batch and scale to multiple GPUs for training deep learning models [39]. PipeDream is a deep neural network training system for GPUs that parallelizes computation by pipelining execution across multiple machines that partitions and pipelines training jobs across worker machines [62]. Ease.ml is a declarative machine learning service platform that focuses on a cost-aware model selection problem in a multi-tenant system. [51]. Some recent works also exploit data parallelism to accelerate the training process. MotherNets can ensembles different models and accelerate the training process by reducing the number of epochs needed to train an ensemble [88]. FLEET theoretically proves that optimal resource allocation in deep learning training is NP-hard and propose a greedy algorithm to allocate resources [23].

Compared with these previous works, our prototype implements a method that can pack diverse models with different batch sizes. We also conduct a comprehensive evaluation that differentiates performance wins from variable elimination v.s. improved utilization, and highlight potential for packed models to train slower than the sum of their parts, which is only apparent with modern architectures. Taking these inspirations, we further deploy our primitives to hyperparameter tuning and show the performance improvement.

2.6 Discussion

Our core contribution is demonstrating the potential benefits (and overheads) of combining similar models into a single computational graph, and thus collapsing common data inputs during training iterations. This was the reason why we chose not to optimize this process at a lower level (e.g., MPS/Hyper-Q [8]), where we found that the majority of benefits could be attributed to simply sharing common inputs and context variables. Thus, the key goal of our proposed optimizer is to decide whether two models share enough to see a potential benefit, and controlling the exact execution order of the computation is orthogonal to our contribution.

Our long-term goal is to build a system for multi-tenant deep learning deployment, and we believe the `pack` will be one of the core parts of multi-tenancy systems for machine learning. In hyperparameter tuning there is a single-user and a clear SLO (find the best model configuration over all), then to extend to more general multi-tenancy settings where concurrent models are trained, we will reason about multiple users, priorities, and user-specified objectives. For this, we decide to first make a deep investigation on a single GPU so that we will know how to optimize when there are multiple GPUs. Thus, any distributed training and the regarding optimization is out of the scope of the paper.

We implemented `pack` on TensorFlow to conduct a comprehensive evaluation and highlight its benefits in hyperparameter tuning. Although we believe that a custom execution platform could improve performance, `pack` doesn't require the modification of any specific framework and can be implemented across frameworks. We focus `pack` as a higher-level primitive due to (1) the optimizations will be more transferable across ML execution frameworks and thus increase the impact or applicability of our insights, and (2) many low-level libraries are highly optimized and introducing these changes (e.g. supporting jagged arrays) we believe are interesting research questions on their own.

2.7 Conclusion

We analyze the benefits and limitations of packing multiple models together to take advantage of available GPU resources for model training. Under the proper conditions, this packing can bring up to 40%

reduction in latency per model packed, compared with training the models sequentially on a GPU. We further demonstrate that `pack` primitive can be used to accelerate a state-of-the-art hyperparameter tuning algorithm. Our end-to-end tuning system demonstrates a 2.7x speedup in terms of time to find the best model by improving GPU utilization. Our analysis opens many interesting optimization opportunities, such as the training process can be decomposed and scheduled for packing to reduce the overall training time, or trading off accuracy or training time to improve overall resource utilization.

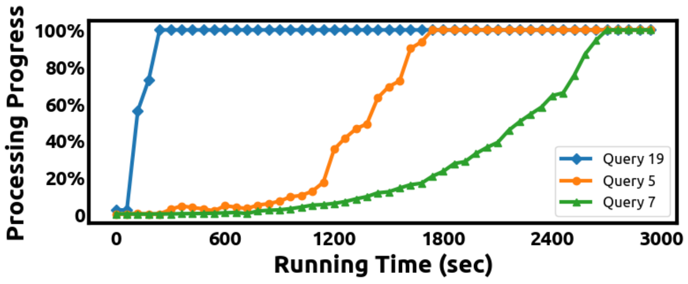
CHAPTER 3

ROTARY: A RESOURCE ARBITRATION FRAMEWORK FOR PROGRESSIVE ITERATIVE ANALYTICS

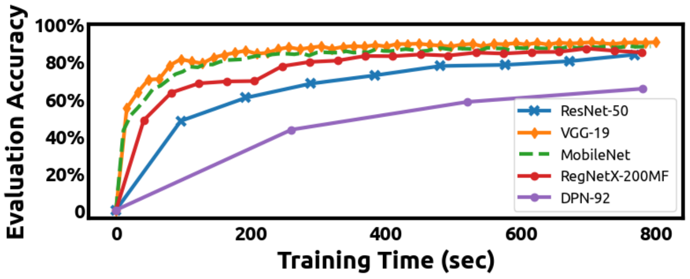
A growing number of organizations are concerned about resource usage in data analytics [41, 78, 12, 72]. The concerns have become particularly acute over the last two years where a confluence of factors, including supply chain shortages and a waning Moore's law, have discouraged organizations from simply scaling out to cope with the ever-increasing analytics demands. In this resource-limited world, every organization needs to determine how to partition and share computational infrastructure to adequately support all of its analytics users [5, 24, 4, 16, 59]. While similar to traditional scheduling and resource allocation problems, existing solutions do not fully exploit properties of modern data analytics workloads.

Modern analytics algorithms are often progressive, where an iterative loop repeatedly refines an answer until a desired convergence criterion is met. In this setting, completion is a matter of user opinion, where a user-defined rule has to be used to terminate the job when the answer is deemed sufficiently accurate or unchanged. A traditional job scheduler would place an immense level of trust in a user's ability to appropriately specify these termination criteria. For example, consider a user training a convolutional neural network for a fixed time of 500 epochs. Suppose the model actually converges in accuracy after only 100 epochs, then 80% of this model's training time is a wasteful block on system resources. The longer an analytics job runs, the risk increases that a mis-specified completion criterion (e.g., an overly ambitious accuracy or convergence threshold) can block key resources from others for an extended amount of time. Ideally, a scheduler has introspection into the convergence progress of a progressive analytics job to detect and preempt such anomalies. These decisions cannot be made in a vacuum and need to consider a job's prioritization, specified completion criteria, limited available resources, and other jobs waiting for the resources — a decision we call *resource arbitration*.

We identify two very different analytics applications that fit this resource arbitration paradigm: approximate query processing (AQP) and deep learning training (DLT). In AQP, one executes queries on a subset of the overall dataset or a data stream to return an approximate answer within a user-specified er-



(a) Online aggregation progress of Query 5, 7, 19 of TPC-H. The percentage of data processed achieves 100% when the queries received and processed the entire TPC-H dataset (SF=1) in batches from a data source.



(b) Evaluation accuracy of five well-tuned popular convolutional neural network models on CIFAR-10 with batch 128 and learning rate 0.01.

Figure 3.1: Progress curves of AQP and DLT jobs

ror. Consider an AQP scenario where multiple users are issuing approximate queries to the same OLAP infrastructure, the need for resource arbitration arises because complex jobs with lower error tolerances (higher accuracy) can dominate execution and starve jobs with higher error tolerances (lower accuracy). In DLT, one updates the parameters of the neural network based model with a variant of gradient descent repeatedly until the desired objective (e.g., accuracy or convergence) is reached. Consider a DLT scenario where multiple users use the same GPU cluster to train their models, the need for resource arbitration arises because larger neural networks with tighter convergence criteria can dominate GPU utilization, and starve resources from smaller networks or ones with looser convergence criteria. In both of these scenarios, one needs a resource arbitration system that *can pause a running job at the risk of dequeuing it in a partially complete state, in favor of jobs that could better use the same resources*. This strategy is only useful in a setting where an intermediate result has significant utility to a user, and might even be indistinguishable from the fully complete result.

The connection between AQP and DLT is not arbitrary, we treat them as two representative examples of progressive iterative analytics, and there are three unique traits of these two progressive iterative analytic applications that make them particularly attractive for an initial study in resource arbitration. First, users have diverse completion criteria for different progressive iterative analytic jobs, and there is no unified finish-line as found in classic data processing jobs (i.e., going through all the data). For example, some users can have an ambitious objective (e.g., 99% accuracy), but the others only need to run their jobs for a while (e.g., 10 mins) to get a rough idea. Second, progressive iterative analytic jobs often (eventually) produce diminishing returns. For instance, for a progressive iterative analytic job, the first batch of data can take users from zero knowledge of the final result to a very rough estimation, but the final batch of data may only allow users to go from a very precise estimation to the final result, and in between, and the improvement of each batch are not linear. Third, progressive iterative analytic jobs process data at a different rate, which indicates the jobs may take different amounts of time to reach the same degree of progress. Such a difference would overly prioritize short jobs when the jobs are ranked based on the likelihood that they will complete sooner or achieve higher progress. We plot the progress curve of sample AQP and DLT jobs in Figure 3.1 to demonstrate these three traits.

Motivated by the traits of progressive iterative analytic applications, we propose a resource arbitration framework, which essentially determines how to prioritize progressive iterative analytic jobs for limited resources and if/when to interrupt a currently running job in favor of another. *The need for a resource arbitration framework arises for two reasons: (1) from the perspective of single jobs, it is reasonable to sacrifice precision for a quicker result; (2) from the perspective of the overall workload, it is beneficial to dynamically allocate and preempt resources to different jobs, for example, giving more resources to more promising jobs and constraining the resources for jobs stop progressing.*

While scheduling systems [84, 13, 82, 28, 66, 90, 22, 34, 100, 85, 67] and resource arbitration sound similar in nature, they actually solve different, complementary problems. Scheduling systems are generally designed to optimize the execution and placement of the jobs according to the users' resource requirements and ensure the jobs can be completed on time. By contrast, resource arbitration systems are responsible for continuous resource allocation and preemption, determining when to start (or resume) and stop (or checkpoint) the progressive iterative analytic jobs based on the processing progress, real-time available resources, and users' completion criteria. One application scenario of resource arbitration is hyperparameter optimization [50] for deep learning models, where a set of hyperparameter configurations are sampled from a hyperparameter space and formed a number of training trails that run iteratively and keep returning intermediate training results. Such a process is executed repeatedly until the best-performed hyperparameter configurations are selected. Thus, resource arbitration could stop the trials that contain unpromising hyperparameter configurations prematurely and allocate more resources to the promising ones so that the best-performing hyperparameters can be discovered as soon as possible.

To realize this framework, we implement two prototype systems *Rotary-AQP* and *Rotary-DLT* for approximate query processing and deep learning training applications. For Rotary-AQP, we first extend a single-user progressive query processing system based on Apache Spark [74] and modify it to a multi-tenant AQP system. Then, we build the resource arbitration system on top the multi-tenant AQP system. We evaluate Rotary-AQP using the TPC-H benchmark, and the evaluation results show that Rotary-AQP outcompetes the state-of-the-art system and other baselines by allowing more TPC-H queries to reach their goals within the same amount of time. For Rotary-DLT, we build the system on top of TensorFlow [81] and conduct an evaluation using the workloads derived from a survey of 30 deep learning researchers across multiple research organizations. The evaluation results demonstrate the performance of Rotary-DLT by outperforming three heuristic baselines across a variety of optimization objectives. The two system implementations and their outstanding performance confirm the generality and practicality of our resource arbitration framework.

To summarize, our primary contributions include: (1) defining the resource arbitration for iterative applications and highlighting its importance; (2) proposing a general resource arbitration framework, Rotary, and a new cost model that leverages the estimation of progress and resource consumption for job prioritization; (3) implementing two resource arbitration systems for approximate query processing and deep learning training, following the proposed framework.

3.1 Resource Arbitration Framework

Here, we describe our resource arbitration framework, Rotary.

