THESIS

CONTAINERIZATION OF MODEL FITTING WORKLOADS OVER SPATIAL DATASETS

Submitted by

Menuka Warushavithana

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2021

ABSTRACT

CONTAINERIZATION OF MODEL FITTING WORKLOADS OVER SPATIAL DATASETS

Spatial data volumes have grown exponentially over the past several years. The number of domains in which spatial data are extensively leveraged include atmospheric sciences, environmental monitoring, ecological modeling, epidemiology, sociology, commerce, and social media among others. These data are often used to understand phenomena and inform decision making by fitting models to them. In this study, we present our methodology to fit models at scale over spatial data. Our methodology encompasses segmentation, spatial similarity based on the dataset(s) under consideration, and transfer learning schemes that are informed by the spatial similarity to train models faster while utilizing fewer resources. We consider several model fitting algorithms and execution within containerized environments as we profile the suitability of our methodology. Our benchmarks validate the suitability of our methodology to facilitate faster, resource-efficient training of models over spatial data.

ACKNOWLEDGEMENTS

This research was supported by the National Science Foundation [OAC-1931363, ACI-1553685] and the National Institute of Food & Agriculture [COL0-FACT-2019].

TABLE OF CONTENTS

ABSTRACT ii
ACKNOWLEDGEMENTS iii
LIST OF TABLES v
LIST OF FIGURES vi

Chapter 1 Introduction 1
1.1 Goals and Objectives 1
1.2 Challenges 2
1.3 Research Questions 2
1.4 Approach Summary 3
1.5 Paper Contributions 4

Chapter 2 Related Work 5

Chapter 3 Methodology 9
3.1 Segmentation 9
3.2 Partitioning and Segmenting Spatial Extents 9
3.3 Rigorous Training of Models 10
3.4 Sizing containers based on distance from Centroids 11
3.5 Warm-starting model training tasks via transfer learning 12
3.6 Inferring Suitable Container Sizes 12

Chapter 4 Performance Benchmarks 14
4.1 Datasets 14
4.2 Experimental Setup 14
4.3 Assessing the Transfer Learning scheme 15
4.3.1 Linear Regression 15
4.3.2 Gradient Boosting 16
4.3.3 Time-Series Regression 16
4.4 Exploration of Container Sizing 17
4.5 Rigorous, Light, and Moderate training using containers 18

Chapter 5 Conclusions and Future Work 28

Bibliography 29

LIST OF TABLES

4.1 Completion Times for different model fitting algorithms on differently sized clusters . 17
4.2 Cluster Sizes 18
4.3 Maximum memory utilization for a single spatial extent 18

LIST OF FIGURES

3.1 Partitioning spatial extents based on their distance from the cluster centroid. 12

4.1 Percentage decrease in convergence time for Linear Regression (over Macav2 dataset) with Transfer Learning. 15
4.2 Percentage decrease in convergence time for Gradient Boosting (over Macav2 dataset) with Transfer Learning 16
4.3 Percentage decrease in convergence time for Time-Series Regression (over COVID-19 dataset) with Transfer Learning. 17
4.4 Convergence times for Linear Regression on NOAA dataset. 19
4.5 Completion times for Linear Regression on Macav2 dataset. 19
4.6 Linear Regression on Macav2 data - Memory Utilization for containerized vs. bare-metal worker nodes. 20
4.7 Linear Regression on Macav2 data - CPU usage for containerized vs. bare-metal worker nodes. 20
4.8 Linear Regression on NOAA data - CPU Usage for a container. 21
4.9 Convergence times for Gradient Boosting over the NOAA dataset. 21
4.10 Convergence times for Gradient Boosting over the Macav2 dataset. 22
4.11 Gradient Boosting on Macav2 data - Memory Utilization for containerized vs. bare-metal worker nodes. 22
4.12 Gradient Boosting on Macav2 data - CPU usage for Containerized (Optimized for time), containerized (Optimized for memory), and bare-metal worker nodes. 23
4.13 Gradient Boosting on NOAA data - CPU Usage for a container. 23
4.14 Gradient Boosting on Macav2 data - CPU Usage for a container. 24
4.15 Completion times for Time-Series Regression on the COVID-19 dataset. 24
4.16 Comparison of CPU Utilization for Time-Series Regression on the COVID-19 dataset (Bare-metal vs. Containerized Dask). 25
4.17 Comparison of Memory Utilization for Time-Series Regression on the COVID-19 dataset (Bare-metal vs. Containerized Dask). 25
4.18 Time-series Regression on the COVID-19 dataset: CPU Usage on a single container. . 25
4.19 Time-series Regression on COVID-19: CPU Usage on a single container. 26
4.20 Completion times for Time-Series Regression on the NOAA dataset. 26
4.21 Time-series Regression on NOAA: CPU Usage on a single container. 27

Chapter 1

Introduction

Data volumes generated in several commercial, research, and academic domains have grown exponentially. These data provide opportunities to extract insights. Modeling, via fitting models to the data, is a dominant way to accomplish this. Models allow researchers to extract and discern patterns from the data. These models encompassing multiple variables (of features) of interest allow researchers and planners to understand phenomena and inform decision-making.

Model building is resource-intensive and involves the entire resource hierarchy encompassing the CPU, RAM, and network I/O. There are several contributing factors to this resource-intensive nature of modeling. The model building process is iterative as parameters are tuned to best fit the data. The dimensionality of the data and data volumes exacerbate these challenges. Depending on the complexity of the model fitting algorithm, the number of parameters that need to be tuned also increases. All these are further compounded by the required tuning of model hyperparameters that govern learning rates, regularization schemes, loss functions, etc.

1.1 Goals and Objectives

The crux of this effort involves leveraging containers to manage orchestration of model training workloads. Containerization offers several advantages. Besides providing isolation between processes, containers are lightweight compared to virtual machines. Containers also allow us to specify thresholds for resources such as CPU, memory, and network I/O. However, containers are also subject to provisioning issues. Containers that are overprovisioned claim increasingly more resources that can impact other co-located processes and containers. On the other hand, when containers are under provisioned the encapsulated tasks hit their resource limits (CPU, memory, etc.) impeding their forward progress. The tasks complete, albeit much slower, because most of the time is spent making incremental progress as the containerized task receives a smaller share of the CPU and relies increasingly on the paging subsystem to ensure memory-residency of relevant data within the specified thresholds.

In this study, we focus on spatial modeling. To capture subtle variability in phenomena and to ensure accuracy of models, rather than build an all-encompassing model often a collection of models are built (one for each spatial extent). This allows individual model instances to capture variability patterns that are unique to the spatial extent under consideration.

1.2 Challenges

Our model training workloads encompass voluminous data and a large number of model instances. Combined with resource-intensive nature of the model fitting process this introduces several challenges.

1. Model fitting tasks occur in shared clusters. As such, there is a high probability of interfering with other colocated processes or containers.

2. Model building is resource-intensive: The process is iterative involving tuning model coefficients and weight vectors alongside hyperparameters and regularization schemes that inform overall model characteristics.

3. Data is voluminous and high-dimensional: Brute force model fitting operations are computationally expensive.

1.3 Research Questions

To effectively containerize model fitting tasks over spatial data we explore the following research questions.

RQ-1: How can we segment the data space?

