IT UNIVERSITY OF COPENHAGEN

Department of Computer Science
Data-intensive Systems and Applications

THESIS

Resourceful Learning: Training More Models with Fewer Resources

Author:
Ties Robroek

Supervisor:
Pınar Tözün

Submitted on
August 31, 2024

Imprint

Project:	Thesis
Title:	Resourceful Learning: Training More Models with Fewer Resources
Author:	Ties Robroek
Abstract Translation:	Morten Clausen
Date:	August 31, 2024
Copyright:	IT University of Copenhagen

Supervisor:
Pınar Tözün
IT University of Copenhagen
Email: pito@itu.dk

Acknowledgements

A PhD is not a journey that you walk alone, but together with your colleagues, friends and family.

I was originally going to be part of a different research project at a different university, but when that was cancelled last minute my world and self-esteem collapsed. I want to thank *Nikki* for always being there for me and guiding me whenever I needed it, both before and during the PhD journey.

The world of academia has its bright and dark sides. Having heard stories of bad student-supervisor relationships from friends and online, I am indefinitely grateful to *Pınar* for being an ever-supportive supervisor that I could only have hoped for. Furthermore, I want to thank my academic brother *Ehsan* with whom I've walked and explored this path together.

DASYA has been a great research group to be a part of. I am thankful for all the great (and not-so-great) conversations we have had during lunch and the bad movies we've watched together. To *Morten* and *Kasper* I owe special thanks for stimulating me to pursue not just mental but also physical activity, and *Dovile* for making me think about things. Finally, I want to thank my roommates *Aaron*, *Neil*, and *Robert* for sharing what is probably the least-ventilated room in our building.

During my research exchange in Zürich I discovered what working in a different research group can be like, with thanks to *Ana* for hosting me. I want to especially thank *Maximilian* for his working ethos, *Ben* for making me feel at home and want to again apologise to *Foteini* for consistently mispronouncing her name during my first month.

Last but not least, I want to thank my parents for always being there for me, even though my decision to move to Denmark has not always been the easiest or the most practical.

Abstract

Data Science is a field that has seen rapid development as of late due to the introduction of more powerful and specialised hardware. Massive algorithms such as Deep Neural Networks can be feasibly run on current-day accelerators. This hardware is by design efficient for solving embarrassingly parallel tasks, such as matrix multiplication.

New models are sometimes trained on systems spanning over 10000 nodes with these accelerators, pushing the state of the art further but incurring massive resource costs [1, 2, 3]. Firstly, larger hardware setups require significant space, data-centre cooling and electricity [4, 5]. Secondly, with the state-of-the art utilising more and more expensive hardware setups, doing groundbreaking research in Machine Learning is becoming more expensive, and by extension, less attainable. It is vital to keep research accessible to not just those with private budgets [4].

Previous research has shown that the price of server infrastructure does not linearly reduce the time to train a Deep Learning model to accuracy. In some cases, models can be trained to similar accuracy with only a slight increase in training time on cheaper hardware [6]. It is thus crucial that we do not simply look at maximum accuracy, but also look at how long it takes us to get there [7, 1].

Paramount to training models in a resource-conscious way is to understand how Deep Learning training interacts with the hardware. Therefore, we have introduced the data collection and visualisation framework radT to make this information accessible. This allows Deep Learning researchers to make more informed choices on how they use their hardware. Additionally, we have used radT ourselves to run extensive benchmarking of Deep Learning training on a plethora of configurations.

Furthermore, current-day GPU hardware may be more powerful than what is required to train a singular model. Rather than letting the unused resources go to waste, one can train multiple models at the same time on the same GPU (collocation). With radT we investigated the effectiveness of several methods of GPU collocation. This showed that GPU collocation can be very effective when models are small, or when models complement each other's hardware requirements.

Another way to increase efficiency via collocation is to streamline data loading and processing pipelines. We introduced a system that takes care of data redundancies by decoupling data loading from model training. This way, a server runs a single data loading process that loads data for all models being trained at the same time. This results in significant CPU savings and can even lead to GPU savings in cases where parts of the data pre-processing pipeline run on the GPU.

While the aforementioned projects improve the transparency of resource utilisation and efficiency via collocation, they do not improve the training performance of a

model training in isolation. Our final contribution makes use of our data loading expertise to design a new data loader that progressively increases the complexity of the data. By starting with easier data points before progressing to more complex data points, akin to how a human would learn, we are reducing the flops required for similar training steps. This leads to sharper accuracy growth and overall improved training speeds.

With all of these resource-aware techniques, this thesis demonstrates that it is possible to achieve more in model training by using fewer resources.

Resumé

Data Science som videnskabeligt felt har for tiden set hurtigt udvikling grundet introduktionen af mere kraftfuld og specialiseret hardware. Massive algoritmer som Deep Neural Networks er mulige at eksekvere på nutidens acceleratorer. Denne hardware er af design effektiv til at løse åbenlyst parallelle opgaver, så som matrix multiplikation.

Nye modeller er somme tider trænet på systemer der inkluderer over 10000 knudepunkter med disse acceleratorer, hvilket skubber den seneste teknologi fremad, men med massivt forhøjede omkostninger [1, 2, 3]. For det første kræver større hardware-konfigurationer en betydelig mængde plads, nedkøling og strøm [4, 5]. For det andet, da de nyeste teknologier anvender dyrere og dyrere hardware-konfigurationer, medfører det en prisstigning på at lave banebrydende forskning inden for Machine Learning, hvilket lukker muligheden af for flere mennesker. Det er essentielt at holde forskning åben for så mange som muligt, og ikke kun dem med mange midler [4].

Tidligere forskning har vist at prisen på server-infrastruktur ikke lineært reducerer tiden det tager at træne en Deep Learning model til en givet rigtighedsprocent. I nogen tilfælde kan modeller trænes til cirka den samme rigtighedsprocent med kun en lille forøgelse i træningstid på billigere hardware [6]. Det er derfor vigtigt at man ikke blot ser på den højeste rigtighedsprocent, men også hvor lang tid den tager at opnå [7, 1].

Kernen til at træne modeller med vægt på ressourceforbrug er at forstå hvordan Deep Learning interagerer med hardwaren. Derfor har vi introduceret dataindsamlings- og visualiserings-værktøjet radT, for at gøre denne information tilgængelig. Dette gør det muligt for forskere i Deep Learning at foretage begrundede valg omkring deres brug af hardware. Desuden har vi selv anvendt radT til at foretage omfangsrige performance-tests af Deep Learning træning, på et hav a forskellige konfigurationer.

Hvad mere, så kan morderne GPU-hardware være mere kraftfuldt end hvad der er nødvendigt til at træne en enkelt model. I stedet for at lade de ubrugte ressourcer gå til spilde, kan man træne flere modeller samtidigt på den samme GPU (kollokation). Med radT har vi undersøgt effektiviteten af adskillige metoder til GPU-kollokation. Dette viste at GPU-kollokation kan være yderst effektiv når modellerne er små, eller når modellerne er hinandens modsætninger i forhold til deres hardware-krav.

En anden måde at øge effektiviteten gennem kollokation er at strømline data-indlæsning og processeringspipelinen. Vi har introduceret et system der tager sig af data-overflødighed ved at frakoble data-indlæsning fra modeltræning. På denne måder kører en server en enkelt data-indlæsningsproces til at indlæse alle dataene

for alle modeller som trænes samtidigt. Dette resulterer i en betydelig reduktion af CPU-brug, og kan endda føre til reduktion af GPU-brug i tilfælde hvor dele af præ-processeringen kører på GPUen.

Selvom de fornævnte projekter klarlægger ressourceforbruget og forbedrer forøger effektiviteten gennem kollokation, forbedrer de ikke træningsperformance a modeltræning i isolation. Vores sidste bidrag gør brug af vores data-indlæsningsekspertise til at designe en ny data-indlæser der gradvist øger kompleksiteten af dataene. Ved at starte med simple datapunkter og efterfølgende arbejde mod mere komplicerede datapunkter, i samme stil som et menneske lærer, reducerer vi mængden af flops påkrævet til lignende træningstrin. Dette fører til større vækst af rigtighedsprocenten og i sidste ende forbedret træningshastighed.

Med alle disse ressource-opmærksomme teknikker demonstrerer denne afhandling, at det er muligt at opnå mere inden for modeltræning med færre ressourcer.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine learning has developed from a research direction under computer science to a global phenomenon. This growth, which exploded around the 2012 Imagenet vision challenge [8] with the introduction of the deep neural network AlexNet [9], has yet to slow down.

Looking back at the development from 2012 on reveals a line of deep learning architectures that have provided increasingly higher performance and widened the breadth of applications that deep learning has impacted. Analysis on text data saw success under model architectures such as the RNN and later LSTM, only to be later completely replaced by the Transformer architecture [10]. Similarly, image classification that started with AlexNet saw these CNN's replaced by visual transformers and later multi-modal solutions [10].

All these model developments, however, have one thing in common; size is leading the way. Increasingly impressive applications of machine learning are fueled by increasingly large models and datasets. Figure 1.1 shows the development of the model size of major machine learning models featured in academia and industry. The y-axis to denote this scaling, crucially, is *log-scale*. The model that started the current deep learning revolution, AlexNet, is millions of times lighter in required computing than recent state-of-the-art models produced by industry.

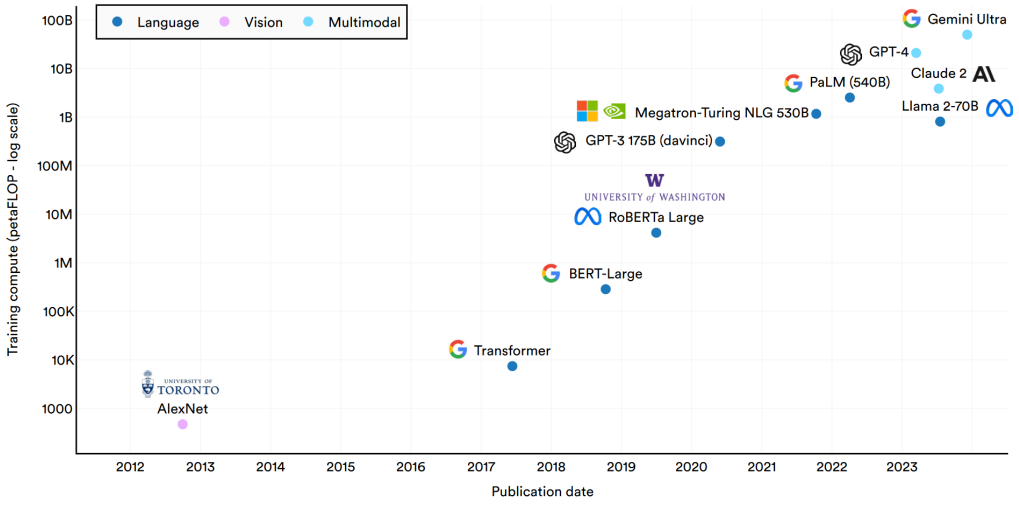


FIGURE 1.1: Training compute of notable machine learning models by domain. (Figure source [11])

This growth in machine learning has many similarities to the exponential hardware growth seen in the last decades following Moore's Law [12]. Exponential growth in these cases is not sustainable and inevitably leads to a slowdown of growth due to induced limitations. With Moore's Law the limitation comes from physics; how many transistors can fit on a chip. For deep learning models, the limit is defined by how much can be computed. The size of models is thus limited by the availability of computing.

There are two dimensions in which compute scales. Firstly, the amount of compute on a single computational node can be scaled up. This is the strategy that fueled AlexNet's original breakthrough [9] as they accelerated their network training by utilising a node fitted with graphics cards (GPUs) as accelerators. As the decade progressed, the compute capabilities of GPUs has increased significantly, and thus the compute one single node can offer to train deep networks has followed suit.

Secondly, the amount of nodes, and thus GPUs, can be scaled out. Distributed training allows multiple nodes to train a single model cooperatively by splitting the task on one dimension. Following the recent focus on Large Language Models, which require a lot of compute, companies such as Meta have acquired hundreds of thousands of GPUs [13] to train as large models as possible.

More powerful hardware could lead to more efficient training if the models do not increase in complexity. Figure 1.1 reveals that instead of reducing the footprint of model training, models have followed suit. While the expansion of resources, going from a single CPU to many GPUs, has led to the compute requirements of large models becoming attainable, that is only part of the story. The manufacturing of all the hardware for the data centres used to train these models comes at a significant price.

Additionally, running all the chips is not free, as it induces resource costs in terms of energy to run the chips and provide cooling for the setup. The result is that the explosive growth in model compute requirement is mirrored by a similarly explosive growth in compute cost. This cost, as exemplified in Figure 1.2, has easily exceeded one million USD for the largest of models.

This leads us to *resource-aware machine learning*. If we want machine learning to be sustainable, it is paramount to be aware of the resources that we use to train models and reduce this consumption wherever possible.

The height of the resource costs for training state-of-the-art deep learning models has serious consequences. High costs diminish the net value of models as training them can outweigh the benefits they bring. Furthermore, high costs restrict accessibility to contributing to state-of-the-art research for those with lower budgets, especially those in developing countries.

Apart from the monetary costs of all these resources, deep learning has grown so much that training models has a significant impact in terms of CO2 emissions. While carbon emissions can be difficult to estimate and compare due to differences in energy procurement [14], increased energy consumption generally leads to increased emissions. A small transformer model trained with 6 billion parameters is estimated to emit about an order of magnitude more CO2 than a US household does in a year [15]. Note that a workload like that is among the small models listed in Figure 1.1

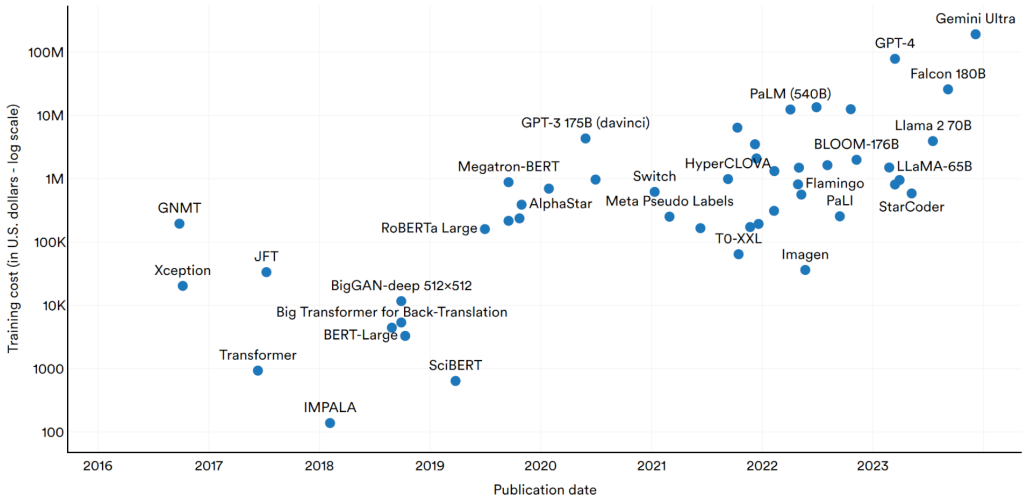


FIGURE 1.2: Training cost in USD of notable machine learning models [11].

and Figure 1.2. While the climate impact of deep learning training might have been modest a decade ago, we now face serious climate impact due to AI research.

1.1 Adding Transparency to Deep Learning

In general, the AI scene is heavily focused on accuracy and similar performance metrics. Simply put, the value of new deep learning models lies in producing results that outperform the previous state-of-the-art models. As a result, the majority of deep learning research is focused on improving this performance over anything [14].

Deep learning training is a complicated process and can be optimised in many different directions. Instead of focusing on accuracy, models can compete e.g. by being trained faster or with less energy. This is why energy usage and similar resource metrics should be reported on [4, 5]. Turning these into key metrics promotes further research in optimising models on these metrics.

The issue then, however, is that the results reporting for model training is severely lacking. Even simple metrics that can provide real value to researchers such as the time it took to train a model are rarely reported on in literature. More involved metrics, including memory requirements, carbon emissions, and energy draw, are practically non-existent. The absence of reporting on such metrics prevents innovation in reducing the resource consumption of training [4, 5].

Collecting and reporting resource metrics can be a hassle. For example, collecting energy consumption numbers heavily depends on the hardware being used to run the experiment. If we want to develop deep learning applications that are more resource-aware, the field needs to start reporting these metrics. The collection of resource metrics needs to be attractive in order to make the reporting of resource metrics to be attractive. While there is a range of tools available to record a whole range of hardware metrics [16, 17, 18], managing this data collection takes away valuable time that can be spent on model development.

For resource-aware data science, we need to promote resource-awareness by making collecting and reporting resource metrics attractive and hassle-free.

1.2 Keeping GPUs Busy

With deep learning model training riding the wave of computational power delivered by modern-day accelerators such as GPUs and TPUs, these chips have become the most important piece of hardware for training AI models. Resource-efficient training does not just involve reducing energy consumption, but also refers to utilising the resources we have the best we can.

GPU utilisation in modern-day cloud datacentres, sadly, leaves much to be desired, with the vast majority of jobs using less than two-thirds of allocated resources [19]. One major contribution of GPU resource loss is the relative inflexibility of GPU resource allocation. GPU resources are not shared between multiple jobs and thus GPUs cannot be partly allocated to training tasks. This becomes wasteful when the training task cannot be adjusted in size to the resource offering, either due to the nature of the task or the user's preferences.

A common method for increasing utilisation on CPU hardware is collocation [20, 21, 22]. Running multiple tasks at the same time allows for using more of the available resources when the hardware resources exceed what the individual tasks require. Furthermore, this can cover individual fluctuations in resource requirements as other work can fill in the gaps. Tasks that stress different parts of the hardware can especially benefit from collocation as the sum of their utilisation may highly utilise multiple aspects of the hardware.

GPU hardware is predominantly designed to run certain compute-heavy tasks, so-called *massively parallel tasks*, in an efficient manner, but context switching is expensive. To this end, multiple methods have been developed for collocating tasks on GPUs. These may provide a valuable link in improving the efficiency at which we train deep learning models.

For resource-aware data science, we need to make sure that acquired hardware is used to its fullest potential.

1.3 Reducing Data Redundancy

Collocating does not require any assumptions to be made about what is being trained. This does offer high flexibility, for example when training different models on different datasets. What this does not do, however, is improve the efficiency of every task itself. It does not matter whether the tasks are collocated or not; they act as black boxes and encompass the same operations.

Some collocation configurations may involve overlap between the training tasks. For example, the same model may be trained on different datasets, or the same dataset can be used to train different models. The latter is a common occurrence for exploration in deep learning training. It is common to evaluate multiple models on a dataset to find the best-performing configuration. This exploration can involve the hyperparameter space of the model (*hyperparameter search*) or the structure of the model (*neural architecture search*).

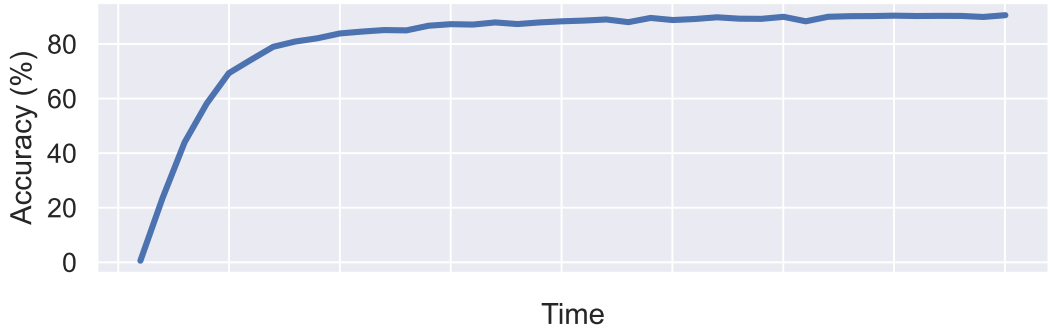


FIGURE 1.3: Development of model accuracy when training a ResNet [23] model on the ImageNet2012 dataset [8]. Accuracy rises quickly at first, after which it slowly fine-tunes to the final accuracy.

When multiple tasks are collocated that partly overlap with each other, we are introducing redundancy. After all, the models are trained completely separately, albeit on the same hardware, even though they operate e.g. on the same data. Considering that hyperparameter search and neural architecture search can be very expensive steps in the training lifecycle, reducing this redundancy can lead to significant resource gains. By unifying as many data operations as possible, fewer resources are required to run the training workload, allowing the workload to run on smaller hardware or more models to be collocated.

For resource-aware data science, we need to make sure that any resources we spend on training are required and not spent on redundant work.

1.4 Accelerating Time-To-Accuracy

In machine learning literature, the predominant focus is on the final accuracy the model can attain after training and optimising it. As discussed before, there is little attention to metrics regarding resource consumption, such as energy expenditure or time to train the model. Considering the importance of both the accuracy of the model and the time or resources it took to train the model, one particularly interesting metric is *time-to-accuracy*. This involves setting a target accuracy depending on the task and model (e.g. "90%") and measuring how fast the training process results in the model reaching that performance.

Considering time-to-accuracy as our target yields a strong incentive to improve the efficiency of the training steps. After all, considering we report and do not change the hardware used for training, we either have to improve how well the training steps are computed on the hardware, or modify the training steps so that fewer are required to reach the target accuracy.

When training a deep learning model from scratch, the model typically starts out with randomised weights. The result is that initial training sees large accuracy jumps as the model is starting to get a rudimentary understanding of the task. A typical image classification model training curve can be seen in Figure 1.3. As the model training progresses, the model has to recognise more intricate and difficult patterns. This leads to a slowdown in training progression, with the final (fine-tuning) stages requiring a lot of time for the last few percentage points of accuracy.

The heterogeneity of this training curve exposes the potential for optimisation. Model training is an iterative process that repeatedly exposes the model to the dataset. One epoch equates to one full pass over this dataset. The dataset does not change; the model goes over the same data points during initial learning as it does on fine-tuning. However, the complexity of what the model needs to extract from the dataset rises sharply. Hence, instead of considering the dataset as immutable, we can adjust the difficulty of the challenge to the model's needs. When the model has just started training, it can perhaps train on cheaper, smaller, data. This leads to faster time-to-accuracy and reduced resource consumption. Scalability in deep learning training is something that requires attention for both the feasibility of scaling training and the resource impact it has on our planet.

For resource-aware data science, We need to look at the training loop from new perspectives such as adopting a more dynamic approach with the input data sizes in order to improve the efficiency of training.

1.5 Thesis Statement and Contributions

This thesis contributes to the effort to reduce the energy and resource impact of machine learning.

Thesis Statement

Deep learning training is an expensive, iterative process that consumes significant on-prem and cloud resources. Practitioners use whatever resources it takes to deliver state-of-the-art models. Transparency is required to reduce the resource consumption of training models and practitioners should report on resource metrics, turning resource consumption into a target. Resource (hardware and data) sharing and progressive data loading are two ways to get more done in deep learning training with fewer resources.

The contributions of this thesis are summarised below:

- We design a system, *radT*, that automatically handles the collection and processing of deep learning metrics, including hardware resource metrics.
  - *radT* is designed to be as hassle-free as possible; attractive for data scientists and easy to use. Reporting resource metrics as well as accuracy will result in resource consumption becoming a target for innovation.
  - Additionally, we include a visualisation front-end that allows for quick exploration of this data.
  - Finally, we add automated support for managing GPU collocation.
- We thoroughly benchmark the current options for GPU collocation on modern-day machine learning workloads and illustrate when GPU collocation is beneficial for model training.

  - We evaluate both homogeneous, including multiple of the same model, and heterogeneous, including different-sized or typed models, and workloads.

  - We release a set of general guidelines that provide actionable recommendations on when and how to use GPU collocation.

- Expanding on GPU collocation, we isolate the case where models train on the same dataset and unify their data loading.

  - We design a system, *TensorSocket*, that automatically takes care of this data sharing with minimal effort from the user.

  - We thoroughly benchmark this system and compare it to state-of-the-art solutions for reducing data redundancy when training multiple models.

  - We provide suggestions as to when unified data loading is most beneficial, including but not limited to reducing cloud expenditure by switching to cheaper CPU hardware as a result of the more efficient data loading.

- We analyse data loading when training models under *time-to-accuracy*, and introduce *progressive data loading*.

  - We benchmark state-of-the-art manual methods of progressively making data loading more complex.

  - We combine our findings with resource readings resulting in a data loading strategy that scales up data progressively without hurting GPU resource utilisation.

  - Similar to our other research, we distil our results into a package that provides an accessible way of progressive data loading.

Furthermore, these contributions have resulted in the following publications and open-source software:

- Publications:

  - *Ties Robroek, Aaron Duane, Ehsan Yousefzadeh-Asl-Miandoab, and Pınar Tözün.*
    Data Management and Visualization for Benchmarking Deep Learning Training Systems.
    DEEM 2023

  - *Ties Robroek, Ehsan Yousefzadeh-Asl-Miandoab, and Pınar Tözün.*
    An Analysis of Collocation on GPUs for Deep Learning Training.
    EuroMLSys 2024.

  - (Under Review) *Ties Robroek, Neil Kim Nielsen, and Pınar Tözün.*
    TensorSocket: Shared Data Loading for Deep Learning Training.

  - And not yet submitted work for progressive data loading, as seen in Chapter 6.

- Open-source software:

- **radT: Resource Aware Data science Tracker**
  (https://github.com/Resource-Aware-Data-systems-RAD/radt)
  Data tracking and management tool introduced at DEEM.

- **migedit**
  (https://github.com/Resource-Aware-Data-systems-RAD/migedit)
  Command-line utility for management of NVIDIA Multi-Instance GPU.

- **TensorSocket**
  (https://github.com/Resource-Aware-Data-systems-RAD/tensorsocket)
  Share PyTorch tensors over ZMQ sockets, under review.

- **Progressive Training (working title)**
  (-)
  Automatically rescale training samples for faster training, not yet submitted.

Additionally, contributions were made to the following publications and open-source software not covered by this thesis:

- Publications:

  - *Ehsan Yousefzadeh-Asl-Miandoab, Ties Robroek, and Pınar Tözün.*
    Profiling and monitoring deep learning training tasks.
    EuroMLSys 2023.

  - *Maximilian Böther, Victor Gsteiger, Ties Robroek, and Ana Klimovic.*
    Modyn: A Platform for Model Training on Dynamic Datasets With Sample-Level Data Selection.

  - (Under Review) *Maximilian Böther, Ties Robroek, Viktor Gsteiger, Robin Holzinger, Xianzhe Ma, Pınar Tözün, and Ana Klimovic.*
    Modyn: Data-Centric Machine Learning Pipeline Orchestration

- Open-source software:

  - **Modyn**
    (https://github.com/eth-easl/modyn)
    Open-source platform for model training on growing datasets.

1.6 Roadmap

The four contributions of this thesis and necessary background reading are covered by the chapters as follows:

- Chapter 2 contains background on deep learning training, including the training loop itself, and especially data loading. It also covers the basics of the architecture of GPUs and briefly introduces GPU collocation.

- Chapter 3 introduces issues and limitations found with current resource tracking for deep learning training. It then follows this up with *radT*, a tracking framework that removes the hassle of managing resource tracking for training models. Furthermore, we introduce and explore the visualisation front-end

that is bundled with it. Finally, *radT* stands as a foundational work for the other chapters of this thesis.

- Chapter 4 focuses on the aforementioned GPU collocation. Three different methods for collocation are introduced, which are then benchmarked against each other to find their strengths and weaknesses. The chapter concludes with actionable recommendations for deep learning researchers who want to accelerate their own training with GPU collocation.

- Chapter 5 expands on the GPU collocation results from the previous chapter by investigating data redundancies when collocating model training. In this chapter, the system, *TensorSocket* is introduced for unifying data loading in order to resolve these redundancies.

- Chapter 6, similar to the previous chapter, tackles resource efficiency from the data side by reducing consumption from the data loading side. It introduces *progressive data loading*, reducing the difficulty of the data samples to more closely align with the requirements of the model at that point in time.

- Chapter 7 briefly revisits the previous chapters and discusses future directions that are being explored, or that might be worth exploring in the future, and concludes the thesis.

Chapter 2

Background

This chapter contains a brief background required for reading this thesis, including:

- a brief illustration of the variety in application domains of deep learning (Section 2.1).
- an introduction on deep learning neural network training (Section 2.2).
- an overview of the data pipeline for network training (Section 2.3).
- an illustration of the design of modern-day GPU accelerators and why they have become so relevant for training neural networks (Section 2.4).
- a list of metrics available to evaluate deep learning training on (Section 2.5).

2.1 Domain Variety in Deep Learning

Deep learning models have increased in size, but not all tasks have. Models such as AlexNet [9] and ResNetv2 [24] have been surpassed in accuracy by more modern models on the Imagenet2012 [25] challenge yet remain popular in some domains. Similarly, non-Deep Learning methods, such as support vector machines [26] and random forests [27], remain popular in some domains. This indicates that there is demand for optimising models, even on a smaller scale. Examples include:

- *Computer Vision.* Computer vision used to be dominated by convolutional neural networks, but now includes variety introduced by the Vision Transformer and mixes of both techniques. Additionally, Computer vision networks are often used with smaller datasets or other constrained resources for specific tasks.
- *Natural Language Processing.* Natural language processing, as aforementioned, has undergone a transformation away from recurrent networks to transformers. Transfer learning is common to optimise a large pre-trained network for a specific task.
- *Medical Imaging.* Medical imaging is very similar to computer vision, but introduces sharp requirements for diagnosing diseases with respect to false negatives. Additionally, datasets are often restricted in size and privacy-sensitive.
- *Recommendation Systems.* Recommendation systems generally use embedding tables to generate recommendations. They distinguish themselves as being

light on compute but requiring extensive VRAM to store these embedding tables.

2.2 Deep Learning Training

Deep learning is a subset of machine learning that tries to solve difficult tasks with a variety of deep models, most commonly deep neural networks. A deep neural network is a neural network that consists of more than two layers, including input and output. These layers consist of two operations each; a linear and non-linear combination. This attribute allows for deep neural networks to be universal approximators. As long as the network is large enough, the network will be able to approximate any function. This is shared among deep network architectures, such as recurrent neural networks, convolutional neural networks, and transformers.

Teaching a network to approximate a function is called *training*. This is an iterative process called *gradient descent* where the *weights* of the model are slowly tuned in order to study the intended task. The model type, architecture and *hyper-parameters* form the *inductive bias* of the model. This is the set of assumptions used by the model defined by the machine learning practitioner.

The base weight configuration of a model is sourced randomly. During training, the data is exposed to the model in full passes called *epochs*. With modern-day dataset sizes, it would be excessive to train on the whole epoch before making any changes to the weights. Instead, the data is sampled in subsets called *(mini)batches*. The weights are updated after every batch, and the data samples that are in the batches are randomised every epoch.

When training on data, we isolate two different steps. A batch is first sent through the neural network as the *forward pass*. The data is transformed following the layers of the network into model output. This is followed by the *backward pass*, which moves the gradients back up the model in order to calculate how to update the model weights. The backward pass is skipped when running inference instead of training as the model does not need to be updated.

The model is being trained to optimise on a specific *loss function*. The *loss* determines how well the model performed on a data point and is what the gradient is based on that is used to update the model. While the loss gives insight into how well the model is performing, prolonged training can cause *overfitting* which involves the model learning unwanted features from the data. To combat this, a separate *test set* is used to validate the model performance on data it has not seen during training.

While training deep learning models allows for universal approximation, it requires a properly sized network to do so. Deep learning networks have seen a sharp increase in size over the last ten years. This has allowed models to solve remarkable challenges, reaching near-human or even superhuman performance [28, 29]. An increase in model size is usually paired with an increase in dataset size. A larger model has more variables to train and tune. The additional data introduces a larger variety, preventing the model from overfitting the data, which is when the model learns an unintended representation.

2.3 Data Pipeline

Modern deep learning training happens on vast quantities of data. Imagenet2012 [8], the dataset on which AlexNet was trained [9], contains over a million training images in about 150 GB of data. Dataset sizes have grown over the decade, with the LLama line of LLM models being trained on trillions of data points [30, 31, 32]. This data needs to be made available to the model, which for most training pipelines means that it needs to be prepared and transported to the accelerator.