3.1.1 Terminology and Setup

First, we define a common set of terms to describe progressive iterative analytic jobs. In a progressive iterative analytic job, data are processed in batches, where each *batch* is a subset of the entire dataset or a data stream that is progressively sampled from the overall data, each batch has the (approximately) same batch size. A progressive iterative analytic job moves one *step* when it finishes processing a single batch. After a fixed number of such steps (called an *epoch*), the job's performance can be evaluated based on *convergence metrics* on the returned results. Convergence metrics are usually some proxy for result accuracy. One progressive iterative analytic job typically runs for multiple epochs until the user is satisfied with its convergence.

Example 1. Approximate Query Processing in SQL

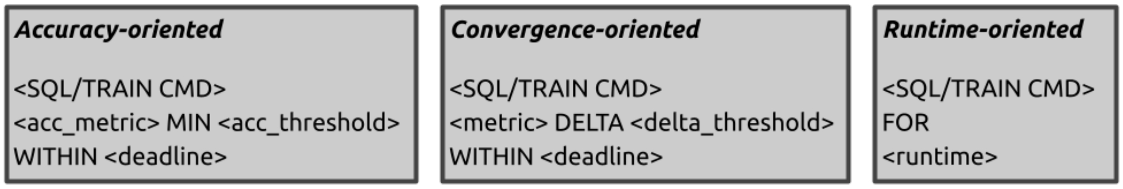
Approximate query processing can provide quick, approximate results to users by running queries on a subset of the overall dataset or a data stream. One technique to realize AQP is online aggregation [46]. Online aggregation systems process data iteratively using data batches, each progressive sampling of the data is a batch and processes roughly the same amount of data, as they are each of approximately the same size. Online aggregation systems calculate error bounds such as confidence intervals after each batch is processed so that users can make decisions about whether to continue processing. In AQP, a batch and an epoch can be synonymous, and the convergence metric is the size of the confidence interval.

Example 2. Deep Learning Training

A typical DLT job consists of a neural network model (e.g., ResNet [25] or Bi-LSTM [20]), a dataset for training and evaluation, and a set of hyperparameters (e.g., batch size, learning rate, optimizer, etc.). During the training process, the training dataset is iteratively sampled in batches, and each training step is one optimization step updating the parameters (or gradients) of the neural network model based on the batch. In the context of DLT, an epoch normally is a complete pass of the training data. Once the neural network model has been trained for one epoch, it will be evaluated on the evaluation dataset in terms of either training loss or validation set accuracy. This process is applied repeatedly until the desired convergence target is achieved. As models have become more complex, DLT largely relies on specialized hardware devices like GPUs and TPUs.

3.1.2 User-defined Completion Criteria

Rotary allows users to define three types of completion criteria based on the common practice of DLT and AQP. As presented in Figure 3.2, there are ❶ *accuracy*-oriented completion criteria, ❷ *convergence*-oriented completion criteria, and ❸ *runtime*-oriented completion criteria. Essentially, such completion criteria are add-ons to the regular query and training commands and should be orthogonal to the execution of AQP and DLT.



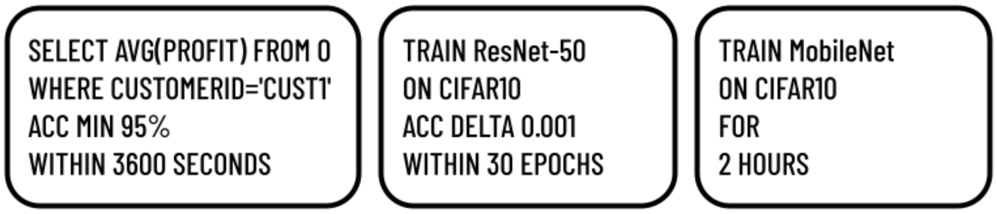
```
Accuracy-oriented

<SQL/TRAIN CMD>
<acc_metric> MIN <acc_threshold>
WITHIN <deadline>
```

```
Convergence-oriented

<SQL/TRAIN CMD>
<metric> DELTA <delta_threshold>
WITHIN <deadline>
```

```
Runtime-oriented

<SQL/TRAIN CMD>
FOR
<runtime>
```

Figure 3.2: Templates of user-defined completion criteria

Figure 3.3 shows examples of completion criteria templates. Specifically, in the three examples, the left one illustrates how to add a completion criterion of achieving at least 95% accuracy within 3600 seconds, the middle one defines a completion criterion for training a ResNet model until reaching the convergence of 0.001 within 30 epochs, the right one will train the MobileNet model for 2 hours and return the training results anyway.



```
SELECT AVG(PROFIT) FROM O
WHERE CUSTOMERID='CUST1'
ACC MIN 95%
WITHIN 3600 SECONDS
```

```
TRAIN ResNet-50
ON CIFAR10
ACC DELTA 0.001
WITHIN 30 EPOCHS
```

```
TRAIN MobileNet
ON CIFAR10
FOR
2 HOURS
```

Figure 3.3: Examples of user-defined completion criteria

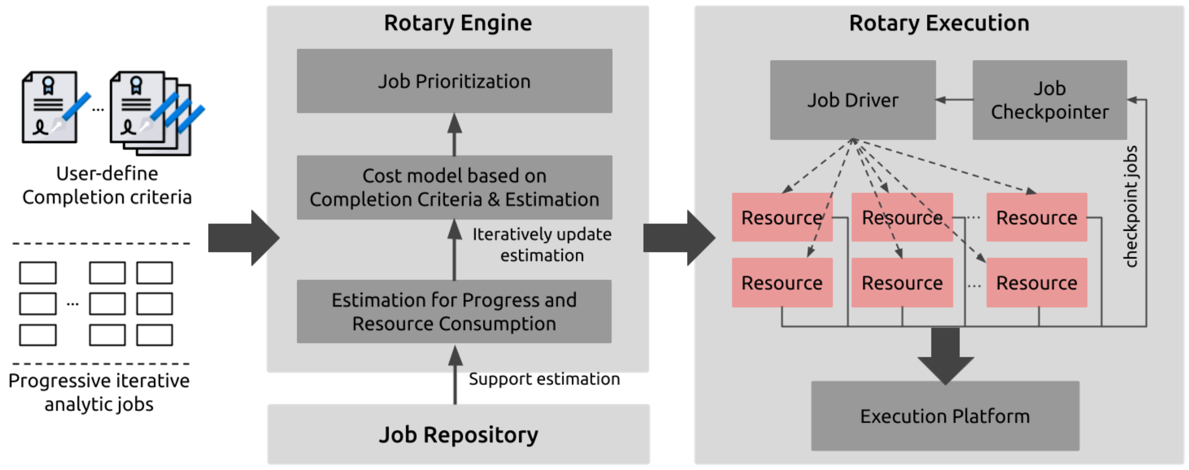
❶ Accuracy-oriented completion criteria are widely used and allow users to explicitly specify an expected accuracy within maximum training epochs. In the above example, we use *ACC* (i.e., training accuracy) which is a common metric, but other user-defined metrics such as F1 score and Perplexity are supported as well. Additional error bound such as confidence interval are optional as well. The deadline could be expressed in epochs or time units such as seconds, minutes, and hours.

❷ Convergence-oriented completion criteria are also typical, especially for DLT jobs. With this kind of criteria, a job is considered "complete" once its performance is found to no longer increase. In the middle example of Figure 3.3, *ACC* is used for measuring convergence, but other metrics such as *LOSS* [18] are also supported for convergence. The convergence-oriented criteria also allow users to specify a deadline, which means a job will be terminated if it fails to converge until the deadline.

❸ Runtime-oriented completion criteria are proposed for the users who want to execute their progressive iterative analytic jobs for a while without any explicit objective or threshold. As the *WITHIN* predicate we have in the other two completion criteria, the runtime can be the number of epochs or a period of time such as training a model for 100 epochs or running a query for 6 hours.

3.1.3 Framework Architecture

We identify three opportunities to address resource arbitration problem. First, the diverse completion criteria of progressive iterative analytics bring the opportunity to allocate various amounts of resources to different jobs while still achieving their objectives. For example, it makes more sense to give less resource to the job that only need to achieve an effortless objective. Second, diminishing returns of progressive iterative analytic jobs indicates that the value of two data batches to a user may be poles apart. This makes iterative resource allocation and preemption practical and valuable, for example, the jobs that are processing the data batch which can provide more valuable results to users can be finished sooner if more resources are allocated continuously. This leads to a cost model which should balance the progress improvement (i.e., providing more valuable results) and resource consumption (the cost to improve the progress or produce the results). An example of this can be seen in Figure 3.1b, where we show that the earlier training epochs could improve the deep learning models' accuracy more significantly than the older ones, and the users could get a decent trained model more quickly if more resources are given to the jobs with more potential for improvement, however, the trade-off between performance improvement and the models' GPU memory requirements need to be addressed as well. Third, different data processing rate of progressive iterative analytic jobs rationalizes the adaptive running epochs, namely, long-running jobs should be allowed to have a longer running epoch after arbitrating and allocating resources so that they can return expected intermediate results. This can be exemplified by Figure 3.1a, where we present that the process of query 19 increases more expeditiously than query 7 and 19 when they are all checked every 60 second, however, we can observe all the queries will have the similar pattern of progress improvement if query 5 and query 7 are check every 120 and 180 seconds.


User-define Completion criteria
Progressive iterative analytic jobs
Rotary Engine
Job Prioritization
Cost model based on Completion Criteria & Estimation
Iteratively update estimation
Estimation for Progress and Resource Consumption
Support estimation
Job Repository
Rotary Execution
Job Driver
Job Checkpointer
Resource
checkpoint jobs
Execution Platform
Figure 3.4: Framework architecture of Rotary

To exploit these opportunities, we proposed the resource arbitration framework, Rotary. Here we discuss the framework's architecture and highlight the core components in Figure 3.4. Rotary allows users to submit their progressive iterative analytic jobs along with the corresponding completion criteria. Once submitted, Rotary considers these jobs active and is ready to run them. Rotary's engine is responsible for resource arbitration. It can estimate how much processing progress a job can achieve in terms of completion criteria and how many resources the job will consume for such progress. Rotary can prioritize jobs according a cost model and arbitrate the resources for them. Once the process of resource arbitration is finished, the selected jobs will be deployed in Rotary execution, where the resource is allocated and preempted to the jobs so that they can run in an execution platform (e.g., PyTorch or TensorFlow for deep learning, Spark for query processing). When the jobs complete the current epoch, they can be checkpointed or materialized if they are not granted resources for the next running epoch. Furthermore, it is beneficial to store the progressive iterative analytic jobs and track intermediate processing results since such information can be used to provide a better estimation.

Rotary should be able to consider and select the jobs had been deferred before when it is beneficial to do so. This ability provides the resource arbitration system a wider range of running option for progressive iterative analytic jobs. More specifically, for each available resource, only the selected job by Rotary can take it, which could result in deferring some other jobs too far into the future. Such deferment may be computationally expensive and worsen the diminishing returns. However, Rotary can re-evaluate the jobs that are not selected for the current available resource so that deferring job far into the future are less likely to hold.

3.1.4 Resource Arbitration Problem Statement

Workloads. Consider a workload W that consists of n progressive iterative analytic jobs $\{j_1, \cdots, j_n\}$, each job processes data batch-by-batch and returns the intermediate processing results for every epoch. Each i^{th} job emits a time-series per epoch measuring the convergence $acc_{i,0}, acc_{i,1}..., acc_{i,T}$. Each job has a specific user-defined completion criterion c, which terminate if $c(acc_{i,t}) ==$ `true`. Thus, there is a list of criteria $C = \{c_1, \cdots, c_n\}$ associated with jobs in the workload W. Once a job w reaches its completion criteria, it is de-queued $W = W \setminus w$.