RQ-2: How can we leverage this segmentation to inform spatial similarity?

RQ-3: How can we leverage segmentation to inform the sizing of containers?

RQ-4: How can we reduce cumulative resource requirements (and consumption) for containers to complete model fitting tasks?

1.4 Approach Summary

Our containerizations of model training workloads are based on the nature of the model fitting workloads. We posit that characteristics of the data space over which the particular class of models are being built can be used to inform the sizing of containers. To capture subtle regional variations in phenomena, we build models for individual spatial extents. Given that a large number of model training tasks need to be containerized, our methodology extracts efficiencies at several levels: (1) minimize duplicate work, (2) reduce resource overheads relating to computing, memory, and disk accesses, and (3) rightly sizing the containers.

Containers allow us to specify resource thresholds. Our methodology targets effective provisioning of these containers to avoid situations where the model fitting process may interfere with other colocated processes on that machine.

Container sizing is different based on the model fitting algorithm and spatial extent.

Rather than have a one-size-fits-all solution that leads to all containers being sized identically, our container sizing scheme is aligned with models being fitted to the data. Even within a particular model fitting algorithm, we ensure that containers can be sized differently.

Our goal is to partition spatial extents based on their data similarity. Spatial similarity is then used to inform how models are containerized during training.

A key step in the model building process is identifying the set of independent variables (or features) over which the model is being built. Once this set of features is available, we construct a surrogate model.

The surrogate model (based on random forests) is used to estimate feature importance and ranking. The normalized ranking is then used to select a subset of features, M, to cluster spatial extents. Only a subset of the features are chosen to avoid the curse of dimensionality that arises in high dimensional spaces – for example, the ratio of the pairwise distances between the closest points and the farthest points approaches one.

Each spatial extent is represented using an N-dimensional vector. Next, we cluster these spatial extents to produce a set of k clusters. Each cluster represents spatial extents with similar data characteristics based on the features of interest.

We use this similarity based on data characteristics to further extract efficiencies by significantly reducing duplicate processing within containers. Within each cluster, we train one model rigorously and then use the trained final parameters of those models as starting points for other points within the cluster. Because the data characteristics of points within the cluster are similar, the final parameters of rigorously trained models allow the modeling tasks to complete faster compared to exhaustive training.

This allows us to minimize processing overheads for model convergence. We reduce processing and memory requirements within containers by warm-starting model training tasks with parameters from rigorously trained models that allow tasks to complete faster while consuming fewer resources.

We do more with less: the number of rounds, memory residency, etc. are all reduced.

1.5 Paper Contributions

In this study, we describe our novel scheme to containerize model training workloads over voluminous spatial datasets. Our methodology

1. Reduces resource requirements significantly.
2. Is agnostic of the model-fitting algorithm.
3. Includes design of segmentation schemes to identify spatial similarities that are deeply aligned with the model fitting tasks.

Our methodology allows lower-latency, high-throughput completion of tasks with reduced resource requirements allowing us to do more with less.

Chapter 2

Related Work

Transfer learning involves the process of transferring knowledge gained from one learning task to another, provided the source domain and the target domain have similar data distributions [1,2]. Using knowledge learned in the source domain to train a model in the target domain has been shown to require less training data, along with having faster convergence rates, greater accuracy, and fewer resource requirements [3]. Machine learning models built over geospatial data (which includes a time and space component) provide new avenues for transfer learning schemes since similarities in data linked to time and space dimensions can be leveraged to reduce the number of computations [4]. In their study on using spatial-temporal patterns to automatically identify people's faces, Jianming Lv et al. [5] have used a transfer learning approach that can transfer visual classifiers across datasets from a small labeled source dataset to an unlabeled target dataset. Xin Qin et al. have used spatial similarities in Human Activity Recognition (HAR) data collected through sensors to acquire knowledge from labeled datasets and apply it to other unlabeled data [6]. Furthermore, they have utilized spatial features to accurately select suitable source and target domains for transfer learning using an approach named Adaptive Spatial-Temporal Transfer Learning or ASTTL. Major strides have been made in the field of medical image recognition due to the availability of large-scale annotated datasets which can be utilized in training Deep Convolutional Neural Networks (CNNs). There are three techniques used for training CNNs on a large dataset; training the network from scratch, using a pre-trained model, or using transfer learning to train models across datasets. Hoo-Chang Shin et al. have studied the suitability of using spatial features in medical imagery to perform transfer learning by using neural networks trained on a particular dataset and re-apply the same network to a different dataset by fine-tuning the network parameters [7].

Virtualization was the first technology to effectively isolate hardware resources to be used by applications and provide the illusion of running multiple machines on a single physical machine.

However, virtualization comes at a cost of resource overhead. Containerization was introduced as a lighter and faster alternative to virtual machines [8]. The lower overhead of deploying containers compared to virtual machines has enabled the development of fault-tolerant cloud-computing applications [9]. Containers have transformed software engineering practices and deployments especially relating to microservice architectures [10, 11] and Internet of Things (IoT) applications [12]. Containerization of applications introduces new challenges relating to how to manage multiple containers that need to interact with each other in a distributed system. Popular container orchestration engines such as Kubernetes [13] and Docker Swarm [14] address these challenges by introducing controller nodes to manage deployments of multiple containers. Although Kubernetes and Docker Swarm are the most popular frameworks, the first unified container-management system was developed at Google and was named Borg [15], which was used for managing long-running services and batch jobs [16].

Containerization and container orchestration has been used in studies to model systems to achieve faster results as containers allow fast deployment on highly scalable environments [17]. Zhou et al. [18] have attempted to use Kubernetes to orchestrate deep learning workloads by deploying Convolutional Neural Networks (CNNs) on containers launched in edge devices on Internet-of-Things (IoT) environments.

Deep learning workloads are dominated by GPUs; containerization of these resources is still in early phases and given the comparably smaller size of GPU RAMs, not typically shared across multiple containerized processes. This study does not target deep learning workloads and is focused on algorithms that leverage CPUs and CPU RAMs.

Machine learning workloads are resource-intensive. Additionally, concurrent model-building in cases of a multi-user environment adds to the complexity of this problem [18]. To avoid resource contention in parallel model building in distributed clusters, efficient resource management and scheduling are crucial in ensuring good throughput [19].