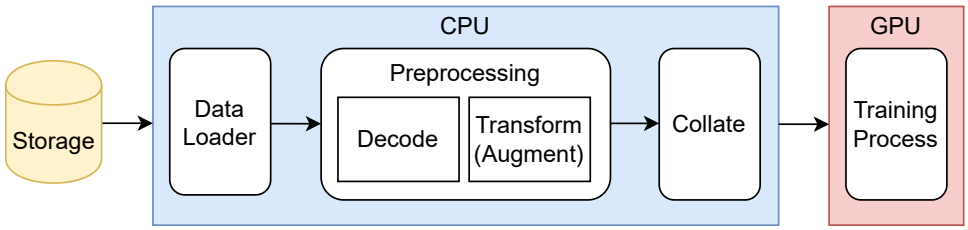
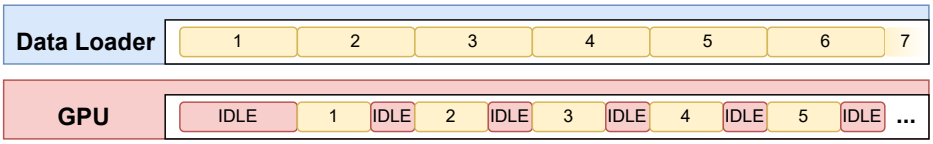


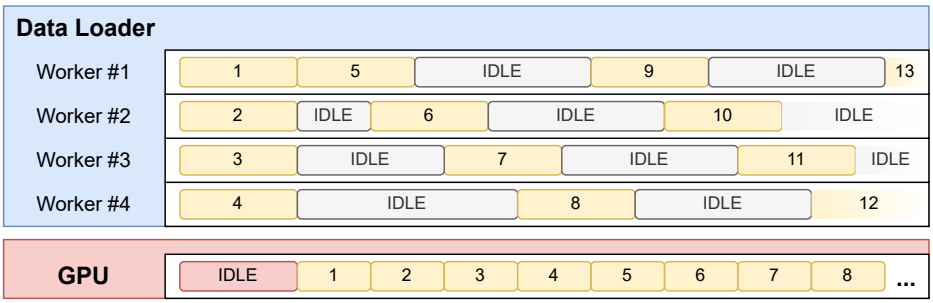
FIGURE 2.1: General overview of the data loading pipeline.

The dataset is often stored on disk as the full dataset generally cannot fit in RAM in its entirety. The (online) data loading pipeline, which encompasses the whole journey the data takes during training, thus goes from disk to accelerator. Figure 2.1 shows an overview of this pipeline, broken down into several steps:

- **CPU:** Group of operations required to fetch and translate the raw data into what the model expects.
  - *Data Loading:* Data is retrieved from storage or from cache. Considering the size of modern datasets and the fact that epochs only visit every data point once, the vast majority of access goes to storage. On the largest of models, it is common that data is streamed from storage further away as the size of the dataset exceeds local storage capacity.
  - *Decode:* The raw data format is decoded into the format required by the model. Depending on the training pipeline, data is either saved in a raw format, e.g. *.jpg* images or natural text files, or in a processed format, e.g. saved tensors.
  - *Transform (Augment):* While the data is now in a format recognised by the machine learning framework, it often requires additional transformation to facilitate correct model training. The transform step includes operations such as resizing or clipping the data so that it conforms to the expected input dimensions of the model. Furthermore, augmentations can be added, which are random transformations that increase the variety in the dataset.
  - *Collate:* Data is sent to the model in batches, which are groups containing a predefined amount of data points. During collation, data points are packed together in one big tensor, reducing the amount of data movement operations required to move the data.
- **GPUs:** Finally, the data is sent to the GPU and the model trains on the data.



(A) Single data worker.



(B) Multiple data workers.

FIGURE 2.2: Using multiple data loading workers in order to provide enough data loading throughput to match compute throughput. Yellow blocks denote batches. A batch needs to be loaded before it can be used by the GPU. The general aim is to minimise GPU idle time.

Keep in mind that the aforementioned data pipeline runs continuously, as batches are iteratively trained on. This, however, results in downtime for the GPU, resulting in significant efficiency losses. Once the GPU is done with the current batch, the next batch needs to be fetched, processed and moved. There are multiple methods used to combat this downtime:

- **Async execution:** The GPU being its own chip can process a batch of data without further management of the CPU. This means that while the GPU is processing, the CPU can already take steps in preparation for the next batch.
- **Data workers:** GPUs are designed to be able to process large amounts of data. If the data loading throughput is lower than how fast the GPU processes the data, the GPU will inevitably stall, as illustrated in Figure 2.2a. This can be resolved by having multiple data workers fetching data in parallel. This increases the total data loading throughput, preferably matching or exceeding the throughput of the GPU (Figure 2.2b).
- **GPU pre-fetching:** GPU pre-fetching involves pre-loading multiple batches on the GPU. This hides the last-mile latency induced by the connectivity between CPU and GPU (i.e., a PCI-e, NVLink, etc. connection). Without GPU pre-fetching the batch always needs to be moved from memory to GPU just before training on the batch. This delay can be overcome by storing a small buffer of batches on the GPU at the cost of some GPU memory.

In addition to online data processing, there is an offline data input pipeline that encompasses the introduction of new data to the dataset to be trained on. This is generally characterised as an ETL (extract, transform, load) process. Raw data is retrieved from its source, any pre-processing that can be done offline is applied, and the result is written to the dataset. Some operations, such as transforming the data

to the correct dimensions, could technically be done offline but are often done online regardless. This can depend on multiple factors, e.g. whether the extra storage required to save the separate dataset is worthwhile because of the data reuse, or whether the operation is computationally prohibitive to warrant offline processing instead of the I/O overhead.

2.4 GPU Acceleration

Deep learning training has developed hand-in-hand with accelerators. GPUs, initially designed for graphical applications, are able to efficiently process heavily parallelisable operations. After all, graphical operations operate on image data, which are essentially large matrices of values. GPU acceleration allows models to be larger and process more data, resulting in models that can efficiently use the GPU. Note that the CPU still controls the training process; the GPU is simply used to accelerate certain steps in the process as orchestrated by the CPU.

Model training involves the aforementioned forward and backward pass. During the forward pass, the data batch goes through the network layer by layer. In general, the output of a layer is computed by taking the results of the previous layer, multiplying these with the weights, and summing the result. This can efficiently be done on the GPU using a single matrix multiplication. While access to this compute leads to powerful deep learning models, networks must be designed around leveraging parallel computations such as matrix multiplications in order to benefit from this.

Not all model architectures can fully optimise GPU horsepower. In Natural Language Processing (NLP), recurrent architectures such as Recurrent Neural Networks (RNN) and Long Short-Term Memory (LSTM) have been dominant for most of the last decade. These architectures cannot process input data in a massively parallel way as there are dependencies between the data points due to their recurrent nature. Nowadays, NLP is dominated by Transformer models, which are able to provide a much more parallel-friendly compute structure.

Depending on the workload, different parts of the GPU can form a bottleneck. As a general role, any data movement is very expensive when dealing with GPU acceleration, causing data movement to often be challenging to manage. Ideally, data movements should be hidden by having the GPU run computation while the movement happens. Bottlenecks in data movement and compute can cripple the overall throughput of the GPU. Furthermore, large models such as Large Language Models (LLMs) are often restricted by the available GPU memory due to the sheer size of their weights. Running out of memory is strictly worse than having a compute or data movement bottleneck as it will kill the process.

2.5 Measuring Deep Learning

Machine Learning progress is predominantly advertised in the form of performance metrics such as accuracy, but there is a wide selection of software and hardware metrics available to measure training with. A selection of interesting metrics is detailed below:

- Metrics from the machine learning framework.

  - *Loss:* The degree to which the model training predictions do not match the ground truth. Loss is the target metric of the gradient descent loop and thus the model is trained to minimise loss.

  - *Accuracy:* How well the model performs on the test set. Accuracy notes the share of correct predictions. For more difficult challenges, a top-N accuracy can be used instead in order to ease the task difficulty.

  - *Iterations/s:* Throughput measure. One iteration usually refers to an individual sample, though sometimes can refer to a batch of samples. Higher throughput indicates more efficient processing of the training data.

- Metrics from the host (CPU) side, collected by tooling such as TOP [18] and iostat [33].

  - *CPU Utilisation:* How active the CPU is. The CPU is mostly used for data processing tasks during training and is often underutilised.

  - *CPU Memory Usage:* How much CPU's DRAM is used to train the model. This memory is primarily used as an intermediary between disk and GPU memory.

  - *MB Read/Written per Second:* How active the disk is. Data loading is usually the bulk of disk communication. Excessive disk communication may bottleneck data throughput and in term idle the GPU.

- Metrics from the accelerator (GPU) side, collected by NVIDIA tooling such as SMI [17] and DCGM [16].

  - *GPU Utilisation:* How active the GPU is. The GPU is used to accelerate deep learning by performing parallel operations efficiently. The neural network weights and neural network computations are done on the GPU. NVIDIA's default GPU utilisation metric can be unreliable as it measures how much of the GPU saw *any activity* during the last sampling step. A better indication for GPU utilisation is gathered from more specific metrics available in NVIDIA tooling [34], such as SM Activity (SMACT) and Graphics Engine Activity (GRACT). In general, we want the utilisation of the GPU to be maximised due to the importance of the GPU.

  - *GPU Memory Usage:* How much VRAM of the GPU is in use. This memory is used to store e.g. the model weights, gradient results and data batches. Exceeding the memory available on the GPU will crash the model training process.

  - *Memory Utilisation:* The share of time that the memory is either being read or written to. A high memory utilisation indicates heavy pressure on GPU memory bandwidth. Excessive memory utilisation can be an indicator of bottlenecks due to the relative expense of memory operations on the GPU.

  - *PCIe Throughput / NVLink Throughput:* Throughput of connections to and from the GPU. PCIe typically serves as the connection to the rest of the system, while NVLink-enabled systems can exchange data with other GPUs directly. Usage of the appropriate connection, e.g. NVLink when

training with multiple GPUs at the same time, is paramount for efficient training.

– *Power Draw:* Measures the wattage of the GPU. Power draw reported by NVIDIA tooling can be inaccurate when using the default readings, though there are workarounds to get within a 5% variance [35]. Energy measurements are key in estimating the resource consumption of deep learning.

Chapter 3

radT - Resource Aware Data Tracker

3.1 Introduction

Deep learning has become a staple in data science. Large deep learning models provide state-of-the-art functionality solving many problems not solvable by conventional algorithms [9, 24, 36]. Models need to be trained before being deployed in production. This training is an expensive iterative process in which the model iterates over a dataset multiple times. The growth in deep learning has been paired with an exponential growth in model and dataset size. More powerful and optimised hardware is required to facilitate the training of such models. This, in addition to the increase in training times, has inflated the resource requirements of deep learning training to a level where it can no longer be ignored.

GPUs are the de facto commodity hardware for meeting the resource requirements of deep learning. Today's GPUs are significantly more powerful than those of ten years ago. In order to improve the utilisation of hardware resources it is paramount that we use GPUs to their maximum potential. This requires tuning the training process to properly fit the hardware. On the other hand, a problem may not have a large enough dataset to warrant a large model, or the ideal training setup for a model might not be able to utilise all of the GPU resources [6, 37, 38]. This poses an issue when training neural networks as this process usually takes exclusive access to a GPU. This may lead to resource wastage as the model may not be large enough to saturate the GPU. As the scale of the hardware increases, underutilisation of hardware resources becomes a serious consideration for data scientists training deep learning models. All these issues underline the need for performing systematic experiments that evaluate the impact of certain configurations on deep learning training and hardware utilisation.

In the machine learning training space, there has been considerable work to provide insights into the training process. Techniques to improve model selection are focused on e.g. model accuracy instead of hardware utilisation [39]. Platforms such as WandB [40] and MLFlow [41] provide extensive tracking and management functionalities, but their hardware monitoring is limited. MLOps tools like Polyaxon [42] and Kubeflow [43] provide a solution for deploying training tasks on clusters and may log hardware metrics if the user wants them to, but are not specifically designed for keeping and exploring detailed benchmarking data with hardware metrics. Umlaut [44] provides accessible and flexible metric collection, but does not offer GPU

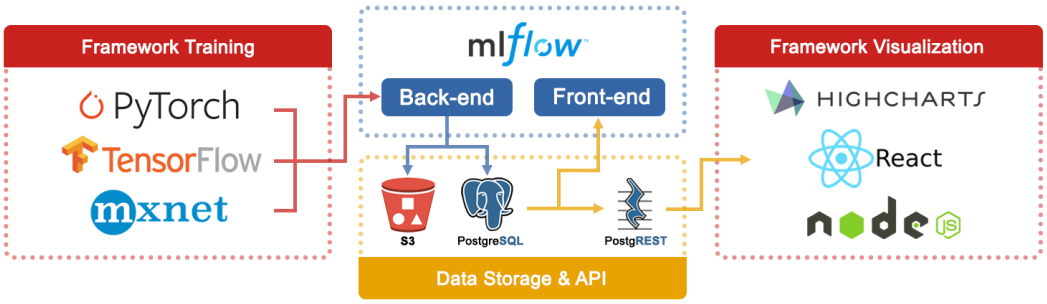


FIGURE 3.1: Dataflow architecture of our solution.

metrics. Finally, automated machine learning offers a variety of exploration tools [45, 46], though again focusing on model accuracy.

In this chapter, our goal is to build and demonstrate a framework that aids data scientists and machine learning systems researchers when performing systematic experiments that also takes hardware into account. We have identified six requirements and challenges for such a framework. Firstly, in order to provide a rigorous analysis of model training performance, a large amount of configurations has to be examined. This requires a large system and is made more challenging by the time required for training a single workload. Even when using the aggressive limiting measure of capping training to 5 epochs (training iterations) will not guarantee that workloads take less than a day to train. Secondly, a combination of software and hardware metrics, such as training accuracy and power consumption, have to continuously be collected during this training process. This requires both integration with the training script and a variety of hardware profiling and monitoring tools. Thirdly, the data, in the form of time series, quickly grows as training goes on. This results in gigabytes of numeric data which then needs to be sifted through using a flexible yet efficient process. Fourthly, multiple different data sources must be compared with each other, yielding a range of different data visualisation use cases. Fifthly, most of the data timeline may be inconsequential and repetitive, and interesting parts must be identified and investigated. Lastly, the solution needs to be as convenient to use as possible. Training a model via the framework should only impose minimal code intrusion and exploring results should be self-explanatory.

This chapter presents our framework, which allows for benchmarking and visualising deep learning model training in a reproducible manner. Our design takes the novel approach of repurposing the well-established machine learning lifecycle platform MLFlow [41] for machine learning systems analysis. Our extensions transform the platform to reach all of our aforementioned requirements in both back- and front-end while ensuring compatibility with pre-existing workflows and models. We describe how multiple combinations of models and datasets can be evaluated, in both isolated and collocated manner. Additionally, we show that there is support for different machine learning environments, and how existing models can be easily retrofitted to be supported by our framework. Researchers in our lab have extensively used our framework using a combination of datasets and machine learning models representative of a variety of deep learning workloads. Lastly, we delve into the visualisation front-end and illustrate some of the results from our test runs. The framework is publicly available on GitHub[1].

[1] https://github.com/Resource-Aware-Data-systems-RAD/dl-training-viz

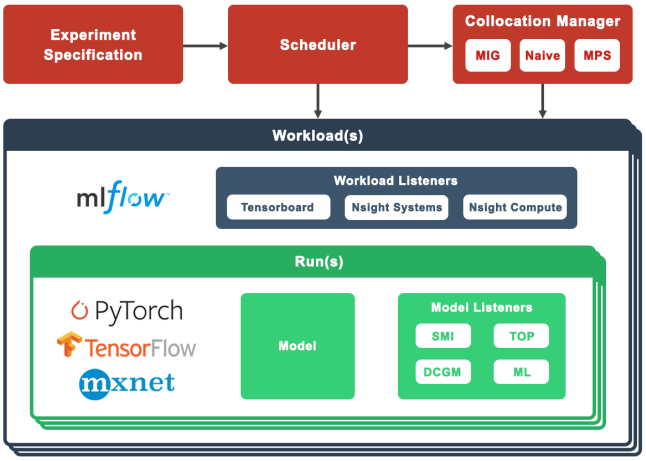


FIGURE 3.2: Data collection framework architecture. Experiments consist of workloads that can contain one or multiple model runs.

3.2 Framework

Our framework is split into a back-end and a front-end. Our data lifecycle extends that of MLFlow in order to meet our requirements. We illustrate the data architecture in Figure 3.1. Model training happens as an MLFlow client. This client sends data to the MLFlow server whenever there is a metric to report. The data flow between the client and server contains many events every second due to the large number of supported hardware metrics. This connection is frequent for all run-level listeners. Within MLFlow we distinguish the storage of relational data (experiment setup data, collected hardware monitoring metrics, etc.) and the storage of artifacts (stdout logs, profiling tool traces, etc.).

For our general data storage, we need a solution that can serve data collection, MLFlow, and our front-end quickly and reliably; the first and last of which are particularly susceptible to performance bottlenecks. Data collection happens continuously throughout model training and requires storage to be available at all times. Data exploration requires sifting through a large amount of data quickly, presenting considerable throughput and responsiveness requirements for the front-end.

We found that hosting the relational data in a separate PostgreSQL database yields the best results. While MLFlow defaults to local data file storage, we have found this to scale poorly with respect to data size and be unreliable. Furthermore, file storage requires data access to happen through MLFlow, which inhibited the performance of our front-end. We store artifacts in S3 storage on a different server, again forgoing the native file storage. We host our React front-end on a separate server that connects to the database via PostgREST, which is an automatic REST API extension to PostgreSQL databases. We host this REST API on the same server as the database, though it may be more beneficial to run these on separate servers to improve scalability. Similarly, larger setups may benefit from two copies of the database, where one is to write metrics to and the other is for the front-end to read from. This would remove any interference during the training process where metrics are repeatedly written to the database.

```
Experiment,Workload,Status,Run,Devices,Collocation,Environment,Model,Data,Listeners,Params
20,1,,,0,-,pytorch,timm.resnet26,/raid/datasets/cifar10/,smi+top+dcgmi,batch-size=128
20,2,,,2,3g.20gb,pytorch,timm.resnet26,/raid/datasets/cifar10/,smi+top+dcgmi,batch-size=128
20,2,,,2,3g.20gb,pytorch,timm.resnet26,/raid/datasets/cifar10/,smi+top+dcgmi,batch-size=128
20,3,,,0,MPS,pytorch,timm.resnet26,/raid/datasets/cifar10/,smi+top+dcgmi,batch-size=128
```

FIGURE 3.3: Experiment .csv file. Every row corresponds to a model training.

3.3 Back-end

Figure 3.2 illustrates the hierarchy presented to the user. Following the structure of MLFlow, individually trained models are called *runs* and are organised in experiments. We introduce a new layer in between runs and experiments called *workloads*. This is required as we include testing of multiple models at the same time as a requirement to test the impact of workload collocation. A *workload* consists of one or more model runs and a collection of workloads form an *experiment*. We will now go over the concepts introduced in the back-end pipeline, including the scheduler, environments, collocation, and listeners.

3.3.1 Scheduling

The execution of training experiments is managed by the workload scheduler. Models can be trained individually by use of a command line interface or be structured into experiment files. These are CSV files that can be edited in any text editor as shown in Figure 3.3. Every row is a single to-be-trained model. Models can be trained in a collocated fashion by sharing a workload ID. The numbering system of workloads is up to the user. In our case, we opt to use three digits for workload identification. All models in a workload are trained concurrently. When model training terminates, either due to success or failure, the row is tagged as such. Once all the model training jobs terminate, the next workload is started. This repeats until all rows have been executed. Rows list all parameters required for training the model with the framework. In particular, the *params* field specifies any pass-through parameters that are sent directly to the model training script.

3.3.2 Environments

MLFlow as a platform allows machine learning researchers to train and store their models in a controlled fashion. We leverage their environments feature to ensure that our training is reproducible. Models can be trained in either anaconda environments or docker containers. We collectively call these the environments supported by the framework. Whenever a model is added, it is included in one of these environments. Any deep learning library, such as Tensorflow [47], Tensorflow-Keras [48], or Pytorch [49], is supported as long as an environment for it is included. This ensures compatibility and reduces the number of abstractions required to adopt an existing codebase to the framework. Code intrusion is kept to a minimum for the actual model training code. Achieving the basic hardware tracking functionalities only requires two lines of code:

```
from mldgpu import MultiLevelDNNGPUBenchmark
...
with MultiLevelDNNGPUBenchmark() as run:
  ...
```

Support for training the model outside of the framework can be kept by encapsulating these lines in conditional checks.

3.3.3 Collocation

Collocation allows for multiple models to be trained simultaneously, increasing hardware utilisation. Multiple models can be trained on multiple GPUs, but can also share the same GPU. We support multiple technologies for sharing the same GPU resource:

- *MIG*, Multi-Instance GPU, is a hardware mechanism for recent NVIDIA workstation graphics cards that allows for hardware partitioning of the GPU [50]. This allows for multiple models to train without interference.
- *MPS*, Multi-Processing Service, is a software mechanism by NVIDIA for managing collocated processes on GPUs [51].
- *Naive* refers to simply launching multiple processes to use the same GPU without any other measures taken.

3.3.4 Listeners

In addition to the model performance metrics collected by MLFlow we require a host of hardware metrics to evaluate training performance. We introduce a group of additional processes called listeners that automatically record this information. By default, we launch a full set of listeners to capture both host system and GPU hardware metrics. This preset can be overwritten by setting the listeners parameter of the run. Additionally, we provide an interface for intuitively adding new listeners. The following run-level listeners are included by default:

- *TOP* is a tool for recording CPU hardware metrics [18].
- *NVIDIA-SMI* is a tool by NVIDIA for recording GPU metrics. NVIDIA-SMI yields GPU-wide metrics such as GPU utilisation, memory usage, and power consumption [17].
- *DCGMI*, or Data Centre GPU Management Interface, is a NVIDIA tool for recording more advanced GPU metrics. Additionally, DCGMI supports metric collection under MIG for individual MIG-partitions [16].

We support workload-level listeners in addition to these run-level listeners. These collect more detailed information but are significantly more likely to impact the performance of the model training. NVIDIA Nsight Systems [52] and Compute [53] in particular can have a pronounced effect on training performance [34]. Workload-level listeners are therefore disabled by default but can be enabled easily inside the framework.

3.4 Front-end

While MLFlow comes with its own set of data exploration tools, they do not fulfil many of our research requirements. Firstly, the tools have not been built around

comparing runs to each other and often provide extremely limited functionality. Secondly, they are not designed to handle large amounts of data points which can frequently take over 10 seconds to render a single time series. Lastly, the concept of a *workload* is central in our data architecture but does not exist in the MLFlow workflow.

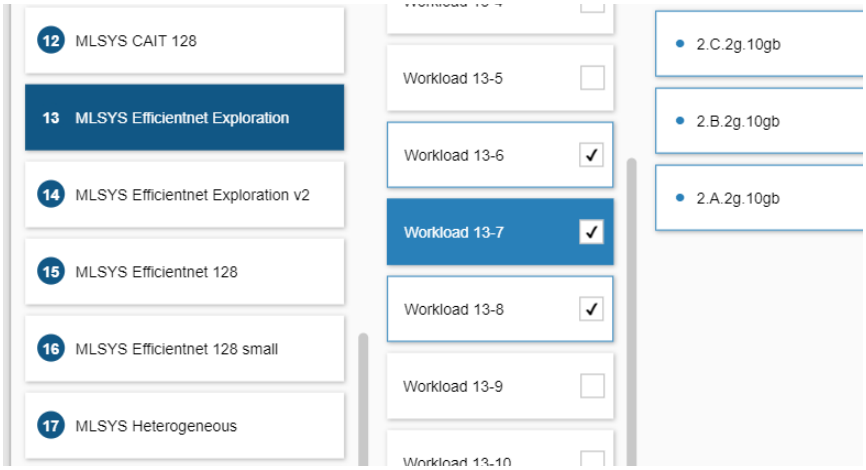


FIGURE 3.4: Model runs are organised into workloads and experiments which the user can then explore and visualise.

To address these issues and serve our preferred workflow, we introduce a novel front-end application layer which consists of two primary components: an interface for data selection and an interface for data visualisation. Before describing these components in detail, we will first establish the front-end's primary use cases which were identified from dissecting our experimental results:

- *Intra-workload*, where the models trained within a workload are to be compared to each other.
- *Inter-workload*, where workloads have to be compared to each other by taking the aggregate of their contained runs.
- *Mixed*, where specific runs of different workloads are to be compared to each other.

Figure 3.4 depicts the front-end's data selection interface where data sources can be browsed in a hierarchical manner. The user first specifies what experiment to explore, after which the workload and corresponding runs can be navigated and optionally selected via checkboxes. As a shortcut, all runs in a workload can be selected by clicking the workload's checkbox. Data from multiple workloads and multiple experiments can be selected at the same time and the system will automatically decide how to deal with the selected data. At any time, all currently selected runs are viewable by the user on the rightmost side of the interface. This section also serves as a shortcut to remove selected runs instead of having to locate their checkboxes again.

After confirming their data selection, the user is directed to the second primary component where the data can be visualised. This interface initially appears blank and simply provides a dropdown list of the available metrics which can be visualised with the currently selected runs. Once the user has chosen a metric, the interface

FIGURE 3.5: Using the Highcharts.js library, generated charts are interactive and responsive, allowing for quick dissection of the data.

will reload with a corresponding graph for that metric (see Figure 3.5). There is no limit to the amount of graphs which can be generated and the selected runs can be changed between graphs. This allows for the comparison of different datasets within the same interface. Visualisations are also fully interactive and support toggling, zooming, and clipping, as well as exporting to PDF, PNG, or SVG formats. Finally, an interactive version of any visualisation can be shared via a small file which will re-fetch the data from the server.

3.5 Conclusions

We have presented our framework for benchmarking and evaluating machine learning training. We have identified the challenges connected to collecting and processing real-time training data in an efficient and accessible manner. Additionally, we have tackled the visualisation of this data, allowing for efficient and effective data exploration. We are able to compare data in experiments and between experiments with a unified interface. In addition to our own experimental analysis work with collocated workloads [54], our framework has been actively used in our lab to do experiments for medical imaging research[2]. We invite other researchers interested in both hardware and software metrics to consider it for their own research pipelines.

[2] https://purrlab.github.io/

Chapter 4

GPU Collocation

4.1 Introduction

Today's GPUs are significantly more powerful than those of a decade ago. Modern GPUs, together with larger datasets, facilitate the exponential growth of deep learning models. Many data scientists, however, do not require large models in practice. For example, a problem may not have a large enough dataset to warrant a large model or the ideal batch size for training the model may not be large enough to utilise all of the GPU resources [6, 37, 38, 19]. This poses a hardware underutilisation issue when training neural networks as the training process usually takes exclusive access to a GPU. This problem gets exacerbated with each new GPU generation offering more hardware resources.

Workload collocation is a method for increasing hardware utilisation by running multiple applications at the same time over the same hardware resources. When a workload does not require all of the resources available on a device, a workload with additional applications can be considered. That way, the device and its resources are shared among the collocated applications. While workload collocation is heavily studied for CPUs [20, 21, 22], its opportunities and challenges have been largely unexplored for modern GPUs. In addition, unlike CPUs, GPUs lack sophisticated resource-sharing methods such as virtual memory and fine-grained sharing.

Today, there are several methods for workload collocation on a GPU. Firstly, multiple processes can be assigned to the same GPU simultaneously without any explicit process management. Alternatively, the collocation can be more precisely managed, for example via NVIDIA's *Multi-Process Service (MPS)*. Finally, the latest generations of NVIDIA GPUs can be partitioned into fully isolated GPU instances at the hardware level via *Multi-Instance GPU (MIG)*.

This chapter analyses different ways of collocating deep learning model training on NVIDIA GPUs. Specifically, we investigate the strengths and limitations of the new MIG technology in contrast to the older methods. We characterise the performance of the above-mentioned collocation methods on an A100 GPU. We diversify our workload by considering three datasets (ImageNet, ImageNet64x64, CIFAR-10) representing different sizes (large, medium, small). Furthermore, we acknowledge that the current deep learning landscape employs a wide variety of model architectures. We investigate two popular convolutional models (ResNet, EfficientNetv2) and one transformer model (CaiT). Additionally, we collocate a recommender model with a vision model to demonstrate the merits of workloads containing models that

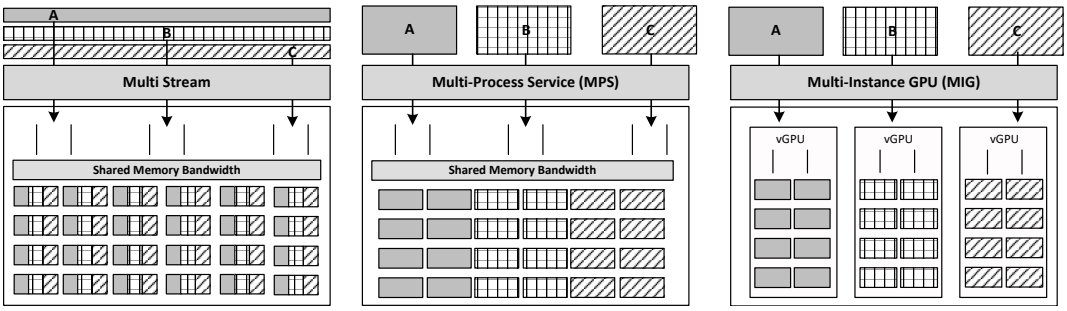


FIGURE 4.1: Collocation methods on modern NVIDIA GPUs. A, B, and C are different processes launched by the host CPU to run on the same GPU in a collocated manner using naïve approach (left-hand side), MPS (middle), and MIG (right-hand side).

stress different parts of the hardware. Our results highlight that:

- When model training is unable to utilise the full GPU on its own, i.e., when running on our small- and medium-sized training cases, or when running cases that stress different parts of the GPU, training multiple models in collocated fashion presents considerable benefits. On the other hand, for large model training, collocation provides either limited improvements to throughput as the GPU becomes over-saturated or cause model training to crash when the available GPU memory is not big enough to hold the combined memory footprint of the collocated models.

- On all the combinations we evaluated, MPS performs better than naïve and MIG collocation, achieving up to 80% and 40% higher throughput, respectively. It is also incredibly flexible, allowing single-user workloads to get the most out of the hardware with minimal setup required.

- MIG offers strict separation of the GPU's memory and compute resources across the collocated workloads, eliminating interference. It also allows multi-user collocation, unlike MPS, and can achieve higher energy efficiency when the partitions are set well. On the other hand, MIG-based collocation is more rigid, since MIG requires creating hardware partitions a priori. For the cases of well-defined workloads, one can create the ideal MIG partitions and leverage MIG-based collocation. However, for more dynamic workloads where the workload mix changes over time, MIG would require re-partitioning to perform well, whereas other collocation methods can still provide benefits.

The rest of the chapter is organised as follows. Firstly, Section 4.2 gives background on GPU collocation techniques and surveys related work. Then, Section 4.3 describes our experimental methodology and setup, and Section 4.4 presents the results. Section 4.5 outlines guidelines for collocation based on our results and touches on some challenges we encountered during our analysis, and finally, Section 4.6 concludes the chapter. Our artifacts are publicly available on GitHub[1].

[1] https://github.com/Resource-Aware-Data-systems-RAD/collocation-analysis-artifacts

4.2 Background

This section first provides background on different methods of collocation. Then, we survey related work on workload collocation for deep learning.

4.2.1 Collocation on GPUs

Figure 4.1 illustrates the three collocation methods we study in this chapter. We describe each of them briefly here.

