Leveraging Machine Learning for Improved Distributed System Performance

A dissertation

submitted by

Hifza Khalid

In partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Computer Science

TUFTS UNIVERSITY

February 2025

ADVISOR: Alva Couch

Leveraging Machine Learning for Improved Distributed System Performance

Hifza Khalid

ADVISOR: Alva Couch

Our research aims to optimize the performance of large distributed systems, which operate across multiple machines, by applying machine learning techniques. In our first project, we intended to use a large dataset of performance data for the Linux operating system to suggest optimal tuning for network applications in a client-server setting. We conducted a series of experiments to select hardware and Linux configuration options that are significant to network performance. Our results showed that network performance was mainly a function of workload and hardware. Investigating these results showed that our dataset did not contain enough diversity in configuration settings to infer the best tuning and was only useful for making hardware recommendations. Based on these experiments and their outcomes, we concluded that one should not take data diversity, even of huge datasets, for granted. We also recommend a set of preliminary tests for anyone working on similar complex datasets and planning to use machine learning.

In our second project, we considered the problem of using a publicly released dataset by Alibaba to model the batch tasks that are often overlooked compared to online services. The dataset contains the arrivals and resource requirements (CPU, memory, etc.) for both batch and online tasks. Our trained model predicts, with high accuracy, the number of batch tasks that arrive in any 30 minute window, their associated CPU and memory requirements, and their lifetimes. It captures over 94% of arrivals in each 30 minute window within a 95% prediction interval. The F1

scores for the most frequent CPU classes exceed 75%, and our memory and lifetime predictions incur less than 1% test data loss. The prediction accuracy of the lifetime of a batch-task drops when the model uses both CPU and memory information, as opposed to only using memory information.

Our third project proposes a deep reinforcement learning approach to task scheduling, aiming to maximize cloud resource utilization by strategically delaying and consolidating batch tasks onto fewer machines. We explore Policy Gradient (REINFORCE) and Deep Q-Network (DDQN) methods to develop a self-learning scheduler that adapts to dynamic workload conditions. Experimental results show that REINFORCE increases average CPU and memory utilization by 125-200% compared to Best-Fit and Packer, efficiently reduces the number of machines required, and achieves a 5-30% reduction in resource fragmentation. Although DDQN also reduces machine usage compared to traditional methods, its performance declines under high loads due to job drops and sub-optimal long-term planning in partially observable environments. Moreover, REINFORCE is computationally more efficient with lower memory requirements, while DDQN is more sample efficient.

To my hardworking late father, my supportive mother, my beloved husband, and my dearest daughter

Acknowledgments

I am deeply grateful to everyone who supported me on this challenging yet rewarding journey, particularly my teachers, advisors, and mentors. I would like to extend my heartfelt appreciation to my PhD advisor, Dr. Alva Couch. He not only guided me through my research but also provided invaluable encouragement during difficult times, reminding me to persevere even when the path seemed daunting. I am especially thankful for the freedom he allowed me during my PhD – studying from my home country during COVID, exploring research topics that ignited my passion, and adapting to remote study after I moved in with my partner. His calm demeanor and patience made this experience much more manageable, and I truly appreciate his support.

I want to express my gratitude to Dr. Raja Sambasivan for his honest feedback whenever I needed it; I genuinely wish I could have worked more closely with him. I am also grateful to Dr. Zartash Uzmi and Dr. Ihsan Ayyub Qazi for their invaluable mentorship during my Master's program, their guidance in my PhD applications, and for teaching me the foundations of conducting research. A special thanks goes to my undergraduate advisor, Dr. Irfan Ullah Chaudhary, who encouraged me to pursue a PhD. Under his mentorship, I grew both professionally and personally, learning the importance of striving for excellence and embracing my uniqueness.

I would like to thank Dr. Arunselvan Ramaswamy and Dr. Simone Ferlin-Reiter for their generous guidance throughout my research. Red Hat has been a pivotal part of my journey, and I am especially grateful to Peter Portante for being a steadfast pillar of support. His unwavering belief in me, along with his positivity and encouragement, has been truly invaluable. I also want to thank Hijab and Sahara for being my only family in the U.S. during the first few years of this journey.