Cloud infrastructures have recently gained popularity with their ability to deploy disjoint machine learning jobs over containers [20–22], mainly due to containers being lightweight, with low resources requirements and having the capability to rapidly scale up/down based on workload constraints. In their study to evaluate the impact of containerizing deep learning workloads, Pengfei Xu et al. have evaluated performance benchmarks of system components such as IO, CPU, and GPU while running deep learning experiments on Docker containers [23]. Their results indicate that running computationally intensive jobs on CPUs/GPUs has little overhead compared to running the jobs directly on top of the operating system. David Brayford et al. have successfully used containers to deploy three-dimensional convolutional GAN (3DGAN) with petaflop performance on High-Performance Computing (HPC) resources [24]. They state that using containers allowed them to use HPC clusters without sacrificing the security of the cluster, and helped the execution of privileged operations to be performed inside the container without escalating permissions up to root on the host environment. Krishan Kumar et al. have employed containerization on cloud-computing environments to train traditional sentiment analysis applications in Natural Language Processing [25]. Their results show that the sentiment analysis tasks run on a containerized environment meet the requirements of real-time applications over a cloud deployment. Shreya S Rao et al. have developed a scalable Machine-Learning-as-a-Service (MLaaS) offering which minimizes the deployment costs for machine learning tasks using containerization on the cloud [26]. They demonstrate the suitability of their system using factors such as time-to-deploy, resource usage, and training metrics. Wonjun Lee et al. have incorporated containers to train machine learning models that are tasked with identifying kernel-level rootkits in a system [27]. They have used one of the architectural designs used in containerization; the isolation of containers through the host kernel's namespaces, to inform learning tasks about kernel-level rootkits. Gabriel de Oliveira Ribeiro has attempted to aggregate popular machine learning libraries such as SciKit Learn, TensorFlow, Spark MLLib into a REST (Representational State Transfer) API (Application Programming Interface) to orchestrate machine learning workflow on containerized cloud environments [28]. The results of their experiments indicate that the API simplifies and streamlines iterative machine learning processes. Going beyond the off-the-shelf containerization frameworks available, Omar S. Navarro Leija et al. have designed and implemented a new variety of containers (named DetTrace) which provides a layer of reproducible abstraction for Linux [29]. In DetTrace, all computations that occur inside a container are expressed as pure functions of the initial file system of the container. These reproducible containers can be used to deploy machine learning tasks in a more effective way compared to traditional containers. In another study, researchers have utilized used deep learning to effective allocate CPU resources for containers based on diminishing marginal returns [30].

In this work, we couple the lightweight scalability of a containerized environment with transfer learning over segmented data domains to further improve the resource utilization and throughput of such parallel machine learning workflows in cases of spatiotemporal datasets.

Chapter 3

Methodology

To achieve our objective of effectively utilizing a containerized environment to train regression models, we start by segmenting the multi-dimensional data space to identify spatial similarity in geographical areas. We then leverage the segmented data space to calibrate computational and memory resources needed to fit models using machine learning algorithms over a particular dataset; this then informs the sizing of containers. This process is explained in the following sub-sections.

3.1 Segmentation

We bucket observations into their smallest administrative units. Each administrative unit has a hierarchical prefix associated with it. Prefix matching can be used to aggregate smaller administrative spatial extents into larger ones. For example, Census tracts can be aggregated into cities/towns, which in turn can be aggregated into counties, states, etc.

The data is sharded so that data from a prefix are co-located on the same machine. The process is deterministic and ensures that all data can be funneled to the correct machine. The deterministic sharding scheme also ensures data locality during model fitting operation. When building models for larger spatial extents, the data is hosted on a smaller subset of machines.

3.2 Partitioning and Segmenting Spatial Extents

Each data item is represented as an N-dimensional feature vector. We use a ranked subset of these features to segment spatial extents.

To ensure our transfer learning scheme functions well, we start by building reliable source models. Since exhaustive training is resource-intensive, we initiate training by selecting 1% sampling of the data representing all spatial extents. Then we rank the feature importance using the Random Forests algorithm, after which we select features based on normalized ranking that add up to 85% of cumulative variance. Isolating the most important features helps get rid of curse of dimensionality. After selecting the features, we perform an aggregation of columns based on each spatial extent. Then we cluster the spatial extents using a popular clustering algorithm such as K-Means Clustering. The results of the clustering operations informs the spatial extents that are closest to the cluster centroids, and the distance to each spatial extent from its own cluster centroid.

3.3 Rigorous Training of Models

We use the spatial extents associated with cluster centroids to perform hyperparameter tuning. Root Mean Squared Error (RMSE) is used as the stopping criterion i.e. when the RMSE improvements do not meet the specified thresholds.

Each algorithm has its own set of hyperparameters. We selected a subset of parameters available for each algorithm to perform a grid search, in order to find the optimal set of parameters for each centroid (parent) model. In this study, we considered four model-fitting algorithms that we discuss below. We note that our methodology is broadly applicable and does not preclude using other model-fitting algorithms.

Gradient Boosting: Hyperparameters that we considered for Gradient Boosting included the loss function to be optimized, learning rate (that shrinks the contribution of each tree), the number of estimators, and the maximum depth (of the individual regression estimator) [31].

Linear Regression: The hyperparameters considered for Linear Regression were whether to calculate the intercept for the model (boolean), whether to normalize the regressors before regression, and the number of jobs to be used for the computation [32].

Support Vector Regression: We used a multi-class logistic regression algorithm to predict labels in the datasets we used. The hyperparameters considered for the grid search were the *kernel* which could be one of *linear*, *poly*, *sigmoid*, the degree of the polynomial kernel function *poly*, and the regularization parameter C [33].

Time-Series Regression: Hyperparameters in a time-series model belong to two major classes: *changepoint* and *seasonality*. Changepoints are the points in data where there is a sudden change in the trend. Seasonality can be either *additive* or *multiplicative*. Thus, the key hyperparameters are (1). The number of changepoints or *n_changepoints*, (2). *changepoint_prior_scale* which captures how flexible the changepoints are, (3). *seasonality*, and (4). *seasonality_prior_scale* which captures the flexibility of seasonality in data [34].

3.4 Sizing containers based on distance from Centroids

For a given cluster, we assign each spatial extent a sampling percentage that is proportional to the distance to the spatial extent from the cluster centroid. First, a percentage distance is calculated. We define the coordinates of a random data point i, the centroid, the point farthest from the centroid, the point closest to the centroid are represented as λ_i, λ_C, λ_{max}, and λ_{min} respectively. Then, the normalized distance of i is calculated using formula 3.1.

$$NormalizedDistance(i) = \frac{\lambda_i - \lambda_{min}}{\lambda_{max} - \lambda_{min}} \tag{3.1}$$

Then, we use the calculated normalized distance to assign a sampling percentage for i. A minimum and (S_{min}) a maximum (S_{max}) sampling percentage is pre-determined for each dataset. Sampling percentage is calculated using formula 3.2.

$$SamplingPercentage(i) = S_{min} + (S_{max} - S_{min}) \cdot NormalizedDistance(i) \tag{3.2}$$

The calculated sampling percentage can be used to segment the spatial extents into multiple classes (See Fig. 3.1). For instance, class Y will represent spatial extents that are assigned a 5-15% sampling rate, and Z will represent spatial extents that use a 15-25% sampling rate. The centroid models (represented in X) are rigorously trained using 50-100% of the data based on progressive sampling and RMSE thresholds, as they are our source models. The objective of this segmentation is to identify spatial extents that can be trained using containers that are sized differently from each other. We can use containers with maximum resource allocation for training points in class X as they need to be rigorously trained. Class Y will be trained with lightweight containers, and

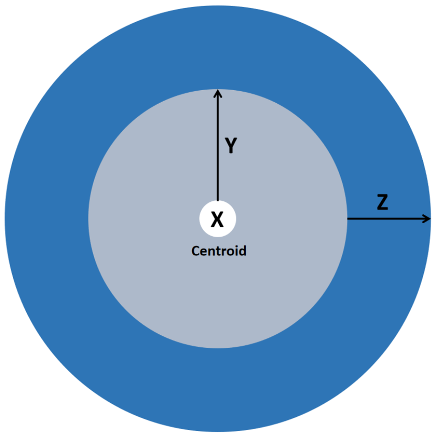

Figure 3.1: Partitioning spatial extents based on their distance from the cluster centroid.

class Z will be trained using moderately-sized containers (both Y and Z points are trained with transfer-learned weights from X models).