Naïve (or Multi-Stream)

With CUDA 7, the option of running multiple processes at the same time using their own CUDA stream on the same GPU is introduced. A *CUDA stream* [55] is a sequence of operations that execute on the GPU (i.e., kernels and data transfers) in the order in which they are issued. While operations within a stream are guaranteed to execute in the prescribed order, operations in different streams can run concurrently. This concurrency greatly helps with overlapping the stall time due to the data transfers between the host CPU and GPU in one stream with work from another stream.

We call this type of workload collocation the *naïve* method since it offers a limited way for sharing GPU resources. This is because the streams have to share the GPU compute resources in a time-based manner rather than having resources explicitly dedicated for each stream (Figure 4.1 left-hand side).

Multi-Process Service (MPS)

The *multi-process service (MPS)* [56] enables the host CPU to launch multiple processes on a single GPU. Similar to naïve collocation, these processes share the GPU memory and memory bandwidth. However, unlike naïve collocation, the streaming multiprocessors (SMs) of the GPU are split across the different processes. Assignment of the SMs is done by the MPS daemon automatically, unless explicitly stated by the user, based on the provisioning of the GPU compute resources needed for each process (Figure 4.1 middle). While this provisioning introduces some process management overhead, splitting resources this way avoids context switching of kernels from different processes on the same SM. This reduces interference across the different processes compared to the naïve approach.

One limitation of MPS is that the processes have to be launched by a single user for security reasons. Therefore, MPS cannot be used to collocate applications launched by different user accounts.

Multi-Instance GPU (MIG)

Multi-instance GPU (MIG) [50] is the most recent collocation technology introduced with NVIDIA's Ampere GPUs. It provides hardware support for splitting a GPU into smaller GPU instances of varying sizes. These GPU instances may run different processes each allowing these processes to run in parallel on the same GPU (Figure 4.1 right-hand side).

MIG-capable A100 GPUs consist of multiple slices. The memory of the GPU is split into 8 memory slices and the compute side is split into 7 compute slices, plus one

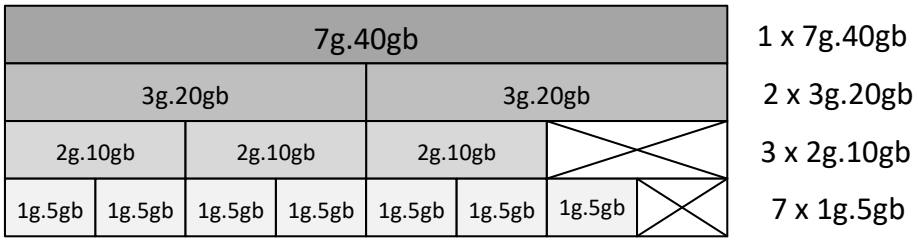


FIGURE 4.2: Possible MIG partitioning schemes on a NVIDIA A100-40GB GPU. Horizontals can overlap (collocation) but verticals cannot. For example, having a 3g.20gb instance is not compatible with five 1g.5gb instances but is compatible with two 2g.10gb instances (figure adapted from [50]).

reduced slice for the partition management overhead. These can be combined into GPU instances providing a partitioning of the GPU. A limitation of enabling MIG is that it does not allow for one model to be trained on multiple GPUs or GPU instances. On the other hand, each partition is strictly separated in terms of hardware resources preventing any form of interference across partitions.

An A100 GPU with 40GB memory supports several available partitioning profiles (see Figure 4.2). The smallest possible GPU instance is one with just one memory slice and one compute slice, `1g.5gb`, with 14 streaming multiprocessors (SMs) and 5GB of memory. Consecutively, a `2g.10gb` profile consists of two compute slices (28 SMs) and two memory slices (10 GB of memory). The other available profiles are `3g.20gb`, `4g.20gb`, and `7g.40gb`. The last profile consists of almost all of the GPU resources. However, using the GPU without MIG mode is not analogous to running this large profile as the compute capability of the GPU is hampered slightly due to MIG management overhead; i.e. the reduced compute slice as mentioned above (10 SMs). The ideal GPU instance sizes may vary from workload to workload following the memory and compute needs of the models.

Many different partitions are possible as long as the maximum resource capacity is not exceeded. For example, splitting the GPU into a `4g.20gb` and `1g.5gb` instance is possible but two `4g.20gb` instances would exceed the compute resources of the device. There is, however, a notable exception. While a split of one `4g.20gb`, one `2g.10gb`, and one `1g.5gb` instance is possible, one cannot proceed with a split of one `4g.20gb` and one `3g.20gb` instance, despite the values appearing to sum up to the maximum resources of the device. Such partitioning rules are set by the GPU itself, and the allowed set of instances and configurations varies across different types of Ampere GPUs (A100, A30) as well as the NVIDIA GPU architectures that come after (e.g., H100).

Finally, the amount of memory slices and the amount of compute slices may differ in a partitioning. Specifically, a GPU instance may be split into multiple compute instances from the compute side with unified memory. This can be useful when compute and memory requirements do not follow the same pattern. For example, one could run a memory-intensive model and a compute-intensive model with isolated compute instances on a single GPU instance.

4.2.2 Related work

Weng et al. [57] observe 6000 GPUs on Alibaba clusters and highlight the challenges of cluster scheduling mechanisms that result in dramatically low GPU utilisation. Jeon et al. [19] perform a similar experimental analysis focusing on deep learning training on a Microsoft GPU cluster. These works motivate studying workload collocation on GPUs as a way to overcome such low utilisation.

Collocation on GPUs has been studied in two dimensions: software-based and hardware-based approaches.

Software-based approaches either focus on developing better primitives for collocation on GPUs or provisioning the resources of GPUs for running multiple applications. cuMAS [58] is a host-side CUDA task scheduler, which receives multiple CUDA calls and reorders them based on data transfer behaviour to increase overall system utilisation. Ravi et al. [59] propose a framework as a transparent layer for executing applications within virtual machines to share one or more GPUs. Horus [60] uses machine learning for predicting GPU utilisation of deep learning training tasks. Afterwards, it feeds the cluster scheduler and resource manager with the information to make better decisions for collocating different workloads.

In contrast, hardware approaches propose micro-architectural changes to GPUs to enable finer-grained and more precise multi-application execution within a GPU considering performance, utilisation, and quality of service trade-offs [61, 62, 63, 64, 65, 66].

MIG is a relatively new technology and there have not been many works that thoroughly explore its possibilities. HFTA [38] is a mechanism to fuse multiple model training runs for hyper-parameter tuning into one training run. The authors show the effectiveness of HFTA compared to using MPS or MIG to run multiple training runs in parallel. MISO [67] runs MPS on a `7g.40gb` MIG instance to predict the best MIG configuration for different jobs.

Finally, similar to our work, Li et al. [68] characterise the performance of MIG using deep learning models focusing on time and energy metrics. Their methodology is different than and complementary to ours. It covers a variety of deep learning use cases but doesn't consider sizing the models up and down. Furthermore, they don't compare against other forms of collocation such as MPS.

In general, our work is complementary to these works since we focus on an experimental methodology to investigate the strengths and limitations of MIG in contrast to the older collocation techniques such as MPS and naïve collocation and by using workloads of different sizes.

4.3 Setup & Methodology

This section details our experimental setup and methodology. First, Section 4.3.1 introduces the hardware system used for conducting our experiments. Next, Section 4.3.2 defines our metrics, their relevance, and how we measure them. Section 4.3.3 describes the models and datasets used in this study. Finally, Section 4.3.4 describes the list of the experiments and how we run them.

4.3.1 System

Our experiments run on a DGX Station A100, composed of an AMD EPYC 7742 CPU and four A100 40GB GPUs. The system is a pre-packaged solution provided by NVIDIA running *DGX OS*, a variant of *Ubuntu 20.04.4 LTS*. The CPU consists of 64 cores, 128 threads, operating at a base clock of 2.25 GHz with a boost clock of 3.4GHz [69]. There is 256MB of L3 cache and 512GB of DRAM available. Each of the A100 GPUs has 40GB of VRAM and supports up to 7 MIG instances with at least 5 GB of memory per instance (see Section 4.2.1). The A100 GPUs can communicate using NVLink among them, while there is a 16x PCIe 4.0 ($\sim$32GB/s bandwidth) connection between each GPU and the CPU.

4.3.2 Metrics

The goal of this chapter is to investigate the performance of different GPU collocation techniques instead of improving the accuracy of models for a particular use case. Therefore, the set of metrics we focus on is related to how the model training interacts with and gets impacted by the GPU resources.

Time per epoch is the time it takes to finish a single epoch of training for a particular model. GPUs are used in deep learning in order to reduce training time by exploiting the embarrassingly parallel nature of most deep learning computations. Therefore, time per epoch is the most fundamental metric to look at in our study. We time the second epoch of training, skipping the first one as a warm-up epoch.

GPU utilisation depicts how much the GPU is being used. We are interested in how active the whole GPU is depending on the workload executed and the collocation mechanism used.

We use **`SMACT`** (SM Activity) to track GPU utilisation. SMACT is the fraction of active time on an SM, averaged over all SMs. This provides a good indication of whether the GPU is in use [34]. This information is reported by the Data Centre GPU Manager (`dcgm`) [70]. SMACT can be tracked for the whole GPU when there are no MIG instances available and per MIG instance when there are.

When reporting utilisation under MIG, we aggregate the SMACT readings across MIG instances to compare to the readings from the entire GPU under naïve and MPS. Furthermore, this aggregate SMACT is based on the reduced available computer resources when MIG is enabled (7g 98 SMs vs the entire GPU with 108 SMs).

Memory footprint is the total memory space (in GB) allocated by all of the collocated models on the GPU. This metric is especially crucial for reasoning about the failed collocation attempts. Specifically, we measure the memory requirement after a full epoch of training to signify how much memory is needed for the model to train. We use `nvidia-smi` to collect the memory consumption for the whole GPU.

As naïve collocation and MPS share memory across all the runs within the workload, the available GPU memory must be able to accommodate the memory footprint for the collocated runs (Section 4.2.1). If the sum of memory required exceeds the capacity of the device, the workload will fail due to running out of memory. For MIG (Section 4.2.1), the memory available per GPU instance needs to be large enough to accommodate the memory footprint of the models mapped to that instance.

CPU-GPU interconnect utilisation details the amount of data transferred from the CPU to the GPU. This is useful as a measure of the activity between the host system and the GPU. This activity can differ between varying collocation methods and will likely increase with increasing degrees of collocation. We measure this utilisation with `dcgm` using `pcie_rx_bytes` readings.

GPU energy consumption is the amount of energy consumed by the GPU while training. We look at both the GPU power usage over time and the overall GPU energy consumption of training. Our goal is to observe the power consumption characteristics across different collocation methods and degrees of collocation in addition to the energy efficiency of each collocation scenario. A higher GPU utilisation or shorter time per epoch may not necessarily lead to a higher energy efficiency. We measure the power usage and energy consumption via `dcgm` using `power_usage` (watts) and `total_energy_consumption` (joules), respectively.

4.3.3 Models & Datasets

TABLE 4.1: Models & Datasets

Model	Dataset	#Parameters	Size
ResNet26	CIFAR-10	17M	small
ResNet50	ImageNet64	24M	medium
ResNet152	ImageNet	59M	large
EfficientNet_v2_s	CIFAR-10	22M	small
EfficientNet_v2_s	ImageNet64	22M	medium
EfficientNet_v2_s	ImageNet	22M	large
CaiT_xxs24_224	ImageNet	12M	large
DLRM	Criteo Terabyte	24B	very large

Deep learning achieves state-of-the-art models for a variety of use cases, such as speech recognition, image classification, and sentiment analysis. These fields feature a plethora of models with different compute and memory requirements. Table 4.1 lists the models and datasets used in our experiments.

Models

We select three model architectures representing a large range of image classification models in addition to a recommender model. In terms of the interaction with the hardware, deep learning applications that leverage GPUs are either compute- or memory-intensive on the GPU. This includes other models such as for speech recognition and object detection. The type of interaction with the hardware is the determining factor for the behaviour of the collocated training runs. By including a variety of compute-intensive neural network architectures under image classification and the memory-intensive recommender model, this chapter aims to cover a good representative set of training cases.

ResNet is a deep convolutional neural network that has been around since 2016 [23, 24]. In addition to being a popular choice for image classification and segmentation, ResNets [71, 72] can be scaled up and down in size, which makes them ideal for benchmarking over varying hardware resources. This helps with creating workloads of varying sizes for testing the different workload collocation options, especially

the different MIG partitions. We train ResNet26, ResNet50, and ResNet152 models to create *small*, *medium*, and *large* workloads, respectively. The medium model has significantly more parameters compared to the small one, and the large model has about twice the parameters of the medium model. We train these models on datasets corresponding to their size (see Section 4.3.3 for details).

EfficientNetv2 is a recent convolutional neural network [73] for image classification. EfficientNetv2's architecture is focused on delivering high performance while limiting the size of the model. In order to satisfy memory constraints on a single GPU, we exclusively train the small version of EfficientNetv2. We train the small version on all three image datasets creating *small*, *medium*, and *large* workloads.

CaiT is a visual transformer model. Unlike many other transformers for image classification, CaiT achieves high performance without the need for extra data [74]. We use the smallest version of CaiT (*xxs*) to satisfy memory constraints on the GPUs in our system (Section 4.3.1). Since most transformer model architectures for image classification start at a relatively large size compared to their convolutional counterparts, we only create a *large* workload with CaiT, training it on the largest image dataset.

DLRM is a recommendation model. Unlike the previous vision models, this model is used to provide personalisations and recommendations based on past user behaviour [75]. The model is significantly less GPU compute-heavy than vision models but requires large amounts of CPU and GPU memory. We use the MLPerf configuration of the model as provided by the authors. We train DLRM on the Criteo Terabyte dataset [76]. Training DLRM for an epoch on this dataset takes significantly longer than training any of the vision models for an epoch. We therefore look at the rate at which DLRM goes through the data instead of the time that a full epoch takes ($\sim$ 4.3 days).

Datasets

We accompany the vision models with three datasets of varying sizes, forming small, medium, and large workloads. We also include a very large dataset for the recommender model.

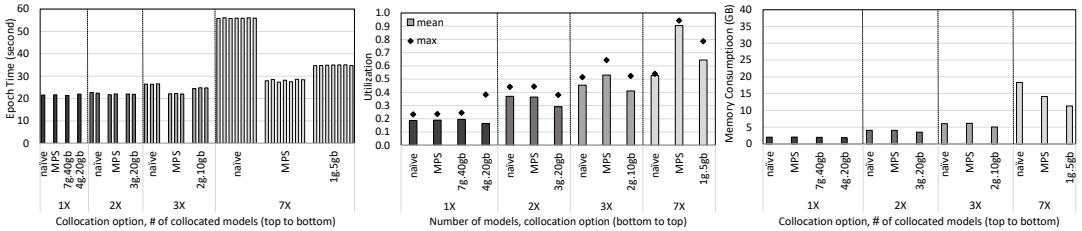
For our ***small*** dataset we have CIFAR-10 [77] (163 MB), containing 60,000 labelled 32×32 pixel images divided over 10 classes. The dataset is split into 50,000 training and 10,000 test images.

Our ***medium*** dataset is a downsampled version of the large dataset, ImageNet2012, called ImageNet64$\times$64 [78] (12 GB). We will refer to this dataset as ImageNet64.

For our ***large*** dataset, we use the unmodified ImageNet2012 [8] (138 GB), referred to as ImageNet. Imagenet2012 is a collection of 1,431,167 labelled images from 1,000 different classes. The dataset is split into 1,281,167 training, 50,000 validation, and 100,000 test images. Unlike CIFAR-10, the dataset is not balanced and the images are not all uniform in size. Every picture is resized to 224×224 using the *nearest pixel* interpolation method to conform with the size of images used in the original ResNet specification [23].

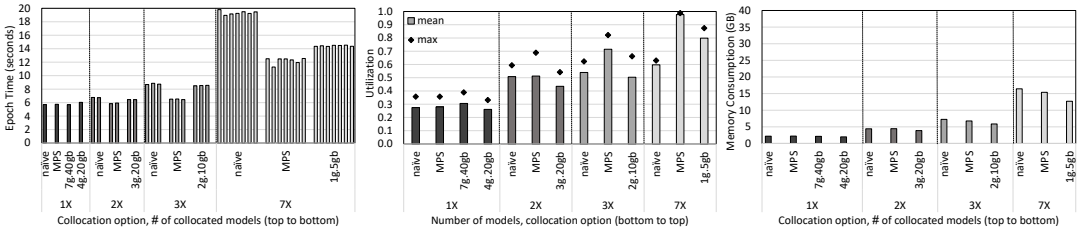
Finally, we use the Criteo 1TB Click Logs dataset [76] for training the DLRM model.

The dataset consists of online advertisement click-through logs and contains 24 days of data. Crucially, we run the model in `memory-map` mode. This pre-processes the data without loading all in CPU memory at once, preventing the system from running out of memory.



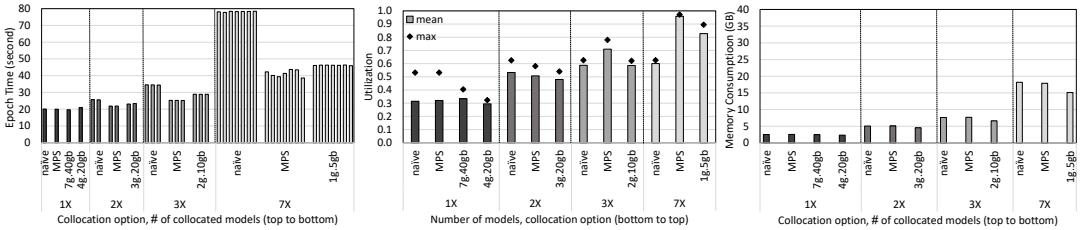
(A) Epoch time (B) GPU utilisation (C) Memory footprint

FIGURE 4.3: Small: ResNet26 + CIFAR-10 (batch size = 32).



(A) Epoch time (B) GPU utilisation (C) Memory footprint

FIGURE 4.4: Small: ResNet26 + CIFAR-10 (batch size = 128).



(A) Epoch time (B) GPU utilisation (C) Memory footprint

FIGURE 4.5: Small: EfficientNet_s + CIFAR-10 (batch size = 128).

4.3.4 Experiments

We devise experiments with varying dataset sizes and models to assess the performance of collocating deep learning training under different loads.

We orchestrate the execution of the workloads via a benchmarking framework [79] built on the machine learning platform MLflow [80]. This allows running the experiments in a systematic and controlled way. We create workloads containing either a single or collocated model(s). Workloads are automatically executed in sequence by the framework and in controlled conda environments. Any setup requirements for MIG partitioning and the MPS daemon are performed before training. The GPUs are cleaned after every workload, removing any previous configurations such as MIG partitions. The framework comes with a set of listeners that automatically take measurements while the models are training. In this case, every model has a DCGMI and NVIDIA-SMI listener bundled with it to collect the metrics of interest (Section 4.3.2). We source the vision models from the TIMM library [81], the recommender model

from Facebook Research [75], and are using the latest version of PyTorch as of the start of our experiments (2.0) [49].

Uniform collocation. We default to a batch size of 128 for most of our experiments. We additionally train the *ResNet* models on a batch size of 32 to observe the impact of the batch size. Based on our preliminary experiments with some of the large, hence longer-running, models and dataset, we observed that the behaviour of time-per-epoch, GPU utilisation, memory consumption, etc. (Section 4.3.2) does not drastically change from the second epoch on. As the first epoch of the vision models tends to be slower than subsequent epochs, we let the vision models warm up for one epoch and report the measurements from the second epoch, providing representative information.

We determine several model training collocation options following the available MIG profiles (Section 4.2.1):

- One model: MIG 7g.40gb or 4g.20gb
- Two models: MIG 3g.20gb
- Three models: MIG 2g.10gb
- Seven models: MIG 1g.5gb

These are based on the maximum amount of instances that can be allocated at the same time for a given MIG profile. We also create the corresponding collocation experiment for the non-MIG collocation methods (Section 4.2.1). For example, for the `1g.5gb` profile, there can be a maximum of 7 MIG instances present on the GPU at the same time allowing for 7 models to be trained in parallel, each model on a separate instance. We contrast this setup with training 7 models in parallel using naïve collocation and MPS. These form our initial set of experiments collocating uniform training runs.

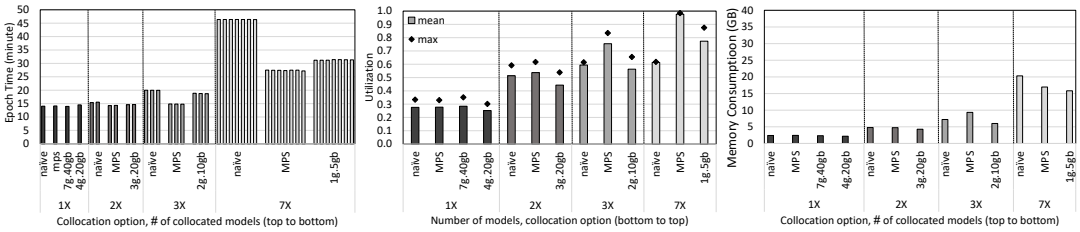
The `7g.40gb` and `4g.20gb` profiles do not allow for any parallel instances of the same size since there is not enough compute or memory left for such an instance.

Finally, the experiments with the full MIG profile, `7g.40gb`, and MPS without collocation have the purpose of exploring the performance impact of enabling the respective technologies for a GPU, in comparison to a case where they are disabled.

Mixed collocation. We also create collocation experiments with mixed sets of ResNet models and datasets, based on the performance of the models in the prior experiments. In order to satisfy the memory constraints, we train the small and medium models with batch size 128 and the large models with batch size 32. We limit the amount of collocated models to four in order to provide a clear scope that should cover most use-cases. Table 4.3 lists all the mixed ResNet training collocation options we experiment with in the corresponding section (Section 4.4.6).

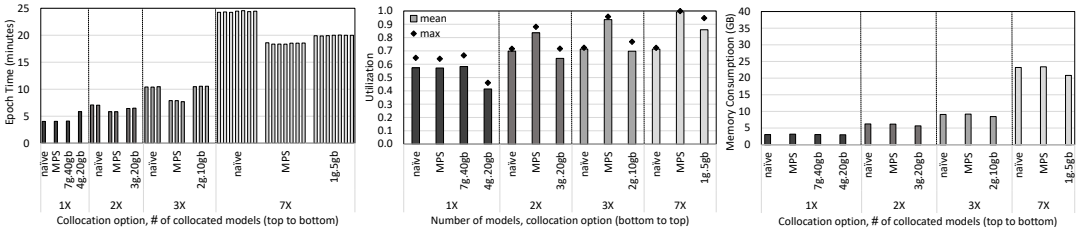
Similarly, we collocate the recommender model with a single large ResNet model with batch size 32. Due to the size of the recommender model, we limit our experiment to 2-way collocation. As the memory requirements of the recommender exceed that of even the 20GB MIG partitions, we issue a `7g.40gb` MIG GPU instance and split this instance into two compute instances (see Section 4.2.1). This

yields a `3c.7g.40gb` and a `4c.7g.40gb` compute instance to train the models on, where the 40GB memory is shared between the collocated training runs.



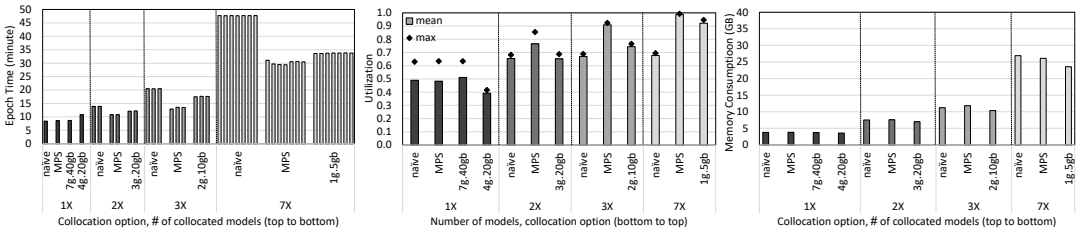
(A) Epoch time (B) GPU utilisation (C) Memory footprint

FIGURE 4.6: Medium: ResNet50 + ImageNet64 (batch size = 32).



(A) Epoch time (B) GPU utilisation (C) Memory footprint

FIGURE 4.7: Medium: ResNet50 + ImageNet64 (batch size = 128).



(A) Epoch time (B) GPU utilisation (C) Memory footprint

FIGURE 4.8: Medium: EfficientNet_s + ImageNet64 (batch size = 128).

4.4 Results

Figures 4.3-4.10 illustrate the results for our uniform collocation experiments. Each figure shows the results of a particular model and dataset combination (as listed in Table 4.1). Bars that are grouped together form one collocated workload with models trained in parallel. The different degrees of collocation are separated by dotted vertical lines. The four non-collocated cases, which do not run any models in parallel, are the first four bars and form our baselines. We omit the figures for two of the large model and dataset combinations: (1) Resnet152 with batch size 128 and ImageNet and (2) EfficientNet with ImageNet. These two cases were too large to allow for any collocation; they ran out of memory on the GPU for all collocation mechanisms even when training just two models in parallel. Sections 4.4.1-4.4.3 present the results each focusing on a metric we collect (Section 4.3.2) for Figures 4.3-4.10.

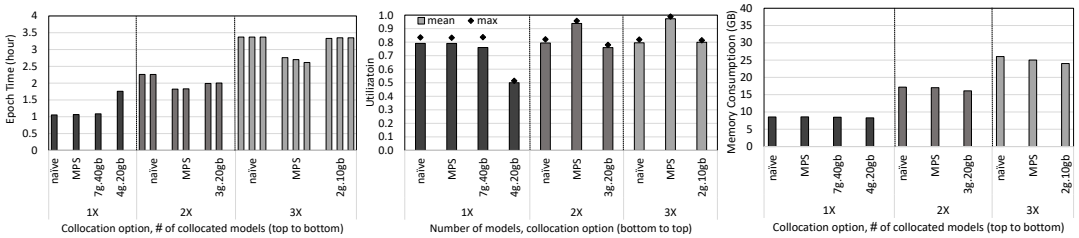
After presenting the results for the uniform collocation, Section 4.4.6 describes our findings on the collocation runs using a mixed set of vision models, and Section 4.4.7

presents the effectiveness of collocation when training a recommender and a vision model in a collocated fashion.

4.4.1 Time per Epoch

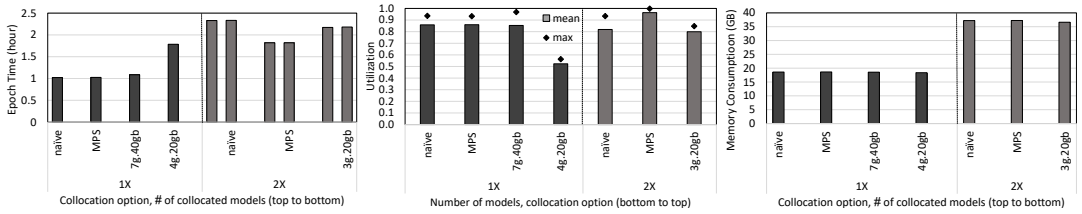
As mentioned in Section 4.3.2, time per epoch is our main performance metric when comparing the effectiveness of different collocation methods. The rest of the metrics are used to explain certain trends in the time per epoch results.

Starting with our baselines, we need to verify whether enabling the MPS daemon or MIG partitions introduces any visible overheads. Looking at the first four bars of Figures 4.3a-4.10a, reveals that there is a little variation between the first three non-collocated workloads: `naïve`, `mps`, and `7g.40gb`. This indicates that there is negligible overhead of having MPS or enabling MIG over the naïve case. On the other hand, we see the impact of having fewer resources available on the `4g.20gb` MIG instance as the workloads get larger. In Figures 4.7a-4.10a, the single training run exhibits a larger time per epoch on the `4g.20gb` instance compared to the other non-collocated runs.



(A) Epoch time (B) GPU utilisation (C) Memory footprint

FIGURE 4.9: Large: ResNet152 + ImageNet (batch size = 32).



(A) Epoch time (B) GPU utilisation (C) Memory footprint

FIGURE 4.10: Large: CaiT + ImageNet (batch size = 128).

Going over to the collocated runs, comparing the different collocation mechanisms on Figures 4.3a-4.10a reveals that MIG-based collocation performs better as the degree of parallelism increases (especially to 7). MPS reveals itself as a clear winner, offering the best performance across the board. In contrast, naïve collocation is the least effective.

For the small ResNet models and 7-way collocation, the benefits of collocation become very visible. With ResNet's embarrassingly parallel nature and the larger batch size allowing even more parallelism, they manage a high utilisation of the GPU compute resources without overloading the GPU, as demonstrated by Figure 4.4b. The medium ResNet models reflect the same pattern, though start hitting compute resource boundaries under 7-way collocation, as seen in Figure 4.7b. As a result, collocation provides considerable benefits for the small and medium cases with MIG

and especially with MPS. We attribute the superior performance of MPS to its more flexible resource management allowing more effective parallel collocated runs (as Section 4.4.6 and Section 4.4.7 also show) and the lower compute resources that are available to MIG (Section 4.2.1).

As expected, collocation impacts the time it takes to train the individual models due to interference across the collocated runs. Aside from 2-way collocation with our smallest workload (Figure 4.3), training more models in parallel increases the time to finish training a model. Additionally, as the degree of collocation increases, so does the total time to train the models. On the other hand, multiple models finish training simultaneously, increasing training throughput. For example, except for the large workloads, 2-way collocation delivers two models in roughly the same time as no-collocation delivers one model. 3-way collocation with MPS and MIG leads to a 15-35%, 50-160%, and 45-110% increase in time per epoch compared to the non-collocated case for ResNet with batch size 32 (Figures 4.3a & 4.6a), ResNet with batch size 128 (Figures 4.4a & 4.7a), and EfficientNet (Figures 4.5a & 4.8a), respectively, while delivering three model training runs instead of one. 7-way collocation with MPS and MIG only increases the runtime by 40-80% for our smallest workload (Figure 4.3) while delivering 7 models in parallel. These results clearly show that collocation is valuable when a single training run is not large enough for the available GPU compute and memory resources; e.g., the *small* and *medium* cases.

However, the picture shifts considerably with the large workloads (Figures 4.9 & 4.10). We no longer see improvements for all of the collocated runs. MPS remains strong and is the only form of collocation that remains beneficial even at 3-way collocation. Under naïve collocation, one epoch of training takes roughly as long as training the models in sequence without collocation. MIG fairs little better under 2-way collocation, but is not advantageous. Additionally, 7-way collocation becomes impossible due to memory constraints.

Take-away. *Collocation provides a significant increase in throughput, especially for smaller models. MPS collocation consistently beats the other forms of collocation, with MIG just slightly behind for workloads with smaller models.*

4.4.2 GPU Utilisation

Observing GPU utilisation in Figures 4.3b-4.10b helps us understand when collocation provides benefits and when it does not. In general, the lower the GPU utilisation for the non-collocated scenarios, the higher the benefits of collocation.

As explained in Section 4.3.2, the utilisation % reported for MIG is with respect to the aggregate available compute resources (98 SMs) to MIG. A small amount of compute resources of the GPU is lost to MIG overhead. Results for naïve and MPS depict the utilisation of the whole GPU (108 SMs).

Starting with the small workloads (ResNet26 with batch size 32 in Figure 4.3b), we can see that non-collocated runs do not fully utilise the streaming multiprocessors. Collocation massively increases the utilisation, allowing for more of the GPU to be useful when training 2, 3, or 7 models at the same time. Both SM activity and occupation do not meet the saturation point for this small use case, explaining the excellent collocation performance as discussed in Section 4.4.1.

When interpreting the throughput benefits of collocation, looking at the GPU utilisation of the individual runs help. More specifically. when the GPU utilisation for the runs with no collocation is less than 50%, the 2-way collocation easily doubles the throughput under MPS. For 3-way or higher degrees of collocation, the throughput improvements start being sub-linear as a result of higher GPU utilisation of the collocated runs, pressure on the GPU internals, and coordination efforts for the collocated runs.