Resources. These jobs have to be assigned to a particular "computing resource" (e.g., an available GPU or CPU core). There are M such resources considered and they are possibly heterogeneous. These resources can only process one job at a time and are not sub-dividable. A job holds on to a particular resource for at least an epoch. Thus, at any given time t, the current resource usage can be modeled as a bi-partite assignment where a subset of jobs are mapped to unique resources $\texttt{assign}(W, M)$. As these assignments change, jobs have to be loaded to the resources and check-pointed accordingly.

Resource Arbitration Policy. A resource arbitration policy is a function that produces assignments decisions based on the current state of the queue Q_t, which is the convergence state of all of the jobs currently in the queue.

$$\pi : Q_t \mapsto \texttt{assign}(W, M)$$

The application of this policy results in a sequence of resource assignment decisions at each time-step.

Objective. The queue is empty if all of the jobs in W have been successfully completed. At each time-step t, $A_t = |W|$ quantifies the number of jobs remaining in the queue (or the attainment rate). The objective of Rotary is to maximize the attainment rate of the workload W. However, the objective is implementation-specific, which means various objectives can be supported in different implementations.

Algorithm 3: Algorithm Sketch for Resource Arbitration

```
while not all jobs reach completion criteria do
    for jobs is active but not attained $j_i$, $i \leftarrow 1$ to $n$ do
        Estimate the processing progress $\hat{\phi}_{j_i}$ for next epoch;
    end
    Resource arbitration for active jobs based on $\{\hat{\phi}_{j_i} | \forall i = 1..n\}$;
    for selected jobs do
        Executing the selected job;
        Observe the current progress for the selected job;
    end
end
```

We propose an algorithm sketch for addressing the problem, as presented in Algorithm 3. However, the system implementations for various applications may have different algorithms to address the problem. Following the algorithm sketch, we design two algorithms in the system implementations for AQP (§3.2.1) and DLT (§3.3.1).

3.2 Rotary for AQP

Following the proposed framework, we illustrate how we implement and evaluate the resource arbitration system, Rotary-AQP, for approximate query processing applications.

3.2.1 Rotary-AQP Implementation

To implement Rotary-AQP, we modify a single-user progressive query processing system based on Apache Spark [74] and make it support multi-tenant environments by adding concurrency control and checkpoint mechanisms. This system serves as our execution platform to run the AQP jobs.

Rotary-AQP supports AQP jobs with the accuracy-oriented completion criteria, which is a widely-used metric in AQP [96, 76, 3, 72], namely, each job is attached with an accuracy threshold and a deadline to reach such threshold, thus the processing progress in the framework is measured in terms of accuracy in this implementation of AQP. Rotary-AQP processes AQP jobs and arbitrates the resources for them so that more jobs can reach their accuracy threshold. Rotary-AQP focuses on online aggregation [27]. The accuracy of an aggregation is calculated as $accuracy = \frac{\alpha_c}{\alpha_f}$, where α_c is the current aggregation result and α_f is the final aggregation result. Considering the aggregation operations are column-oriented, the accuracy of an AQP job that performs multiple aggregation operations on multiple columns can be calculated as $accuracy = \frac{1}{k}\sum_{i=0}^{k} \alpha_c^k/\alpha_f^k$, where α_c^k is the current aggregation result on column k and α_f^k is the final aggregation result on column k. This is based on the assumption that all columns are of equal importance (which is applied to our evaluation). However, Rotary-AQP also allows the users to specify the importance of each column by assigning weights.

We use a non-parametric confidence interval estimator to assess convergence. The technique is based on envelope functions from empirical process theory [50]. Rotary-AQP keeps tracking the least and largest aggregation results within a time window (e.g., t epochs) and uses this gap to determine convergence [1]. Given that the aggregation will eventually converge, the gap between the least aggregation result (denoted by p) and the largest aggregation result (denoted by q) can be considerable but should be shrunk gradually over time. Thus, the accuracy progress can be expressed as $\frac{p}{q}$, which can provide an approximate estimate for the accuracy progress of an aggregation operation in the AQP jobs.

Following the architecture in Figure 3.4, Rotary-AQP has two core components for estimating the accuracy progress and memory consumption. The former is used to prioritize jobs, and the latter is to make sure there will be sufficient memory to support jobs. The accuracy progress estimator in Rotary-AQP is responsible for estimating the potential accuracy of a job j for the next epoch if the resources are granted. The core idea is to fit a progress-runtime curve leveraging the historical and real-time data. The historical data can be obtained based on the archived jobs in the job repository which are similar to the job j according to various factors including predicate, table, dataset, and batch size. The real-time

1. The formal derivation of this estimator has been cut for brevity.

data can be conveniently obtained since Rotary-AQP tracks the running AQP jobs. Weighted linear regression [38] is exploited in the accuracy progress estimator for combining the historical and real-time data. More specifically, the estimator selects top-k similar historical data for an AQP job from the job repository to fit an initial progress-runtime curve that can be used for the first estimation. Then, when the job is placed and launched, Rotary-AQP records the real-time intermediate aggregation results and continuously adjusts the fitted curve by adding these real-time results of the job. Due to the importance of the real-time results, each of them should share equal weights with all the historical data when fitting the progress-runtime curve. For example, when adding a real-time result, it is granted 0.5 weight and all the historical results of the selected historical jobs from the job repository as a piece gets another 0.5 weight. This continuous joint fitting method makes the estimated progress-runtime curve reasonably close to the ground truth and sufficient for estimating the progress if the jobs are allocated with the resource and could run for a specific time.

The memory consumption estimator is designed to predict the memory consumption of the AQP jobs. Inspired by the cost-based optimization [14] in Apache Spark, the estimator can estimate the memory consumption of AQP jobs based on the table and column statistics of each batch and the predicates in the AQP. The column statistic and the predicates can be obtained by utilizing *EXPLAIN*, which can provide logical and physical plans for a query statement, before actually running the AQP jobs. Rotary-AQP also tracks the number of table rows has been scanned, filtered, and aggregated. This estimate need not be perfect but it needs to be close enough to allow the system to estimate how much memory an AQP query will require.

The memory consumption is also used to support adaptive running epoch and determine the length of the running epoch (e.g., the number of batches in an epoch). Due to our observation that the AQP jobs consume larger memory usually proportionally take a longer time to process a batch, which deserves a longer running epoch. Thus, Rotary-AQP makes the length of the running epoch of every AQP job proportionate to the estimated memory consumption.

In light of the estimated accuracy progress and memory consumption, Rotary-AQP can arbitrate resources, namely CPU cores, for the jobs. This process is presented in Algorithm 4.

Algorithm 4: Resource Arbitration for AQP

```
Input  : Workload $W = \{j_1, \cdots, j_n\}$
         Completion Criteria $C = \{c_1, \cdots, c_n\}$
         Total CPU cores $D$
         Total memory $M$
//All jobs are placed in an active queue when arriving
for job $j_i \in W$ that arrives do
    Mark $j_i$ as active and place it to the active queue $AQ$;
end
//Resource arbitration for the jobs in the waiting queue
while $AQ \neq \varnothing$ do
    Initialize priority queue $PQ$;
    for active jobs $j_i$, $i \leftarrow 1$ to $n$ do
        Estimate the memory consumption $\hat{m}_{j_i}$;
        Assign running epoch $e_{j_i}$ for job $j_i$;
        Estimate the accuracy progress $\hat{\phi}_{j_i}$;
        Place $j_i$ in $PQ$ due to $\hat{\phi}_{j_i}$;
    end
    ResourceArbitration(active jobs);
    Run active jobs, and mark them as running;
    for active jobs $j_i$, $i \leftarrow 1$ to $n$ do
        if $j_i$ finish one epoch $e_{j_i}$ then
            Observe the accuracy progress $\phi_{j_i}$ for current epoch;
            if job $j_i$ meets $c_{j_i}$ then remove from $AQ$ ;
            Mark job $j_i$ as active;
        end
    end
end
Function ResourceArbitration(jobs):
    for job $j_k$ in jobs do
        if $\hat{m}_{j_k} \leq M$ then
            allocate 1 core to job;
            $D = D - 1$;
            $M = M - \hat{m}_{j_k}$;
        else
            if job $j_k$ in $PQ$ then remove job $j_k$ from $PQ$ ;
        end
    end
    while $D \neq 0$ do
        for job $j_k$ in $PQ$ do
            allocate extra 1 core to job $j_k$;
            $D = D - 1$;
        end
    end
End Function;
```

3.2.2 Rotary-AQP Evaluation

Our evaluation for Rotary-AQP addresses the following questions:

- Can resource arbitration improve the number of jobs that attain their performance objective, compared to greedy baselines and a state-of-the-art approach? (§3.2.2)
- What is the overhead of resource arbitration? (§3.2.2)
- How does the distribution of job resource requirements impact the performance of Rotary-AQP? (§3.2.2)
- How does the progress estimation impact the performance of Rotary-AQP? (§3.2.2)

All the experiments are conducted on a server with two Intel Xeon Silver CPUs (2.10GHz, 12 physical cores) and 192GB memory, running Ubuntu Server 18.04. For all experiments, we use 20 physical cores and leave the rest for the OS (Ubuntu 18.04). We use an Apache Kafka [36] cluster on a different machine with the same hardware configuration as the data source for AQP queries.

We implement four baselines for comparison: ReLAQS [76], EDF (Earliest Deadline First), LAF (Least Accuracy First), and roundrobin. As a naive baseline, roundrobin allocates one core to each job in turn until there are no more cores and run them for an epoch per time until they reach their completion criteria (either achieve the accuracy threshold or beyond the deadline). EDF and LAF are two heuristic baselines, which always prioritizes the jobs have the earliest deadline and least accuracy respectively. ReLAQS is the state-of-the-art work, which is a multi-tenant system for AQP that aims to reduce the average latency of a workload by scheduling CPU cores to jobs with the most potential for improvement. In ReLAQS, the potential improvement of each job is simply estimated according to previous processing results. Compared with ReLAQS, Rotary-AQP considers both CPU and memory for resource arbitration, combines historical and real-time data to estimate the accuracy progress and overcome issues like cold-start, and supports adaptive running epoch for short-running and long-running AQP jobs.

AQP Workload

We evaluate Rotary-AQP using the TPC-H benchmark. Rotary-AQP supports all 22 queries and runs them on the TPC-H dataset with the default scale factor. Larger scale factors should not affect the performance of Rotary-AQP but require more memory to run multiple AQP jobs simultaneously. Given the number of concurrent jobs and Spark's in-memory requirement, we limit the scale-factor to 1.

The workload consists of 30 AQP jobs, each of which is a random query selected from the 22 TPC-H queries. According to the observed memory consumption of queries, we categorize the TPC-H queries into three groups: light, medium, and heavy query. The workload is a mixed collection of jobs for the queries from the three groups, and the proportions of the jobs in the three groups can be tweaked to reflect different purposes. In the workload, each job is attached with a accuracy threshold and deadline, which are both randomly selected from two parameter spaces. Furthermore, to simulate users submitting approximate queries to the shared cluster, the job arrives according to a Poisson distribution with the mean arrival time of 160 seconds. The configurations of the workload are elaborated in Table 3.1.