3.5 Warm-starting model training tasks via transfer learning

To implement our transfer learning scheme, instead of naively training spatial extents (using randomly initialized hyperparameters), we initialize the hyperparameters for child models based on those identified during hyperparameter tuning for the parent models (models associated with cluster centroids). Additionally, we use a different data sampling rate for each spatial extent calculated using the formula mentioned in section 3.4.

3.6 Inferring Suitable Container Sizes

For a given model-fitting algorithm on a particular dataset, and based on the categorization based sampling percentage, we use a heuristic method to infer how resources should be allocated for the containers (workers).

First, we use a single spatial extent (this could be a state, county, or census tract) belonging to the category Y or Z (Fig. 3.1). Then, we train the selected spatial extent on containers and run the modelling algorithm within a single containerized worker (single Kubernetes pod) to capture the total convergence time, memory and CPU usage. We iteratively reduce the container sizes while noting the convergence time in each case. The goal of this process is to identify the amount of resources that are needed to perform the modelling task while keeping the resource allocations at a minimum. When the convergence time starts to increase, we stop the iterative process. The container size at that point identifies the effective sizing/allocation of resources for the containers. We proceed to train the entire dataset using the identified container configuration.

Chapter 4

Performance Benchmarks

Our benchmarks profile the suitability of our methodology by assessing its impact on (1) completion times, (2) resource utilizations, (3) and throughput. Further, these benchmarks are performed with diverse model fitting algorithms over multiple datasets to demonstrate applicability to a broad class of problems.

4.1 Datasets

We evaluated three datasets in our experiments.

1. Multivariate Adaptive Constructed Analogs Applied to Global Climate Models (Macav2) [35]: a collection of outputs from 20 climate models which covers the continental United States

2. North American Mesoscale Forecast System Dataset (NOAA NAM) [36]: is a collection of weather forecasts encompassing multiple meteorological variables over the North American continent at a multiplicity of resolutions

3. COVID-19 Dataset [37]: A collection of records for that United States that include the daily number of confirmed COVID-19 cases and mortality within each county, from the beginning of the pandemic to date.

4.2 Experimental Setup

In this study, we leveraged Dask [38], a framework for scalable analytics using Python and the MongoDB [39] distributed, data store. We used a cluster of 25 physical nodes to set up a Dask cluster where we would launch all our model fitting tasks. MongoDB was deployed on a cluster of 50 nodes; Each node (8 x 2.1 GHz HP-dl60, 64 GB RAM, and 4 SATA hard disk drives) running CentOS (version 8.4.2105). We use version 3.8 of Python, 20.10.7 of Docker, and 1.20.8 of Kubernetes.

We used Scikit Learn [40] to implement Linear Regression, Gradient Boosting, and Support Vector Regression. Time-series models were built using Facebook's Prophet library [34]. Facebook Prophet consists of APIs implemented in Python and R. In their basic form, both Scikit Learn and Facebook Prophet runs primarily on a single machine. However, by integrating Scikit Learn and Facebook Prophet with Dask, we were able to distribute model fitting workloads over a cluster of nodes, and use a containerized environment.

4.3 Assessing the Transfer Learning scheme

Here, we focus on profiling the suitability of our transfer learning scheme for model fitting tasks over spatio-temporal data. These experiments were conducted on a Dask cluster with 25 worker nodes (not containerized).

4.3.1 Linear Regression

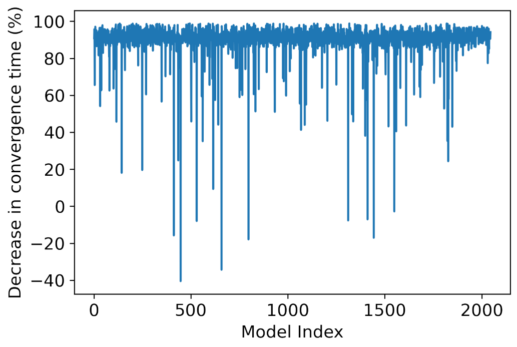

Figure 4.1: Percentage decrease in convergence time for Linear Regression (over Macav2 dataset) with Transfer Learning.

In the case of Linear Regression, non-transfer-learned models took 7.15 seconds on average to converge while it took 0.62 seconds for the transfer-learned models to converge, resulting in a decrease of 91.33% in average convergence time (Fig. 4.1); this substantiates the suitability of our transfer learning scheme for Linear Regression.

4.3.2 Gradient Boosting

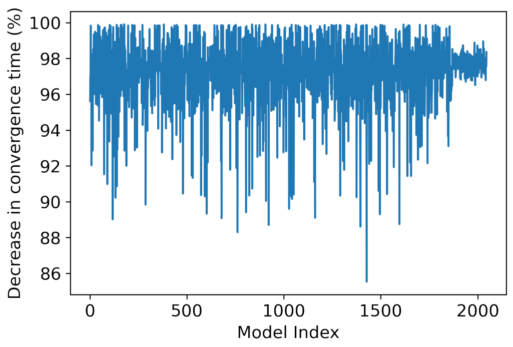

Figure 4.2: Percentage decrease in convergence time for Gradient Boosting (over Macav2 dataset) with Transfer Learning

When fitting models using gradient boosting, non-transfer-learned models took 385.48 seconds on average to converge while it only took 10.54 seconds for the transfer-learned models to converge; (Fig. 4.2). In particular, out transfer-learned models converged 36.57 times faster compared to tradition cold-start training.

4.3.3 Time-Series Regression

Time-series models built (using Facebook Prophet) over the COVID-19 dataset took on average 70.22 seconds to converge without transfer learning, while the transfer-learned models converged in 2.25 seconds on average; representing a 31.2x fold speedup (Fig. 4.3). using our methodology.

Our empirical benchmarks with these three different types of models, demonstrate the applicability of our transfer learning schemes to regression models to achieve faster convergence.

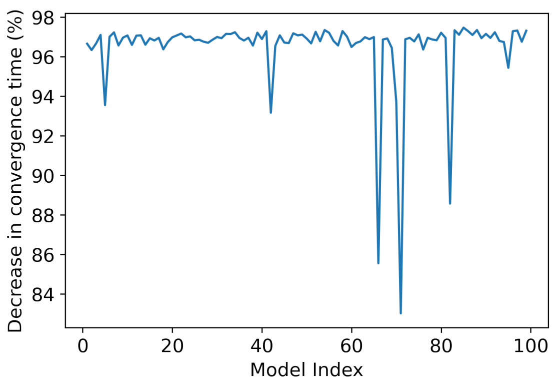

Figure 4.3: Percentage decrease in convergence time for Time-Series Regression (over COVID-19 dataset) with Transfer Learning.

4.4 Exploration of Container Sizing

Here, we assess the suitability of using differently sized containers. We profile completion times for multiple model fitting tasks over three containerized Dask clusters, each sized differently. The three types of clusters were sized as follows.