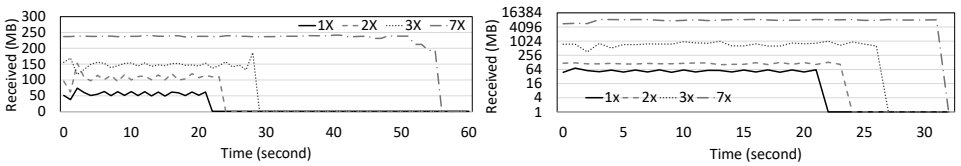
Most of the patterns seen in the small case are present in the medium ResNet case (Figure 4.6b) as well. The utilisation is notably higher, with MPS 7-way collocation (`1g.5gb`) approaching maximal SM activity, though occupation does not max out. The results for large ResNets (Figure 4.9b) notably deviate from these two experiments. The non-collocated runs already have notably high utilisation (∼80%) and the 2-way collocated runs get close to the limit. Going one step further, the 3-way collocated runs over-saturate the compute side of the GPU leading to diminishing returns in terms of the throughput achieved under collocation (Section 4.4.1).

As we increase the batch size from 32 to 128 for ResNet, GPU utilisation jumps by up to 75% and 110% in, respectively, the small (Figure 4.4b) and medium (Figure 4.7b) cases. The GPU utilisation under collocation with MPS is especially high, reaching almost 100% SM activity, with MIG reaching similar numbers on 7-way collocation in the medium case.

As in the case of time per epoch results, the GPU utilisation of EfficientNet (Figures 4.5b & 4.8b) is similar to the results of ResNet with batch size 32. There is a big jump in utilisation from non-collocation to collocation. MIG and MPS collocation feature high utilisation, showing very similar numbers for 7-way collocation, with MPS having a small utilisation lead on MIG for 2- and 3-way collocation.

Finally, for CaiT (Figure 4.10b), there is little variety in the GPU utilisation across different cases. Even though the utilisation numbers are very similar, MPS still manages to provide a throughput benefit in this case over training the models in series, unlike naïve or MIG collocation.

Take-away. *MPS provides the highest GPU utilisation regardless of workload, thanks to the more flexible resource sharing it provides. Also, throughput under MPS fares better despite heavy GPU utilisation.*



(A) Naïve (B) Multi-Process Service

FIGURE 4.11: Data traffic from CPU to GPU during the second epoch of ResNet26 + CIFAR-10 (batch size 32) training. Note that the y-axis is different between the graphs.

4.4.3 Memory Footprint

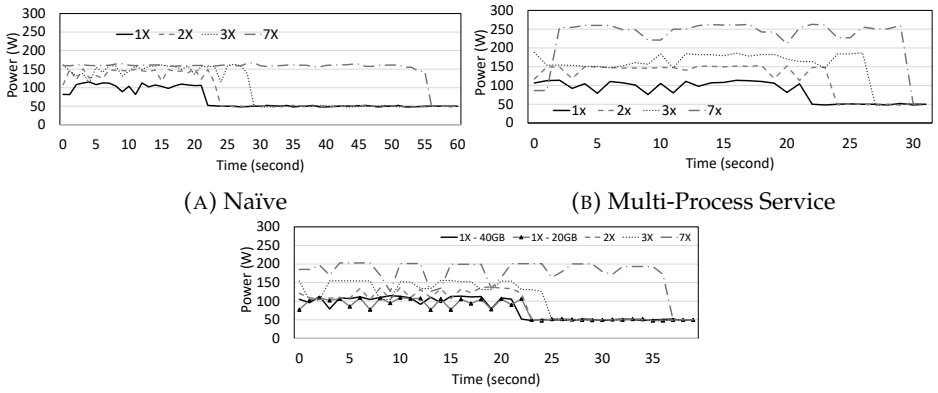
The GPU memory footprint of the models while training is crucial as it is the main determinant for whether models can be trained in a collocated fashion. As we have

seen in the previous two sections, when there is high utilisation of compute resources on a GPU, the collocated runs can still make forward progress, even though the collocation might not be beneficial in terms of throughput. On the other hand, when there is not enough memory available for the aggregate memory footprint of the collocated training runs, then these models run out of memory when assigned to the GPU.

Figures 4.3c-4.10c report the aggregate memory footprint on the GPU for different collocation methods for each workload. They demonstrate that the increase in memory footprint with collocation is proportional to the degree of collocation. Notably, MIG collocation shows slightly smaller memory footprints than the two other options. In general, 2-way, 3-way, and 7-way collocation results in roughly two, three, and seven times the memory footprint of no collocation, respectively. This is an expected result as the models are not sharing data across collocated runs in these experiments.

The memory footprint for MIG prompted us to delve deeper into PyTorch's memory allocation. The reduced memory allocation for MIG shows up in both nvidia-smi readings and PyTorch's advanced memory statistics. However, PyTorch's basic memory allocation and reservation trackers, which count space allocated for the tensors and reserved by the allocator, respectively, do not show any difference across the collocation methods. Training the models on a separate GPU with less available GPU memory displays the same pattern of slightly reduced memory footprint, confirming that this is not a symptom unique to MIG, but is rather due to the memory available to PyTorch. PyTorch adjusts the memory footprint depending on the total available memory, which is less in the case of non-`7g.40gb` MIG instances compared to whole GPU memory available under MPS and naïve. Switching the memory allocator back-end from PyTorch's native implementation to CUDA's built-in asynchronous allocator removes the differences in the memory footprint of different collocation methods. However, we do not recommend this switch as it slows down the training process.

Take-away. *Memory requirements for uniformly collocated models can be effectively estimated by multiplying the memory required by a single model. PyTorch allocates slightly less memory under MIG instances that have a fraction of the whole memory than under other collocation methods.*



(A) Naïve (B) Multi-Process Service

(C) Multi-Instance GPU

FIGURE 4.12: GPU power usage during the second epoch of ResNet26 + CIFAR-10 (batch size 32) training.

4.4.4 Interconnect Traffic

Figure 4.11 reports the number of bytes received by the GPU under naïve and MPS collocation over time during the second epoch of small ResNet training with batch size 32. We pick this case as it benefits greatly from collocation and can highlight the differences across the collocation scenarios more easily. MIG is omitted here due to `dcgm` not providing the readings for this metric under MIG as a result of the GPU being split into multiple instances.

For lower degrees of collocation, naïve collocation leads to a linear increase in data transferred over PCIe from CPU to GPU with respect to degree of collocation. On the other hand, for the 7X case, there is less work being done per unit of time for each training run leading to sub-linear PCIe traffic. This is likely caused by the throughput benefits of collocation taking a huge hit under naïve collocation, as shown in Figure 4.3.

In contrast, MPS exhibits a super-linear increase in PCIe utilisation when collocating models. In addition to the data transfers for the collocated runs, MPS increases the kernel launch processes since it is able to eliminate false dependencies and share the GPU resources more effectively across the collocated kernels (Section 4.2.1).

Overall, across all evaluated scenarios, we did not observe bottlenecks due to PCIe connectivity. This is crucial for our comparison of the different collocation scenarios, since scenarios with critical I/O bottlenecks may lead to misleading results for this type of study.

Take-away. *MPS heavily increases communication between the CPU and the GPU to facilitate faster collocated model training.*

TABLE 4.2: Total energy consumption for GPU to complete the second epoch of ResNet26 + CIFAR-10 (batch size 32) training.

collocation	small case	Energy (KJ)
1X	naïve	4.465
	MPS	4.417
	4g.20gb	4.498
	7g.40gb	4.417
2X	naïve	6.367
	MPS	6.210
	3g.20gb	5.547
3X	naïve	8.477
	MPS	7.716
	2g.10gb	7.058
7X	naïve	18.984
	MPS	15.047
	1g.5gb	13.159

4.4.5 Energy Consumption

Finally, we look at the power usage and the energy consumption of the GPU for small ResNet training, similar to Section 4.4.4. Figure 4.12 show that GPU power consumption is highly correlated with utilisation since the collocation scenarios that

achieve higher utilisation in Figure 4.3b also result in higher GPU power usage. MIG exhibits significantly lower wattage under 7-way collocation than MPS while training slightly slower. The benefits of this can be seen in Table 4.2, which reports the total GPU energy consumption of the second epoch of the model training. While requiring higher power usage per unit of time, MPS spends less energy compared to naïve collocation thanks to finishing training faster. While not as fast as MPS, MIG in general exhibits the lowest GPU energy footprint.

Take-away. *GPU wattage is heavily correlated with GPU utilisation. The fastest way for training models might not be the most energy efficient on GPUs, since MIG exhibits the lowest energy footprint among all the collocation methods.*

TABLE 4.3: Mixed Vision Workloads

	Small (S)	Medium (M)	Large (L)
Model	ResNet26	ResNet50	ResNet152
Dataset	CIFAR-10	ImageNet64	ImageNet
Batch size	128	128	32
Instance	1g.5gb	2g.10gb	4g.20gb
S+M	1x	1x	-
S+S+M	2x	1x	-
S+S+M+M	2x	2x	-
S+S+S+M	3x	1x	-
S+M+M+M	1x	3x	-
S+L	1x	-	1x
M+L	-	1x	1x

4.4.6 Mixed Vision Workloads

The results presented so far focused on homogeneous collocation scenarios. Such cases can be extremely useful in practice when a data scientist is performing hyperparameter tuning to come up with the ideal set of parameters for a model repeatedly running the same model with a different set of parameters. On the other hand, there is also value in investigating non-homogeneous collocation scenarios to observe what happens when individual training runs stress the GPU unequally.

Based on how models of different sizes behaved in our previous experiments, we select combinations of small, medium, and large ResNet models with corresponding dataset sizes to collocate for the heterogeneous runs (as listed in Table 4.3). We opted to keep a static MIG configuration while testing heterogeneous collocation since in a real-world scenario, e.g., in a data centre, the MIG partitions would already be set and re-partitioning after each training run could be impractical.

Figure 4.13 details the total execution time for training the collocated models using the different collocation methods in comparison to training them back to back, *serial*, without collocation. We see that the benefits of collocation vary heavily across workloads. For small workloads such as "S+M" and "S+S+M", naïve and MPS collocation provide sizeable benefits by training the small model without impacting the medium one. In general, the flexibility of both naïve collocation and MPS is a great advantage here over MIG.

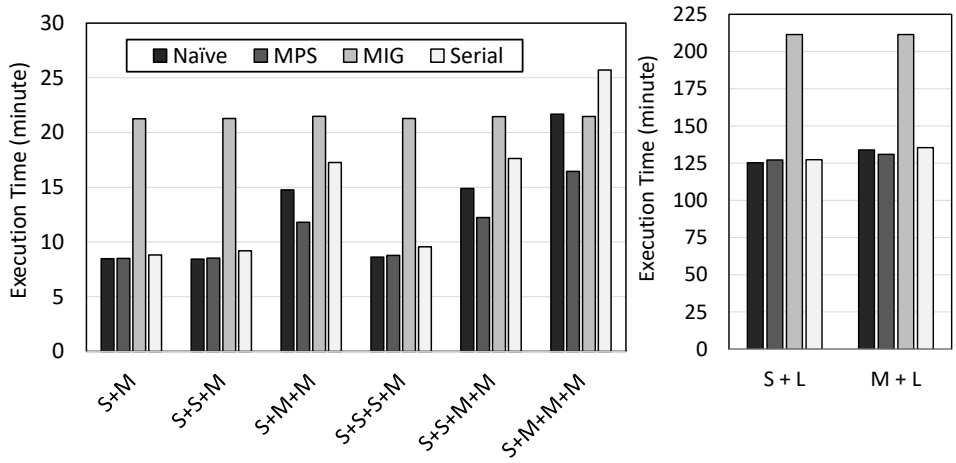


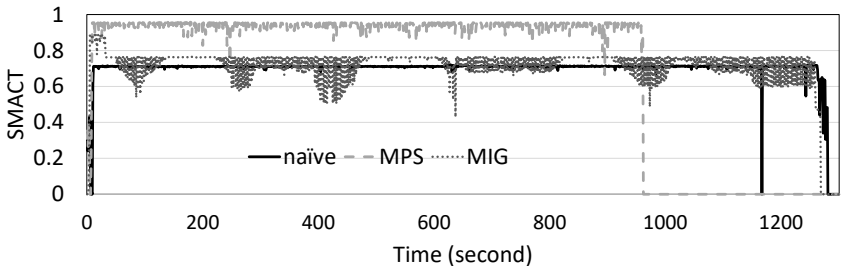
FIGURE 4.13: Total time for training mixed vision workloads with (naïve, MPS, MIG) & without (serial) collocation for two epochs. Workload configurations can be found in Table 4.3.

MIG performs significantly worse for these kinds of workloads as the resources available to the medium model are limited even after the small model finishes. On the other hand, when increasing the workload by including more medium-sized models, e.g. "S+M+M+M", this disadvantage of MIG diminishes, allowing it to slightly surpass naïve collocation. This is in line with conclusions in Section 4.4.1. Note that there is little difference in the execution time achieved by MIG in each of the graphs in Figure 4.13. MIG isolates the collocated models, and the resources available to these medium models are identical under MIG. Thus, with MIG, the total execution time boils down to the execution time of the slowest model training regardless of the mix of the collocation.

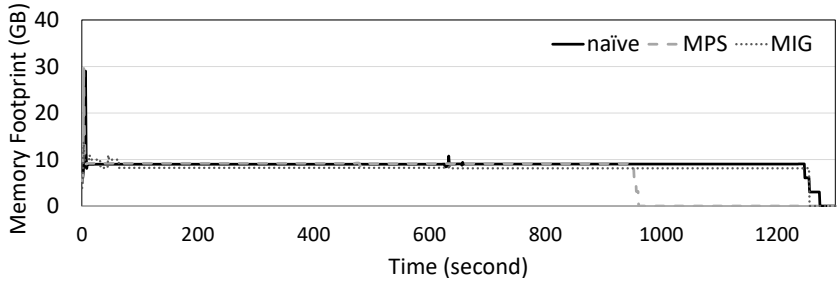
As Section 4.4.2 discussed, the large model runs utilise more of the GPU on their own. "S+L" and "M+L" runs corroborate this earlier finding as collocation offers reduced benefit in these cases. MIG performs significantly worse than the other options as the resource availability of the large model is locked even after the small and medium models finish their execution.

Figure 4.14 dives deeper into the "S+M+M+M" workload to observe how the GPU utilisation and memory footprint changes over time during collocated runs with naïve, MPS, and MIG collocation. We pick this mix as it is the one that utilises MIG instances the best. The GPU utilisation under MIG gets lowered after the small model finishes since MIG is unable to fill up the corresponding instance with more work. On the other hand, naïve and MPS are able to keep similar GPU utilisation throughout. In contrast, the memory footprint follows a similar trend for all collocation strategies. It is higher in the beginning as all four models are training. The values then drop off quickly once the small model finishes training.

Take-away. *MPS collocation provides significant benefits when training a mix of models. MIG collocation is unsuitable when mixing different models as resources can not be reallocated once some of the models finish training.*



(A) Streaming Multiprocessor Activity (SMACT)



(B) Memory footprint

FIGURE 4.14: GPU utilisation and memory footprint over time for S+M+M+M from Figure 4.13.

4.4.7 Mixed Recommender and Vision Workloads

Table 4.4 shows the results for collocating a recommender model with a vision model. As the recommender model takes much longer to train, we measure its training time by blocks instead. One training block contains 102400 training iterations and a validation pass. The ResNet training time is measured per epoch as in previous experiments. We treat the first training block and epoch as warm-up to ensure correct measurements. We run the MIG-based collocation on compute instances that share memory as the recommender model does not fit into the memory of smaller GPU instances.

Adding a memory-heavy model such as the recommender greatly promotes collocation. While training time does increase when collocating these models, it only goes up between 4%-14% and 4%-33% for the recommender and ResNet, respectively. MPS performs especially well, training both models with just a 4% increase in training time. Interestingly, the availability of extra memory under a shared memory MIG configuration benefits the training of ResNet, noting improved performance

TABLE 4.4: Mixed collocation with Recommender model. Recommender time is for one training block (102400 iterations) plus validation. ResNet time is for one epoch. The reported increase in time (%) is relative to the two no-collocation runs.

Workload	Recom. Time (h)	ResNet Time (h)	SMACT	Memory (GB)
Recommender	5.36	-	5%	29.14
ResNet152	-	1.05	82%	8.47
Naïve	6.09 (+14%)	1.11 (+5%)	81%	37.75
MPS	5.57 (+4%)	1.10 (+4%)	81%	37.62
MIG (shared)	5.60 (+5%)	1.40 (+33%)	39%	37.86

over `4g.20gb` in Figure 4.9a even though the model is collocated with the recommender.

As before, memory consumption roughly corresponds to the sum of both models. SM activity, however, does not increase on collocation. This suggests that the slowdown under collocation is due to another resource contention. Under MIG, only part of the compute power of the GPU can be assigned to ResNet, even though the recommender requires little.

Take-away. *Collocation of models that stress different parts of the GPU may greatly increase throughput in exchange for a minimal increase in training time.*

4.5 Guidelines & Challenges

Based on the results of our experiments in Section 4.4, we now provide some guidelines for deep learning training collocation in Section 4.5.1, and highlight the challenges faced when doing performance analysis in this domain in Section 4.5.2.

4.5.1 Collocation Guidelines

Workload collocation is highly beneficial when the aggregate compute and memory needs of the collocated deep learning training runs fit the GPU. In Section 4.4, we observe significant throughput benefits (up to four times) for small compute-intensive workload setups despite the increase in the epoch time of individual training runs. Medium-sized compute-intensive workloads also exhibit similar throughput benefits, though less pronounced. Similarly, collocating compute- and memory-intensive training together leads to a more effective use of the hardware resources without significantly hindering training time.

Collocation gives diminishing returns when the SM activity of an individual training run is already close to 100%. For example, even without collocation, the training of CaiT hits SMACT numbers of 90% and up. This shows that the GPU's compute resources are utilised almost completely. Hence, the large workload scenarios, like CaiT, do not benefit as much from collocation. A user can make an educated guess for the most effective degree of collocation (no collocation, 2-way, 3-way, etc.) based on the SMACT values of an individual run.

The aggregate memory footprint of the collocated runs can simply be estimated by the sum of the memory footprints of the individual runs and cannot exceed the available memory on the GPU. As a result, for large workload scenarios, we either cannot collocate any training runs (e.g., ResNet152 with batch size 128) or cannot reach beyond 2- or 3-way collocation (e.g., Figures 4.9 & 4.10). A user can determine whether a set of runs can be collocated effectively a priori based on the known or expected memory footprints of individual runs.

MPS achieves better performance across the board thanks to its flexible distribution of hardware resources among the collocated runs. Hence, for setups where just a single user is submitting training jobs, MPS-based collocation will always be the better option over naïve and MIG.

MIG is able to support collocation effectively when a strict separation is required among the runs thanks to its rigid partitioning even though this partitioning leads

to sub-optimal performance compared to MPS. When there are multiple users submitting training jobs or when even a single user requires non-interfering runs due to e.g. privacy concerns, MIG is the only option for collocation. If the workload is known a priori, the ideal set of MIG instances can be created accordingly. This way, MIG-based collocation can still be beneficial over training serially even though it comes at a slight cost in performance compared to MPS.

MIG exhibits higher energy efficiency on GPUs when the instances are configured well for the workload. Whenever MIG's performance is close to MPS in terms of time-to-train, its energy consumption for end-to-end training is consistently lower than that of MPS. Thus, if the workload is known and the ideal set of MIG instances is set, then, MIG is the more energy-efficient choice.

4.5.2 Challenges

Benchmarking in a rapidly evolving field like deep learning, has its challenges, which we encountered in our study.

Maturity of the toolset for MIG. DCGM is capable of tracking metrics per MIG instance as mentioned in Section 4.3.2. However, earlier in our study, it did not reliably report the metrics for the `4g.20gb` instance. This issue has since been resolved, allowing us to include `4g.20gb` in Section 4.4. There are, however, other reporting anomalies under MIG with DCGM. For example, metrics that track data movement across PCIe, which connects the CPU and the GPU, do not report anything when MIG instances are used on A100 GPUs.

PyTorch improvements. Our initial experiments for this study were conducted under CUDA 11.6 and PyTorch 1.13. This configuration was unable to efficiently utilise MPS. The introduction of PyTorch 2.0 has fixed these issues and has significantly increased the effectiveness of MPS as reported in our results.

4.6 Conclusion

In this chapter, we did a performance characterisation on a modern GPU device that has support for multiple means of GPU collocation: naïve, MPS, and MIG. Our results demonstrate that GPU collocation is highly beneficial for small- and medium-sized workloads that cannot fully saturate the whole GPU. Although per-model training is overall slower, more work can be done per unit of time by executing workloads in parallel, which utilises the GPU resources more effectively and increases the training throughput. MIG notably requires a rigid setup while providing full isolation across its instances. If the workload across the instances are imbalanced, runs that finish early will leave some instances idle, unless there is other work that could be allocated over those instances. Naïve collocation and MPS, on the other hand, can utilise the resources released by the finished work, increasing the training performance of models that require more time to train. In general, MPS provides the best collocation performance, if not the most energy efficient.

In this work, we limited our focus to training on a single GPU, since NVIDIA doesn't allow multi-GPU training with MIG. In a data centre, many workloads can be collocated not only on the same GPU but also on the same server. Therefore, studying the impact of collocation while running other workloads on other GPUs on the same

device would be interesting future work. Furthermore, considering the results with the recommender model, further investigations of the shared memory instances of MIG would be worthwhile.

Chapter 5

Data Sharing via TensorSocket

5.1 Introduction

The process of training a deep learning (DL) model is computationally expensive, mandating the use of powerful accelerators such as GPUs to match the computational needs. However, while the core of the training process can naturally be accelerated this way for many DL models, some training pipelines feature computationally expensive data pre-processing operations such as augmentation and decoding [82]. Such operations often cause bottlenecks on the host-, or input-side of the training pipeline, where the dominating processing unit is still the CPU [83, 84].

Compute offerings of cloud providers are popular for addressing the computational needs of deep learning training thanks to their on-demand availability. On the other hand, the range of CPU-to-GPU configurations is rather limited, as shown in Figure 5.1. Furthermore, an instance with a high vCPU to GPU ratio can cost up to 16 times as much as an instance with minimal vCPU count with the same GPU [85]. This high trade-off for the need for more CPU availability, combined with the wide range of DL workloads and their differing computational requirements, lead to several bottlenecks [86, 87, 88]. Specifically, DL training processes that are bottlenecked by their input processing render expensive high-performance accelerators underutilised [19, 57]. In turn, this wastes both CPU and GPU resources. Underutilisation of cloud resources is financially wasteful for everyone as compute that has been paid for is not used effectively. Furthermore, it creates an unsustainable carbon footprint in order to address the demand for AI [15, 14, 4, 89, 90].

In practice, it is common to train several models to accomplish a task. The feasibility of DL models heavily depends on finding a model architecture (neural architecture search) or set of parameters (hyper-parameter tuning) that responds well to the data. This results in model training scenarios that exhibit shared tasks, especially in their input pipelines. There has been recent work that has proposed ways to leverage such shared tasks [87, 91, 92, 93] and effectively demonstrated the premise of sharing. On the other hand, these works either focus on the cloud scale, paying less attention to the finer-grained cooperation across training processes on the same server, or put a heavy burden on the CPU resources, essentially locking the CPU into being a data feeder for a hardware accelerator instead of facilitating efficient resource utilisation.

In this chapter, we draw inspiration from these prior works on data sharing across DL training tasks in addition to the work on database systems that leverage the shared work done across concurrent database requests [94, 95, 96, 97]. Our goal is

to increase opportunities for work sharing and collocation across DL training jobs while minimising the hardware resource requirements for such jobs. Rather than viewing these jobs as big monolithic isolated tasks that have to get scheduled exclusively on some CPU and GPU resources, we propose TENSORSOCKET, a novel data loader that is shared across models being trained on the same dataset. TENSORSOCKET turns the *competition* for data and hardware resources into a *cooperation* allowing for more effective workload collocation across concurrent training tasks. As a result, it alleviates resource underutilisation and the aggregate costs of DL model training.

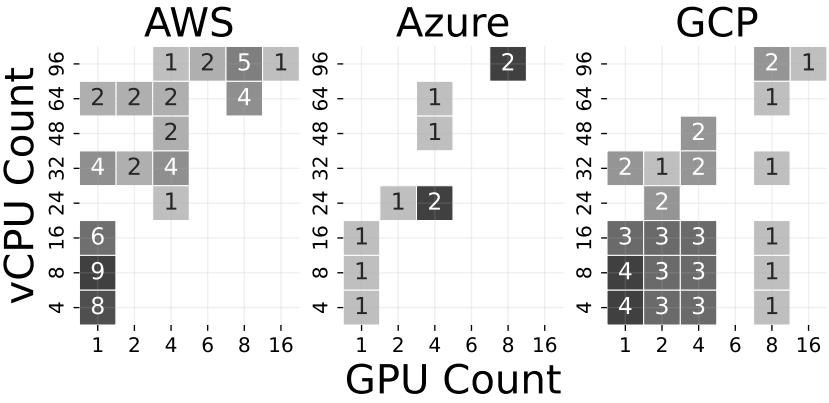
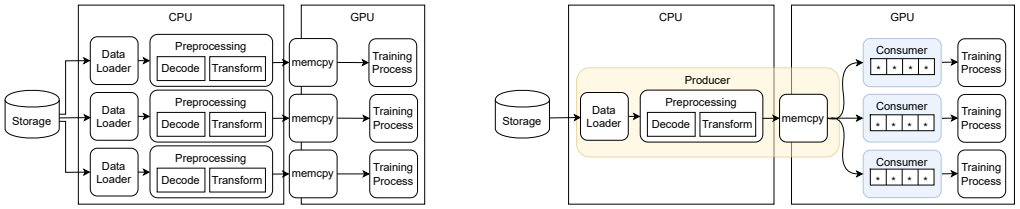


FIGURE 5.1: Cloud instances by vCPU to GPU ratio offered by Amazon Web Services (AWS) [85], Microsoft Azure [98], and Google Cloud Platform (GCP) [99]. The colour scale denotes the number of instances offered with the specified vCPU-GPU pairing.

Our contributions are as follows:

- We present the design and implementation of TENSORSOCKET and demonstrate how easily it can be adopted as an alternative data loader in DL training pipelines implemented in PyTorch [49] in a plug-and-play manner.
- We highlight and evaluate the benefits of TENSORSOCKET across a variety of training scenarios (image analysis, audio classification, generative AI) and hardware setups (powerful on-premises hardware, different cloud offerings). Our evaluation shows that TENSORSOCKET enables scenarios that are infeasible without data sharing, doubles training throughput, and, most importantly, when utilising cloud instances, can halve the cloud costs by reducing the CPU needs by up to four times.
- We compare TENSORSOCKET to the state-of-the-art techniques for shared data loading, CoorDL [87] and Joader [93], and demonstrate that TENSORSOCKET either outperforms or matches their training throughput while requiring less CPU resources and lower deployment effort.

The rest of this chapter is structured as follows. Firstly, Section 5.2 gives an overview of the data loading step in DL model training, before Section 5.3 presents TENSORSOCKET and motivates concrete use case scenarios for it. Then, Section 5.4 demonstrates the multi-dimensional benefits of TENSORSOCKET over a variety of training pipelines and hardware setups, and Section 5.5 discusses further applicability and potential limitations of TENSORSOCKET. Finally, Section 5.6 surveys related work,



(A) Conventional DL input pipeline. (B) Shared input pipeline with TensorSocket.

FIGURE 5.2: Collocated DL model training (a) without sharing the data loader and (b) with sharing the data loader using TensorSocket. The arrows denote the flow of data from storage to training processes. The shared data loading process (producer) is shown in yellow, and the training processes (workers) in blue.

and Section 5.7 concludes the chapter.'

5.2 Data Loading in Deep Learning

Characteristics and bottlenecks. Figure 5.2a shows the different training processes that deal with reading and transforming data. In general, a DL training process consists of a model configuration, a dataset partitioned for training and validation, and a training loop. The training process iterates over the full training partition of the dataset a number of times, specified as *epochs*. The *data loader* is tasked with fetching and readying the data for the model to train on. During each iteration, a *batch* of data is prepared from either disk or memory. This preparation includes fetching, decoding, and transforming the samples. Furthermore, data may be augmented in order to improve the accuracy of models trained on it and their ability to generalise to future unseen data [82, 100].

Decoding, transforming, and augmenting data are all steps that handle and modify data during training and are collectively called *data pre-processing* operations. The more extensive the pre-processing, the higher the computational overhead during training, potentially introducing input-bound bottlenecks. In some real-world training processes, these operations can amount to half of the energy costs [101].

Furthermore, even though the data used for training is in fast local storage, such as main memory or SSDs, this data commonly exceeds memory capacity. The result is that the data has to be repeatedly read from disk, swapping out the data that has already been trained on. The damage done by this swapping and OS thrashing depends on how much data can fit in memory as well as the storage back-end. This introduces I/O as a potential bottleneck on the critical path [87].

Alleviating the bottlenecks. Data loaders can be configured to alleviate problems that may arise due to inadequate host-side resources that result in the training process idling the GPU [86]. For example, pre-fetching overlaps the work done for data preparation with model training by reading and processing data prior to when it is required during training. Similarly, scaling up the number of workers contributing to reading and pre-processing the data can also help hide the input bottlenecks in training. While increasing the worker count does not speed up the pre-processing time of individual batches, it does increase the total batch throughput that can be fed to the model training. On the other hand, a high degree of pre-fetching and parallel

workers incur higher host-side CPU utilisation and memory consumption, increasing the required resource cost and potentially causing contention for resources at the host side.

If the CPU is fully utilised while the GPU is not, another option is to offload pre-processing operations to the GPU. Tools like NVIDIA DALI [102] or techniques like FusionFlow [103] and FastFlow [104] offer methods for resolving data loading bottlenecks that may arise in such scenarios. However, offloading pre-processing to the GPU reserves compute resources that could otherwise be used for training the model itself and should therefore be done with care.

Opportunities for sharing. Developing an effective DL model often requires multiple models to be trained and evaluated in quick succession over the same or similar datasets. Model selection is a common practice where model architectures and training configurations are empirically compared. Hyper-parameter tuning evaluates different hyper-parameter settings, such as learning rate, weight decay, and optimiser settings. These types of tasks are essential for landing on the best performing model [105, 106, 107, 108], as the range of different model architectures and hyper-parameters available increase with the introduction of new models.

Our goal is to propose a mechanism to alleviate the data loading bottlenecks in deep learning training that is complementary to the ones listed above. More specifically, motivated by the repetitive nature of tasks during the model search and hyper-parameter tuning for deep learning training, we would like to leverage the shared data and work required by these tasks. In our solution, TENSORSOCKET, we aim to dedicate the maximum amount of GPU resources to the training loop itself and minimise CPU resource requirements for data loading. Furthermore, by extending existing data loader implementations instead of replacing them, our solution is compatible with other optimisations that can be done for data pre-processing.

State-of-the-art sharing techniques. Motivated by these opportunities for sharing, prior work has also advocated for data sharing in DL [87, 91, 92, 93]. Here, we more specifically detail the proposals that are closest to TENSORSOCKET, which we also compare TENSORSOCKET against in Section 5.4.

CoorDL [87] is an extension to NVIDIA DALI that coordinates data pre-processing. It is designed for the cluster level and can be used to share data directly between training processes. It can distribute a batch of data to any number of training processes in the cluster. Once all training processes are done with the data, CoorDL continues to the next batch. CoorDL's focus on the cluster level and design around DALI, however, surfaces some limitations. Firstly, CoorDL is designed for model training on separate GPUs and cannot utilise leftover GPU compute power to train multiple models on a single GPU in a collocated fashion. Secondly, CoorDL performs poorly when the models that train simultaneously are not very similar, as in this case the models that are faster have to wait for the slower ones. This rigid design also prevents CoorDL from being deployed as a live service with training processes arriving at different moments. Thirdly, CoorDL requires the data loading and pre-processing pipeline to be implemented in DALI, requiring substantial extra engineering if the pipeline implementation is not already using DALI. Finally, the existing CoorDL codebase [109], and the DS-Analyzer project around it, is written in Python 3.6, which has been deprecated since 2021 by PyTorch and reached Python end-of-life in December that year.