Queries	Light	q1, q2, q4, q6, q10, q11, q12, q13, q14, q15, q16, q19, q22
	Medium	q3, q5, q8, q17, q20
	Heavy	q7, q9, q18, q21
Completion Criteria	Accuracy	55%, 60%, 65%, 70%, 75%, 80%, 85%, 90%, 95%

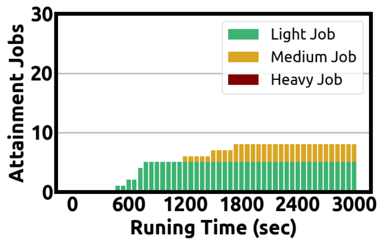
	Deadline	Light Queries Deadline (sec):
		360, 420, 480, 540, 600, 660, 720, 780, 840, 900
		Medium Queries Deadline (sec):
		1080, 1200, 1320, 1440, 1560, 1680, 1800, 1920, 2040, 2160
		Heavy Queries Deadline (sec):
		1440, 1620, 1800, 1980, 2160, 2340, 2520, 2700, 2880, 3060

Workload	40% AQP jobs with light queries 30% AQP jobs with medium queries 30% AQP jobs with heavy queries	

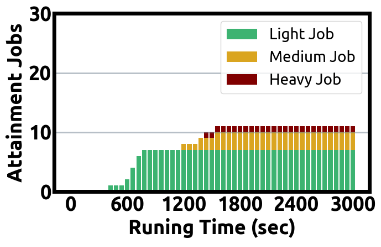
Table 3.1: Synthetic AQP workload. The selection of query type, accuracy threshold, and deadline, are all random and based on a uniform distribution. Job arrival is based on a Poisson distribution.

Attainment

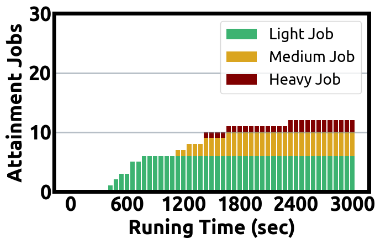
Attainment rate serves as the most important benchmark since it measures how many jobs reach their accuracy threshold, namely, users are satisfied with the results. Figure 3.5 shows the overall number of attained jobs (e.g., jobs that met their convergence criteria before their deadline) under Rotary-AQP, which exceeds those using the four baselines. Although Rotary-AQP can attain more jobs for light, medium, and heavy queries, it performs best for the jobs with heavy queries. This is mainly due to two reasons. First, Rotary-AQP can provide better progress estimation by jointly leveraging historical and real-time data, so that it can always find the jobs with the most potential for improvement. Second, Rotary-AQP can give the proportional running epochs to various jobs according to their job size (i.e., estimate memory consumption in the implementation) so that the heavy jobs which often are long-running jobs can return progressive results and be fairly compared with the short-running jobs during resource arbitration. Thus, compared with the baselines, Rotary-AQP allows the heavy jobs to have a higher chance to gain more resources for running. Such results confirm the efficiency and effectiveness of Rotary-AQP in terms of attainment rate.



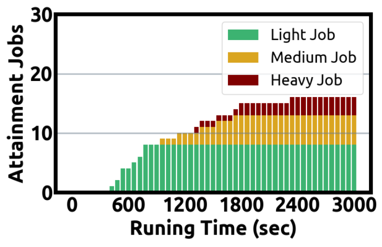
(a) Roundrobin



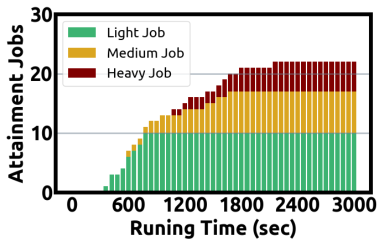
(b) EDF



(c) LAF



(d) ReLAQS

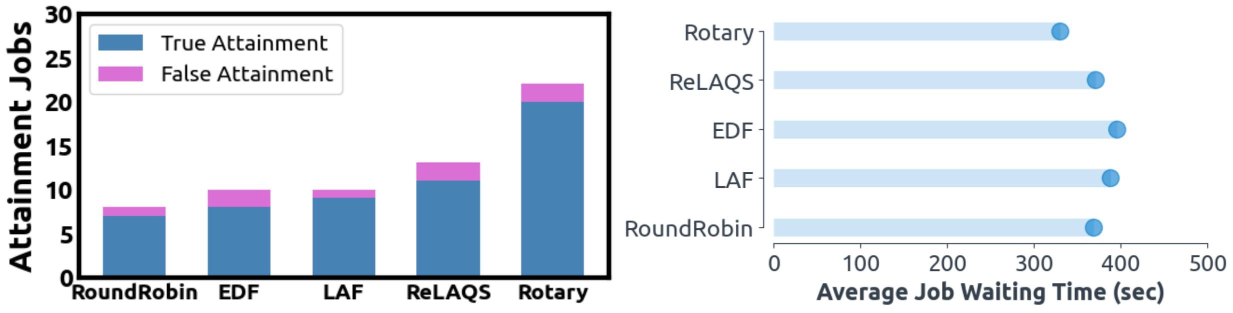


(e) Rotary-AQP

Figure 3.5: Evaluation of Rotary-AQP and four baselines (Roundrobin, EDF, LAF, ReLAQS) on the synthetic AQP workload

False Attainment and Waiting Time

Since we use an envelope function to determine when to stop the jobs, it is possible for the envelope function to make mistakes, namely stopping the jobs which are not supposed to be permanently terminated, which we consider as false attainment. We present the false attainment for Rotary-AQP and the baselines in Figure 3.6a. As we can see, the envelope function can provide reliable decisions generally, but still make mistakes. This issue can be mitigated by lengthening the time window of the envelope function.



(a) False Attainment

(b) Average Waiting Time

Figure 3.6: False attainment and waiting time of Rotary-AQP

We also tally the average waiting time of the jobs in the workload, as shown in Figure 3.6b. The waiting time of a single job is calculated as the difference between its running time under Rotary or other baselines and the time of running it independently and isolated. Our system also outperforms other baselines due to the adaptive running epochs. More specifically, unlike Rotary-AQP, other baselines are in favor of short-running jobs which can achieve higher accuracy progress in a short time which may defer the heavy job far into the future and lead to an unexpected long waiting time for the long-running jobs.

Job Distribution

To evaluate Rotary-AQP on a balanced workload, we have 40% jobs with light queries, 30% jobs with medium queries, and 30% jobs with heavy queries. However, it is also reasonable to fathom the performance of Rotary-AQP on the workload with various job distributions. For this, we deploy Rotary-AQP and the baselines in three "extreme" case: the workloads only consist of jobs with light jobs, medium jobs, and heavy jobs.

As we can see from Figure 3.7, Rotary-AQP can achieve the best performance for all three skewed workloads, especially in the workload that only contains heavy jobs. Rotary-AQP and ReLAQS can defeat other baselines due to the ability of progress estimation, whereas Rotary-AQP performs better than ReLAQS because Rotary-AQP can collect more accurate real-time intermediate results due to the adaptive running epochs to make more reliable progress estimation for the next epoch.

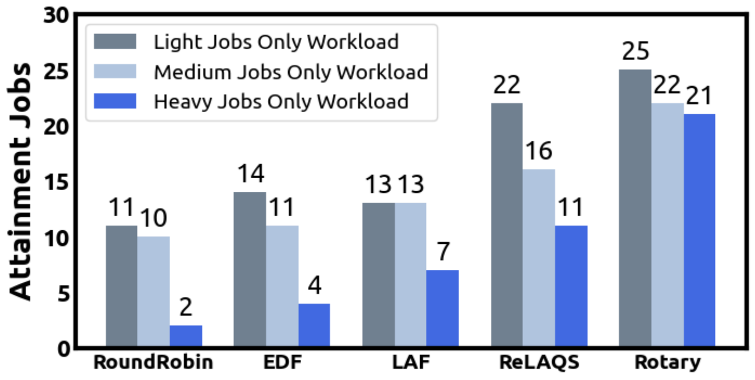
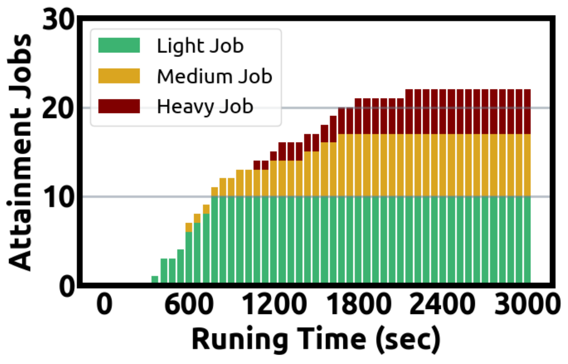


Figure 3.7: Attained jobs in the various workloads (30 jobs)

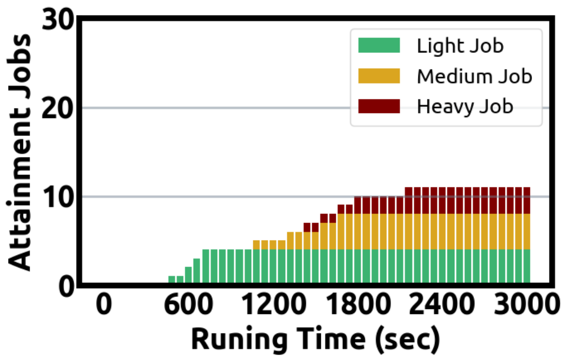
Progress Estimation Sensitivity

Since the accuracy progress estimator serves as a core component to Rotary-AQP, we investigate how much it affects the performance of Rotary-AQP. Thus, we design a new baseline which is essentially Rotary-AQP but their accuracy progress estimator will randomly return the estimated progress following a uniform distribution from 0 to 1. Such artificial progress estimation is misleading, and Rotary-AQP may make unwise resource arbitration accordingly.

Figure 3.8b displays the number of attained jobs under such artificial estimation, which is slightly better than roundrobin (Figure 3.5a) and almost tied to EDF (Figure 3.5b) and LAF (Figure 3.5c). More specifically, the artificial estimation attains fewer light jobs than EDF and LAF but outperforms them in terms of the attainment rate of medium and heavy jobs. Such results indicate that the accuracy progress estimator is vital to Rotary-AQP or the resource arbitration framework even if the adaptive running epochs still can help some medium and heavy jobs to attain their goals.



(a) Rotary-AQP



(b) Rotary-AQP with artificial progress estimation

Figure 3.8: Impact of progress estimation

3.3 Rotary for DLT

We implement a resource arbitration system, Rotary-DLT, for deep learning training applications and evaluate it using a workload we developed based on a survey. We also discuss similarities and differences between Rotary-DLT and Rotary-AQP.

3.3.1 Rotary-DLT Implementation

Rotary-DLT follows the architecture in Figure 3.4. Compared to Rotary-AQP, Rotary-DLT has the following differences: (1) two estimators to predict the number of training epochs to achieve a specific accuracy and the memory usage of a deep learning model; (2) a time recorder to measure the time of a training epoch; (3) GPU resource arbitration for the DLT jobs; and (4) TensorFlow is deployed as the execution platform to run the DLT jobs. Furthermore, Rotary-DLT stores the information of the historical DLT jobs in a repository, so that the system can provide more accurate estimates for attainment progress and memory consumption. Specifically, all the completed jobs' information including model architecture, training hyperparameters, training epochs, and evaluation accuracy are stored.

A key feature of Rotary-DLT is the ability to estimate the number of epochs for training DLT jobs to achieve specific accuracy, which is accomplished by training epoch estimator (*TEE*). Considering that DLT jobs always center on the accuracy metric, TEE is beneficial for Rotary-DLT to know whether it should allocate or preempt resources for the jobs. Similar to the progress estimator of Rotary-AQP, the basic idea of TEE is to fit an accuracy-epoch curve by jointly leveraging the historical data in the repository and the real-time data of active jobs during the training process. The only difference is that Rotary-DLT selects the similar historical data that have the same training dataset and hyperparameters (e.g., batch size, learning rate, and optimizer).