- Cluster A: Memory - 64 GB, CPU - 16 cores
- Cluster B: Memory - 32 GB, CPU - 8 cores
- Cluster C: Memory - 8 GB, CPU - 1 core

Table 4.1: Completion Times for different model fitting algorithms on differently sized clusters

	Completion Time (s)		
Algorithm	**Cluster A**	**Cluster B**	**Cluster C**
Gradient Boosting	600.16	570.46	603.96
Linear Regression	35.58	35.66	41.39
Support Vector Regression	382.45	391.56	392.14
Time-Series Regression	381.58	401.87	405.12

Table 4.2: Cluster Sizes

Cluster	Memory (GB)	CPU Cores
C1	16	8
C2	12	4
C3	8	2
C4	4	1

Table 4.1 shows that, for Linear Regression, Support Vector Regression, and Time-series Regression, the completion time increases as the size of containerized workers get smaller i.e. when less resources are allocated for training. In the case of Gradient Boosting, the lowest completion time was observed when the data was trained using **Cluster B**. These results validate our hypothesis that it is possible to find a compromise between completion times and allocation of resources to size containers differently to cater to different model fitting workloads, and to make intelligent use of the finite resources available on the physical nodes in a cluster.

4.5 Rigorous, Light, and Moderate training using containers

For the datasets selected in this study, we attempted to infer the optimal sizing of containers (workers) using cluster settings mentioned in Table 4.2. In the following sections, we explain the results obtained by running different regression algorithms using each of the aforementioned clusters.

Table 4.3: Maximum memory utilization for a single spatial extent

Model fitting Algorithm	Dataset	Maximum amount of memory utilized (MB)
Linear Regression	Macav2	725
	NOAA	970
Gradient Boosting	Macav2	930
	NOAA	955
Support Vector Regression	Macav2	980
	NOAA	1094
Time-series Regression	COVID-19	985
	NOAA	1115

Linear Regression

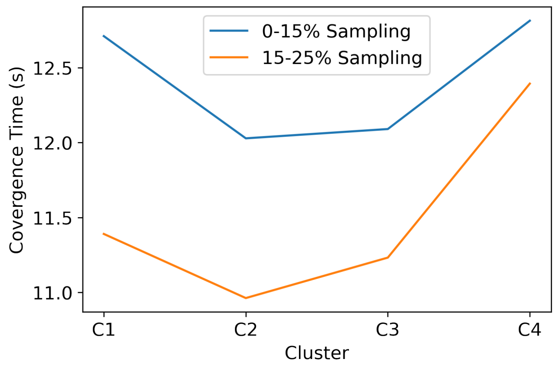

Figure 4.4: Convergence times for Linear Regression on NOAA dataset.

For Linear Regression models trained over the NOAA dataset, based on the lowest convergence times, the optimal sizing of containers (workers) were identified as C2 (12GB memory, 4 CPU cores) for both 0-15% and 15-25% sampling categories (Fig. 4.4).

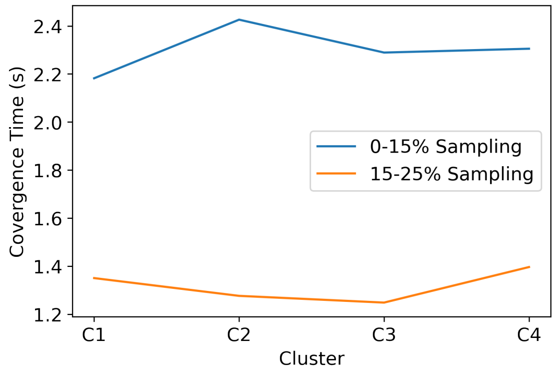

Figure 4.5: Completion times for Linear Regression on Macav2 dataset.

For Linear Regression models trained on the Macav2 dataset, based on the lowest completion time, the optimal sizing of containers (workers) were identified as C1 (16GB memory, 8 CPU cores) for the 0-15% sampling category, and C3 (8GB memory, 2 CPU cores) for 15-25% sampling (Fig. 4.4).

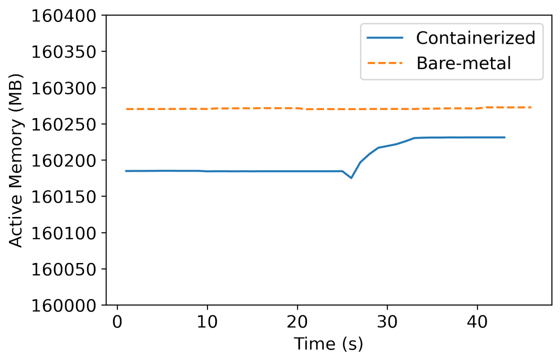

Figure 4.6: Linear Regression on Macav2 data - Memory Utilization for containerized vs. bare-metal worker nodes.

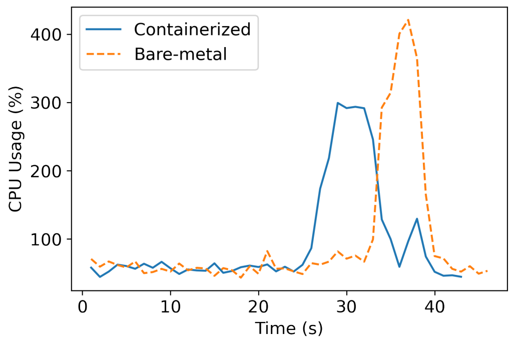

Figure 4.7: Linear Regression on Macav2 data - CPU usage for containerized vs. bare-metal worker nodes.

While training Linear Regression models on the Macav2 dataset (for all US counties) using the optimally-sized containers, we observed a 0.4% reduction in the average memory usage (Fig. 4.6) and 0.15% reduction of average CPU usage (Fig. 4.7) compared to the same workload run on bare-metal workers. (A bare-metal environment is where the workers and not containerized and no explicit constraints on the resources are enforced.)

Fig. 4.8 shows the CPU usage for a Linear Regression model trained on a single county of the NOAA dataset using a single containerized worker. The limits set in each of the 4 cases (4 cluster settings mentioned in Table 4.2) are also depicted in the graph. The CPU utilization was below 2 CPU cores for the entirety of the training but it was above 1 core; as such, the allocation needed for this experiment was set to 2 CPU cores. As outlined in Table 4.3, the maximum amount of memory utilized by the containerized worker is 970 MB, which implies that it is possible to

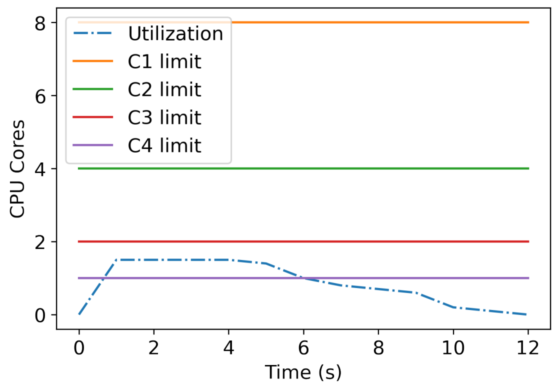

Figure 4.8: Linear Regression on NOAA data - CPU Usage for a container.

run this experiment by allocating workers 1GB of memory in an environment where resources are limited.