Joader [93] is a standalone shared data loading solution that supports sharing over multiple datasets. A server is configured in which all datasets have been registered. Training clients, also known as jobs, then communicate with this server using RPC. Joader reduces CPU utilisation by having one server that does the data loading and pre-processing for multiple jobs, even if those jobs require datasets that are not identical but just overlap. In the scenario where models train on the same base dataset, this allows models to train at different speeds and still share part of their data loading. Joader achieves this flexibility across different datasets through a technique called dependent sampling. However, this dependent sampling also comes with an important drawback; it requires intersection calculations to run at every iteration, which adds a high CPU cost. Furthermore, Joader only has a proof-of-concept implementation [110], which is written in Rust making it very difficult to adapt existing deep learning training codebases to. Datasets have to be converted to the proprietary format that Joader expects and only image data with specific parameterisation is supported. There is no support for a variety of image pre-processing operations other than the pipeline that is hardcoded in Rust. Finally, data reaches the training jobs as NumPy matrices which require tensor conversion and host-to-device transfer, and batching during training is not supported, which are all detrimental to data loading and training performance.

Next, we delve deeper into TENSORSOCKET.

5.3 TENSORSOCKET

This section presents TENSORSOCKET[1], our shared data loader that capitalises on the redundancy among similar but separate data loaders of collocated training processes. We propose a solution to inefficient hardware utilisation and resource wastage by minimising redundant work and hardware resource consumption while ensuring that downstream training processes are not impacted. By detaching the data loading pipeline from each training process, we can merge several of them into a single data loading pipeline. This single data loader can expose the training data for use in each collocated training process.

5.3.1 Overview

Figure 5.2b illustrates how our shared data loader works. The figure shows an example of three collocated training processes, where, in yellow, the tasks of the *producer* are shown, and in blue, the training process with the *consumer* is shown.

At its core, our system is composed of a *producer* and a number of *consumers*. The *producer* holds a single data loading process along with some bookkeeping, while the *consumers* iterate on data sent by the producer. This producer-consumer workflow can be swapped in place of DL training framework-specific `DataLoader` objects, such as the PyTorch DataLoader.

Given that the producer is the owner of the data loading pipeline as well as responsible for data generation, it would be regressive to copy each batch of data into every collocated training process. Instead, once the producer has prepared a batch of data, the workers are all given the location of the data batch to use in their respective

[1]Code repository for TENSORSOCKET itself `https://anonymous.4open.science/r/tensorsocket-CC3E`.

processes. Every consumer has a queue that holds up to a few of these locations. This introduces some flexibility to prevent training hiccups (e.g., a training process falling behind during a batch) from interfering with the other training processes.

5.3.2 Implementation

TENSORSOCKET is a library built around PyTorch as it is the most widely used deep learning framework available. A crucial limitation common in other data sharing solutions [87, 93] discussed in Section 5.2 is that the solution itself implements the complete data loader. This limits the adoption of the implementation as it requires the user to adapt to the specific codebase, in addition to the library version dependencies, of that solution. We prevent these shortcomings by setting TENSORSOCKET up as a wrapper around the existing PyTorch data loader instead of a separate data loader itself, ensuring out-of-the-box compatibility with any PyTorch training script. While this does mean that the current implementation is PyTorch-specific, implementations of similar wrappers around the data loader's frameworks such as TensorFlow won't be prohibitive as these frameworks generally follow the same principles for their data loading. We leave the more detailed discussion on TENSORSOCKET's adoptability in other frameworks to Section 5.5.2.

Producer

TENSORSOCKET splits data loading from training. The data loading producer becomes a server that can dynamically process and serve incoming consumer clients. A TENSORSOCKET producer instance is initialised with a data loader object. It is exposed as an iterator that itself iterates over the nested data loader it is initialised with. The producer repeatedly requests the contained data loader to fetch the data from disk. It will pause iterating over the data loader whenever the consumers currently do not need extra data, notified by communication between the producer and consumers. This can also be the case when there are no consumers present as in this case there is no need for any data loading.

Consumer

Abstracting away the data loader into the producer allows the consumer to be lightweight. As TENSORSOCKET's producer and consumer directly replace the data loading batch iterator in the training script, the consumer similarly takes the form of an iterable object that fetches new data whenever available. If there is no data available the consumer halts and waits. This design makes for a minimal one-line swap in training script code (as exemplified in Figure 5.3).

Communication

The communication between the producer and consumers is done using ZeroMQ sockets. ZeroMQ is a tiny open-source library that allows for sharing atomic messages with low latency. In TENSORSOCKET's case, we use ZeroMQ sockets for communication between the producer and consumers using a PUB/SUB pattern [111]. This is a multicast pattern that is flexible and scalable, allowing one producer to connect to multiple consumers without any performance risks.

The data is shared over these sockets by the producer. Once a consumer has fetched a readied batch of data for use in training, it notifies the producer by sending an acknowledgement message back to it, allowing for the producer to continuously keep the consumers fed new data. When multiple consumers are training simultaneously, the producer will wait for an acknowledgement from all consumers before releasing a piece of data. This ensures that all consumers have iterated on a batch before it is deleted.

Depending on the dataset and models, consumers may take a long time to go through their training data batches. In order to be continuously aware of consumers, producers send and receive heartbeat messages from their consumers over a different socket. The producer will time-out consumers that it has not received a heartbeat from in a while.

Data sharing

Data sharing is at the heart of TENSORSOCKET. If the data sharing implementation is not efficient, it will become a bottleneck that would outweigh the benefits of sharing. There are two ways that data sharing can be implemented.

Some solutions share the data bytes directly with the training processes via interprocess communication [93]. This surfaces some concerns regarding sharing efficiency and data duplication. Increasing the size of the training data directly increases the size of the network messages in such a solution, potentially leading to slowdowns. Furthermore, while the data loading itself is unified, the resulting data is then duplicated for every client, spiking memory consumption and data movement costs.

In TENSORSOCKET's case, we share small packets containing pointers to the data instead of the data itself. Following our earlier design philosophy, we heavily borrow from PyTorch's existing data management. PyTorch introduces Tensor objects which are data matrices, similar to NumPy matrices, that contain all data that PyTorch runs on. While PyTorch is most commonly known as a Python library, much of the internals such as Tensors are defined in C++. We can use this to our advantage by extracting the data pointer and other necessary information. This pointer is then shared by the producer to the consumers, which in turn use this data to reconstruct the Python tensor object without any data duplication.

PyTorch as a library is heavily optimised for running with multiple threads as data workers and on multiple GPUs and machines. The tensor implementation contains methods for dealing with concurrency and distribution, including tensor rebuilding. By using this somewhat hidden PyTorch functionality we can share tensors without requiring an external implementation.

Tensors in PyTorch hold data that can be on the host system (i.e., CPU), but can also be put on the GPU. Transferring data to the GPU is a costly operation. By inheriting PyTorch tensor methods, TENSORSOCKET can reconstruct tensors on both the CPU and GPU. This means that the producer can put the data on the GPU once, after which all consumers collocated on that GPU can access it. Furthermore, we can rely on PyTorch's tensor management for our shared data. Tensors are kept in memory as long as any of the producers or consumers hold a reference to it.

```python
# train.py (without TensorSocket)
data_loader = DataLoader(dataset)
for batch_idx, (input, target) in enumerate(dataset):
    output = model(input)
    ...
```

(A) Conventional training script without TENSORSOCKET.

```python
# producer.py
data_loader = DataLoader(dataset)
producer = TensorProducer(data_loader)
for _ in range(epochs):
        for _ in producer:  # Loop over the dataset
            pass
producer.join()
```

(B) TENSORSOCKET producer script.

```python
# consumer.py (or train.py)
data_loader = SharedLoader()
for batch_idx, (input, target) in enumerate(data_loader):
    output = model(input)
    ...
```

(C) TENSORSOCKET consumer script.

FIGURE 5.3: Example TENSORSOCKET implementation requiring minimal code changes. The top listing depicts a standard DL script. The data loader is split from the main training process, creating a producer process and a consumer process.

Finally, using capabilities of frameworks such as PyTorch gives us access to fast GPU-to-GPU communication methods like NVLink while sharing the tensors. This allows TENSORSOCKET to efficiently share data even if the models train on different GPUs. Data can be loaded on one of the GPUs after which it can be directly shared to the other GPUs with direct NVLink interconnects.

Synchronisation

Considering that consumers may train different models and training is stochastic, we should expect that consumers do not process a batch in identical time. This led us to introduce a batch buffer on the consumer side. Instead of actively requesting the next batch on iteration, consumers can hold up to N batches (i.e., pointers to the tensors of batches) in their buffer. This allows for the producer to actively pre-fetch data, and for the consumers to drift at most N batches apart. Both the buffering and the pre-fetching hide the latency of various parts of the data loading pipeline. When designing this queue, we experimentally found that a buffer as small as two batches is enough to provide maximum training throughput while training similar tasks. Increasing the buffer size can be beneficial when training processes fluctuate more widely in their speed. It should be noted that increasing the buffer size does increase the GPU memory requirement of the system as more batches need to be kept on the GPU simultaneously.

While the buffering scheme described above relaxes the conditions for data sharing among the consumers, TENSORSOCKET by design targets scenarios where the consumers train on the same dataset at or around the same time. Therefore, the consumers have to be balanced in terms of their training speed even if they do not process data in identical time. Whenever a process trains too fast, the consumer iterator will automatically halt as there is no data available. This frees up resources for other consumers to make up for the difference. In this case, the GPU can be time-shared among the consumers. Modern GPUs enable this through services such as NVIDIA multi-streams or Multi-Process Service (MPS) [54]. Especially, MPS allows efficient time and spatial sharing of the GPU. The GPU sharing and inclusion of the consumer buffer allow to balance the load of the consumers automatically, resulting in higher training throughput and GPU utilisation.

Since the TENSORSOCKET producer acts as a server producing data for the consumers, we also need to account for consumers that connect at different times. Once an epoch has already started, any new consumers lack behind and have to wait for the next epoch to start training. We introduce a leniency measure called rubberbanding to provide a window for consumers to join training. If a consumer joins before 2% of the dataset has been iterated on in an epoch, the producer will halt all other consumers to let that consumer synchronize and join training. The percentage of dataset that serves as the cutoff point can be configured. We found that rubberbanding is an effective method for allowing users to spawn multiple consumers without fear of them not joining fast enough.

Usage

TENSORSOCKET is built on top of PyTorch 2 with a high focus on creating as seamless a solution as possible. Our implementation enables a drop-in replacement and otherwise offers the same functionality as a PyTorch data loader.

Figure 5.3 shows an example usage of TENSORSOCKET. The PyTorch *data loader* is isolated to a different process and wrapped in a *TensorProducer*, which is then iterated over similar to a conventional data loader. The training process itself receives the data automatically by iterating over a *SharedLoader*, which is an abstraction of a *TensorConsumer* specific to this example. The full example can be found in our accompanying code repository[2]. In general, implementing TENSORSOCKET involves copying the data loading logic of a training script to a separate producer script and adding a consumer to the training script.

5.3.3 Use Case Scenarios

Here, we go over a few use case scenarios that would benefit from TENSORSOCKET, and how our implementation of TENSORSOCKET makes these scenarios possible.

Centralised Always-Available Loading.

When exploring a dataset, it is invaluable for users to seamlessly start and stop training jobs. TENSORSOCKET allows for a high degree of flexibility and stability by abstracting away the data loading from the training job itself. Once TENSORSOCKET is running on a server, consumers can come and go as they please. The consumers

[2] https://github.com/Resource-Aware-Data-systems-RAD/data-sharing.

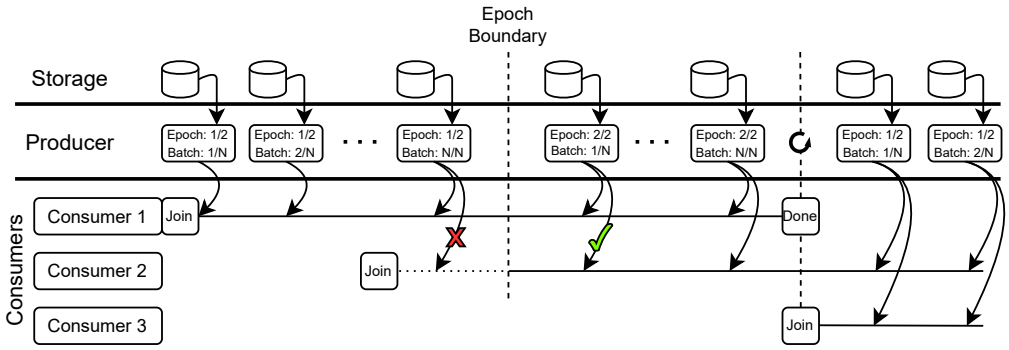


FIGURE 5.4: Illustration of how TENSORSOCKET allows the producer to run continuously and supply new consumers with batches.

ping the producer with heartbeats such that the producer knows how many training processes are active at a given time. Consumers may either join training at any point in an epoch or wait until a new epoch starts, depending on the configuration. In the latter case, the producer buffers the first few batches at the start of a new epoch to provide a short window in which new consumers are accepted. In such a case the producer will halt the other consumers in order for the new consumer to quickly iterate over the buffer.

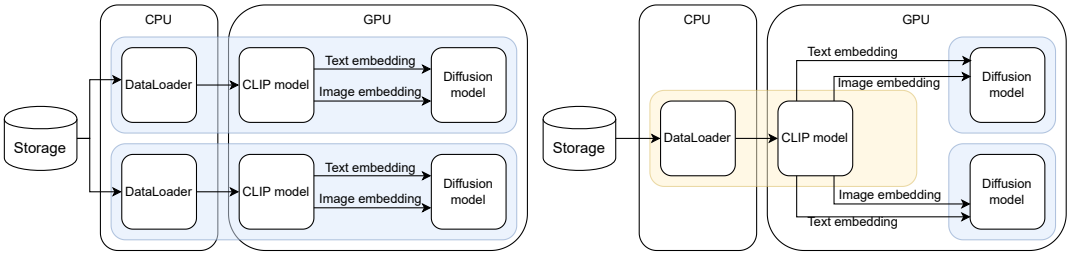
Figure 5.4 shows an example of how new jobs, represented as consumers, are handled by our shared data loading system. In the example, consumer 2 joins in too late during the first epoch, having to wait until the second epoch starts. Consumer 3 joins in at an epoch boundary and immediately starts consuming data batches.

Native Inter- and Intra-GPU Sharing.

One way of increasing the hardware utilisation for DL training is training multiple models at the same time, i.e., *workload collocation* [20, 21]. Workload collocation can improve training throughput [38, 54, 112, 113] when the hardware resource needs of the individual training processes are not large enough to utilise all the available CPU and GPU resources [37, 6] or are bottlenecked by their input pipelines [114]. This often reduces the aggregate runtime when more than one model has to be trained, even though the training time per model usually goes up due to not having exclusive access to the GPU.

TENSORSOCKET allows for sharing data on a single GPU between any number of consumers, boosting efficiency in collocation scenarios that can benefit. This additionally reduces redundant memory consumption, as the memory requirement for training processes is lowered due to not needing a data loader for each separate training process.

Our implementation also supports collocation across multiple GPUs. Data batches are seamlessly moved between GPUs when the data is needed on a different device. Thus, TENSORSOCKET is able to leverage GPU-GPU interconnects with lower latency and higher bandwidth than CPU-GPU interconnects, as also mentioned in Section 5.3.2. We evaluate this scenario in Section 5.4.2.



(A) Conventional deep learning input pipeline. (B) Shared input pipeline with TensorSocket.

FIGURE 5.5: Sharing example for generative DALL-E image generation models.

Sharing for Mixed Workloads.

Mixed workloads that train models at different speeds, for instance when model complexity differs significantly, can be difficult to optimise from a data loading perspective. For instance, prior proposals such as CoorDL [87] are not able to compensate for training speed differences and thus can only be used efficiently when models are very similar, such as in hyper-parameter tuning tasks. TensorSocket supports mixed workloads by allocating more hardware resources to heavier training processes than lighter processes. We bound the models to be within a certain amount of batches from each other (as described in Section 5.3.2). The result is that slower models are sped up and lighter models are slowed down so that all models traverse the epoch in the same amount of time. We evaluate this scenario in Section 5.4.5.

Sharing Generative Tasks Online.

In some cases, it is beneficial to move more tasks to the *producer*. For example, the training of some generative models requires pre-computed data representations in the form of embeddings for training the diffusion prior. These embeddings are usually generated before training [115], *offline*, but can be generated on the fly, *online*, via a model inference task on the GPU. Online generation offers flexibility for new data and circumvents having to use extra disk space to store the embeddings generated a priori but is more taxing on the hardware resources while training. TensorSocket can move not only the data loading operations but also the embedding generation task to its *producer* as it is essentially part of the data loading pipeline. This minimises the computational footprint on not just the CPU but also the GPU when sharing. Figure 5.5 illustrates this scenario for the DALL-E model with and without sharing, and Section 5.4.4 evaluates it.

5.4 Results

We now quantify the expected benefits of the data and work sharing enabled by TensorSocket. Our evaluation aims to answer the following questions:

- What is the impact of TensorSocket on training efficiency?
- What are the cost savings TensorSocket can provide?
- How do the benefits of TensorSocket vary across different hardware setups and machine learning pipelines?

- How does TENSORSOCKET compare to state-of-the-art data sharing solutions for model training?

To answer these questions we evaluate TENSORSOCKET in a variety of scenarios inspired by the use cases listed in Section 5.3.3. The rest of this section first describes our experimental setup for these scenarios in Section 5.4.1. Then, the results for each scenario are presented in Sections 5.4.2-5.4.5. Finally, we compare TENSORSOCKET to CoorDL and Joader in Section 5.4.6.

5.4.1 Experimental Setup

Use cases. We seek to demonstrate the value of TENSORSOCKET on a range of workloads that benefit from different degrees of shared data loading. We therefore evaluate DL models from the domains of computer vision, audio classification, and image generation. We investigate a wide range of popular computer vision models from TIMM [81] and use CLMR as our audio classification workload [116], and the image generation model is sourced from a well-known and tested PyTorch implementation of DALL-E 2 [117, 115]. The datasets chosen for our evaluation are ImageNet-1K [8], LibriSpeech [118], and Conceptual Captions (CC3M) [119], respectively. Table 5.1 lists the evaluated models and the corresponding datasets.

Hardware setup. We evaluate the scenarios on multiple hardware configurations. Table 5.2 details the cloud instances and on-prem servers used in our evaluations. The cloud configurations allow for testing the CPU utilisation benefits of TENSORSOCKET by varying the number of vCPUs while keeping the GPU count the same. The A100 server features multiple GPUs allowing us to evaluate data sharing when each GPU trains a separate model. Finally, the H100 server's GPU is large enough to collocate multiple DALL-E 2 training tasks. As a result, the variety of the hardware setups allows us to evaluate the impact of TENSORSOCKET on different environments, use case scenarios, and collocation options.

Modern GPUs support different primitives for workload collocation on a single GPU [79]. In this work, we utilise NVIDIA Multi-Process Service (MPS) [51], unless stated otherwise, since it is shown to allow flexible collocation while exhibiting high performance [54]. Processes executed under MPS share both GPU memory and streaming multiprocessors (SMs). The MPS daemon automatically handles the sharing of the SMs across the collocated processes.

Application	Model	Dataset
Image Classification	RegNetX 002 RegNetX 004 ResNet18 MobileNetV3-Small 0.75 MobileNetV3-Large 1.00	ImageNet
Audio Classification	CLMR	LibriSpeech
Image Generation	DALL-E 2 (Diffusion Prior)	CC3M

TABLE 5.1: Evaluated models and datasets.

Metrics. We train the same models on the same dataset without changing the learning process and thus without impacting accuracy. Instead, we focus on the training speed and hardware utilisation as the performance metrics. We quantify the training speed via *samples/s*, the number of training samples processed by the training loop per second. *CPU Utilisation* is measured via `top` [18]. Finally, we measure *GPU Utilisation* with SM Activity, the fraction of active time on the streaming multiprocessors of the GPU, monitored by DCGMI [70]. SM activity is shown to illustrate a finer-grained view of GPU utilisation compared to other GPU utilisation readings from the `dcgm` tool [34].

Training runs. All experiments are run with Python 3 and PyTorch 2, and the latest versions of the respective model repositories as of writing, using the radT platform [79]. We ran everything twice to validate the results and stick to the default model parameter settings as specified by the model repositories. We set the total number of data loading workers across the collocated workloads to the number of available CPU cores (or to 48 for the A100 server as explained in Table 5.2). These workers are split equally among the training processes in the experiments with conventional data loading (no sharing).

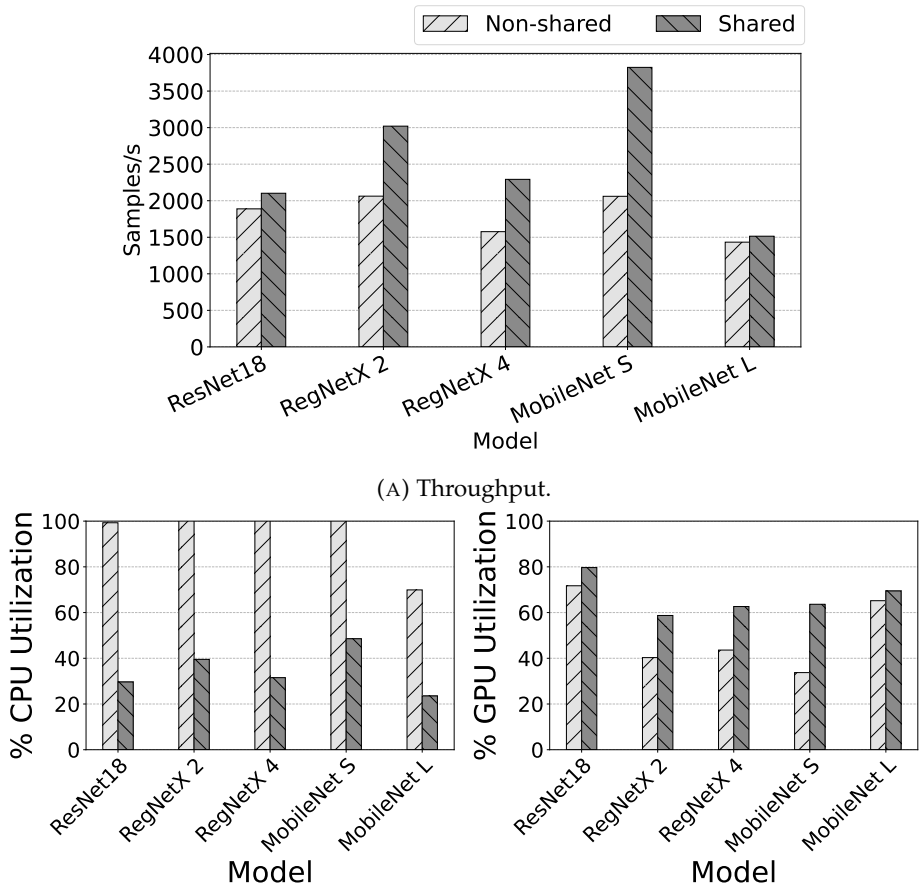
5.4.2 Image Classification

We first evaluate TensorSocket's impact on the training efficiency over the most basic collocation scenario, where the same model is trained on a separate GPU available on the server (e.g., a hyper-parameter tuning scenario). We train a variety of image classification models, as listed in Table 5.1, on ImageNet. ResNet18, RegNetx 4 and MobileNet L are more demanding models to train, while RegNetX 2 and MobileNet S are smaller. Among our hardware setups (Table 5.2), the A100 server is the only one with multiple GPUs available. With 12 CPU-cores per GPU, this scenario additionally showcases TensorSocket's benefits when the CPU-to-GPU ratio is too low to fully utilise the whole system, which is common among lower-priced cloud offerings (Figure 5.1).

Figure 5.6 reports the per-model training throughput and hardware utilisation. In the case of no shared data loader, the training script runs separately on each GPU. When using TensorSocket, we direct the producer to GPU 0 and launch a consumer on each of the four GPUs. The producer and the consumers can communicate via NVLink, which is available on this server across the GPUs.

Instance	(v)CPUs	GPU	VRAM	Cost
H100 Server	24	H100	80 GB	-
A100 Server	128 (*48)	4x A100	4x 40 GB	-
AWS g5.2xlarge	8	A10G	24 GB	$1.212
AWS g5.4xlarge	16	A10G	24 GB	$1.624
AWS g5.8xlarge	32	A10G	24 GB	$2.448

TABLE 5.2: On-prem servers and cloud instances used in evaluation. Costs are on demand per hour costs for corresponding cloud instances [120]. The A100 server is limited to a max of 48 cores to mimic Azure offerings with A100 GPUs (Figure 5.1), which provide a 12:1 vCPU to GPU ratio.



(A) Throughput.

(B) CPU Utilisation

(C) GPU Utilisation

FIGURE 5.6: Image classification training on the A100 server with 4-way collocation, where each GPU has one instance of the same model training, w/o sharing via TENSORSOCKET.

TENSORSOCKET increases the training throughput across all workloads. In MobileNet S's case, the throughput almost doubles, whereas for models such as ResNet18 and MobileNet L the increase ranges from 5% to 10%. The degree of improvement correlates with the computational complexity of the models.

For models such as ResNet18, RegNetX 2, RegNetX 4, and MobileNet S, Figure 5.6 reveals that under traditional data loading the CPU is fully utilised while the GPUs are not. This implies that the CPU becomes the bottleneck causing underutilisation of the GPU resources. Sharing via TENSORSOCKET resolves this bottleneck by reducing the stress on the CPUs while achieving a higher GPU utilisation in addition to the throughput benefits.

On the other hand, for models that are not CPU-bound such as MobileNet L, TENSORSOCKET provides marginal benefits on the throughput and GPU utilisation. However, it frees up 70% of CPU resources. The savings in CPU resources can allow for collocating additional workloads on the CPU side in an on-prem setting and cutting the costs in a cloud setting.

We also provide a sensitivity analysis with respect to varying levels of collocation in Figure 5.7. For this, we use both MobileNets as they are the models that exhibit the

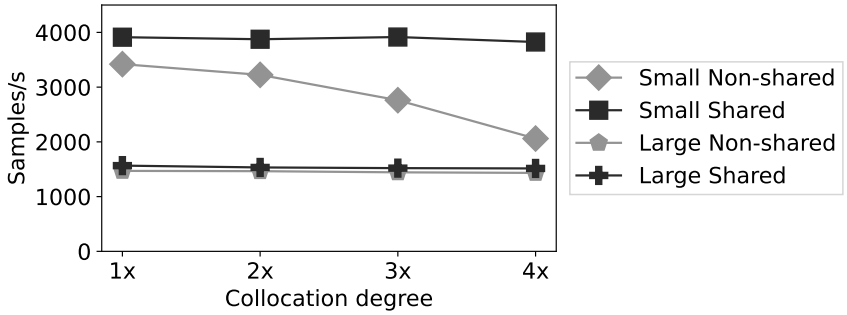


FIGURE 5.7: Per-model training throughput of MobileNet Small and Large with increasing degree of collocation on the A100 server. Each collocated model is trained on a separate GPU.

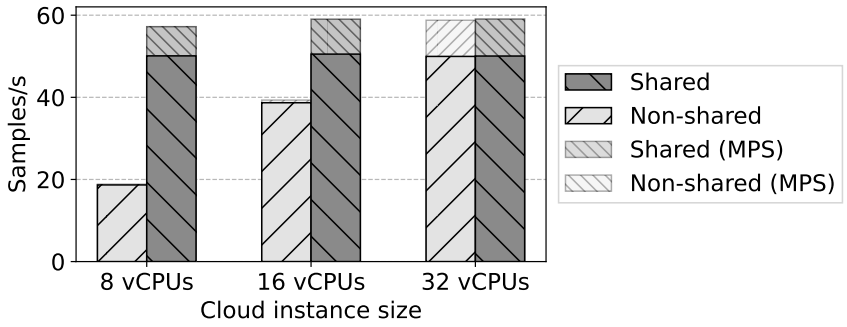


FIGURE 5.8: Samples/s per collocated training on AWS G5 Instances for CLMR - 4-way collocation on the same GPU using MPS and multi-streams across different vCPU counts w/o data sharing via TENSORSOCKET.

most and least benefit from TENSORSOCKET. TENSORSOCKET yields a throughput increase for both the small and large MobileNet in all configurations. On the other hand, increasing the number of models trained simultaneously has little effect on the large model as the CPU is not the limiting factor (Figure 5.6). Conversely, scaling up collocation for the small MobileNet relies on TENSORSOCKET to maintain high throughput.

5.4.3 Audio Classification

We train the CLMR audio classification models in a 4-way collocated fashion on different AWS instances in order to showcase the impact of data and work sharing on host-side resources. The results are shown in Figure 5.8.

For this setup, in addition to MPS-based sharing, we evaluate running collocated processes as a separate GPU stream, which provides more restricted sharing, but may sometimes be the only option in a shared hardware setup such as the cloud. The blurred parts of the bars, therefore, highlight the additional throughput benefits of MPS-based sharing over multi-streams. Regardless, TENSORSOCKET is compatible with any form of GPU sharing primitive.

From Figure 5.8, on the machine with the highest number of vCPUs, the workload with and without sharing achieve the same throughput. This indicates that the number of vCPUs necessary to sustain the GPU throughput for this workload is met.

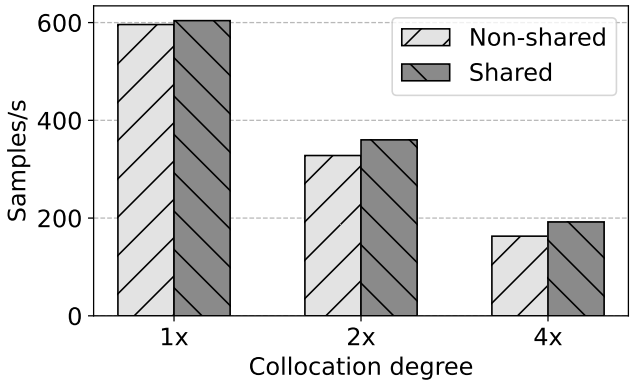


FIGURE 5.9: Samples/s per collocated online training of DALL-E on the H100 server with 1-, 2- and 4-way collocation and w/o sharing via TENSORSOCKET.

Without TENSORSOCKET, however, the smallest instance size of 8 vCPUs performs drastically worse than the largest with 32 vCPUs. TENSORSOCKET effectively reduces the vCPU requirement by 75%. The result is that under TENSORSOCKET all three sizes of cloud instances achieve high training throughput. Based on the costs reported in Table 5.2, this leads to cloud cost savings of about 50%.

5.4.4 Image Generation

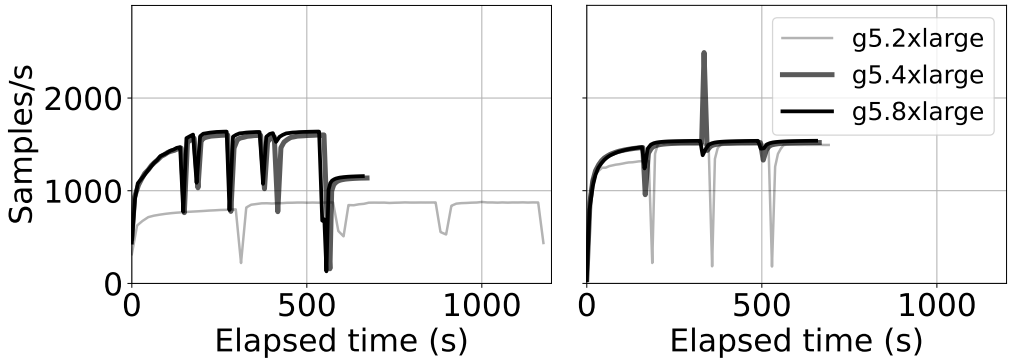
As mentioned in Section 5.3.3, TENSORSOCKET can be used to share not just tasks on the CPU but also on the GPU. We analyze such a pipeline using the DALL-E 2 image generation (diffusion) workload. When training DALL-E 2, data passed to its training process must pass through a CLIP model (Section 5.3.3 and Figure 5.5). CLIP translates the input data into a representation that can be used by the trained model. The CLIP model can be seen as a model with frozen weights when training the diffusion model. This essentially boils down to running inference tasks on the GPU as part of the data preparation process for DALL-E training.