The training memory estimator (TME) is another key component, and its primary function is to predict the maximum GPU memory usage of the models in the jobs. This is because a DLT job can only be launched if the target GPU has sufficient available memory. As we mentioned in §3.1.1, a DLT job is fed a batch of training data of the same size during each iteration, thus it is viable to estimate the memory usage for every training iteration if the model specification information and batch size are given. The batch size of training data decides how much data will be transferred from host to GPU during each batch, and all the learnable parameters in the deep learning model require space in memory and these parameters where historic gradients are being calculated and used also accumulate in memory. We also fit a batch size-memory curve by leveraging the data from historical jobs for TME. More specifically, when estimating the memory usage of a job, TME first retrieves all the data of historical jobs that use the same training dataset and then computes the similarity between the job for estimation and other historical jobs according to the number of model parameters (i.e., model size). The similarity between two jobs is defined as $similarity(x, y) = 1 - \frac{|x-y|}{max(x,y)}$, where x and y are the number of model parameters of two jobs respectively. TME then picks top-k similar historical jobs from the job repository to fit the batch size-memory curve. We also exploit the weighted linear regression to fit the curve but in a different way, the more similar a historical job is, the higher weights the job will be granted. Furthermore, we pad the estimated memory by an additional offset, such as 5%, to minimize the likelihood of out-of-memory (OOM) issues occurring.

There are two fundamental differences between AQP and DLT, which should be considered for implementing Rotary-DLT. First, DLT jobs can be evaluated every one or multiple epochs using an evaluation dataset, thus it is unnecessary for Rotary-DLT to have a mechanism like an envelope function in Rotary-AQP to approximately evaluate the progress of each job. Second, the batch processing time of AQP jobs can be quite different due to the query predicates and heterogeneous data batch, for example, a batch may trigger numerous join and aggregation operations, but the others may not. However, DLT jobs usually have similar batch processing time due to the stable model architecture and the same batch size. Thus, Rotary-DLT has a side component, training time recorder (TTR), to record the training time of a single step or an epoch. TTR records the time of a training step or a training epoch for each DLT job on different devices to reduce the overhead of recording. Due to a CUDA warm-up issue [54], the very first training step always takes a longer time, so we always discard the first training step when TTR is running and recording. Since the deep learning training job is launched in an iteration fashion, recording the single training time of each job is sufficient to measure the overall time of the training process.

Following the problem statement (§3.1.4), we propose a practical resource arbitration algorithm for DLT, as shown in Algorithm 5. The crux of the algorithm is to prioritize the jobs and iteratively allocate and preempt the available GPU resources for specific objectives such as fairness or efficiency. For this, the algorithm ranks all the active jobs based on their training progress ϕ and prioritizes them in terms of an adaptive threshold T. This adaptive threshold allows the resource arbitration algorithm to first picks up the jobs that have the lowest ϕ so that no single job will considerably fall behind to maintain fairness, and the algorithm starts to select the jobs with the highest ϕ once all the jobs in the workload achieve at least T progress so that more jobs can be attained in a shorter time with a higher chance. Furthermore, our resource arbitration algorithm is adaptive and spans all solutions between fairness and efficiency by tuning the threshold T. The algorithm becomes pure-fairness when $T = 100\%$, and it could only care about efficiency if $T = 0\%$.

As a core in the resource arbitration algorithm for DLT, the computation of training progress ϕ differs for various completion criteria. For example, for the jobs with runtime-oriented completion criteria, calculating ϕ is trivial, which is the ratio of current runtime (e.g., number of epochs) to the runtime threshold. For the jobs with accuracy-oriented and convergence-oriented completion criteria, ϕ can be obtained by estimating the current accuracy and comparing it with the target accuracy. We present the computation of training progress in Algorithm 6.

Algorithm 5: Adaptive Resource Arbitration for DLT

```
Input  : Workload $W = \{j_1, \cdots, j_n\}$
         Completion Criteria $C = \{c_1, \cdots, c_n\}$
         Total GPU $D$
         GPU memory $\{M_1, \cdots, M_D\}$
//All jobs are placed in an active queue when arriving
for job $j_i \in W$ that arrives do
    Place job $j_i$ to the active queue $AQ$
end
//Resource arbitration
while $AQ \neq \varnothing$ do
    if MeetProgressThreshold($W$) then
        Initialize a priority queue $PQ$ that prioritizes the jobs with highest training process;
    else
        Initialize a priority queue $PQ$ that prioritizes the jobs with lowest training process;
    end
    for $i \leftarrow 1$ to $n$ do
        Estimate the resource consumption $\hat{m}_{j_i}$ for job $j_i$;
        Estimate the training progress $\hat{\phi}_{j_i}$ for job $j_i$;
        Place job $j_i$ in $AQ$ according to $\hat{\phi}_{j_i}$
    end
    for $d \leftarrow 1$ to $D$ do
        for job $j_k$ in $AQ$ do
            if $m_{j_k} \leq M_d$ then
                Run job $j_k$ on GPU $d$;
                Remove job $j_k$ from $PQ$;
            end
        end
    end
    for job in $AQ$ do
        if job achieves the completion criteria then
            Remove $job$ from $AQ$
        end
    end
end
Function MeetProgressThreshold($W$):
    if all jobs meet $T$ then return True ;
    return False;
End Function;
```

Algorithm 6: Progress Computation in Rotary-DLT

```
Input  : Workload $W = \{j_1, \cdots, j_n\}$
         Completion Criteria $C = \{c_1, \cdots, c_n\}$
for $i \leftarrow 1$ to $n$ do
    $e_i^* \leftarrow$ job running progress (training epochs) of $j_i$;
    if $j_i$ has runtime-oriented completion criteria then
        Obtaining expected training epoch $e_i$ according to $c_i$;
        $\phi_i = \frac{e_i^*}{e_i}$;
    else if $j_i$ has accuracy-oriented completion criteria then
        Obtaining maximum training epoch $e_i^{max}$ according to $c_i$;
        Estimating the necessary epochs $\hat{e}_i$ according to $c_i$;
        if $\hat{e}_i > e_i^*$ then $\phi_i = \frac{e_i^*}{e_i^{max}}$ ;
        else $\phi_i = \frac{e^*}{\hat{e}_i}$ ;
    else if $j_i$ has convergence-oriented completion criteria then
        Obtaining maximum training epoch $e_i^{max}$ according to $c_i$;
        Obtaining expected accuracy $acc_i$ according to $s_i$;
        Estimating the necessary epochs $\hat{e}_i$ according to $acc_i$;
        if $\hat{e}_i > e_i^*$ then $\phi_i = \frac{e_i^*}{e_i^{max}}$ ;
        else $\phi_i = \frac{e^*}{c_i}$ ;
    end
end
```

3.3.2 Rotary-DLT Evaluation

To evaluate Rotary-DLT, we conduct a synthetic workload based on a survey of 30 experience deep learning researchers. We implement Rotary-DLT on top of TensorFlow 1.15 [81]. All the experiments are conducted on a server with Intel Xeon Silver CPU (2.10GHz), 192GB memory, and 4 GPUs (RTX 2080 8GB graphic memory), running Ubuntu Server 18.04. All the results in the evaluation are averaged over 3 independent runs.

Survey-based DLT Workload

We surveyed 30 experienced deep learning researchers across the following affiliations listed alphabetically: Microsoft Research, National University of Singapore, Northeastern University, Singapore Management University, University of California-Berkeley, University of Chicago, University of Illinois at Urbana-Champaign, and University of Toronto. According to their responses about training infrastructure, model architecture, running time, and completion criteria, we synthesize a DLT workload. The

Model	Architecture	Inception[79], MobileNet[30], MobileNetV2[69], SqueezeNet [32], ShuffleNet[99], ShuffleNetV2[55], ResNet[25], ResNeXt[92], EfficientNet[80], LeNet[45], VGG[73], AlexNet[43], ZFNet[95], DenseNet[31], LSTM[29], Bi-LSTM[20], BERT[83]
	Batch size	Computer vision models: 2, 4, 8, 16, 32 [58] Natural language processing models: 32, 64, 128, 256
	Optimizer	SGD, Adam, Adagrad, Momentum
	Learning rate	0.1, 0.01, 0.001, 0.0001, 0.00001
	Dataset	Computer vision models: CIFAR-10 [42] Natural language processing models: UD Treebank [15], Large Movie Review Dataset [56]
Completion Criteria	Convergence-oriented criteria (delta accuracy)	5%, 3%, 1%, 0.5%, 0.3%, 0.1%, 0.05%, 0.03%, 0.01%, 0.005%, 0.003%, 0.001%
	Accuracy-oriented criteria (final accuracy)	70%, 72%, 74%, 76%, 78%, 80%, 82%, 84%, 86%, 88%, 90%, 92%
	Runtime-oriented criteria (epoch)	From scratch 5, 10, 30, 50, 100 Pre-trained (Fine-tuned): 1, 2, 3, 4, 5
	Maximum epoch for criteria	1, 5, 10, 15, 20, 25, 30
Workload	Synthetic workload	60% DLT jobs with convergence-oriented completion criteria 20% DLT jobs with accuracy-oriented completion criteria 20% DLT jobs with runtime-oriented completion criteria

Table 3.2: Synthetic DLT workload. The selection of model architecture and proportion of jobs with various completion criteria distribution are based on the responses to our survey, and the selection of other hyperparameters and the parameters about completion criteria follow the uniform distribution.

elaborate the configurations of the synthetic workload are presented in Table 3.2. We implement a number of representative deep learning models in Computer Vision (CV) and Natural Language Processing (NLP) with randomized hyperparameters and completion criteria. We use the small batch sizes to training the CV models due to the empirical study [58] but choose bigger sizes for NLP models due to common practice [6]. For other specific hyperparameters of some models (e.g., growth rate for DenseNet), we follow the design in their original paper. We also have pre-trained versions of BERT, VGG, and ResNet since the jobs of fine-tuning pre-trained models are also common.

For the models that have multiple variants like ResNet, DenseNet, ShuffleNet, VGG, BERT, we use the shrunk variants (e.g., ResNet-18, ResNet-34, DenseNet-121) to fit them on a single GPU.

Attainment

We consider fairness and efficiency as two vital but opposite optimization objectives. Achieving fairness can guarantee that no single job is stalled due to a myriad of jobs being in front of it or some upfront jobs taking an unexpectedly long time. Efficiency focuses on completing more jobs in a shorter time if possible, and this objective can be only achieved by always picking up the jobs that can be finished faster. If we stick with fairness, the jobs that can be completed quickly may have to wait a long time. On the contrary, concentrating on efficiency can result in zero progress in some jobs, which means they are never triggered.

We define metrics of attainment rate for DLT jobs with various completion criteria to evaluate the performance of Rotary-DLT.

Accuracy-oriented attainment rate: Similar to attainment progress ϕ, this shows the completion percentage of a job with accuracy-oriented completion criteria but from the perspective of accuracy, which is defined as $\frac{\textit{current accuracy}}{\textit{completion criteria}}$. For instance, if a job has an accuracy target of 80% and obtains 56% accuracy after training one epoch. The current attainment rate of this job is $\frac{56\%}{80\%} = 70\%$.

Convergence-oriented attainment rate: We measure the attainment rate of jobs with convergence-oriented completion criteria in terms of epochs. When retrospecting the training process, if the jobs converged before the max training epochs, we mark the epoch when the model is converged as the *convergence-line* and define the attainment rate as $\frac{\textit{current epoch}}{\textit{convergence-line}}$. For the jobs that failed to converge, we define the progress rate as $\frac{\textit{current epoch}}{\textit{max epochs}}$.