I am indebted to my committee members – Prof. Raja Sambasivan, Prof. Fahad Dogar, Prof. Mark Hempstead, and Dr. Ashish Kamra – for their insightful feedback, which significantly enhanced the quality of my thesis.

I am profoundly thankful to my family for their unconditional support and prayers. My siblings have always been my rock; Ali, thank you for taking care of our parents during this time. Your presence there brought me peace, and your generosity and selflessness never cease to amaze me. Anum Baji, I cherish our long, thoughtful discussions; they provided a perfect escape from my work. Ayesha, my angel sister, your kindness and support are unmatched. I am grateful for the way you light up our family with your warmth and positivity. Misba, thank you for helping with my wedding preparations and for your encouraging words. I am also deeply grateful to my in-laws for their unwavering support and prayers. Aunty, thank you for always taking care of me like a daughter and for traveling to the U.S. to support me when Alisha was born. I am indebted to you and uncle for all your kindness during my PhD journey. I would also like to thank Mehreen for being the sweetest and most positive person I know, and Hammad for your thought-provoking ideas and support.

Words cannot fully express my gratitude for the unwavering support from my parents and the sacrifices they made to provide me with quality education and resources. I owe a special debt of gratitude to my late father, whose progressive mindset encouraged me to pursue education abroad. Despite his health challenges, he always sought to provide us relief and instilled in us the value of hard work and education. I deeply wish he could have witnessed the successful culmination of this journey. My heartfelt appreciation goes to my mother, who worked tirelessly to ensure we received quality education. I can never repay you for the sacrifices you have made, and I believe your prayers have played a pivotal role in my success.

Lastly, I want to express my love and gratitude to my husband and daughter, Alisha. Farhan, I feel incredibly lucky to have you as my life partner. It is no surprise to me why your family calls you a diamond of a person. Your love, support, and belief in me kept me motivated during this journey, especially during challenging times. Your patience when I was overwhelmed was remarkable, and I cannot thank you enough for taking care of Alisha during the final stretch of my thesis without complaint. You always knew how to lift my spirits when I was stressed. I genuinely

believe I could not have completed my PhD without you. Alisha, my lovely daughter, you have brought us so much joy and laughter. Thank you for being such a cheerful and easygoing baby; you made the difficult final days of my PhD journey much more bearable.

Hifza Khalid

TUFTS UNIVERSITY

February 2025

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the age of high-speed internet, most large information systems are structured as distributed systems Distributed Systems with components running on several machines [85]. Optimizing the performance and resource utilization of distributed and cloud-based systems Cloud Computing is hindered by the complexities of hardware and software configuration tuning, accurate workload prediction, and efficient task scheduling. To address these challenges, there is a need for advanced methodologies in addressing performance optimization challenges and advancing our understanding of resource management in complex systems.

The performance of distributed systems is generally characterized by their throughput and response time. It is very difficult to determine the causes of poor performance because of the complex interactions between different sub-components and the possibility of the problem occurring at many possible places along the communication path [4]. The problem can occur within the network or at the end-hosts. On the fastest networks, the performance is limited by the host's ability to generate, transmit, process, and receive data [19]. Within the end-hosts, the bottlenecks could be the application, the operating system, the network adapter, or any other unit involved [96]. Our first project aims to improve distributed system performance by optimizing hardware and configuration parameters of the machines involved.

Businesses today often perform resource-intensive computations on the cloud, which relies on shared clusters within data centers to reduce costs and enhance re-

source utilization [106]. While co-location improves machine utilization, it introduces challenges such as security risks, scheduling difficulties, and performance interference [55, 102]. To address these challenges and improve cloud operation, efficient planning and optimization is required [41]. Our second and third projects tackle these challenges through a comprehensive approach combining Machine Learning (ML) Machine Learning based workload modeling with the application of deep reinforcement learning (DRL) for task scheduling optimization.