Gradient Boosting

For Gradient Boosting models trained on the NOAA dataset, the optimal sizing of containerized workers were recognized as C1 (16GB memory, 8 CPU cores) for 0-15% sampling category, and C4 (4GB memory, 1 CPU core) for 15-25% sampling (Fig. 4.9). For Gradient Boosting models trained on the Macav2 dataset, the optimal container sizes were C2 (12GB memory, 4 CPU cores) for the 0-15% sampling and C3 (8GB memory, 2 CPU cores) for 15-25% sampling (Fig. 4.10).

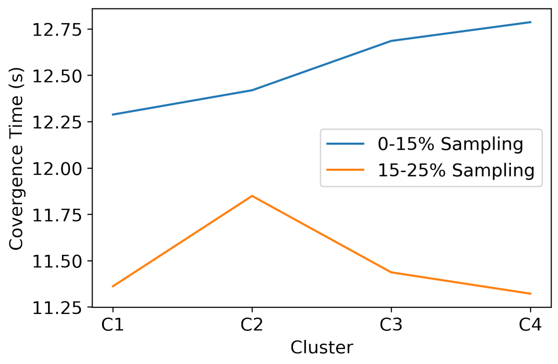

Figure 4.9: Convergence times for Gradient Boosting over the NOAA dataset.

For Gradient Boosting models trained on the Macav2 dataset for all US counties, we observed a 6.46% reduction in overall CPU usage on containerized workers (sized optimally) compared to

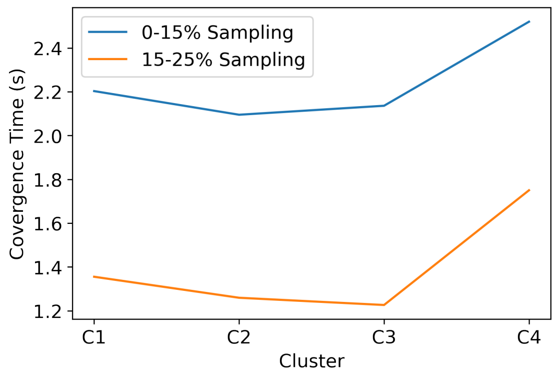

Figure 4.10: Convergence times for Gradient Boosting over the Macav2 dataset.

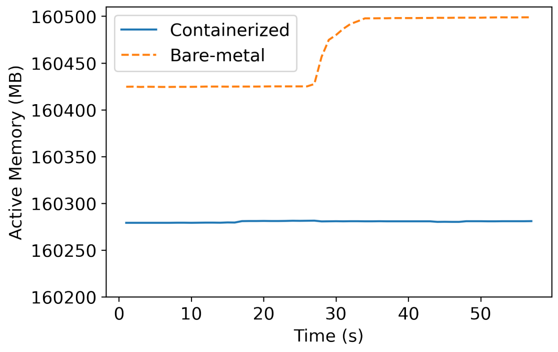

Figure 4.11: Gradient Boosting on Macav2 data - Memory Utilization for containerized vs. bare-metal worker nodes.

bare metal Dask workers as depicted in Fig. 4.12. Furthermore, the average memory utilization was reduced by 0.11% (Fig. 4.11).

Fig. 4.13 and Fig. 4.14 show the CPU usage for a Gradient Boosting model trained on a single spatial extent (county) of the NOAA and the Macav2 dataset respectively using a single containerized worker. The limits set in each of the 4 cases (4 cluster settings) are also depicted in the graphs. The CPU utilization was below 2 CPU cores for the entirety of the training but it lies above 1 core. Therefore, the precise allocation needed for this experiment, for both datasets, should be 2 CPU cores. Furthermore, according to Table 4.3, the maximum amount of memory utilized by this experiment is 955 MB. Therefore, we could limit the memory to 1GB per worker if the resources are scarce.

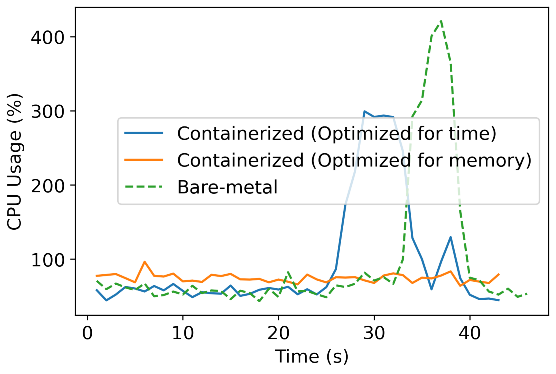

Figure 4.12: Gradient Boosting on Macav2 data - CPU usage for Containerized (Optimized for time), containerized (Optimized for memory), and bare-metal worker nodes.

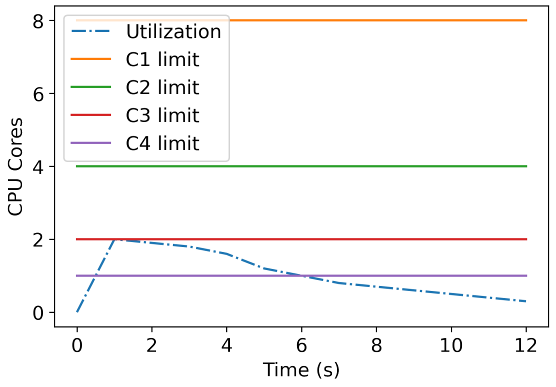

Figure 4.13: Gradient Boosting on NOAA data - CPU Usage for a container.

Support Vector Regression

For Support Vector Regression on the Macav2 dataset (similar to the experiments on other regression models), the optimal container configurations were identified as C3 (8GB memory, 2 CPU cores) for 0-15% sampling, and C2 (12GB memory, 4 CPU cores) for 15-25% sampling. The optimal containers for model-fitting on the NOAA dataset were observed as C3 for both 0-15% and 15-25% sampling.

Time-Series Regression

When implementing time-series models, it is important to select a time period into the future we want to predict a selected variable, based on the time-series data available. The COVID-19 dataset contains daily records of the number of confirmed cases and mortality spanning 15 months. Using

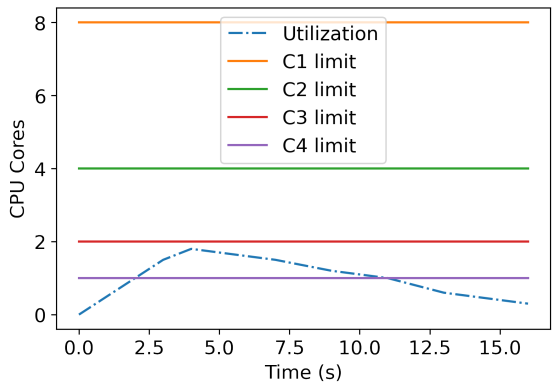

Figure 4.14: Gradient Boosting on Macav2 data - CPU Usage for a container.

the 15 months of data, we predicted the number of COVID-19 cases into 1-3 months into the future. The NOAA dataset contains hourly data recorded related to various weather phenomena spanning multiple years. We selected the variable *Mean Sea Level Pressure (Pascal)* to make predictions for a couple of months into the future.