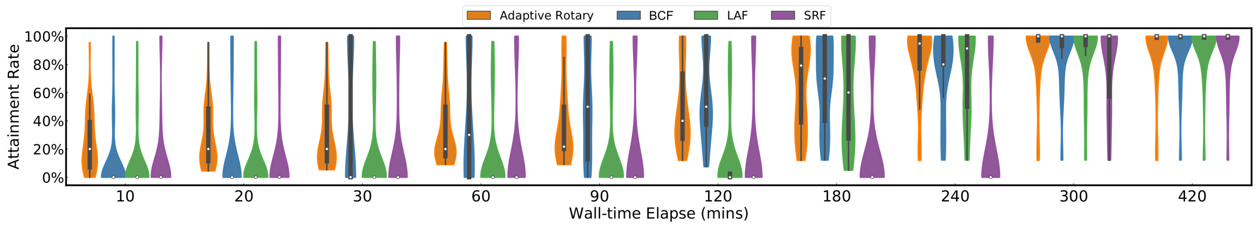
Runtime-oriented attainment rate: The runtime-oriented attainment rate is denoted as $\frac{\textit{current epoch}}{\textit{completion criteria}}$, which is further exemplified by the following case. If a job has a runtime-oriented completion criterion of 15 epochs, and the attainment rate is $\frac{5}{15} = 33.3\%$ after training 5 epochs.

We evaluate the Rotary-DLT against three greedy baselines:

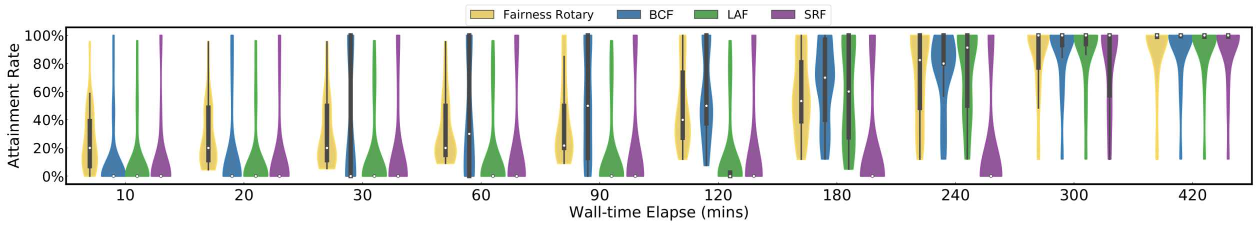
(a) Shortest Runtime First (SRF): it always runs the jobs with the shortest runtime completion criteria first and handles the other jobs following a round-robin strategy.

(b) Biggest Convergence First (BCF): it always runs the jobs with the biggest convergence completion criteria first and handles the other jobs following a round-robin strategy.

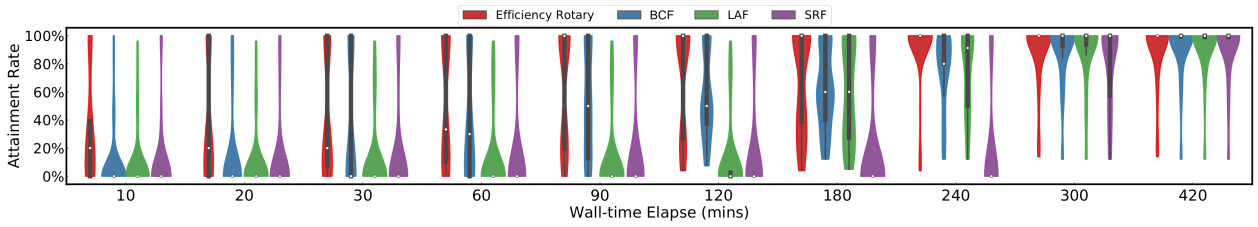
(c) Lowest Accuracy First (LAF): it always runs the jobs with the lowest accuracy completion criteria first and handles the other jobs following a round-robin strategy.



(a) Adaptive Rotary-DLT ($T = 50\%$): Rotary-DLT is pure-fairness from 0 to 120 minutes and can push the minimum attainment rate of the workload. Rotary-DLT becomes more aggressive on efficiency and successfully makes more jobs achieve 100% attainment rate starting from 180 minutes since all the jobs make substantial progress (50%) toward their completion criteria.



(b) Fairness Rotary-DLT ($T = 100\%$): Rotary-DLT always picks up the jobs with the lowest ϕ and can maximize the minimum attainment rate of all jobs in the workloads considerably faster than other baselines. For example, Fairness Rotary-DLT and SRF achieve the same minimum attainment rate of all jobs using 120 minutes and 300 minutes.



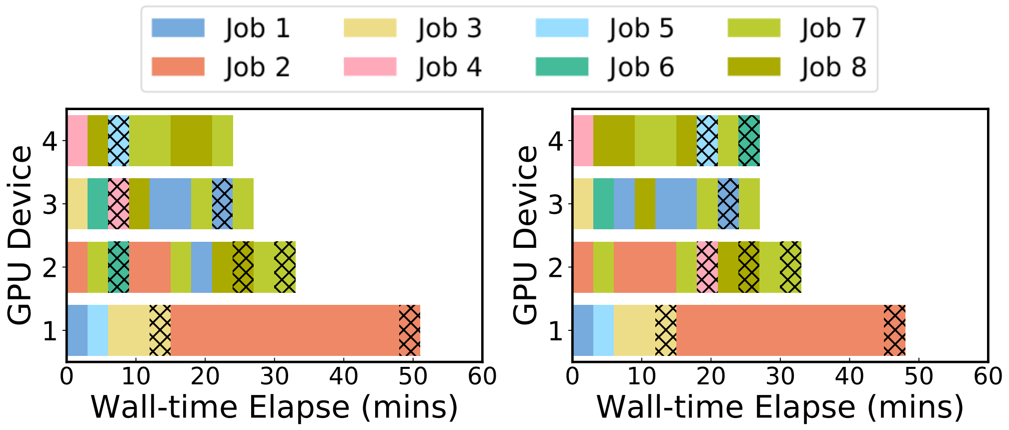
(c) Efficiency Rotary-DLT ($T = 0\%$): Rotary-DLT only selects the jobs with the highest ϕ and makes more jobs meet their completion criteria in a relatively short period. Considering the results at 120 minutes, Efficiency Rotary-DLT completes (achieving 100% attainment rate) more jobs than the other baselines.

Figure 3.9: Evaluation of Rotary-DLT variants and three baselines (BCF, LAF, SRF) on the synthetic DLT workload

We demonstrate all the results in Figure 3.9 using violin plots. In Figure 3.9a, Rotary-DLT is adaptive which fuses the fairness and efficiency policy. It starts with the pure-fairness policy that always selects the jobs with the lowest ϕ. Once all the jobs in the workload achieve at least 50% progress toward their completion criteria, adaptive Rotary-DLT switches to an efficiency-centric policy, which starts to pick up the jobs with the highest ϕ. Figure 3.9b and 3.9c demonstrate the performance of two Rotary-DLT variants that optimizes fairness and efficiency objectives, respectively. All the Rotary-DLT variants outperform the three baselines according to optimization objectives.

Impact of Training Epoch Estimation

Training epoch estimation is positioned at a vital place in developing and evaluating Rotary, and it is critical to understand its effect of it. We conduct a micro-benchmark workload with 8 DLT jobs and track the job placement under efficiency Rotary-DLT with and without accurate epoch estimation, respectively. Among the 8 jobs, $job4$ is for BERT, $job5$ is for Bi-LSTM, and $job6$ is for LSTM. To evaluate how the epoch estimation impacts the performance, we remove all the archived jobs about NLP models in the repository of Rotary-DLT so that the estimation for jobs $4, 5, 6$ is unreliable and even erroneous (e.g., the number of epochs for meeting the completion criteria is 2 but an erroneous estimate can be 100 epochs).



(a) With Reliable Estimation (b) With Erroneous Estimation

Figure 3.10: Job placements under efficiency Rotary-DLT. Rectangles with hatches indicate the jobs meet the completing criteria. The job 4, 5, &6 are completed faster based on accurate epoch estimates (Figure 3.10a), but they are deferred due to the erroneous epoch estimates (Figure 3.10b).

We demonstrate the placements for 8 jobs in Figure 3.10. Each rectangle denotes a job placement and the one with hatches means the job meets the completion criteria. Figure 3.10a presents the job placement under efficiency Rotary-DLT with the accurate epoch estimation. In light of the accurate epoch estimate, jobs 4, 5, and 6 are triggered to run after the trial phase in Rotary-DLT and complete early. However, as shown in Figure 3.10b, the epoch estimate is inaccurate and the placement is inefficient accordingly. For example, job 4 can reach the complete criteria in 2 epochs, but the inaccurate estimate for that is 125 epochs, so its progress ϕ is much lower than others and cannot be placed as it should be. Therefore, jobs 4, 5, and 6 are finished later than those under accurate estimation.

Overhead of TTR, TEE, and TME

We investigate the overhead of recording the training epoch time of DLT jobs, namely measuring how the overhead of TTR and TEE in Rotary-DLT scales when the DLT workload grows. As shown in Table 3.3, taking the workloads with the sizes of 10, 20, 30, and 40 as examples, the overhead of TTR and TEE takes an imperceptible proportion of the whole workload processing time even for the larger workload.

Workload Size	Overall Running Time	Overhead of TTR	Overhead of TEE	Overhead of TME
10	8142s	0.225s	0.74s	0.58s
20	23790s	0.6s	1.31s	1.03s
30	34014s	0.87s	1.98s	1.49s
40	43124s	1.12s	2.56s	2.11s

Table 3.3: The overall process time and overhead in Rotary

3.4 Related Work

To the best of our knowledge, Rotary is the first resource arbitration system for DLT jobs to support user-defined completion criteria. Thus, we broadly review the related works to position our work.

3.4.1 *Scheduling for AQP*

Approximate query processing scheduling works are related to our framework since they focus on orchestrating the AQP jobs. However, to the best of our knowledge, there is not much work in this area [46, 10].

iOLAP is one of the representative works [96], which returns intermediate results by processing the input data a batch at a time rather than running the query on the entire dataset. iOLAP partitions the input data into mini-batches and schedules the delta update query on each batch and collects query results. It also can schedule recomputing jobs to recover the query result when a failure is detected. S-AQP is similar work to iOLAP lies in this area [2]. However, they mainly focus on scheduling query plans.

For scheduling AQP jobs, ReLAQS [76], which serves as one of the baselines in our experiments, is the state-of-the-art work. It is can preempt the AQP jobs according to the estimation and try to help more jobs achieve their objectives. However, our framework has additional contributions: (1) ReLAQS only schedules CPU cores, Rotary-AQP further considers memory consumption when preempting resources; (2) Estimation of ReLAQS only uses real-time results to predict the progress of each AQP job for the next running epoch, the estimators in Rotary-AQP jointly utilize historical and real-time data to make predication which can overcome some issues such as cold-start or data bias; (3) Compared with ReLAQS, Rotary-AQP can support adaptive running cycling for short-running and long-running AQP jobs.

3.4.2 Scheduling for Machine Learning

We consider scheduling systems for machine learning as the most relevant works. We first review the works that define fixed scheduling objectives for DLT jobs. MArk allows users to specify the response time for machine learning model serving and schedules by selecting between AWS EC2 and AWS Lambda to support unpredictable workload bursts [97]. Some works like Tiresias [22] and Optimus [66] schedule machine learning jobs with time constraints.