1.0.1 First Project: Tuning System Configuration

Improper configuration settings can lead to performance degradation of up to 30% [90], which increases operational costs and affects user satisfaction. Achieving optimal performance in distributed applications requires a deep understanding of the hardware and configuration parameters involved. Previous studies, such as those by Acher et al. [1] and Chase et al. [20], have focused primarily on application-layer or transport layer tuning, neglecting the Linux system configurations that are essential for network performance [88]. This oversight is significant, as no previous work has leveraged large datasets to optimize Linux configurations specifically for network performance. Given the large parameter space [2] and complex interactions between them, tuning hardware and configuration settings in distributed systems remains a challenging task. Although considerable efforts have been made in this area [1, 20, 21, 101], these have generally targeted application or transport layer optimizations. Our project aims to address this gap by focusing on optimizing both hardware and system configuration parameters in distributed systems.

Red Hat, a provider of enterprise open source solutions, provided us with a large database of benchmark runs covering different hardware and Linux configurations, with different workload characteristics. Our goal was to use this data to automate the process of Linux configuration tuning, a process that typically involves running a benchmark application, monitoring the results and using educated guesses coupled with years of experience to tune the parameter values, until the performance of the application is as expected or the hardware components causing its sub-optimal

performance are identified.

We began working toward our goal of configuration tuning by selecting an initial set of hardware and Linux configuration parameters. Since these parameters were not necessarily the most effective in changing network performance, we used various feature selection methods to eliminate the redundant parameters and find a smaller set of parameters directly impacting network performance. Our results showed that network performance was mainly a function of hardware parameters and workload. This was an unexpected result and we became curious to investigate it. Analyzing the dataset exposed that users who ran these network benchmarks did not typically change the operating system configuration substantively and therefore, we did not have a sufficiently diverse sample of data. Our research also revealed other limitations of the dataset that were only apparent after visualizing the data. Based on our work, we recommend a set of preliminary experiments for researchers looking to determine the worthiness of a dataset for performance tuning and share some of our findings from the dataset analysis.

1.0.2 Second Project: Predicting Workload Characteristics

Co-located workloads can result in resource utilization inefficiencies of over 30% [24], impacting overall system performance and operational costs. As cloud environments increasingly adopt co-location strategies, understanding these workloads becomes critical. Bergsma et al. [8] modeled cloud VM workloads but did not consider the impact of co-location. Costa et al. [28] utilized statistical methods but focused on a centralized architecture, missing the nuances of our problem. Existing studies do not cater to the complexities of managing online and batch services with separate schedulers.

Our second project centers on developing accurate cloud workload models for improved decision-making and planning within cloud management systems. The task of accurately modeling these workloads is inherently challenging due to the "heterogeneous" and "imbalanced" nature of the cloud [98]. Modeling co-located jobs presents an even greater challenge due to additional factors such as interference,

resource contention, complex inter-job dependencies, varying resource demands and isolation requirements, for which simplistic modeling techniques are inadequate.

Addressing this gap, we propose a Machine Learning (ML) based approach to workload modeling using real-world cloud data. Alibaba, one of the leading cloud providers, co-locates transient batch tasks and high priority latency-sensitive online jobs on the same cluster. In our work, we consider the problem of modeling the batch tasks in the dataset that are often overlooked compared to online services. We use the dataset to train four different ML models. Specifically, the Autoregressive Integrated Moving Average (ARIMA) model is trained to forecast arrival counts, while three Long Short-Term Memory (LSTM) networks are trained to predict the CPU and memory requirements as well as lifetimes for batch services. To predict memory requirements, we use the predicted CPU requirements as input. Conversely, for the lifetime model, we use memory requirements as input. Our trained model predicts, with high accuracy, the number of batch tasks that arrive in any 30-minute window, their associated CPU and memory requirements, and their lifetimes.