With TENSORSOCKET, we can move the CLIP model inference to the producer of the shared data loader. In this scenario, we aim to showcase how TENSORSOCKET can also reduce redundancy in the computational footprint on GPUs. By only needing a single CLIP model, we can collocate multiple DALL-E 2 diffusion models without running multiple instances of CLIP inference. This workload is carried out on the H100 Server machine, as it is capable of supporting 4-way collocation of diffusion model training.

Figure 5.9 shows the impact. Since the H100 server has enough CPUs to feed the one GPU (Table 5.2), this evaluation scenario is not CPU-bound. Nevertheless, we observe a speedup over non-shared operation under collocation. Under 2- and 4-way collocation TENSORSOCKET is 10% to 15% faster in aggregate throughput than without when running online training. The throughput per individual training process gets reduced as expected, since in this setup, the GPU is highly utilised even without collocation due to the demanding model. This shows that TENSORSOCKET can enable data and work sharing not only on CPUs but also on GPUs.

5.4.5 Model Selection

In addition to the evaluations that revolved around specific domains, we show how our shared data loader supports mixed workloads, which is useful for model selection. As mentioned in Section 5.3.3, the design of our shared data loader supports training of a mixed set of models through reallocating GPU resources between training processes. We evaluate model selection by collocating two different model training processes on different AWS cloud instances. As the training speed of the models differs, we report on the aggregate training throughput. Figure 5.10 shows the results for a mixed workload consisting of a collocated RegNetX 2 and RegNetX 4. The runs on the left use conventional non-shared data loading whereas those on the right use TensorSocket. For the g5.8xlarge and g5.4xlarge AWS instances, the CPU does not constitute a bottleneck, and we therefore do not see substantial throughput gains by sharing. However, we are able to closely approximate the throughput of these larger instances with the smaller g5.2xlarge instance when sharing. In contrast, the workload throttles heavily on the small instance when not using shared data loading. For the g5.2xlarge instance, sharing is therefore not only strictly necessary for running this workload efficiently but also able to deliver almost the same throughput at half the instance cost, as seen in Table 5.2.



(A) Non-shared (B) Shared

FIGURE 5.10: Runtime and aggregate training throughput of mixed workloads (RegNetX 2 and RegNetX 4) on AWS G5 Instances with and without data sharing via TensorSocket.

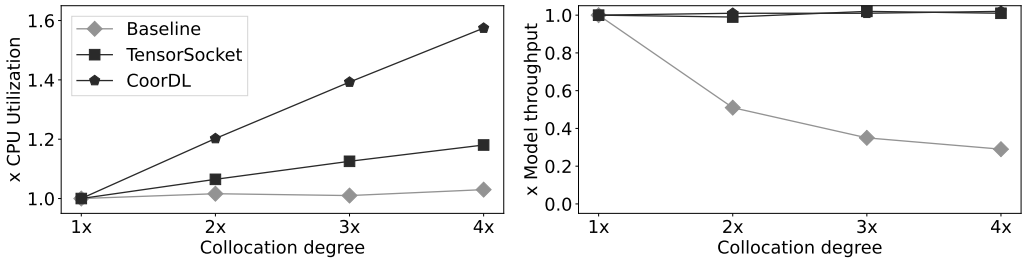
5.4.6 Comparison to other sharing techniques

After having assessed the value of shared data loading via TensorSocket in deep learning training, we compare TensorSocket to other tools that achieve data loading speedups via sharing. We specifically compare against the state-of-the-art methods CoorDL [87] and Joader [93].[3]

CoorDL

Comparing to CoorDL surfaces a couple of challenges due to the age of the library [109]. CoorDL has been designed as a plugin for NVIDIA DALI and is written for Python 3.6. This version of Python, however, has been deprecated by PyTorch since

[3]While we point out the many difficulties in establishing a fair comparison across all the codebases in the rest of this section, we are grateful for the authors of both CoorDL [87] and Joader [93] for providing an open-source implementation.



(A) Normalised CPU utilisation. (B) Normalised per-model training throughput.

FIGURE 5.11: CPU utilisation and throughput scaling of baseline (no-sharing), CoorDL, and TENSORSOCKET on the A100 system under different levels of collocation. The scaling is compared to training a single model with that technique without collocation. Each collocated ResNet18 model runs on a separate GPU. TENSORSOCKET is able to achieve the maximum throughput without the deployment restrictions and higher CPU requirement of CoorDL.

2021, which means modern versions of the framework are not compatible with it. TENSORSOCKET is incompatible with PyTorch 1 as the deep learning framework made sweeping changes with the introduction of PyTorch 2. This complicates establishing a fair comparison between CoorDL and TENSORSOCKET. Nevertheless, we run CoorDL using the evaluation script provided by the authors of the original work [87] to run it as efficiently as possible. We adjust the parameters for TENSORSOCKET accordingly. This means that automatic mixed precision is disabled, the batch size is set to 512, and there are 4 data loading workers. We also choose ResNet18 as the evaluated model following CoorDL's evaluation. Finally, while reporting the results, we normalise the per-model training throughput and hardware utilisation values by dividing them with the values achieved by single-model training (no-collocation). This normalisation is to further eliminate the impact of any unfair differences between the diverging libraries of the corresponding codebases.

Figure 5.11 reports the result of the comparison. These experiments utilise the A100 machine with 4 GPUs. Each instance of the ResNet18 model is being trained on ImageNet and on a separate GPU (as in Section 5.4.2). Figure 5.11a notes the scaling of CPU utilisation as collocation increases. TENSORSOCKET only marginally increases CPU load under higher degrees of collocation, while CoorDL requires more CPU resources to keep up. The CPU utilisation of our baseline, not using either CoorDL or TENSORSOCKET, is close to constant. This can be explained by Figure 5.11b, which shows the throughput scaling as the degree of collocation increases. Both CoorDL and TENSORSOCKEThave no issue keeping the per-model training throughput the same despite higher load. The baseline, however, heavily throttles, losing almost 75% of the performance under 4x load. This throttling can explain the low CPU utilisation, as the data loading workers can not keep up with the training loops and thus the models idle. In general, while both TENSORSOCKET and CoorDL can provide maximum throughput, TENSORSOCKET does so with considerably fewer CPU resources while also being more flexible and less intrusive in usage.

Joader

Joader has a proof-of-concept implementation in Rust that is compatible with the latest version of PyTorch [110]. Specifically, the current implementation of Joader

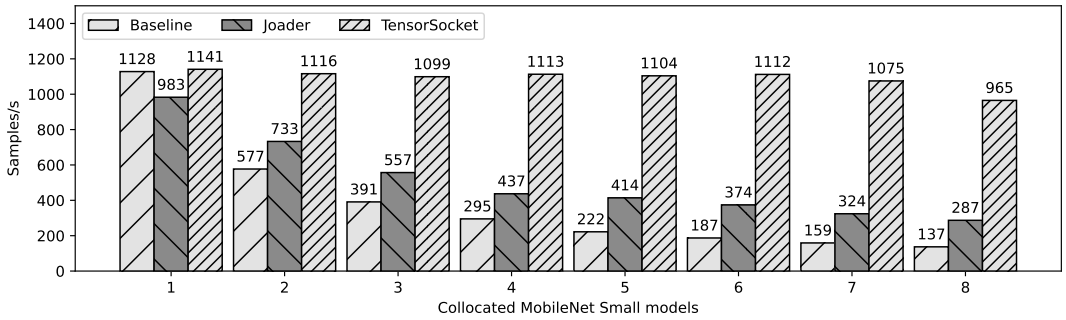


FIGURE 5.12: Comparison of model training throughput with varying degrees of collocation under constrained CPU resources on the H100 system between baseline (no data sharing), Joader [93], and TENSORSOCKET. Each collocated model training is MobileNetV3-Small trained on ImageNet. TENSORSOCKET is able to provide data to many collocated models with little impact on throughput, even under constrained CPU circumstances.

does not require PyTorch at all; rather, it uses Numpy to store data instead. While this does improve the compatibility of the implementation, it raises serious performance issues that hamper its effectiveness in real-world scenarios. Specifically, 1) Joader's image pre-processing and dataset (ImageNet) support is fully hardcoded, 2) images are delivered as NumPy matrices instead of Tensors and 3) Joader does not have support for mini-batches. Therefore, we take a number of concessions in order to support a comparison to Joader that is as fair as possible. Firstly, for (1), we investigate Joader's pre-processing pipeline and configure our TIMM training script to use the same transformations that are hardcoded for Joader. Note that it is not possible to use the exact same transformation code in both, as Joader is written in Rust and pre-processing pipelines available in online model sources like TIMM are typically defined in Python. Then, we address (2) and (3) by having the training script in Joader's case ask for enough data to fill up the batch after which it can construct the tensor and send it to the GPU. This tensor construction from many NumPy matrices, however, is very expensive and will cripple the performance. For the sake of this performance comparison, we are not interested in the loss of the model, we opt to only construct a tensor out of the first batch of data. Subsequent iterations then wait for the new data to come in, only to train on the same, first batch again. This minimises the overhead of batching for Joader.

Figure 5.12 shows the results of comparing Joader this way to TENSORSOCKET on the H100 system. We maximise the performance of all techniques by using MPS for sharing the resources of the H100 GPU. As the baseline, we train 1 to 8 collocated MobileNetV3s without any sharing enabled. Furthermore, we restrict the amount of data loading workers to 8. This means that, under no sharing, every model only has 1 worker when training 8 models simultaneously. Collocation amounts that do not divide 8, such as 5, have the workers divided unevenly in order to sum to 8. Keep in mind that the training scripts themselves are allowed to use further CPU resources. As expected, TENSORSOCKET performs well even when running with high amounts of collocation, dropping no performance up to and including 6-way collocation. Only with 7- and 8-way collocation is there a drop in performance. This comes in stark contrast to non-shared training, where throughput goes down quickly. In fact, the data loading is such a bottleneck for non-shared training here that the summed throughput of collocation never exceeds that of non-collocated

training. Joader comfortably outperforms non-shared training but is far behind the efficiency of TENSORSOCKET. This is likely caused by the extra overhead introduced by Joader's data sampling algorithm (as described in Section 5.2). While Joader's algorithm provides flexibility for training with different speeds and datasets, this sacrifices efficiency in terms of CPU resource use, which in turn impacts the efficiency of training itself.

5.5 TENSORSOCKET Going Forward

Having evaluated TENSORSOCKET's impact on training efficiency, cost savings, and flexibility, this section highlights the key results while discussing further applicability of TENSORSOCKET.

5.5.1 Target Domains and Workloads

TENSORSOCKET alleviates a range of computational and resource-dependent bottlenecks. We believe TENSORSOCKET is valuable for other areas that employ computationally heavy transformations or augmentations, such as with video data. In general, DL training workloads that are prone to exhibit input-bound pipelines or a high degree of CPU utilisation can benefit greatly from shared data loading. If used on workloads in the cloud with similar characteristics as those presented in our work, TENSORSOCKET may reduce costs up to a substantial 50% reduction as shown in Section 5.4.

Achieving these benefits does require some conditions to be met. Training jobs have to be collocated to make use of a shared data loader and are required to train with similar speed on the same dataset. Loosening up these requirements may allow shared data loading to become an attractive option for more workloads, though, that may come at the cost of reduced benefits.

TENSORSOCKET is a solution for training multiple deep learning models on a single node. This works well as in general the GPUs in a single node are the same model. In the perhaps less likely case that the node features different models of graphics cards, it can still be advantageous to run TENSORSOCKET individually per GPU. Finally, to support shared learning over multiple nodes, combining TENSORSOCKET with techniques such as CoorDL[87] or tf.data service[86] may provide an efficient solution.

5.5.2 Generalisability and Customisation

We analysed a complex data loading pipeline that includes generating intermediate representations of our data through an auxiliary model that is not being trained in Section 5.4.4. The ability to support unusual steps like these is a testament to the generalisability of TENSORSOCKET. In addition, we can further dissect the data loading pipeline for finer-grained sharing. This allows for transformations and augmentations that are specific to each training process while only doing costly work, such as image decoding, once. For other tools and techniques that use GPUs for data pre-processing, such as NVIDIA DALI [102] or FusionFlow [103], this means that TENSORSOCKET can be deployed together with these techniques to support GPU-offloading of transformation and augmentation operations while keeping redundancy and computational footprint low.

As of its current implementation, TENSORSOCKET is implemented around PyTorch 2. PyTorch is the leading deep learning framework as of writing and TENSORSOCKET implementation has no other large dependencies. Furthermore, we leave as much of the implementation as possible, such as the data structures, over to PyTorch to minimise the complexity of our codebase. This makes TENSORSOCKET easily maintainable for future PyTorch versions and extendable. Other frameworks, such as TensorFlow, can use TENSORSOCKET right now by using PyTorch as a data sharing intermediate. If demand presents itself, we are interested in considering native TENSORSOCKET support for other deep learning frameworks. As TENSORSOCKET's implementation is compact and the dependencies on PyTorch are isolated, this would require just a small amount of work. Specifically, the wrapper that allows for tensor deconstruction and reconstruction, *TensorPayload*, would need re-implementation to provide native support for the new framework, estimated ∼59 lines of code.

5.5.3 In Conjunction with Related Tooling

Libraries for data pre-processing. Since TENSORSOCKET, by design, serves as a drop-in replacement for framework-specific data loaders, it can easily be integrated with other work or tools that are orthogonal to our work. For instance, PRESTO [84] is a library that seeks to define which data transformations should be carried out offline and online. It does this in order to reach the highest possible throughput, among other target variables. These two systems can be integrated, as from the perspective of PRESTO, the shared data loader of TENSORSOCKET is the same as any other data loader.

GPU collocation primitives. In our evaluation, we mainly utilise MPS for collocating workloads on the same GPU. The GPUs used in our evaluations additionally offer Multi-Instance GPU (MIG) [50] for collocation. Different from MPS, MIG-enabled collocation offers hardware support for splitting up GPU resources across different processes. Therefore, it is less flexible to adjust the resource split on the fly, but the processes execute with a higher degree of isolation and less interference. MIG may be of interest for some collocated workloads using TENSORSOCKET since it is possible to share memory resources of the GPU while keeping computational resources dedicated to every collocated training process.

Tools for hyper-parameter tuning and model selection. Ray Tune [121] is a tool for hyper-parameter tuning that launches training processes with a set of hyper-parameters and replaces any processes that show too high loss values, Cerebro [108] is an efficient training system targeting model selection tasks. As illustrated in Figure 5.4, TENSORSOCKET has the ability to serve training processes that get launched and killed throughout the hyper-parameter optimisation or model selection process. Therefore, it is of interest to see how tools like RayTune or Cerebro can benefit from our shared data loading setup instead of launching a separate data loader for each training process started.

We leave these aspects as future research directions to explore for TENSORSOCKET or alternative shared data loading opportunities.

5.6 Related Work

Historically, there has been a plethora of work on benchmarking and optimising the computational efficiency of the core of model training and serving. However, in recent years, data loading and pre-processing have been gaining more attention, as with ever-increasing model sizes and training throughput requirements, the cost of data loading bottlenecks is growing rapidly [101].

Murray et al. [86] and Mohan et al. [87] emphasize the role of the data loading pipeline and its effect on training efficiency. The former presents the tf.data framework to ease the tuning for the computational efficiency of data pre-processing. The latter proposes CoorDL, a data loading library, and MinIO, a software cache with the goal of reducing cache thrashing. These works, among others [92, 91], also advocate for sharing for DL data loading tasks, but more at the cluster-level rather than the finer-grained view of TENSORSOCKET.

Joader, proposed by Xu et al. [93], provides an alternative to CoorDL that offers extra flexibility when sharing data for training tasks, allowing for shared training on multiple datasets at the same time. It manages this via a novel data sampling solution that optimizes sharing at the cost of some dataset intersection calculations at every training iteration. Such calculations can be costly as Section 5.4.6 demonstrated.

As already mentioned in previous sections, there has also been work to offload data pre-processing tasks onto GPUs [102, 104, 103]. TENSORSOCKET is orthogonal and compatible with these works since it allows for sharing on both CPUs and GPUs.

Behme et al. [114] explore lossy image compression's role in mitigating data loading bottlenecks. They demonstrate that moderately compressed data maintains accuracy comparable to benchmarks while saving 30% storage and how this technique complements the software cache of MinIO [87].

TENSORSOCKET also borrows ideas from works on data and work sharing in databases such as StagedDB [122], QPipe [94], and SharedDB [95], and works that analyse the trade-offs of sharing [97, 123]. These works aim at sharing work that is common across concurrent database queries. In contrast, TENSORSOCKET applies similar sharing ideas to DL data preparation.

Finally, the latest standardised benchmark from TPC, TPCx-AI [124], specifically makes data preparation process an essential part of the benchmark. MLCommons also recently released a benchmark suite focusing on storage [125, 126] unlike its previous benchmarks. These benchmark standardisation efforts emphasise the increasing importance of the data preparation steps of deep learning and the need for investing in crucial optimisations for such steps.

5.7 Conclusion

In this chapter, we presented TENSORSOCKET, a novel data loading mechanism that enables data and work sharing in data pre-processing pipelines of deep learning training. The key insight behind TENSORSOCKET is that tuning efforts to achieve the best model architecture and parameters require training several models on the

same data. This results in shared tasks across these training processes. We demonstrated that TENSORSOCKET can double training throughput while substantially reducing the number of CPU cores to achieve that throughput. Furthermore, it is easy to adopt in existing training pipelines, enables certain training scenarios on restricted hardware resource setups, and can halve cloud setup costs as a result. Finally, TENSORSOCKET is compatible with existing techniques that aim at increasing the computational efficiency of the data pre-processing tasks.

Chapter 6

Progressive Resizing

6.1 Introduction

Data is the driving factor behind the current deep learning boom. The quality of model training heavily depends on it, and the models can only turn out as well as the data lets them. The demand for deep learning, however, has only led to larger and larger datasets, accompanied by larger and larger models. While these can capture great detail, it takes a significant amount of hardware resources, energy, and expertise to train models on such a scale.

Model training happens in an iterative fashion. The model repeatedly passes over the dataset (one epoch) in order to get accustomed to the data. The speed at which a model improves tends to be significantly faster at the start compared to at the end. This is because the model can make quick progress by learning rough features, but has to optimise on smaller details in the end.

What if we use this to our advantage? When we teach children a subject, e.g. Maths, we do not start by introducing them to integrals. Instead, we start with easier concepts first before proceeding to more difficult concepts. We apply this analogy to deep learning. Why should a model learn on fully detailed images from the start, instead of breezing through lower detailed samples first? In this chapter, we advocate for increasing the complexity of training samples during training, *progressive resizing*, based on this insight.

While progressive resizing has been around for a couple of years, it has never been extensively researched. FastAI [127] has been the most vocal supporter of the technique[128]. They, however, just provide a basic set of guidelines for the technique and leave a lot of manual work to the end-user. This, in turn, makes the proposals in this area impractical. Progressive resizing, however, has the potential to reduce training time in exploratory runs and reduce the resource cost of Deep Learning. We thus want to explore this technique to the fullest and provide actionable guidelines and a library to make this technique easy to adopt and attractive to deep learning practitioners.

In this chapter, we analyse the effects of progressive data loading and introduce our own automated progressive resizing data loader. Our method circumvents having to do resizing manually, which is the contemporary approach [128, 73], while still reaping the benefits. We explore the effects of varying image size on network training in depth and make the following contributions:

- Provide an accessible drop-in solution for progressive data loading that yields the benefits of progressive resizing without requiring further customisation.
- Identifying the requirements for hyper-parameters to make progressive resizing automated with high training throughput and hardware utilisation, resulting in up to twice as fast accuracy growth during pre-training.
- Provide two progression methods, ***exhaustive*** and ***mixed***, which are viable in different use cases depending on model and training requirements.

Additionally, we provide extensive empirical testing of our method with multiple models using both PyTorch[129] and FastAI[127].

The rest of the chapter is structured as follows. We delve into the required background first in Section 6.2. We then discuss the methodology behind AdaSize and the setup in Section 6.3. Our experiments and results are detailed in Section 6.4, after which we discuss and conclude in Section 6.5 and Section 6.6, respectively.

6.2 Background

The network itself is at the core of deep neural network training. The depth of the network, the size of the inputs and the size of the layers all play a key role in how much time it takes to go through the training loop. Crucial for training networks quickly is reducing the amount of computations required. The time required for training a model is a result of the amount of floating point operations required. This computational requirement is directly dependent on the amount of data that has to be processed.

Learning Rate and ***Batch Size*** are key hyper-parameters for effective and efficient training convergence. *Learning Rate* dictates how much the gradient computed during backpropagation affects the weights. A higher learning rate results in faster movement through the model space, but can hinder convergence due to overshooting the optimal learning path. As a result, *learning rate decay*, reducing learning rate as training goes on, is fundamental in achieving top accuracies in neural networks. Initial weight optimisation can be done roughly and take large steps at a time, whereas a low learning rate is well suited for final optimisations. Furthermore, some techniques adapt the learning rate on a layer basis [130, 131, 132], improving performance even further. *Batch size* dictates the amount of samples to be processed per batch. Commodity training hardware, such as GPUs and TPUs, are optimised for embarrassingly-parallel operations, and thus run most efficiently on large batch sizes. This in turn leads to a faster training speed as the amount of batches, and by extension, parameter updates per epoch goes down when increasing the batch size. Additionally, large batches are key for enabling distributed, multi-GPU neural network training. In such cases the workload is typically distributed data-parallel, where the samples in the batches are distributed over all of the GPUs, requiring a larger total batch size. This has lead to state-of-the-art training times [133, 134, 130, 131, 135, 132]. Large batch sizes, however, come at a cost of training performance as increasing the batch size can adversely affect attainable accuracy.

Adaptive Batch Size [133, 134] gradually increase the batch size to reap the benefits of increasing the batch size while preventing any of the pitfalls. This is typically done

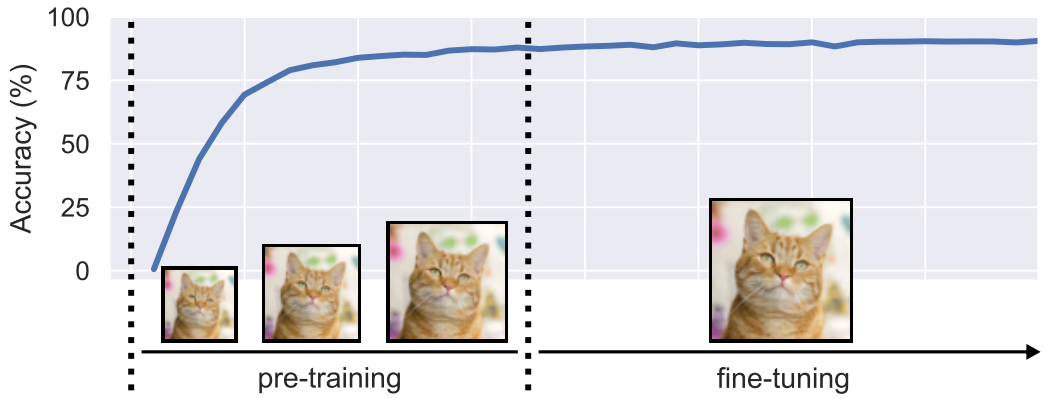


FIGURE 6.1: Progressive resizing involves gradually increasing the size of input data in order to speed up training. The training ends with optimisation on the original image size to ensure high final accuracy.

by coupling the batch size increase to learning rate decay [136]. Coupling batch size and learning rate allows for larger batch sizes than with a static batch size, yielding impressive acceleration in training speed. However, batch size can only be scaled up to a certain point, after which further increases would hamper convergence. Balancing batch size and learning rate can be challenging as earlier suggested rules of thumb [137] do not necessarily generalise well [130], with more aggressive scaling being optimal for some architectures but preventing convergence for others. Alternatively, introducing adaptive learning rates per layer is more intrusive but allows for large batch sizes for a large selection of models, including different vision models and language transformers [130, 131].

Progressive Resizing is a technique where the resolution of the input is gradually increased. Lower-resolution input contains less information but is quicker to train on. Resizing allows for the initial training, which should develop the weights of the network in a productive initial direction, to run on a more efficient dataset, before fine-tuning on higher resolution data. This approach has been successfully combined with progressive regularisation [73], which increases the degree of regularisation as training goes on. Lower-resolution data requires less regularisation for efficient training, prompting this combination. Similarly, Mix&match [138] significantly reduces training time by stochastically providing training steps with reduced input sizes.

Resizing is usually done with Bilinear or Bicubic rescaling, as these techniques have been staple in image processing for a long time. These techniques provide humanly pleasing results, but this may not be the best downsizing for network training. An alternative is using a convolutional network for downscaling, but this induces additional compute requirements as this network should be trained together with the main network [139].

Given all of these dimensions for configurations, it is not clear what one should use or what performs best. Thus, methods to automatically tune progressive resizing and a set of guidelines are required to facilitate wider adoption.

6.3 Adaptive Progressive Resizing

We will now start by discussing convolutional neural networks in depth (Section 6.3.1), after which we explain our resizing strategy (Section 6.3.2 and Section 6.3.3) and our implementation (Section 6.3.4).

6.3.1 Layer Definitions

Convolutional neural networks derive their name from the ***convolutional layers*** they contain. These layers move a shared-weight kernel over the input resulting in activations that are heavily influenced by space-locality. The network is then built up by a repeat application of different layers. In essence, every layer can be defined as a function $Y = F(X)$, with F as the operation, X as the input, and Y being the output of the layer. A (convolutional) network then becomes a chain of such functions.

Applying a layer to an input tensor does not just change the values, but can also change the shape of the output. Specifically for convolutions, we can derive the expected shape of the output tensor via:

$$S_Y = (W_Y, W_Y, Q_F) \tag{6.1}$$

with

$$W_Y = ((W_X - (W_F + 2P_F))/S_F) \tag{6.2}$$

where S_Y is the shape of tensor Y, W_Y is the width and height of tensor Y, W_X is the width of the input X, and W_F, P_F, S_F and Q_F are the width, padding, stride and filter count of the kernel of F respectively. Observe that there is no way to increase the width and height of the output while keeping the same input. The padding term P_F solely exists to compensate for the size-reducing effects of W_F. Convolutional layers thus always result in equal or reduced width and height.

In terms of trained parameters, the input size does not affect a convolutional network in any way. Instead, the parameter count is purely determined by the size of the kernel.

Pooling layers act akin to convolutional layers but do not have trained weights. Instead, they use a simple mathematical function, such as taking the maximum of the kernel inputs.

Convolutional and pooling layers are commonly used in convolutional networks to reduce the size of the data for subsequent layers. The most common way of doing so is by using a stride that is larger than 0. For example, a stride of 1 results in halving both width and height. Most convolutional networks, such as ResNet [23] and EfficientNet [140, 73], group layers together in blocks ending with such a reduction in size. Most importantly, for convolutional networks defined in such blocks, we can simplify the tensor shaping to:

$$W_B = W_X / S_B \tag{6.3}$$

where W_B is the output width and height of the block and S_B is the stride of the last layer of the block. Figure 6.2 illustrates how blocks make up the ResNet architecture.

The output of a convolutional neural network is typically composed of several layers. The result of the last block generally has a small spatial size but a high activation depth and needs to be translated into output predictions. It is common to apply several fully connected layers first. Afterwards, the predictions can be made using for example a softmax layer. The parameter count of a fully connected layer is defined as follows:

$$P_F = C_Y(C_X + 1) \tag{6.4}$$

with P_F the parameter count of the layer, and C_Y and C_X the value counts of output and input respectively. The shape of the outputs of these layers is completely static and thus does not depend on the input size. This means that there is no effect on shape or parameter count for these layers. One should keep in mind, however, that for the first fully connected layer the parameter count can vary, as the size of the input changes.

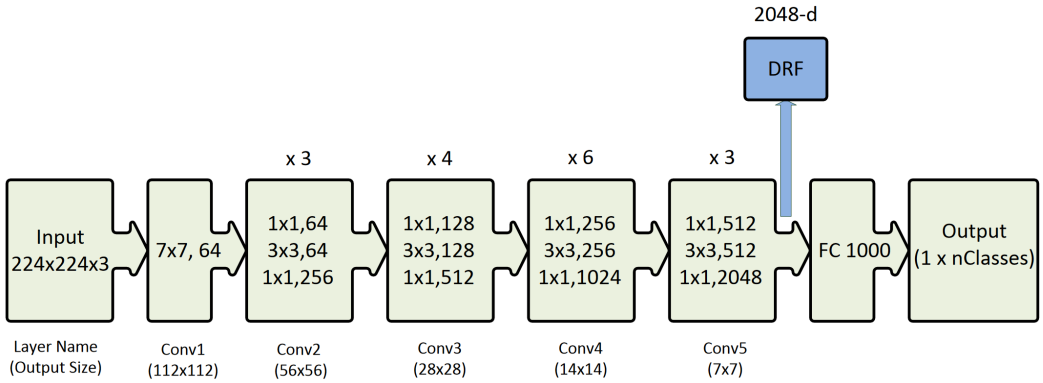


FIGURE 6.2: Layer block definitions and activation size reductions in ResNet[23]. (Figure source [141])

6.3.2 Convolutional Network Image Scaling

When adaptively sampling image sizes we are actively changing the data that is being put into the model. As seen earlier in this section, the size of the activations is directly influenced by the shape of the input. Our adaptive data sampling will thus change the shape of the activations as well. Additionally, for fully connected layers, the size can change the parameter count of the model. This leads to the following two conditions when changing the shape of the input:

- The input sizes must result in valid activation shapes.
- The input resizing must not introduce new parameters.

Shape: Given a convolutional network composed of N blocks. Assuming padding and a stride of 1, the shape of the activation of the last block will equal

$$W_{final} = \lceil W_{data}/2^N \rceil \tag{6.5}$$

We will always be scaling the input data to be of a lower size than the normal maximum training size. W_{data} is thus being reduced from its expected value. Halving the input size will thus, following Equation (6.5), equal adding a singular identity block to the neural network. Crucially, we can keep reducing the data this way, even when the size reaches 1x1. Due to rounding up the shape can never be smaller than a singular convolution.

Parameters: Blocks do not express any dependency on their parameter count to input shape. There is, however, a concern to be had with the connection of the last block to the prediction layers. If the input size exceeds the total reduction by the blocks, and thus the resulting shape is larger than 1x1, then a change in image size can lead to a smaller shaped activation tensor. Popular convolutional networks such as ResNet [23] and EfficientNet [140, 73] have an additional global pooling layer after the last block to force reduce the output into the 1x1 shape, preventing any issues resizing could cause with parameter counts, see Figure 6.2.