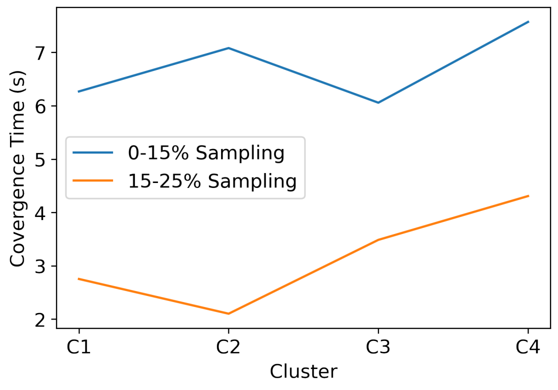

Figure 4.15: Completion times for Time-Series Regression on the COVID-19 dataset.

For time-series models trained on the COVID-19 dataset, the optimal sizing of containers were identified as C3 (8GB memory, 2 CPU cores) for 0-15% sampling category, and C2 (12GB memory, 4 CPU cores) for 15-25% sampling (Fig. 4.15).

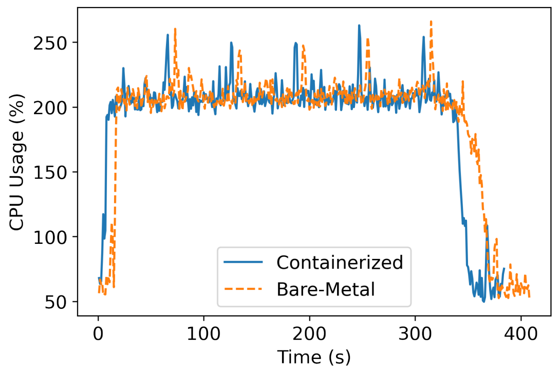

Figure 4.16: Comparison of CPU Utilization for Time-Series Regression on the COVID-19 dataset (Bare-metal vs. Containerized Dask).

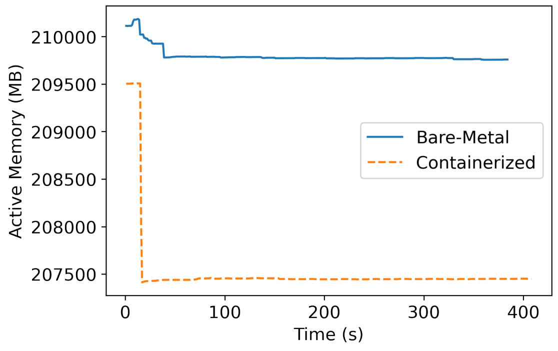

Figure 4.17: Comparison of Memory Utilization for Time-Series Regression on the COVID-19 dataset (Bare-metal vs. Containerized Dask).

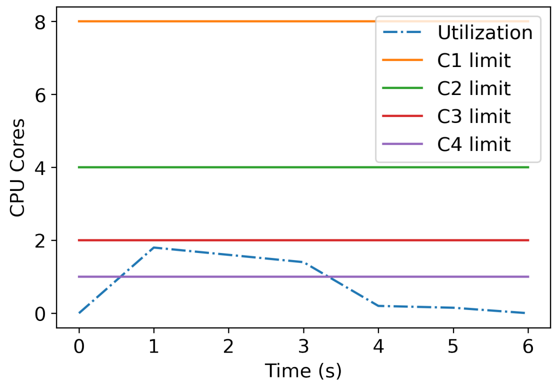

Figure 4.18: Time-series Regression on the COVID-19 dataset: CPU Usage on a single container.

When the entire dataset (all US counties) were used for predicting the number COVID-19 cases for a selected time-frame into the future using the optimally-sized containers, we observed a 0.21% reduction of overall CPU usage (Fig. 4.16 and a 1.1% reduction of overall memory usage (Fig. 4.17) compared to the bare-metal workers.

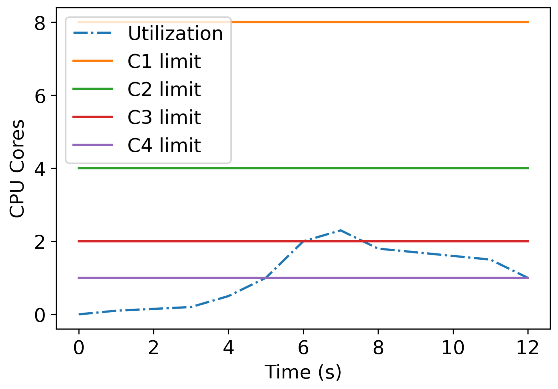

Figure 4.19: Time-series Regression on COVID-19: CPU Usage on a single container.

Fig. 4.21 shows the CPU usage for a time-series model built on COVID-19 data for a single county in the US, using a single containerized worker. It can be observed that the precise allocation of CPU cores required is 4 for this model fitting task. According to Table 4.3, the maximum amount memory utilized by the worker in this case is 985 MB. If we wanted to prioritize frugal memory usage over faster convergence, we could have allocated 1GB memory per worker for this particular model fitting task.

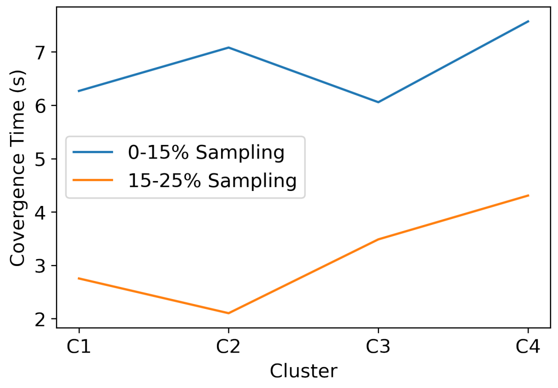

Figure 4.20: Completion times for Time-Series Regression on the NOAA dataset.

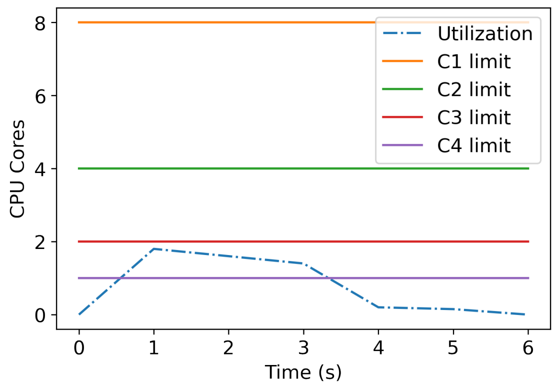

Figure 4.21: Time-series Regression on NOAA: CPU Usage on a single container.

We also trained time-series models on the NOAA dataset for hourly *Mean Sea Level Pressure (Pascal)* values. After identifying the most suitable sizing of containers based on the time it took for a single time-series model (built for a randomly selected US county), we proceeded to build time-series models for all US counties. These workloads consumed 2.31% less CPU on average, and 3.4% less active memory.

Chapter 5

Conclusions and Future Work

We described our methodology to orchestrate model fitting tasks over spatial datasets in a containerized environment. Our methodology focusses on reducing resource requirements by sizing containers effectively and reducing processing overheads during model training.

RQ-1: Our data space segmentation is aligned with the dataset under consideration and relies on both dimensionality reduction and spatial similarity. We rank features based on their importance and considering only a subset of these features reduces dimensionality and achieves more effective spatial segmentation.

RQ-2: Since our spatial segmentation schemes are specific to a dataset and reduce dimensionality, we can identify spatial similarity based on the phenomena under consideration. Further, this allow spatial similarity considerations to be dynamic rather than statically assigned.