Scheduling systems for machine learning are widely deployed as well. Gandiva is a cluster scheduling framework that utilizes the cyclic predictability of intra-batch in a DLT job and the feedback of early training to improve training latency and efficiency in a GPU cluster [90]. Philly analyzes a trace of machine learning workloads run on a cluster of GPUs in Microsoft and schedules the jobs according to a trade-off between locality and GPU utilization [34]. HiveD [100] is designed to be a Kubernetes scheduler extension for Multi-Tenant GPU clusters, which can guarantee resource reservation for DLT jobs. PipeDream [63] is a deep learning training system that schedules computation by pipelining execution across multiple machines to accelerate the training process. AntMan [91] is a large-scale deep learning multi-tenant infrastructure in Alibaba, which utilizes the spare GPU resources to co-execute multiple jobs on a shared GPU and dynamically scales memory and computation. Pollux [67] is resource-adaptive deep learning (DL) training and scheduling framework which optimizes inter-dependent factors both at the per-job level and at the cluster-wide level.

As we emphasized before, scheduling systems and our resource arbitration framework, Rotary, solve different but complementary problems. The scheduling systems pay more attention to resource reservation and job placement according to jobs' requirements, however, Rotary addressed the issues about resource allocation and job preemption.

3.4.3 Multi-tenant Systems

Multi-tenant systems, which don't focus on job scheduling but also have been deployed for AQP and DLT applications, should also be mentioned.

BlinkDB [3] is an AQP system that is based on Apache Hadoop and devises effective strategies to select proper samples (offline generated) in distributed clusters to answer newly coming queries. Quickr [37] is designed for executing ad-hoc queries on big-data clusters that do not need any pre-computing of the whole dataset spread over the clusters. SnappyData [60] is a platform to support OLTP, OLAP, and stream analytics based on Apache Spark.

Multi-tenant systems for deep learning are also proposed and deployed recently. FfDL is a deep learning platform in IBM to support the multi-tenant distributed training of models based on Kubernetes [33]. Facebook also reveals some design choices for building a datacenter to handle multi-tenant training and inference, like the importance of co-locating data with computation [24]. Ease.ml is a declarative machine learning service platform that focuses on a cost-aware model selection problem in a multi-tenant system [59]. CROSSBOW [39] is a system that supports users to select a small batch and scale to multiple GPUs for deep learning training.

3.5 Discussion

Here we discuss some implementation limitations and future work.

Implementation Limitations: We faced several designs trade-offs when implementing Rotary-AQP and Rotary-DLT. However, it should be noted that all the trade-offs are implementation-specific and framework independent, which could be mitigated by different implementations. We discuss two examples.

One implementation trade-off is how to persist the AQP jobs that have been paused (i.e., deferred to future execution) due to resource arbitration. When a job is paused, its intermediate states and results should be persisted either in memory or disk so that it can be resumed. Persisting AQP jobs in memory is more efficient from the perspective of performance, but may quickly saturate the memory, which is a relatively scarce resource compared with disk and may lead to an out-of-memory error. Therefore, we checkpoint the AQP jobs in disks. Such a mechanism will bring additional overhead but allows more jobs to run simultaneously. The same issue happens when we implemented Rotary-DLT, however, checkpointing DLT jobs in disks is a common practice due to the unique characteristics of GPU.

The second implementation choice we made is that we assume the AQP and DLT jobs are executed in a single machine even though our framework and system implementations support distributed execution. This is because we decide to first make a deep investigation of a resource arbitration framework and its implementations so that we can have a better understanding of progressive iterative analytic jobs and verify our framework design. Our system implementations, Rotary-AQP and Rotary-DLT, and the corresponding evaluations confirm the generality and practicality of the proposed framework. Thus, any distributed iterative processing job is out of the scope of this paper.

Materialization for Progressive Iterative Analytics: Progressive iterative analytic jobs need to be persisted. Such requirement essentially asks for a materialization mechanism as in database systems and brings a similar trade-off between cost and efficiency [1]. How and when to materialize the progressive iterative analytic jobs is an interesting and pivotal research question, and we leave the answers for future work.

Unified Resource Arbitration Framework: While we compare AQP and DLT and treat them as two alike progressive iterative analytic applications in different areas and implement two systems for both of them, it is more interesting to have a unified resource arbitration system on a cluster to handle AQP and DLT jobs together, such a system can serve more users and enormously improve the resources utilization.

3.6 Conclusion

In this paper, we argue that resource arbitration is vital but neglected for progressive iterative analytic applications due to their unique characteristics. We proposed a framework, Rotary, to highlight the core features and components for resource arbitration. It allows diverse user-defined completion criteria, prioritizes the jobs for resources, and supports adaptive running epochs. To realize and verify the framework, we implement two resource arbitration systems for AQP and DLT and evaluate them using the TPC-H benchmark and a survey-based workload respectively. The evaluation results show that Rotary-AQP and Rotary-DLT outperform the state-of-the-art and heuristic baselines, and confirm that Rotary is an appealing solution for efficient resource utilization for iterative applications. Our work also opens interesting opportunities to explore the connection between research problems in ML and DB, such as balancing accuracy and running time in approximate queries processing and deep learning training.

CHAPTER 4

RATCHET: EFFICIENT, ROBUST, AND RESOURCE-AWARE CHECKPOINTING FOR ITERATIVE AND PROGRESSIVE QUERY PROCESSING

4.1 Research Proposal

As a prevailing case of data-intensive computation, iterative and progressive has recently received substantial attention. Such long-running computation jobs keep receiving the data and processed them in a progressive way so that the previous results can be reused and merged. Such incremental query processing jobs are prohibitively expensive in terms of time and resource consumption especially in a resource-shared and -constrained environment, and interruptions to these jobs are inevitable due to resource reallocation that is common in a multi-tenant environment. When interruptions occur, the long-running, stateful, incremental query processing jobs terminate abruptly, wiping out the query and the associated data from the memory, and erasing the current processing process, which is a significant waste when the interruptions happen frequently. This, therefore, necessitates an operation that can suspend and store the job if there will be an interruption and resume it when possible, which can be called intermediate checkpointing. However, supporting intermediate checkpointing in multi-tenant environment poses several unique challenges which we would like to address as describe blow:

- **Checkpointing Triggering:** there is no one-size-fits-all checkpointing triggering point that works across queries, hardware, and multi-tenant environments. The triggering point depends on several factors such as query operators, data size, and computation contention. This suggests that triggering a checkpoint operation might not be worthwhile and may lead to additional performance overhead. For example, an incremental query processing job may have to re-run some long-running operators like join if the checkpoint happens in the middle or even at the nearly-end point of the operators. Moreover, we can checkpointing a job just before a long-running operator in favor of some other jobs that only consist of short-running operators if it is beneficial to do so. Therefore, the system that can automatically decide when and which jobs should be checkpointed is necessary in a multi-tenancy environment.

- **Checkpointing Staleness:** when the query processing job is interrupted, the on-going data may be either lost or missed, which leads to rerun or skip the current processing batch. This may result in accuracy loss if the checkpointed job is mis-matched with the current and arriving data. Therefore, we need to add robustness to overcome this checkpointing staleness. This robustness should be able to capture the data progress even in the presence of interruptions and checkpointing so that makes correct, batch-level checkpointing feasible for incremental query processing.

- **Resource-Aware Checkpointing:** Checkpointing incremental query jobs can consume a substantial amount of memory that is a scarce resource in multi-tenant environments. One solution is hierarchical storage for checkpointing, which allows some jobs to be checkpointed on disks which usually have large storage capacity. However, checkpointing on disks will bring additional I/O overhead and may significantly affect the processing performance. Therefore, a system that can arbitrate which jobs should be checkpointed on low-performance devices is desired in a resource-constrained environment.

4.2 Research Plan

We outline our research plan and milestones for Ratchet as below.

Sep.2022 - Nov.2022: Design and implement a prototype checkpointing system for incremental query processing with basic functions.

Nov.2022 - Feb.2023: Implement Ratchet on top of the checkpointing system.

Feb.2023 - May.2023: Implement more baselines and have an experiment plan; tackle technical challenges and collect experiments result to write the paper.

REFERENCES

[1] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. Materialization strategies in a column-oriented DBMS. In *International Conference on Data Engineering (ICDE)*, pages 466–475, 2007.

[2] Sameer Agarwal, Henry Milner, Ariel Kleiner, Ameet Talwalkar, Michael I. Jordan, Samuel Madden, Barzan Mozafari, and Ion Stoica. Knowing when you're wrong: building fast and reliable approximate query processing systems. In *International Conference on Management of Data (SIGMOD)*, pages 481–492, 2014.

[3] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *European Conference on Computer Systems (EuroSys)*, pages 29–42, 2013.

[4] Ankur Agiwal, Kevin Lai, Gokul Nath Babu Manoharan, Indrajit Roy, Jagan Sankaranarayanan, Hao Zhang, Tao Zou, Jim Chen, Min Chen, Ming Dai, Thanh Do, Haoyu Gao, Haoyan Geng, Raman Grover, Bo Huang, Yanlai Huang, Adam Li, Jianyi Liang, Tao Lin, Li Liu, Yao Liu, Xi Mao, Maya Meng, Prashant Mishra, Jay Patel, Rajesh Sr, Vijayshankar Raman, Sourashis Roy, Mayank Singh Shishodia, Tianhang Sun, Justin Tang, Jun Tatemura, Sagar Trehan, Ramkumar Vadali, Prasanna Venkatasubramanian, Joey Zhang, Kefei Zhang, Yupu Zhang, Zeleng Zhuang, Goetz Graefe, Divy Agrawal, Jeffrey F. Naughton, Sujata Kosalge, and Hakan Hacigümüs. Napa: Powering scalable data warehousing with robust query performance at google. *VLDB Endowment*, 14(12):2986–2998, 2021.

[5] Ashvin Agrawal, Rony Chatterjee, Carlo Curino, Avrilia Floratou, Neha Godwal, Matteo Interlandi, Alekh Jindal, Konstantinos Karanasos, Subru Krishnan, Brian Kroth, Jyoti Leeka, Kwanghyun Park, Hiren Patel, Olga Poppe, Fotis Psallidas, Raghu Ramakrishnan, Abhishek Roy, Karla Saur, Rathijit Sen, Markus Weimer, Travis Wright, and Yiwen Zhu. Cloudy with high chance of DBMS: a 10-year prediction for enterprise-grade ML. In *Conference on Innovative Data Systems Research (CIDR)*, 2020.

[6] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural Networks: Tricks of the Trade - Second Edition*, volume 7700, pages 437–478. 2012.

[7] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, and Sebastian Benjamin Wrede. Systemds: A declarative machine learning system for the end-to-end data science lifecycle. In *Conference on Innovative Data Systems Research (CIDR)*, 2020.

[8] Thomas Bradley. Nvidia hyper-q example. `http://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf`, 2013.

[9] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016.

[10] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. Approximate query processing: No silver bullet. In *ACM International Conference on Manageme of Data (SIGMOD)*, pages 511–519, 2017.

[11] C. L. Philip Chen and Chun-Yang Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information Sciences*, 275:314–347, 2014.

[12] Andrew Crotty, Alex Galakatos, Connor Luckett, and Ugur Çetintemel. The case for in-memory OLAP on "wimpy" nodes. In *IEEE International Conference on Data Engineering (ICDE)*, pages 732–743, 2021.

[13] Carlo Curino, Djellel Eddine Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based scheduling: If you're late don't blame us! In *ACM Symposium on Cloud Computing (SoCC)*, pages 2:1–2:14, 2014.