6.3.3 Hyperparameter Rebalancing

While the amount of trainable parameters does not change and the rescaling is safe for convolutional networks, we do need to keep extra care for balancing hyperparameters to ensure good convergence. The reduction in image size has multiple effects on neural network training:

- Reduce problem size for GPU kernels
- Regularise against overfitting

GPU utilisation is paramount for optimising training efficiency. If we consider high GPU utilisation when training the network on high-resolution data, it is expected that reduced image sizes will leave parts of the GPU idle. We aim to accelerate training as much as possible to reduce training time and thus overall resource usage. This requires compensation for the reduction in problem size and keeping the GPU busy as much as possible. Consider the FLOPS estimate for calculating a forward pass on a convolutional layer:

$$FLOPs_F = (B \cdot B_S) \cdot Q_F \cdot W_Y^2 \cdot W_F^2 \tag{6.6}$$

with B being the batch size and B_S the ***batch size scalar***. The code that computes convolutional layers on the GPU parallelises in all these dimensions. We can thus keep roughly the same GPU utilisation by keeping the $FLOPs$ the same. As the width and height of our input decreases so does the amount of floating point operations. The batch size presents itself as the most flexible way to compensate for reductions in $FLOPs$. More specifically, we introduce a batch size scaling factor B_S in Equation (6.6) that compensates for any change in image size. Assuming that the original configuration for the training task utilises the hardware well, this results in the utilisation we are looking for.

$$B_S = (W_{Yoriginal} / W_{Ydownscaled})^2 \tag{6.7}$$

Learning rate. Increasing the batch size is commonly used in distributed machine

learning [132] but is known to affect the learning quality of the model. Batch size and learning rate are often linked together to compensate for this. Specifically, consider what happens on the gradient descent with a momentum model weight update:

$$W_{t+1} = W_t - V_t \tag{6.8}$$

$$V_{t+1} = \beta V_t + (\eta * \eta_S)\Delta W_{t+1} \tag{6.9}$$

where W_t and V_t are the weights and momentum, respectively, at step t, β is the momentum coefficient, and η and η_S are the learning rate and ***learning rate scalar***. The weights are directly scaled by the learning rate; doubling the learning rate will double the size of the weights. The frequency of the weight update in Equation (6.8) depends on the batch size. A larger batch size means that there are fewer batches required to cover the whole dataset in an epoch. Increasing the batch size will thus reduce the amount of model updates per epoch. Similar to existing learning rate scaling schemes [130], we compensate for this similar to Equation (6.6) by scaling learning rate with a factor η_S:

$$\eta_S = B_S \tag{6.10}$$

Warm-up is a common method to reduce overfitting by the model and improve overall training performance. Instead of straight starting with a high learning rate, warm-up introduces a couple of epochs that gradually increase the learning rate. This prevents over-influence by early training samples and improves the performance of adaptive optimisers. With progressive data loading, however, we argue that there is no need for a warm-up period. Downscaled images differ greatly from their higher-resolution counterparts which resolved any overfitting that might occur in early training steps. Additionally, for time-to-accuracy-sensitive workloads, warming up can considerably slow down training speed. Not warming up can thus save considerable time and computation.

6.3.4 Implementation

We have designed a progressive dataloader that provides an interface for progressive data loading to PyTorch as well as tries to automate the process as much as possible. Our method extends native PyTorch dataloading classes and is compatible with any PyTorch training pipeline. Additionally, one can replace a standard existing data loader with our library in-place, accelerating the training curve without extra setup required.

Figure 6.3 depicts the data loading pipeline in PyTorch. The dataset interfaces with the data loader first to provide the length of the dataset, and secondly with the workers to provide the actual images. The sampler builds a permutation of sample IDs which are grouped in batches when iterated upon. It is these batches that are sent to the workers to be fetched. The worker count usually spans most of the CPU cores, which means that many batches are prepared at the same time. This reduces stalling of the training process due to an increase in data throughput. A batch is sent to the training process whenever the worker has finished loading and pre-processing it, after which the worker can work on the next batch.

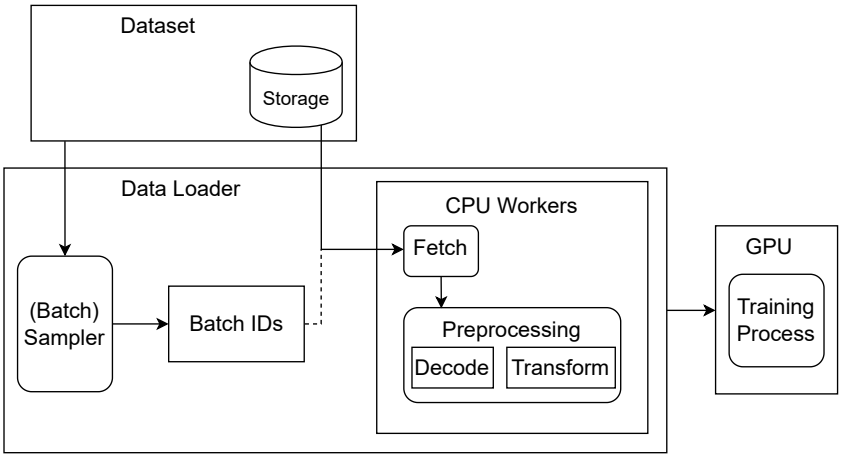


FIGURE 6.3: Data loading pipeline in PyTorch. The data loader provides and loads based on a permutation of samples based on the length of the dataset. Batches of sample identifiers are sent to data workers which then fetch and process the data.

Size Buckets

Doing progressive resizing manually, as for example advocated by FastAI[128], usually causes large jumps in input image size. This is done by manually creating a new data loader object per image size. Automating this process this way to produce a gradual training curve requires a large amount of differently-sized images of the training stages.

The PyTorch data loader is able to load and prepare multiple batches at the same time due to multiprocessing over the data workers. This allows for increasing the data loading throughput, providing enough images for training to be run as fast as possible. When scaling down the image size, however, we are drastically increasing the batch size to keep the GPU saturated, as discussed in Section 6.3.3. This increase in batch size means that the data workers have to load significantly more data in the same timeframe. This can lead to excessive CPU utilisation and even CPU or I/O throttling.

In order to provide an option to circumvent these issues we run training on a couple of predetermined image sizes and interpolate with other means, still resulting in smooth learning curves. Images can be stored on disk in these predetermined sizes to some storage for a reduction in CPU cycles and memory movement. We call these image sizes ***size buckets***. With our method we amend this pipeline in two different ways, providing two options that can be valuable in different situations.

Exhaustive Loading

Exhaustive progression is the least invasive form of automated progressive resizing. Similar to the manual method, we train on the size buckets sequentially from smallest to largest. Instead of jumping from one size to another between two epochs, however, we gradually upgrade images that we deem to be less interesting to the model to the next bucket. We keep training on the lowest-size bucket that still contains images. Eventually, all images will have progressed to the largest buckets, which forms the optimisation phase. Exhaustive loading does not change any parameters other

than what we would require from progressive resizing. It does, however, change the definition of an epoch, as now training an epoch on the smallest bucket no longer includes all samples. Exhaustive loading thus causes some images to be visited less frequently than others.

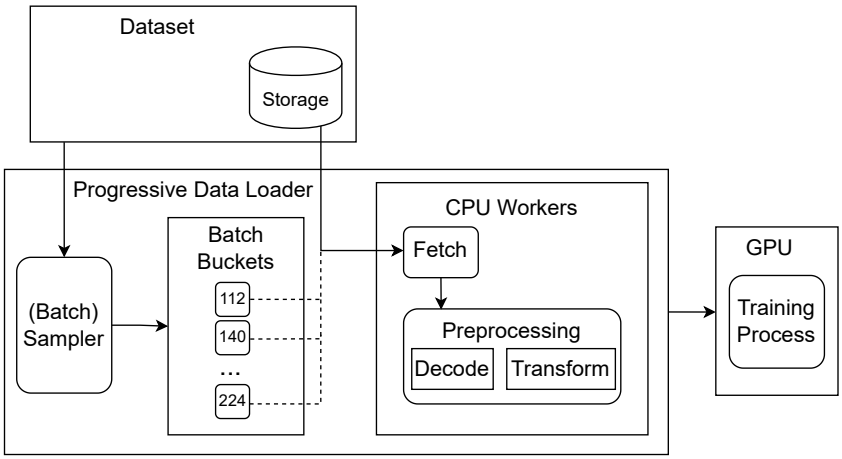


FIGURE 6.4: Mixed data loading in our method. The data loader holds a reference to the current required image size for every image. This groups the images in buckets, which when full are sent to the data workers which then fetch and process the data.

Mixed Loading

Mixed progression offers an epoch-safe alternative to exhaustive progression. We change the data loader to collect samples in buckets instead of sending them straight to the CPU workers. This results in Figure 6.4. Once a batch bucket has filled up the IDs are assigned to a worker. Keep in mind that, following eq. (6.7), the number of items of a bucket varies depending on its image size. Smaller image sizes have buckets with more samples.

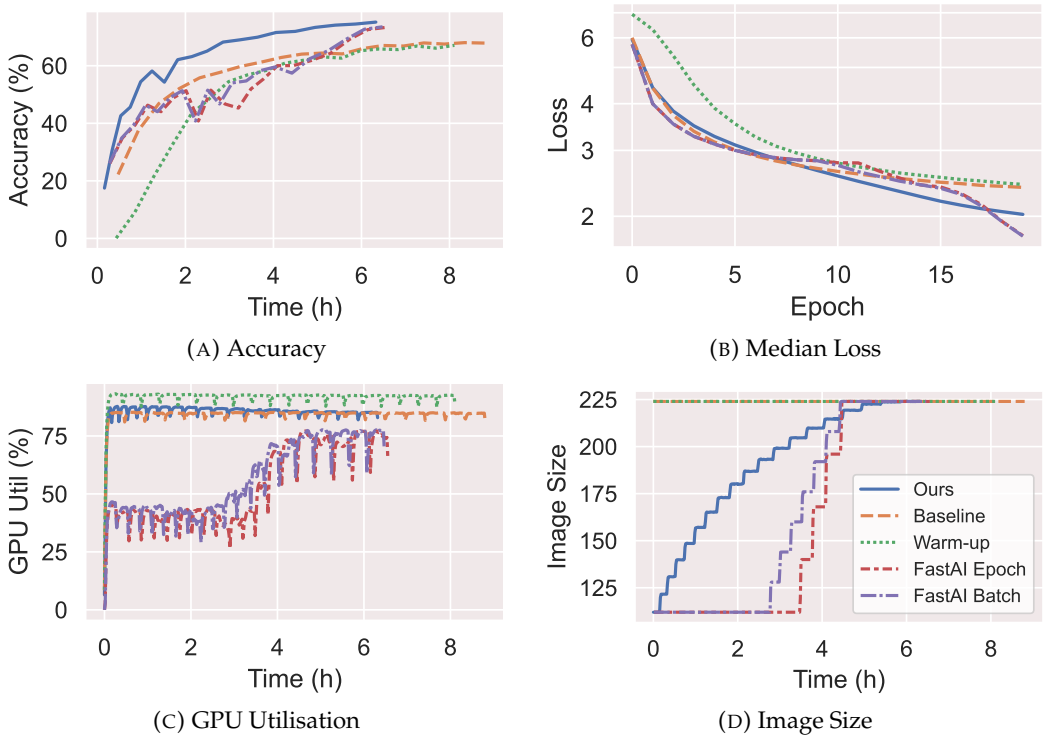
While mixed loading is epoch-safe, it does introduce extra restrictions for the model that is trained on the data. With mixed loading the batch size and image size is not constant within an epoch. This is not an issue for most convolutional neural networks but does pose a restriction for other architectures such as transformers.

6.4 Results

We evaluate our implementation of progressive resizing against conventional training and manual progressive resizing [128]. Furthermore, we explore the impact of progressive resizing on performance and hardware. We evaluate both of our resizing strategies, vary the upgrade rate and explore whether CPU utilisation may become a bottleneck.

6.4.1 Setup

We evaluate everything on a server with 24 vCPUs and an NVIDIA H100 GPU with 80 GB of VRAM. Models have exclusive access to the server for training. We source our models and training script from PyTorch Image Models (TIMM) [142], a popular codebase for image classification models, except for the evaluation of FastAI,



(A) Accuracy

(B) Median Loss

(C) GPU Utilisation

(D) Image Size

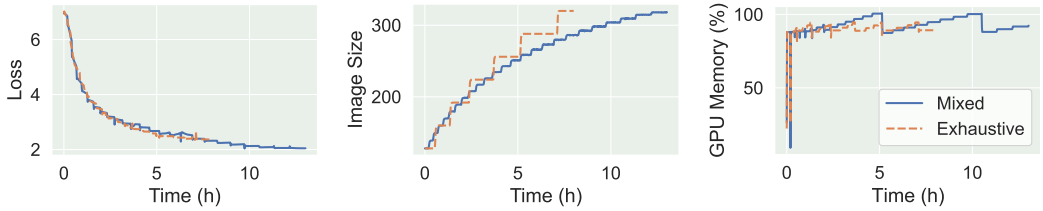
FIGURE 6.5: Training a ResNet152 model from scratch.

which we evaluate on their own codebase instead. We use the progressive resizing strategies that are part of fastxtend [143] for FastAI. We evaluate all models on the ImageNet2012 dataset [9].

6.4.2 Comparison to other techniques

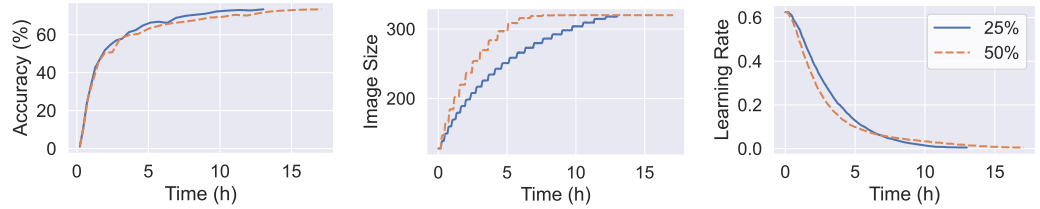
We first evaluate our progressive resizing solution training a ResNet152 model. We compare against a baseline training example that does not utilise resizing and FastAI resizing either during epochs (batch) or in between epochs (epoch). The results of training ResNet152 models using various techniques for 20 epochs can be found in Figure 6.5. Figure 6.5a depicts the top-1 validation accuracy of the trained models. Our method consistently outperforms all other methods due to the sharp accuracy curve. Compared to the baseline, our method is able to hit 65% accuracy in just over 40% of the time (2.5h vs 5.9h), whether warm-up is enabled for normal training or not. Disabling warm-up for baseline training greatly speeds up initial training steps, but both strategies converge around 4 hours in. FastAI performs similarly under both advertised scaling schemes but is ultimately unable to keep up with our method. Interestingly, the accuracy of their initial training is very similar to that of the baseline, but they greatly benefit from short fine-tuning at the end of training.

Analysing loss allows us to delve deeper into the training behaviour of the methods. Figure 6.5b reveals the median loss per epoch, with loss scaled logarithmically. As expected, the baseline with warm-up reduces its loss slower at the start due to the restrictive learning rate. All other methods curve similarly for the first half of training. In the latter set of epochs, our method is able to continue steadily improving on loss while other methods start to slow down. Note the aggressive reduction again



(A) Loss (B) Image Size (C) GPU Storage Utilisation

FIGURE 6.6: Training an EfficientNetV2 Medium model with mixed and exhaustive resizing.



(A) Accuracy (B) Image Size (C) Learning Rate

FIGURE 6.7: Training an EfficientNetV2 Medium model with mixed resizing with an upgrade rate of 25% and 50%.

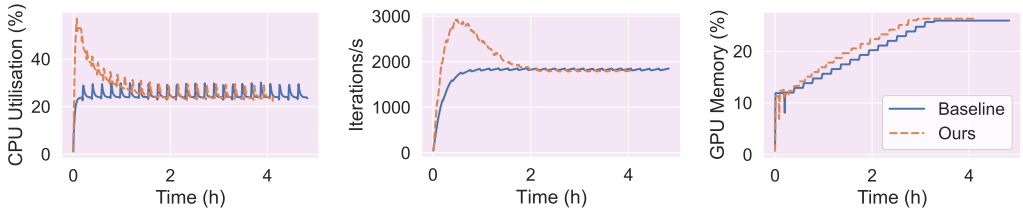
featured by FastAI, which seems to approach overfitting as the loss dips significantly below ours while having lower validation accuracy.

Figure 6.5c reports the GPU utilisation while running these methods. Notably, our method is able to use the GPU to a high degree even though images are scaled down. The GPU utilisation of our method and baseline training without scaling is essentially identical. FastAI does not introduce any compensation for image downscaling and thus is only able to use part of the GPU. This is most pronounced for the first three hours, as during this time FastAI is training on half-sized images, as shown in Figure 6.5d. Our method automatically interpolates a smooth resizing curve based on the target resolution.

6.4.3 Performance under Progressive Resizing

Resizing strategy: Our method supports two different resizing strategies; mixed and exhaustive resizing (Section 6.3.4). Figure 6.6 contains the results of training an EfficientnetV2 Medium model with both strategies for 25 epochs. Both mixed and exhaustive resizing results in very similar loss curves, as seen in Figure 6.6a. As exhaustive resizing changes the sample quantity of epochs, a 25-epoch training period finishes significantly quicker than when using mixed resizing. Under exhaustive resizing, image size is scaled less smoothly than under mixed resizing (Figure 6.6b), though this does not seem to hurt training performance. Both techniques utilise load rebalancing via batch size and thus are able to maintain constant GPU utilisation, as exemplified by the GPU storage utilisation patterns uncovered in Figure 6.6c.

Upgrade rate: The upgrade rate controls how many of the samples are upgraded after every epoch. For the experiments that resulted in Figure 6.7, we evaluate upgrade rates of 25% and 50% over a 25-epoch training window. Our target image size is 320, taking linear steps of 32. This results in 12 and 24 epochs required for scaling respectively.



(A) CPU Utilisation (B) Throughput (C) GPU Storage Utilisation

FIGURE 6.8: Training a MobileNetV3 Large model with and without resizing.

Both models train to comparable validation accuracy (Figure 6.7a). The model that trained with slower upgrading consistently outperforms the faster upgrading one even though the difference is small. The difference in image upgrading speed can be observed in Figure 6.6b. While the accuracy of both models starts similar, the slower upgrading model starts edging out the faster one once the faster model has upgraded the images almost to their full size, roughly 4 hours in. The smaller images are accompanied by larger learning rates, as shown in Figure 6.7c.

Hardware Utilisation: Finally we evaluate the impact our method has on CPU utilisation, shown in Figure 6.8, and relate this to throughput and GPU utilisation. The data workers have to load and scale down full images throughout training. As expected, CPU utilisation spikes heavily at the start of training when using small image sizes (Figure 6.8a). Data loading CPU usage depends on the time it takes to load one image combined with how many images need to be loaded. Figure 6.8b reveals that, due to our resizing policy, we need significantly more image throughput at the start of training. GPU memory utilisation remains constant between training with and without resizing (Figure 6.8c). Our re-balancing makes sure that the GPU is used well without exceeding GPU memory limits, which is required to keep training from crashing.

6.5 Discussion & Future

While our vision of a drop-in progressive data loader has taken shape, we have identified a couple of angles that require further investigation and improvement.

Optimal Hyper-parameterisation: While we have demonstrated our current implementation as a viable proof-of-concept, progressive resizing introduces a vast amount of new hyper-parameters (resizing strategy, upgrade rate, amount of samples to upgrade per step, hyper-parameter balancing strategy) to model training. Our vision for our method is to provide tangible benefits by running with default hyper-parameter values, with further optimisation possible if the user wants to. In order to ensure the robustness of our work, a thorough evaluation is required with a combination of different hyper-parameterisation, datasets and models.

CPU Utilisation: Progressive resizing can provide large initial speed increases by training aggressively with large throughput, facilitated by large batch sizes. This large throughput can significantly strain CPU resources (Section 6.4), potentially causing a CPU bottleneck. When designing our method we have taken into account this shortcoming. In return, we use a step-wise resizing strategy instead of e.g. an interpolated linear one as used for EfficientNetV2 [73]. This limits the amount of

size variants required of every image. Using this to our advantage, we plan to include a dataset *pre-processing* strategy that saves a copy of the dataset for every image size step, removing the need of downscaling during training itself and reducing the amount of data that needs to be read from disk.

Curriculum Learning is an optimisation for machine learning that affects the input data of the model [144, 145, 146, 147, 148, 149]. Instead of just writing a better model to learn a specific task, Curriculum learning tries to quantify the usefulness and descriptiveness of data samples that are going into the model via some heuristic. This way samples that may not be relevant or productive to focus on for the model at a specific epoch can be left out in order to improve convergence or generalisability.

Essential to curriculum learning is figuring out how the model might react to the data. In cases where overfitting is the problem, some or all of the data can be run through the model to fuel this heuristic. The runtime of this forward pass does not matter in this case, and only part of the result is used for the backward pass. When the aim is training speed, however, such methods in general too expensive to employ, as they add additional time complexity when we seek to make things faster. This means that one has to resort to significantly less accurate metrics, such as the results of the last forward or backward pass. In general, even a reduced forward pass based on sampling can lead to massive slowdowns [145]. The network is trained on an accelerator that performs best on large workloads, and thus a small sampled workload still requires significant computing time.

Similarly, data selection algorithms such as ***Coreset search*** attempt to find a subset of the dataset that includes all important concepts to lead to an effective epoch with minimal duplication [150, 151, 152]. This reduced dataset can then be trained on more quickly leading to faster convergence, but data selection algorithms, similar to curriculum learning, are often too expensive to benefit time-to-accuracy [152].

For progressive resizing, techniques from curriculum learning may benefit the way the images are scaled up. Instead of selecting a random subset of samples to scale up, one could use a heuristic to select the samples that would most benefit from scaling up.

6.6 Conclusion

In this chapter, we introduced our vision for progressive resizing. We made progressive resizing accessible by prioritising flexibility and ease-of-use, resulting in a drop-in replacement for PyTorch data loaders. We evaluated the performance of our method in a set of initial experiments, comparing it to the state-of-the-art, and uncovering the effects resizing has on software and hardware. Furthermore, we identify the requirements and a path to make progressive resizing more automatic for end users. In combination with our vision as described in Section 6.5, we hope that our method will become a valuable tool for researchers to evaluate their models faster while tuning them.

Chapter 7

Future Directions and Conclusion

For the sustainability of machine learning, we need to be resource-aware. Deep learning training is growing in scale and machine learning models are used more than ever before. We are facing an ever-increasing breadth of deep learning practitioners in an ever-increasing variety of fields; the scale only leads to more importance for resource-awareness. While the machine learning systems community enjoys sharing new systems and architectures with each other, the validity of any system lies in the hands of the end users. Making systems that are useful and accessible to those users is thus critical.

Besides the papers that have been produced for this thesis, we believe in the advantages of open-source software and hope that making all of our software accessible in such a way leads to further innovation and improvement.

7.1 Experiment Tracking

radT introduces a framework for collecting and visualising training statistics, as introduced in Chapter 3, as well as managing data storage and serving. It includes support for a selection of resource metrics, most of which revolve around NVIDIA hardware. The design of the tooling does not need to be limited to these, and any inclusion is highly appreciated.

As the field matures, more sophisticated metrics become available. While basic metrics offered by tools such as SMI provide valuable insight, they tend to only provide rough indications and may not align with the actual demands of the resource-aware community. As an example, the Carbontracker tool allows for estimating the carbon footprint of model training [90]. Instead of just relying on hardware readings, Carbontracker takes into account the emissions required for power production based on location and time of day. Collection and tracking go hand in hand; the validity of any tool to promote resource-awareness depends on the inclusion of new trackers such as Carbontracker to provide the most actionable data. In general, any command-line tool available on Linux should be straightforward to add to radT.

Improving tooling is a critical step for resource-awareness and transparency in the computational footprint of ML, but does not cover the whole picture. Significant gains might be possible in looking into how awareness is communicated in the field.

Change requires not just technical solutions, but also awareness, education, and collaboration. Resource awareness in IT will require a cross-disciplinary approach going forward, spearheaded by organisations such as ITU's Centre for Climate IT [1].

7.2 Sharing in Deep Learning

In the case of GPU utilisation maximisation, a good amount of this thesis has been spent on advocating for the use of GPU collocation. The reality is that GPU hardware is expensive, both financially and environmentally, and thus any hardware that is procured should be used to its utmost capacity. GPU collocation (Chapter 4) is just one of many ways to do so, but it is a way that seems relatively low-effort to implement. Combining training with inference, similar to Orion [112], and collocating on hardware from other manufacturers such as AMD provide future opportunities to make GPU collocation more accessible and effective.

TENSORSOCKET , as introduced in Chapter 5, provides a slightly more intrusive but more effective way of getting the most out of hardware, building on collocation. When designing such software, we realised that alternatives such as CoorDL [87] above all do not make themselves attractive to users due to their inflexible design and implementation. TENSORSOCKET circumvents this issue by being a minimal insert in PyTorch. This, however, does limit its use to that specific framework, so a version for other popular frameworks such as TensorFlow [47] and MLFlow [80] may prove attractive.

Furthermore, TENSORSOCKET notion of sharing is limited to just the data pipeline in our current implementation. Considering the amount of moving parts in the full training pipeline, there may be the opportunity to isolate and unify other redundancies. For example, models that are training at the same time might be able to share some layers, perhaps frozen, in order to reduce computational complexity. Other development directions include the support for distributed multi-node training, with similarities to e.g. the tf.data service [91].

7.3 Data Selection and Attribution

Firstly, progressive resizing, featured in Chapter 6, provides a general implementation to accelerate training from the perspective of data loading. With this implementation, it can be considered a special case of data selection. In data selection, algorithms are utilised that select optimal training examples to train on. This can lead to a more effective ordering of training samples or a reduction in the effective dataset size. There has been extensive research on strategies for selecting datapoints [153, 154, 150, 152, 155, 151, 156, 157, 158], but many are incompatible with resource-awareness. Most of such techniques require elaborate algorithms in order to decide what samples to select, negating any advantages in terms of resource consumption reduction [152], while not reporting this negative effect on their end-to-end computational footprint. It remains a challenge to combine effective selection with resource investments that do not exceed the selection benefits.

[1] https://ccit.itu.dk/

Furthermore, the field of data attribution [159, 160, 161] is closely related to data selection, as it attempts to explain which training samples contribute to what validation samples. In this vein, attribution can be a valuable key in understanding the effect of data optimisations on the training process. In the case of data scaling strategies, attribution might help in uncovering the black box of training by providing an understanding of what scaling strategy is optimal and how small images compare to large images in terms of attribution.

Finally, in this thesis, deep learning training is considered a process that takes place before serving the model. In practice, however, datasets keep evolving and tasks keep changing. This results in models growing out of date and requires the models to be retrained. Continuous machine learning is closely related to data selection, as it is key to deciding whether there is enough new data to warrant retraining the model. Modyn [162, 163] is a continuous learning platform that focuses on data selection and triggering policies, aiming to minimise the number of resources required to keep a model up-to-date.

7.4 Thesis Summary

This thesis contributes to *resource-aware machine learning* by thoroughly investigating several methods to do more training with fewer resources. Deep Learning has recently seen a massive boom with the introduction and exploitation of the GPU and massive amounts of data. While having been a massive contributor to pushing the state-of-the-art, we cannot endlessly keep scaling to increasingly larger compute clusters.

We identify the problem that resource consumption is not a target for optimisation as it is under-reported, and the collection of resource metrics can be a hassle. To combat this issue, this thesis introduces radT in Chapter 3, a tracking and visualisation framework designed to make tracking hardware metrics hassle-free for deep learning practitioners. Taking to heart as a rule for building systems, we ensure that radT, and the other software contributions, requires minimal installation and integrates seamlessly with popular tooling such as PyTorch and MLFlow.

Using radT, we benchmark available technologies for GPU collocation and provide guidelines for deep learning practitioners. GPU hardware is expensive, and any resources procured should be used with optimal efficiency. GPU collocation provides a way of ensuring that GPU systems are not underutilised, especially when training small models or models that supplement each other's utilisation pattern.

Expanding on GPU collocation, we identified further inefficiencies by exploring data redundancies during collocated training. The result is TENSORSOCKET, a library that orchestrates data pipeline sharing for collocated training processes. We ensure that TENSORSOCKET is easy to use yet flexible, allowing for even advanced sharing, such as CLIP models for DALL-E training.

Finally, we continued exploring data loading with progressive data loading, which directly tackles the time it takes to train a convolutional model. By valuing *time-to-accuracy* over just accuracy, we ensure that we get to a high-accuracy model by using fewer resources. We explore the effects the different hyperparameters have on

progressive learning and distil our findings into the development of a progressive data loading library.

Moving forward, we hope that these findings and resources contribute to enabling more resource-aware and transparent growth in deep learning.

Bibliography

[1] Cody Coleman et al. 2019. Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark. *ACM SIGOPS Operating Systems Review*, 53, 1, (July 25, 2019), 14–25. DOI: 10.1145/3352020.3352024.

[2] Peter Mattson et al. MLPerf Training Benchmark. (Mar. 2, 2020) http://arxiv.org/abs/1910.01500 arXiv: 1910.01500 [cs, stat].

[3] Robert Adolf, Saketh Rama, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2016. Fathom: Reference Workloads for Modern Deep Learning Methods. *2016 IEEE International Symposium on Workload Characterization (IISWC)*, (Sept. 2016), 1–10. arXiv: 1608.06581. DOI: 10.1109/IISWC.2016.7581275.

[4] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and Policy Considerations for Deep Learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics. Association for Computational Linguistics, Florence, Italy, 3645–3650. DOI: 10.18653/v1/P19-1355.

[5] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon Emissions and Large Neural Network Training, 22.

[6] Sebastian Baunsgaard, Sebastian B. Wrede, and Pınar Tozun. Training for Speech Recognition on Coprocessors. (Mar. 22, 2020) http://arxiv.org/abs/2003.12366 arXiv: 2003.12366 [cs, eess, stat].

[7] Cody Coleman et al. DAWNBench: An End-to-End Deep Learning Benchmark and Competition, 10.

[8] Olga Russakovsky et al. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115, 3, (Dec. 1, 2015), 211–252. DOI: 10.1007/s11263-015-0816-y.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (NIPS'12). Curran Associates Inc., Red Hook, NY, USA, (Dec. 3, 2012), 1097–1105.

[10] Papers with Code - The latest in Machine Learning. https://paperswithcode.com/.

[11] Nestor Maslej et al. 2024. Artificial Intelligence Index Report 2024. (May 29, 2024). Pre-published.

[12] Gordon E. Moore. 1998. Cramming More Components Onto Integrated Circuits. *Proc. IEEE*, 86, 1, 82–85. DOI: 10.1109/JPROC.1998.658762.

[13] 2024. Building Meta's GenAI Infrastructure. Engineering at Meta. (Mar. 12, 2024). https://engineering.fb.com/2024/03/12/data-center-engineering/building-metas-genai-infrastructure/.

[14] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. 2019. Green AI. (Aug. 13, 2019). arXiv: 1907.10597 [cs, stat] http://arxiv.org/abs/1907.10597. Pre-published.

[15] Jesse Dodge et al. 2022. Measuring the Carbon Intensity of AI in Cloud Instances. In *2022 ACM Conference on Fairness, Accountability, and Transparency*. FAccT '22: 2022 ACM Conference on Fairness, Accountability, and Transparency. ACM, Seoul Republic of Korea, (June 21, 2022), 1877–1894. ISBN: 978-1-4503-9352-2. DOI: 10.1145/3531146.3533234.

[16] 2015. NVIDIA DCGM. NVIDIA Developer. (Nov. 10, 2015). https://developer.nvidia.com/dcgm.

[17] 2012. NVIDIA System Management Interface. NVIDIA Developer. (June 28, 2012). https://developer.nvidia.com/nvidia-system-management-interface.

[18] Top(1) - Linux manual page. https://man7.org/linux/man-pages/man1/top.1.html.

[19] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. Dahlia Malkhi and Dan Tsafrir, editors. USENIX Association, 947–960. https://www.usenix.org/conference/atc19/presentation/jeon.

[20] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-Aware Cluster Management, 17.

[21] Konstantinos Nikas, Nikela Papadopoulou, Dimitra Giantsidi, Vasileios Karakostas, Georgios Goumas, and Nectarios Koziris. 2019. DICER: Diligent Cache Partitioning for Efficient Workload Consolidation. In *Proceedings of the 48th International Conference on Parallel Processing*. ICPP 2019: 48th International Conference on Parallel Processing. ACM, Kyoto Japan, (Aug. 5, 2019), 1–10. ISBN: 978-1-4503-6295-5. DOI: 10.1145/3337821.3337891.

[22] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 770–778. DOI: 10.1109/CVPR.2016.90.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity Mappings in Deep Residual Networks. In *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV* (Lecture Notes in Computer Science). Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors. Vol. 9908. Springer, 630–645. DOI: 10.1007/978-3-319-46493-0_38.