RQ-3: Spatial similarity underpins our transfer learning schemes and sizing of containers for model fitting tasks. In particular, we train one model within each spatially similar cluster and transfer learn the weights of this anchor model to spatially similar extents as starting points. Since the similarity is based on the dataset(s) under consideration this allows the transfer learned models to converge faster. As substantiated by our benchmarks, the speeds achieved by our methodology include 36x (for gradient boosting) and 31 x for time series models. Calibrating resource utilizations differently based on spatial extents and transfer learning tasks allows us to make frugal utilization of resources.

Finally, our benchmarks demonstrate that a tradeoff can be reached for convergence times and resource utilization by leveraging effectively sized containers (in contrast to tasks running on bare-metal) to deploy model-fitting workloads.

As part of future work, we will explore extending this work in the context of streaming datasets. An additional consideration here is placement of containers to ensure effectively sharing of data processing pipelines across spatially similar clusters.

Bibliography

[1] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. A comprehensive survey on transfer learning. *Proceedings of the IEEE*, 109(1):43–76, 2020.

[2] Tatiana Tommasi et al. Learning categories from few examples with multi model knowledge transfer. *IEEE transactions on pattern analysis and machine intelligence*, 36(5):928–941, 2013.

[3] Lisa Torrey et al. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*, pages 242–264. IGI global, 2010.

[4] Matthias Bussas et al. Varying-coefficient models for geospatial transfer learning. *Machine Learning*, 106(9):1419–1440, 2017.

[5] Jianming Lv, Weihang Chen, Qing Li, and Can Yang. Unsupervised cross-dataset person re-identification by transfer learning of spatial-temporal patterns. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7948–7956, 2018.

[6] Xin Qin, Yiqiang Chen, Jindong Wang, and Chaohui Yu. Cross-dataset activity recognition via adaptive spatial-temporal transfer learning. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 3(4):1–25, 2019.

[7] Hoo-Chang Shin, Holger R Roth, Mingchen Gao, Le Lu, Ziyue Xu, Isabella Nogues, Jianhua Yao, Daniel Mollura, and Ronald M Summers. Deep convolutional neural networks for computer-aided detection: Cnn architectures, dataset characteristics and transfer learning. *IEEE transactions on medical imaging*, 35(5):1285–1298, 2016.

[8] Mathijs Jeroen Scheepers. Virtualization and containerization of application infrastructure: A comparison. In *21st twente student conference on IT*, volume 21, 2014.

[9] Víctor Medel, Omer Rana, José Ángel Bañares, and Unai Arronategui. Adaptive application scheduling under interference in kubernetes. In *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*, pages 426–427. IEEE, 2016.

[10] David Jaramillo et al. Leveraging microservices architecture by using docker technology. In *SoutheastCon 2016*, pages 1–5. IEEE, 2016.

[11] Joe Stubbs, Walter Moreira, and Rion Dooley. Distributed systems of microservices using docker and serfnode. In *2015 7th International Workshop on Science Gateways*, pages 34–39. IEEE, 2015.

[12] Joao Rufino, Muhammad Alam, Joaquim Ferreira, Abdur Rehman, and Kim Fung Tsang. Orchestration of containerized microservices for iiot using docker. In *2017 IEEE International Conference on Industrial Technology (ICIT)*, pages 1532–1536. IEEE, 2017.

[13] Kubernetes. https://kubernetes.io/. (Accessed on 09/16/2021).

[14] Swarm mode overview | docker documentation. https://docs.docker.com/engine/swarm/. (Accessed on 09/16/2021).

[15] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[16] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.

[17] Víctor Medel, Omer Rana, José Ángel Bañares, and Unai Arronategui. Modelling performance & resource management in kubernetes. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pages 257–262, 2016.

[18] Li Zhou, Hao Wen, Radu Teodorescu, and David HC Du. Distributing deep neural networks with containerized partitions at the edge. In *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

[19] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J Freedman. Slaq: quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 390–404, 2017.

[20] Machine learning on aws. https://aws.amazon.com/machine-learning/, 09 2021.

[21] Vertex ai. https://cloud.google.com/vertex-ai, 09 2021.

[22] Azure machine learning. https://azure.microsoft.com/en-us/services/machine-learning/#product-overview, 09 2021.

[23] Pengfei Xu, Shaohuai Shi, and Xiaowen Chu. Performance evaluation of deep learning tools in docker containers. In *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*, pages 395–403. IEEE, 2017.

[24] David Brayford and Sofia Vallecorsa. Deploying scientific al networks at petaflop scale on secure large scale hpc production systems with containers. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, pages 1–8, 2020.

[25] Krishan Kumar and Manish Kurhekar. Sentimentalizer: Docker container utility over cloud. In *2017 Ninth International Conference on Advances in Pattern Recognition (ICAPR)*, pages 1–6. IEEE, 2017.

[26] Shreyas S Rao, S Pradyumna, Subramaniam Kalambur, and Dinkar Sitaram. Bodhisattva-rapid deployment of ai on containers. In *2018 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 100–104. IEEE, 2018.

[27] Wonjun Lee and Mohammad Nadim. Kernel-level rootkits features to train learning models against namespace attacks on containers. In *2020 7th IEEE International Conference on*

Cyber Security and Cloud Computing (CSCloud)/2020 6th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom), pages 50–55. IEEE, 2020.

[28] Gabriel de Oliveira Ribeiro. Learning orchestra: building machine learning workflows on scalable containers. 2021.

[29] Omar S Navarro Leija, Kelly Shiptoski, Ryan G Scott, Baojun Wang, Nicholas Renner, Ryan R Newton, and Joseph Devietti. Reproducible containers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–182, 2020.

[30] Muhammad Abdullah, Waheed Iqbal, Faisal Bukhari, and Abdelkarim Erradi. Diminishing returns and deep learning for adaptive cpu resource allocation of containers. *IEEE Transactions on Network and Service Management*, 17(4):2052–2063, 2020.

[31] sklearn.ensemble.gradientboostingclassifier — scikit-learn 0.24.2 documentation. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.\\GradientBoostingClassifier.html. (Accessed on 09/01/2021).

[32] sklearn.linear_model.linearregression — scikit-learn 0.24.2 documentation. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.\\LinearRegression.html. (Accessed on 09/01/2021).

[33] sklearn.svm.svr — scikit-learn 1.0 documentation. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html. (Accessed on 10/11/2021).

[34] Sean J Taylor et al. Forecasting at scale. *The American Statistician*, 72(1):37–45, 2018.

[35] John T. Abatzoglou and Timothy J. Brown. A comparison of statistical downscaling methods suited for wildfire applications. *International Journal of Climatology*, 32(5):772–780, March 2011.

[36] Russell S. Vose et al. Improved historical temperature and precipitation time series for u.s. climate divisions. *Journal of Applied Meteorology and Climatology*, 53(5):1232–1251, May 2014.

[37] The New York Times. nytimes/covid-19-data.

[38] Dask: Scalable analytics in python. https://dask.org/. (Accessed on 07/16/2021).

[39] The most popular database for modern apps | mongodb. https://www.mongodb.com/. (Accessed on 08/28/2021).

[40] scikit-learn: machine learning in python — scikit-learn 0.24.2 documentation. https://scikit-learn.org/stable/. (Accessed on 09/01/2021).