[14] Databricks. Cost-based optimizer. `https://docs.databricks.com/spark/latest/spark-sql/cbo.html`, 2022.

[15] Universal Dependencies. Universal dependencies. `https://universaldependencies.org`, 2021.

[16] Yupeng Fu and Chinmay Soman. Real-time data infrastructure at uber. In *ACM International Conference on Management of Data (SIGMOD)*, pages 2503–2516, 2021.

[17] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google vizier: A service for black-box optimization. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1487–1495, 2017.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT Press, 2016.

[19] Google. Tensorflow basics. `https://www.tensorflow.org/tutorials/customization/basics`, 2021.

[20] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, 18(5-6):602–610, 2005.

[21] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 485–500, 2019.

[22] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 485–500, 2019.

[23] Hui Guan, Laxmikant Kishor Mokadam, Xipeng Shen, Seung-Hwan Lim, and Robert M. Patton. FLEET: flexible efficient ensemble training for heterogeneous deep neural networks. In *Conference on Machine Learning and Systems (MLSys)*, 2020.

[24] Kim M. Hazelwood, Sarah Bird, David M. Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629, 2018.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision (ECCV)*, pages 630–645, 2016.

[27] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *ACM International Conference on Management of Data (SIGMOD)*, pages 171–182, 1997.

[28] Herodotos Herodotou and Elena Kakoulli. Trident: Task scheduling over tiered storage systems in big data platforms. *VLDB Endowment*, 14(9):1570–1582, 2021.

[29] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[30] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[31] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269, 2017.

[32] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.

[33] K. R. Jayaram, Vinod Muthusamy, Parijat Dube, Vatche Ishakian, Chen Wang, Benjamin Herta, Scott Boag, Diana Arroyo, Asser N. Tantawi, Archit Verma, Falk Pollok, and Rania Khalaf. Ffdl: A flexible multi-tenant deep learning platform. In *International Middleware Conference (Middleware)*, pages 82–95, 2019.

[34] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *USENIX Annual Technical Conference (ATC)*, pages 947–960, 2019.

[35] Hai Jin, Bo Liu, Wenbin Jiang, Yang Ma, Xuanhua Shi, Bingsheng He, and Shaofeng Zhao. Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(3):37, 2018.

[36] Apache Kafka. Apache kafka. `https://kafka.apache.org`, 2022.

[37] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *International Conference on Management of Data (SIGMOD)*, pages 631–646, 2016.

[38] Steven Kay. *Fundamentals of statistical signal processing*. Prentice Hall PTR, 1993.

[39] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter R. Pietzuch. Crossbow: Scaling deep learning with small batch sizes on multi-gpu servers. *VLDB Endowment*, 12(11):1399–1413, 2019.

[40] Sanjay Krishnan, Aaron J Elmore, Michael Franklin, John Paparrizos, Zechao Shang, Adam Dziedzic, and Rui Liu. Artificial intelligence in resource-constrained and shared environments. *ACM SIGOPS Operating Systems Review*, 53(1):1–6, 2019.

[41] Sanjay Krishnan, Aaron J. Elmore, Michael J. Franklin, John Paparrizos, Zechao Shang, Adam Dziedzic, and Rui Liu. Artificial intelligence in resource-constrained and shared environments. *ACM SIGOPS Operating Systems Review*, 53(1):1–6, 2019.

[42] Alex Krizhevsky et al. Learning multiple layers of features from tiny images. 2009.

[43] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Annual Conference on Neural Information Processing Systems (NIPS)*, pages 1106–1114, 2012.

[44] Tung D Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. Tflms: Large model support in tensorflow by graph rewriting. *arXiv preprint arXiv:1807.02037*, 2018.

[45] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceeding of IEEE*, 86(11):2278–2324, 1998.

[46] Kaiyu Li and Guoliang Li. Approximate query processing: What is new and where to go? - A survey on approximate query processing. *Data Science and Engineering*, 3(4):379–397, 2018.

[47] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. Massively parallel hyperparameter tuning. *arXiv preprint arXiv:1810.05934*, 2018.

[48] Liam Li, Kevin G. Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A system for massively parallel hyperparameter tuning. In *Conference on Machine Learning and Systems (MLSys)*, 2020.

[49] Liam Li, Evan Sparks, Kevin Jamieson, and Ameet Talwalkar. Exploiting reuse in pipeline-aware hyperparameter tuning. *arXiv preprint arXiv:1903.05176*, 2019.

[50] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18:185:1–185:52, 2017.

[51] Tian Li, Jie Zhong, Ji Liu, Wentao Wu, and Ce Zhang. Ease.ml: Towards multi-tenant resource sharing for machine learning workloads. *VLDB Endowment*, 11(5):607–620, 2018.

[52] Richard Liaw, Romil Bhardwaj, Lisa Dunlap, Yitian Zou, Joseph E Gonzalez, Ion Stoica, and Alexey Tumanov. Hypersched: Dynamic resource reallocation for model development on a deadline. In *ACM Symposium on Cloud Computing (SoCC)*, pages 61–73, 2019.

[53] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[54] Rui Liu, Sanjay Krishnan, Aaron J. Elmore, and Michael J. Franklin. Understanding and optimizing packed neural network training for hyper-parameter tuning. In *Workshop on Data Management for End-To-End Machine Learning (DEEM@SIGMOD)*, pages 3:1–3:11, 2021.

[55] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet V2: practical guidelines for efficient CNN architecture design. In *European Conference on Computer Vision (ECCV)*, volume 11218, pages 122–138, 2018.

[56] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Annual Meeting of the Association for Computational Linguistics ACL*, pages 142–150, 2011.

[57] Kshiteej Mahajan, Arjun Singhvi, Arjun Balasubramanian, Varun Batra, Surya Teja Chavali, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient gpu cluster scheduling for machine learning workloads. *arXiv preprint arXiv:1907.01484*, 2019.

[58] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *CoRR*, abs/1804.07612, 2018.

[59] Leonel Aguilar Melgar, David Dao, Shaoduo Gan, Nezihe M Gürel, Nora Hollenstein, Jiawei Jiang, Bojan Karlaš, Thomas Lemmin, Tian Li, Yang Li, Susie Rao, Johannes Rausch, Cedric Renggli, Luka Rimanic, Maurice Weber, Shuai Zhang, Zhikuan Zhao, Kevin Schawinski, Wentao Wu, and Ce Zhang. Ease.ml: A lifecycle management system for mldev and mlops. In *Conference on Innovative Data Systems Research (CIDR)*, 2021.

[60] Barzan Mozafari, Jags Ramnarayan, Sudhir Menon, Yogesh Mahajan, Soubhik Chakraborty, Hemant Bhanawat, and Kishor Bachhav. Snappydata: A unified cluster for streaming, transactions and interactice analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2017.

[61] Supun Nakandala, Yuhao Zhang, and Arun Kumar. Cerebro: A data system for optimized deep learning model selection. *VLDB Endowment*, 13(11):2159–2173, 2020.

[62] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for DNN training. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, 2019.

[63] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for DNN training. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, 2019.

[64] Deepak Narayanan, Keshav Santhanam, Amar Phanishayee, and Matei Zaharia. Accelerating deep learning workloads through efficient multi-model execution. In *NIPS Workshop on Systems for Machine Learning*, 2018.

[65] NVIDIA. Nvidia virtual gpu technology. `https://www.nvidia.com/en-us/design-visualization/technologies/virtual-gpu/`, 2019.

[66] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *European Conference on Computer Systems (EuroSys)*, pages 3:1–3:14, 2018.

[67] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[68] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[69] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4510–4520, 2018.

[70] Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. Green AI. *arXiv preprint arXiv:1907.10597*, 2019.

[71] Taro Sekiyama, Takashi Imamichi, Haruki Imai, and Rudy Raymond. Profile-guided memory optimization for deep neural networks. *arXiv preprint arXiv:1804.10001*, 2018.

[72] Zechao Shang, Xi Liang, Dixin Tang, Cong Ding, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. Crocodiledb: Efficient database execution through intelligent deferment. In *Conference on Innovative Data Systems Research (CIDR)*, 2020.

[73] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations (ICLR)*, 2015.

[74] Apache Spark. Apache spark. `https://spark.apache.org`, 2022.

[75] Evan R Sparks, Ameet Talwalkar, Daniel Haas, Michael J Franklin, Michael I Jordan, and Tim Kraska. Automating model search for large scale machine learning. In *ACM Symposium on Cloud Computing (SoCC)*, pages 368–380, 2015.

[76] Logan Stafman, Andrew Or, and Michael J. Freedman. Relaqs: Reducing latency for multi-tenant approximate queries via scheduling. In *International Middleware Conference (Middleware)*, pages 280–292, 2019.

[77] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 3645–3650, 2019.

[78] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for modern deep learning research. In *AAAI Conference on Artificial Intelligence (AAAI)*, pages 13693–13696, 2020.

[79] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[80] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning (ICML)*, volume 97, pages 6105–6114, 2019.

[81] TensorFlow. Tensorflow. `https://www.tensorflow.org`, 2022.

[82] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *European Conference on Computer Systems (EuroSys)*, pages 35:1–35:16, 2016.

[83] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Well-read students learn better: The impact of student initialization on knowledge distillation. *CoRR*, abs/1908.08962, 2019.

[84] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop YARN: yet another resource negotiator. In *ACM Symposium on Cloud Computing (SOCC)*, pages 5:1–5:16, 2013.

[85] Benjamin Wagner, André Kohn, and Thomas Neumann. Self-tuning query scheduling for analytical workloads. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1879–1891, 2021.

[86] Carl A Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[87] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: dynamic GPU memory management for training deep neural networks. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 41–53, 2018.

[88] Abdul Wasay, Brian Hentschel, Yuze Liao, Sanyuan Chen, and Stratos Idreos. Mothernets: Rapid deep ensemble learning. In *Conference on Machine Learning and Systems (MLSys)*, 2020.

[89] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 595–610, 2018.

[90] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 595–610, 2018.

[91] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 533–548, 2020.

[92] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5987–5995, 2017.

[93] Hidehito Yabuuchi, Daisuke Taniwaki, and Shingo Omura. Low-latency job scheduling with preemption for the development of deep learning. In *USENIX Conference on Operational Machine Learning (OpML)*, pages 27–30, 2019.

[94] Peifeng Yu and Mosharaf Chowdhury. Fine-grained GPU sharing primitives for deep learning applications. In *Conference on Machine Learning and Systems (MLSys)*, 2020.

[95] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European Conference on Computer Vision (ECCV)*, volume 8689, pages 818–833, 2014.

[96] Kai Zeng, Sameer Agarwal, and Ion Stoica. iolap: Managing uncertainty for efficient incremental OLAP. In *International Conference on Management of Data (SIGMOD)*, pages 1347–1361, 2016.

[97] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *USENIX Annual Technical Conference (ATC)*, pages 1049–1062, 2019.

[98] Junzhe Zhang, Sai Ho Yeung, Yao Shu, Bingsheng He, and Wei Wang. Efficient memory management for gpu-based deep learning systems. *arXiv preprint arXiv:1903.06631*, 2019.

[99] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6848–6856, 2018.

[100] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C. M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. Hived: Sharing a GPU cluster for deep learning with guarantees. In *USENIX Symposium on Operating Systems Design and Implementation OSDI*, pages 515–532, 2020.

[101] Haoyue Zheng, Fei Xu, Li Chen, Zhi Zhou, and Fangming Liu. Cynthia: Cost-efficient cloud resource provisioning for predictable distributed deep neural network training. In *International Conference on Parallel Processing (ICPP)*, pages 86:1–86:11, 2019.