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60, 6, (May 24, 2017), 84–90. DOI: 10.1145/3065386.

[26] Nello Cristianini and Elisa Ricci. 2008. Support Vector Machines. In *Encyclopedia of Algorithms*. Ming-Yang Kao, editor. Springer US, Boston, MA, 928–932. ISBN: 978-0-387-30162-4. DOI: 10.1007/978-0-387-30162-4_415.

[27] Leo Breiman. 2001. Random Forests. *Machine Learning*, 45, 1, (Oct. 1, 2001), 5–32. DOI: 10.1023/A:1010933404324.

[28] David Silver et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529, 7587, (Jan. 2016), 484–489, 7587, (Jan. 2016). DOI: 10.1038/nature16961.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, 1026–1034.

[30] Hugo Touvron et al. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR*, abs/2302.13971. arXiv: 2302.13971. DOI: 10.48550/ARXIV.2302.13971.

[31] Hugo Touvron et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR*, abs/2307.09288. arXiv: 2307.09288. DOI: 10.48550/ARXIV.2307.09288.

[32] Abhimanyu Dubey et al. 2024. The Llama 3 Herd of Models. (July 31, 2024). arXiv: 2407.21783 [cs] http://arxiv.org/abs/2407.21783. Pre-published.

[33] Iostat(1) - Linux manual page. https://linux.die.net/man/1/iostat.

[34] Ehsan Yousefzadeh-Asl-Miandoab, Ties Robroek, and Pinar Tozun. 2023. Profiling and Monitoring Deep Learning Training Tasks. In *Proceedings of the 3rd Workshop on Machine Learning and Systems*. EuroMLSys '23: 3rd Workshop on Machine Learning and Systems. ACM, Rome Italy, (May 8, 2023), 18–25. DOI: 10.1145/3578356.3592589.

[35] Zeyu Yang, Karel Adamek, and Wesley Armour. 2024. Part-time Power Measurements: nvidia-smi's Lack of Attention. (Mar. 11, 2024). arXiv: 2312.02741 [cs] http://arxiv.org/abs/2312.02741. Pre-published.

[36] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[37] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter Pietzuch. 2019. Crossbow: scaling deep learning with small batch sizes on multi-GPU servers. *Proceedings of the VLDB Endowment*, 12, 11, (July 2019), 1399–1412. DOI: 10.14778/3342263.3342276.

[38] Shang Wang, Peiming Yang, Yuxuan Zheng, Xin Li, and Gennady Pekhimenko. Horizontally Fused Training Array: An Effective Hardware Utilization Squeezer for Training Novel Deep Learning Models. (Mar. 29, 2021) http://arxiv.org/abs/2102.02344 arXiv: 2102.02344 [cs].

[39] Jun Yuan, Changjian Chen, Weikai Yang, Mengchen Liu, Jiazhi Xia, and Shixia Liu. 2021. A survey of visual analytics techniques for machine learning. *Computational Visual Media*, 7, 1, (Mar. 2021), 3–36. DOI: 10.1007/s41095-020-0191-7.

[40] 2020. Experiment Tracking with Weights and Biases. Weights & Biases. https://wandb.ai/site.

[41] Matei Zaharia et al. 2018. Accelerating the machine learning lifecycle with MLflow. *IEEE Data Eng. Bull.*, 41, 4, 39–45.

[42] Mourad Mourafiq. Polyaxon: Cloud native machine learning platform. https://github.com/polyaxon/polyaxon.

[43] Kubeflow. Kubeflow. https://www.kubeflow.org/.

[44] [SW], UMLAUT (Universal Machine Learning Analysis UTility) 2023. HPI Data Engineering Systems. URL: https://github.com/hpides/End-to-end-ML-System-Benchmark.

[45] Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, and Tim Kraska. 2019. Democratizing Data Science through Interactive Curation of ML Pipelines. In *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD/PODS '19: International Conference on Management of Data. ACM, Amsterdam Netherlands, (June 25, 2019), 1171–1188. ISBN: 978-1-4503-5643-5. DOI: 10.1145/3299869.3319863.

[46] Jorge Piazentin Ono, Sonia Castelo, Roque Lopez, Enrico Bertini, Juliana Freire, and Claudio Silva. 2021. PipelineProfiler: A Visual Analytics Tool for the Exploration of AutoML Pipelines. *IEEE Transactions on Visualization and Computer Graphics*, 27, 2, (Feb. 2021), 390–400. DOI: 10.1109/TVCG.2020.3030361.

[47] Martın Abadi et al. TensorFlow: A system for large-scale machine learning, 21.

[48] Chollet, François and others. 2020. Keras: the Python deep learning API. https://keras.io/.

[49] Adam Paszke et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

[50] NVIDIA Multi-Instance GPU User Guide. http://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html.

[51] NVIDIA Multi-Process Service User Guide. https://docs.nvidia.com/deploy/mps/index.html.

[52] 2018. NVIDIA Nsight Systems. NVIDIA Developer. (Mar. 12, 2018). https://developer.nvidia.com/nsight-systems.

[53] 2019. NVIDIA Nsight Compute. NVIDIA Developer. (Aug. 28, 2019). https://developer.nvidia.com/nsight-compute.

[54] Ties Robroek, Ehsan Yousefzadeh-Asl-Miandoab, and Pınar Tözün. 2024. An Analysis of Collocation on GPUs for Deep Learning Training. In *Proceedings of the 4th Workshop on Machine Learning and Systems*. EuroSys '24: Nineteenth European Conference on Computer Systems. ACM, Athens Greece, (Apr. 22, 2024), 81–90. DOI: 10.1145/3642970.3655827.

[55] 2015. GPU Pro Tip: CUDA 7 Streams Simplify Concurrency. NVIDIA Technical Blog. (Jan. 23, 2015). https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/.

[56] 2024. NVIDIA Multi-Process Service.

[57] Qizhen Weng et al. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*. Amar Phanishayee and Vyas Sekar, editors. USENIX Association, 945–960. https://www.usenix.org/conference/nsdi22/presentation/weng.

[58] Mehmet E. Belviranli, Farzad Khorasani, Laxmi N. Bhuyan, and Rajiv Gupta. 2016. CuMAS: Data Transfer Aware Multi-Application Scheduling for Shared GPUs. In *Proceedings of the 2016 International Conference on Supercomputing*. ICS '16: 2016 International Conference on Supercomputing. ACM, Istanbul Turkey, (June 2016), 1–12. ISBN: 978-1-4503-4361-9. DOI: 10.1145/2925426.2926271.

[59] Vignesh T. Ravi, Michela Becchi, Gagan Agrawal, and Srimat Chakradhar. 2011. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*. HPDC '11: The 20th International Symposium on High-Performance Parallel and Distributed Computing. ACM, San Jose California USA, (June 8, 2011), 217–228. ISBN: 978-1-4503-0552-5. DOI: 10.1145/1996130.1996160.

[60] Gingfung Yeung, Damian Borowiec, Renyu Yang, Adrian Friday, Richard Harper, and Peter Garraghan. 2022. Horus: Interference-Aware and Prediction-Based Scheduling in Deep Learning Systems. *IEEE Transactions on Parallel and Distributed Systems*, 33, 1, (Jan. 2022), 88–100. DOI: 10.1109/TPDS.2021.3079202.

[61] Sina Darabi, Negin Mahani, Hazhir Baxishi, Ehsan Yousefzadeh-Asl-Miandoab, Mohammad Sadrosadati, and Hamid Sarbazi-Azad. 2022. NURA: A Framework for Supporting Non-Uniform Resource Accesses in GPUs. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6, 1, (Feb. 24, 2022), 1–27. DOI: 10.1145/3508036.

[62] Hongwen Dai, Zhen Lin, Chao Li, Chen Zhao, Fei Wang, Nanning Zheng, and Huiyang Zhou. 2018. Accelerate GPU Concurrent Kernel Execution by Mitigating Memory Pipeline Stalls. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA). (Feb. 2018), 208–220. DOI: 10.1109/HPCA.2018.00027.

[63] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2016. Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). (Mar. 2016), 358–369. DOI: 10.1109/HPCA.2016.7446078.

[64] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2017. Quality of Service Support for Fine-Grained Sharing on GPUs. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17: The 44th Annual International Symposium on Computer Architecture. ACM, Toronto ON Canada, (June 24, 2017), 269–281. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080203.

[65] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. 2016. Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). IEEE, Seoul, South Korea, (June 2016), 230–242. ISBN: 978-1-4673-8947-1. DOI: 10.1109/ISCA.2016.29.

[66] Xia Zhao, Zhiying Wang, and Lieven Eeckhout. 2018. Classification-Driven Search for Effective SM Partitioning in Multitasking GPUs. In *Proceedings of the 2018 International Conference on Supercomputing*. ICS '18: 2018 International Conference on Supercomputing. ACM, Beijing China, (June 12, 2018), 65–75. ISBN: 978-1-4503-5783-8. DOI: 10.1145/3205289.3205311.

[67] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. 2022. MISO: exploiting multi-instance GPU capability on multi-tenant GPU clusters. In *Proceedings of the 13th Symposium on Cloud Computing*. SoCC '22: ACM Symposium on Cloud Computing. ACM, San Francisco California, (Nov. 7, 2022), 173–189. ISBN: 978-1-4503-9414-7. DOI: 10.1145/3542929.3563510.

[68] Baolin Li, Viiay Gadepally, Siddharth Samsi, and Devesh Tiwari. 2022. Characterizing Multi-Instance GPU for Machine Learning Workloads. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). (May 2022), 724–731. DOI: 10.1109/IPDPSW55747.2022.00124.

[69] AMD EPYC 7742. AMD. https://www.amd.com/en/products/specifications/server-processor.html.

[70] NVIDIA. 2022. Data Center GPU Manager Documentation. http://docs.nvidia.com/datacenter/dcgm/dcgm-user-guide/index.html.

[71] Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. 2022. Image Segmentation Using Deep Learning: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44, 7, (July 2022), 3523–3542. DOI: 10.1109/TPAMI.2021.3059968.

[72] Lars Schmarje, Monty Santarossa, Simon-Martin Schröder, and Reinhard Koch. 2021. A Survey on Semi-, Self- and Unsupervised Learning for Image Classification. *IEEE Access*, 9, 82146–82168. DOI: 10.1109/ACCESS.2021.3084358.

[73] Mingxing Tan and Quoc V. Le. 2021. EfficientNetV2: Smaller Models and Faster Training. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event* (Proceedings of Machine Learning Research). Marina Meila and Tong Zhang, editors. Vol. 139. PMLR, 10096–10106. http://proceedings.mlr.press/v139/tan21a.html.

[74] Hugo Touvron, Matthieu Cord, Alexandre Sablayrolles, Gabriel Synnaeve, and Hervé Jégou. 2021. Going Deeper with Image Transformers. (Apr. 7, 2021). arXiv: 2103.17239 [cs] http://arxiv.org/abs/2103.17239. Pre-published.

[75] Maxim Naumov et al. Deep learning recommendation model for personalization and recommendation systems. (2019). arXiv: 1906.00091.

[76] Criteo. 2015. Criteo Releases Industry's Largest-Ever Dataset for Machine Learning to Academic Community. Criteo. (July 18, 2015). https://www.criteo.com/news/press-releases/2015/07/criteo-releases-industrys-largest-ever-dataset/.

[77] Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images.

[78] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. A downsampled variant of imagenet as an alternative to the cifar datasets. (2017). arXiv: 1707.08819.

[79] Ties Robroek, Aaron Duane, Ehsan Yousefzadeh-Asl-Miandoab, and Pinar Tozun. 2023. Data Management and Visualization for Benchmarking Deep Learning Training Systems. In *Proceedings of the Seventh Workshop on Data Management for End-to-End Machine Learning*. DEEM '23: Seventh Workshop on Data Management for End-to-End Machine Learning. ACM, Seattle WA USA, (June 18, 2023), 1–5. DOI: 10.1145/3595360.3595851.

[80] MLflow - An open source platform for the machine learning lifecycle. MLflow. https://mlflow.org/.

[81] [SW] Ross Wightman, PyTorch Image Models. Hugging Face. URL: https://github.com/huggingface/pytorch-image-models.

[82] Ekin D. Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V. Le. 2019. AutoAugment: Learning Augmentation Strategies From Data. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, Long Beach, CA, USA, (June 2019), 113–123. ISBN: 978-1-72813-293-8. DOI: 10.1109/CVPR.2019.00020.

[83] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A Thekkath, and Ana Klimovic. Cachew: Machine Learning Input Data Processing as a Service.

[84] Alexander Isenko, Ruben Mayer, Jeffrey Jedele, and Hans-Arno Jacobsen. 2022. Where Is My Training Bottleneck? Hidden Trade-Offs in Deep Learning Preprocessing Pipelines. In *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD/PODS '22: International Conference on Management of Data. ACM, Philadelphia PA USA, (June 10, 2022), 1825–1839. ISBN: 978-1-4503-9249-5. DOI: 10.1145/3514221.3517848.

[85] Amazon EC2. Amazon Web Services, Inc. https://aws.amazon.com/ec2/.

[86] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. 2021. Tf.data: A Machine Learning Data Processing Framework. *Proceedings of the VLDB Endowment*, 14, 12, (July 2021), 2945–2958. DOI: 10.14778/3476311.3476374.

[87] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2021. Analyzing and mitigating data stalls in DNN training. *Proceedings of the VLDB Endowment*, 14, 5, (Jan. 2021), 771–784. DOI: 10.14778/3446095.3446100.

[88] Fabio Maschi and Gustavo Alonso. 2023. The Difficult Balance Between Modern Hardware and Conventional CPUs. In *Proceedings of the 19th International Workshop on Data Management on New Hardware*. SIGMOD/PODS '23: International Conference on Management of Data. ACM, Seattle WA USA, (June 18, 2023), 53–62. DOI: 10.1145/3592980.3595314.

[89] David Patterson et al. 2022. The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink. (Apr. 11, 2022). arXiv: 2204.05149 [cs] http://arxiv.org/abs/2204.05149. Pre-published.

[90] Lasse F. Wolff Anthony, Benjamin Kanding, and Raghavendra Selvan. 2020. Carbontracker: Tracking and Predicting the Carbon Footprint of Training Deep Learning Models. (July 6, 2020). arXiv: 2007.03051 [cs, eess, stat] http://arxiv.org/abs/2007.03051. Pre-published.

[91] Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiří Šimša, and Chandramohan A. Thekkath. 2023. Tf.data service: A Case for Disaggregating ML Input Data Processing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*. SoCC '23: ACM Symposium on Cloud Computing. ACM, Santa Cruz CA USA, (Oct. 30, 2023), 358–375. DOI: 10.1145/3620678.3624666.

[92] Aarati Kakaraparthy, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. 2019. The Case for Unifying Data Loading in Machine Learning Clusters. In *11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2019, Renton, WA, USA, July 8, 2019*. Christina Delimitrou and Dan R. K. Ports, editors. USENIX Association. https://www.usenix.org/conference/hotcloud19/presentation/kakaraparthy.

[93] Jingwei Xu, Guochang Wang, Yuan Yao, Zenan Li, Chun Cao, and Hanghang Tong. 2022. A Deep Learning Dataloader with Shared Data Preparation. *Advances in Neural Information Processing Systems*, 35, 17146–17156. https://proceedings.neurips.cc/paper_files/paper/2022/hash/6d538a6e667960b168d3d947eb6207a6-Abstract-Conference.html.

[94] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. 2005. QPipe: a simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD/PODS05: International Conference on Management of Data and Symposium on Principles Database and Systems. ACM, Baltimore Maryland, (June 14, 2005), 383–394. ISBN: 978-1-59593-060-6. DOI: 10.1145/1066157.1066201.

[95] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2012. SharedDB: Killing One Thousand Queries With One Stone. (Feb. 29, 2012). arXiv: 1203.0056 [cs] http://arxiv.org/abs/1203.0056. Pre-published.

[96] Pinar Tözün, Islam Atta, Anastasia Ailamaki, and Andreas Moshovos. 2014. ADDICT: advanced instruction chasing for transactions. *Proceedings of the VLDB Endowment*, 7, 14, (Oct. 2014), 1893–1904. DOI: 10.14778/2733085.2733095.

[97] Iraklis Psaroudakis, Manos Athanassoulis, and Anastasia Ailamaki. 2013. Sharing data and work across concurrent analytical queries. *Proceedings of the VLDB Endowment*, 6, 9, (July 2013), 637–648. DOI: 10.14778/2536360.2536364.

[98] Microsoft Azure Virtual Machines. https://azure.microsoft.com/en-us/products/virtual-machines.

[99] Google Cloud Platform Compute Engine. https://cloud.google.com/products/compute?hl=en.

[100] Francesco Ventura, Zoi Kaoudi, Jorge Arnulfo Quiané-Ruiz, and Volker Markl. 2021. Expand your Training Limits! Generating Training Data for ML-based Data Management. In *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD/PODS '21: International Conference on Management of Data. ACM, Virtual Event China, (June 9, 2021), 1865–1878. ISBN: 978-1-4503-8343-1. DOI: 10.1145/3448016.3457286.

[101] Mark Zhao et al. 2022. Understanding data storage and ingestion for large-scale deep recommendation model training: industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ISCA '22: The 49th Annual International Symposium on Computer Architecture. ACM, New York New York, (June 18, 2022), 1042–1057. ISBN: 978-1-4503-8610-4. DOI: 10.1145/3470496.3533044.

[102] [SW], GitHub: NVIDIA DALI May 19, 2023. NVIDIA Corporation. URL: https://github.com/NVIDIA/DALI.

[103] Taeyoon Kim, ChanHo Park, Heelim Hong, Minseok Kim, Ze Jin, Changdae Kim, Ji-Yong Shin, and Myeongjae Jeon. FusionFlow: Accelerating Data Preprocessing for Machine Learning with CPU-GPU Cooperation.

[104] Taegeon Um, Byungsoo Oh, Byeongchan Seo, Minhyeok Kweun, Goeun Kim, and Woo-Yeon Lee. 2023. FastFlow: Accelerating Deep Learning Model Training with Smart Offloading of Input Data Pipeline. *Proceedings of the VLDB Endowment*, 16, 5, (Jan. 2023), 1086–1099. DOI: 10.14778/3579075.3579083.

[105] Philipp Probst, Anne-Laure Boulesteix, and Bernd Bischl. 2019. Tunability: Importance of Hyperparameters of Machine Learning Algorithms. *J. Mach. Learn. Res.*, 20, 53:1–53:32. http://jmlr.org/papers/v20/18-444.html.

[106] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-Tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A System for Massively Parallel Hyperparameter Tuning, 17.

[107] Chenxi Liu et al. 2018. Progressive Neural Architecture Search. In *Computer Vision – ECCV 2018*. Vol. 11205. Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors. Springer International Publishing, Cham, 19–35. DOI: 10.1007/978-3-030-01246-5_2.

[108] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2020. Cerebro: a data system for optimized deep learning model selection. *Proceedings of the VLDB Endowment*, 13, 12, (Aug. 2020), 2159–2173. DOI: 10.14778/3407790.3407816.

[109] [SW] Microsoft Research, Coordinated Data Loader: CoorDL 2020. URL: https://github.com/msr-fiddle/CoorDL.

[110] [SW], Joader Codebase July 10, 2024. URL: https://github.com/XieJiann/Joader.

[111] 2023. ZeroMQ. (Dec. 11, 2023). https://zeromq.org/.

[112] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 1075–1092. DOI: 10.1145/3627703.3629578.

[113] Connor Espenshade, Rachel Peng, Eumin Hong, Max Calman, Yue Zhu, Pritish Parida, Eun Kyung Lee, and Martha A. Kim. 2024. Characterizing Training Performance and Energy for Foundation Models and Image Classifiers on Multi-Instance GPUs. In *Proceedings of the 4th Workshop on Machine Learning and Systems*. EuroSys '24: Nineteenth European Conference on Computer Systems. ACM, Athens Greece, (Apr. 22, 2024), 47–55. DOI: 10.1145/3642970.3655830.

[114] Lennart Behme, Saravanan Thirumuruganathan, Alireza Rezaei Mahdiraji, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2023. The Art of Losing to Win: Using Lossy Image Compression to Improve Data Loading in Deep Learning Pipelines. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2023 IEEE 39th International Conference on Data Engineering (ICDE). (Apr. 2023), 936–949. DOI: 10.1109/ICDE55515.2023.00077.

[115] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. 2022. Hierarchical Text-Conditional Image Generation with CLIP Latents. (Apr. 12, 2022). arXiv: 2204.06125 [cs] http://arxiv.org/abs/2204.06125. Pre-published.

[116] Janne Spijkervet and John Ashley Burgoyne. 2021. Contrastive Learning of Musical Representations. In *Proceedings of the 22nd International Society for Music Information Retrieval Conference*. ISMIR, (Oct. 2021), 673–681. DOI: 10.5281/zenodo.5624573.

[117] [SW] Phil Wang, DALL-E 2 - Pytorch 2023. URL: https://github.com/lucidrains/DALLE2-pytorch.

[118] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. 2015. Librispeech: An ASR corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). (Apr. 2015), 5206–5210. DOI: 10.1109/ICASSP.2015.7178964.

[119] Piyush Sharma, Nan Ding, Sebastian Goodman, and Radu Soricut. 2018. Conceptual Captions: A Cleaned, Hypernymed, Image Alt-text Dataset For Automatic Image Captioning. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Association for Computational Linguistics, Melbourne, Australia, 2556–2565. DOI: 10.18653/v1/P18-1238.

[120] AWS EC2 Pricing. Amazon Web Services, Inc. https://aws.amazon.com/ec2/pricing/on-demand/.

[121] Anyscale. Ray Tune: Hyperparameter Tuning. https://docs.ray.io/en/latest/tune/index.html.

[122] Stavros Harizopoulos and Anastassia Ailamaki. 2005. StagedDB: Designing Database Servers for Modern Hardware. *IEEE Data Eng. Bull.*, 28, 2, 11–16.

[123] Ryan Johnson, Nikos Hardavellas, Ippokratis Pandis, Naju Mancheril, Stavros Harizopoulos, Kivanc Sabirli, Anastassia Ailamaki, and Babak Falsafi. 2007. To Share or Not To Share? In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*. Christoph Koch et al., editors. ACM, 351–362. http://www.vldb.org/conf/2007/papers/research/p351-johnson.pdf.

[124] Christoph Brücke, Philipp Härtling, Rodrigo D Escobar Palacios, Hamesh Patel, and Tilmann Rabl. 2023. TPCx-AI - An Industry Standard Benchmark for Artificial Intelligence and Machine Learning Systems. *Proceedings of the VLDB Endowment*, 16, 12, (Aug. 2023), 3649–3661. DOI: 10.14778/3611540.3611554.

[125] Oana Balmau. 2022. Characterizing I/O in Machine Learning with MLPerf Storage. *ACM SIGMOD Record*, 51, 3, (Nov. 21, 2022), 47–48. DOI: 10.1145/3572751.3572765.

[126] MLPerf Storage Benchmark Suite Results. MLCommons. https://mlcommons.org/benchmarks/storage/.

[127] Jeremy Howard and Sylvain Gugger. 2020. Fastai: A layered API for deep learning. *Information*, 11, 2, 108.

[128] Jeremy Howard. 2018. Fast.ai - Training Imagenet in 3 hours for USD 25; and CIFAR10 for USD 0.26. fast.ai. (Apr. 30, 2018). https://www.fast.ai/posts/2018-04-30-dawnbench-fastai.html.

[129] Fernando Pérez-García, Rachel Sparks, and Sébastien Ourselin. 2021. TorchIO: a Python library for efficient loading, preprocessing, augmentation and patch-based sampling of medical images in deep learning. *Computer Methods and Programs in Biomedicine*, 208, 106236.

[130] Yang You, Igor Gitman, and Boris Ginsburg. Scaling SGD Batch Size to 32K for ImageNet Training. (2017). arXiv: 1708.03888.

[131] Yang You et al. 2020. Large Batch Optimization for Deep Learning: Training BERT in 76 minutes. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=Syx4wnEtvH.

[132] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2018. Imagenet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*, 1–10.

[133] Aditya Devarakonda, Maxim Naumov, and Michael Garland. 2017. Adabatch: Adaptive batch sizes for training deep neural networks. *CoRR*, abs/1712.02029.

[134] Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le. 2018. Don't Decay the Learning Rate, Increase the Batch Size. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=B1Yy1BxCZ.

[135] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely large minibatch sgd: Training resnet-50 on imagenet in 15 minutes. (2017). arXiv: 1711.04325.

[136] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997. http://arxiv.org/abs/1404.5997 arXiv: 1404.5997.

[137] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. (2018). arXiv: 1804.07612.

[138] Elad Hoffer, Berry Weinstein, Itay Hubara, Tal Ben-Nun, Torsten Hoefler, and Daniel Soudry. Mix & match: training convnets with mixed image sizes for improved accuracy, speed and scale resiliency. (2019). arXiv: 1908.08986.

[139] Hossein Talebi and Peyman Milanfar. 2021. Learning to Resize Images for Computer Vision Tasks. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. 2021 IEEE/CVF International Conference on Computer Vision (ICCV). IEEE, Montreal, QC, Canada, (Oct. 2021), 487–496. ISBN: 978-1-66542-812-5. DOI: 10.1109/ICCV48922.2021.00055.

[140] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*. PMLR, 6105–6114.

[141] Ammar Mahmood, Ana Giraldo Ospina, Mohammed Bennamoun, Senjian An, Ferdous Sohel, Farid Boussaid, Renae Hovey, Robert B. Fisher, and Gary A. Kendrick. 2020. Automatic hierarchical classification of kelps using deep residual features. *Sensors*, 20, 2, 447. https://www.mdpi.com/1424-8220/20/2/447.

[142] [SW] Ross Wightman, PyTorch Image Models Jan. 24, 2022. URL: https://github.com/rwightman/pytorch-image-models.

[143] Fastxtend. https://fastxtend.benjaminwarner.dev/.

[144] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, 41–48.

[145] Amelia Jiménez-Sánchez, Diana Mateus, Sonja Kirchhoff, Chlodwig Kirchhoff, Peter Biberthaler, Nassir Navab, Miguel Ángel González Ballester, and Gemma Piella. 2022. Curriculum learning for improved femur fracture classification: Scheduling data with prior knowledge and uncertainty. *Medical Image Anal.*, 75, 102273. DOI: 10.1016/J.MEDIA.2021.102273.

[146] Yuxing Tang, Xiaosong Wang, Adam P. Harrison, Le Lu, Jing Xiao, and Ronald M. Summers. 2018. Attention-guided curriculum learning for weakly supervised classification and localization of thoracic diseases on chest radiographs. In *Machine Learning in Medical Imaging: 9th International Workshop, MLMI 2018, Held in Conjunction with MICCAI 2018, Granada, Spain, September 16, 2018, Proceedings 9*. Springer, 249–258.

[147] Yiru Wang, Weihao Gan, Jie Yang, Wei Wu, and Junjie Yan. 2019. Dynamic Curriculum Learning for Imbalanced Data Classification. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*. IEEE, 5016–5025. DOI: 10.1109/ICCV.2019.00512.

[148] Guy Hacohen and Daphna Weinshall. 2019. On The Power of Curriculum Learning in Training Deep Networks. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA* (Proceedings of Machine Learning Research). Kamalika Chaudhuri and Ruslan Salakhutdinov, editors. Vol. 97. PMLR, 2535–2544. http://proceedings.mlr.press/v97/hacohen19a.html.

[149] Tambet Matiisen, Avital Oliver, Taco Cohen, and John Schulman. 2020. Teacher-Student Curriculum Learning. *IEEE Trans. Neural Networks Learn. Syst.*, 31, 9, 3732–3740. DOI: 10.1109/TNNLS.2019.2934906.

[150] Baharan Mirzasoleiman, Jeff Bilmes, and Jure Leskovec. 2020. Coresets for data-efficient training of machine learning models. In *International Conference on Machine Learning*. PMLR, 6950–6960. https://proceedings.mlr.press/v119/mirzasoleiman20a.html.

[151] Omead Pooladzandi, David Davini, and Baharan Mirzasoleiman. 2022. Adaptive second order coresets for data-efficient machine learning. In *International Conference on Machine Learning*. PMLR, 17848–17869. https://proceedings.mlr.press/v162/pooladzandi22a.html.

[152] Patrik Okanovic, Roger Waleffe, Vasilis Mageirakos, Konstantinos E. Nikolakakis, Amin Karbasi, Dionysis Kalogerias, Nezihe Merve Gürel, and Theodoros Rekatsinas. 2023. Repeated Random Sampling for Minimizing the Time-to-Accuracy of Learning. (May 28, 2023). arXiv: 2305.18424 [cs] http://arxiv.org/abs/2305.18424. Pre-published.

[153] Angelos Katharopoulos and François Fleuret. 2018. Not all samples are created equal: Deep learning with importance sampling. In *International Conference on Machine Learning*. PMLR, 2525–2534. http://proceedings.mlr.press/v80/katharopoulos18a.html.

[154] Sören Mindermann et al. 2022. Prioritized training on points that are learnable, worth learning, and not yet learnt. In *International Conference on Machine Learning*. PMLR, 15630–15649. https://proceedings.mlr.press/v162/mindermann22a.html.

[155] Mansheej Paul, Surya Ganguli, and Gintare Karolina Dziugaite. 2021. Deep learning on a data diet: Finding important examples early in training. *Advances in neural information processing systems*, 34, 20596–20607. https://proceedings.neurips.cc/paper/2021/hash/ac56f8fe9eea3e4a365f29f0f1957c55-Abstract.html.

[156] Germain Kolossov, Andrea Montanari, and Pulkit Tandon. 2023. Towards a statistical theory of data selection under weak supervision. (Oct. 4, 2023). arXiv: 2309.14563 [cs, stat] http://arxiv.org/abs/2309.14563. Pre-published.

[157] Logan Engstrom, Axel Feldmann, and Aleksander Madry. 2024. DsDm: Model-Aware Dataset Selection with Datamodels. (Jan. 23, 2024). arXiv: 2401.12926 [cs, stat] http://arxiv.org/abs/2401.12926. Pre-published.

[158] Mengzhou Xia, Sadhika Malladi, Suchin Gururangan, Sanjeev Arora, and Danqi Chen. 2024. LESS: Selecting Influential Data for Targeted Instruction Tuning. (June 12, 2024). arXiv: 2402.04333 [cs] http://arxiv.org/abs/2402.04333. Pre-published.

[159] Sung Min Park, Kristian Georgiev, Andrew Ilyas, Guillaume Leclerc, and Aleksander Madry. 2023. TRAK: Attributing Model Behavior at Scale. (Apr. 3, 2023). arXiv: 2303.14186 [cs, stat] http://arxiv.org/abs/2303.14186. Pre-published.

[160] Pang Wei Koh and Percy Liang. 2017. Understanding black-box predictions via influence functions. In *International Conference on Machine Learning*. PMLR, 1885–1894. http://proceedings.mlr.press/v70/koh17a?ref=https://githubhelp.com.

[161] Andrew Ilyas, Sung Min Park, Logan Engstrom, Guillaume Leclerc, and Aleksander Madry. 2022. Datamodels: Predicting Predictions from Training Data. (Feb. 1, 2022). arXiv: 2202.00622 [cs, stat] http://arxiv.org/abs/2202.00622. Pre-published.

[162] Maximilian Böther, Foteini Strati, Viktor Gsteiger, and Ana Klimovic. 2023. Towards A Platform and Benchmark Suite for Model Training on Dynamic Datasets. In *Proceedings of the 3rd Workshop on Machine Learning and Systems*. EuroMLSys '23: 3rd Workshop on Machine Learning and Systems. ACM, Rome Italy, (May 8, 2023), 8–17. https://dl.acm.org/doi/10.1145/3578356.3592585.

[163] Maximilian Böther, Viktor Gsteiger, Ties Robroek, and Ana Klimovic. 2023. Modyn: A Platform for Model Training on Dynamic Datasets With Sample-Level Data Selection. (Dec. 11, 2023). arXiv: 2312.06254 [cs, stat] http://arxiv.org/abs/2312.06254. Pre-published.