This research was inspired by the work of Bergsma et al. [8], who modeled the production virtual machine workload from two real-world cloud providers, Microsoft, and Huawei, and demonstrated its applications in scheduling and capacity planning. While we found their work influential, it did not account for co-located workloads. Co-located workloads have become increasingly prevalent in modern cloud environments, with leading cloud providers like Google and Alibaba adopting the technique to enhance cost efficiency and optimize resource utilization [97]. Costa et al. [28] modeled Google's co-located traces using statistical methods and clustering techniques; however, their work did not address our specific problem. Google's cluster management system operates on a monolithic architecture, utilizing a centralized resource scheduler for resource allocation and management [24], whereas our focus was on online and batch services managed by separate schedulers. Moreover, Google's dataset does not contain workload (online and batch) specific information, making it challenging to characterize different workloads when co-located.

1.0.3 Third Project: Optimizing Resource Utilization

Inefficient resource packing can lead to wastage of up to 40% of available resources, which degrades system responsiveness and increases operational costs [108]. Effective resource allocation is crucial for enhancing service quality. Research by Parkes et al. [79] and others have focused on packing jobs but rely on static heuristics that do not adapt to changing environments. Recent studies, like those by Shanka et al. [73] and Alizadeh et al. [66], have applied Deep RL to address resource packing, but they focus primarily on policy-based methods. Our approach leverages policy and value-based methods, such as DQN, to specifically optimize resource utilization in batch services, an area where current literature is lacking.

Building on this, our third project focuses on cloud task-scheduling optimization. In large production clusters, multiple workloads are collocated on the same machine to achieve operational efficiency at scale [5, 84]. They share the underlying physical resources of the machine such as CPU, cache, memory, disk, and network-bandwidth. However, resources may be underutilized or over-utilized as a result of improper scheduling, which in turn leads to inefficient usage of cloud resources [50]. Resource management Resource Management in a cloud environment is even more challenging because cloud workloads exhibit highly dynamic variations and unexpected bursts in terms of task submission rates. As a result, the task scheduler must be designed to withstand and adapt to these dynamic changes in submitted tasks [100]. To efficiently use cloud resources, the task scheduler must target operational excellence by improving overall cluster utilization and minimizing resource fragmentation [73], while also meeting SLA requirements [81]. Finding optimum initial placement for the workloads is crucial as later migration has a high overhead and causes degraded user-experience [37].

Several solutions have been proposed for task scheduling including methods focused on scheduling problems for offline batch tasks, however they are not suitable for the highly dynamic workloads [100]. For the online task-scheduling methods, existing solutions use hand-crafted heuristics that cannot automatically adapt to the

change of the environment and optimize for specific workloads [62, 63]. Reinforcement learning (RL) approaches are particularly well-suited for resource management systems. They can model complex systems and decision-making policies as deep neural networks Neural Network analogous to the models used for game-playing agents [71]. Moreover, it is possible to train them for objectives that are difficult to tune for directly because they lack reward signals that correlate with the objective [66]. Finally, by continuing to learn, an RL agent can optimize for a specific workload as it changes over time [66].

In this work, we model the task scheduling problem as a deep reinforcement learning (Deep RL) task. A common practice to improve cluster resource efficiency is to maximally pack jobs on the least number of machines [107]. For example, systems like Google Borg [99] and Tetris [42] use multi-resource bin packing to minimize resource fragmentation and optimize server utilization. Therefore, our primary focus is on building a self-learning scheduler that maximizes cloud resource utilization by strategically delaying jobs and consolidating them onto the fewest number of machines possible. We specifically target batch tasks as they are less time-sensitive [54], thus the introduced delays are expected to have minimal impact. Furthermore, since most batch jobs are small and short in duration [43], any delays incurred to optimize packing efficiency are generally insignificant.

Deep RL techniques have been successfully applied to multi-resource cluster scheduling problems in the past [32, 66, 73, 109]. While some of this work focused on the policy gradient method REINFORCE [66, 73], others used a value-based algorithm such as a Deep Q-Network (DQN) [32, 109]. We investigate the use of both Policy Gradient and DQN approaches to enhance batch tasks scheduling. The differences observed between these two methods in terms of achieving optimal results, and computational time can assist in identifying the most appropriate approach for various scenarios.

1.0.4 Publications

Over the course of my PhD research, I have had the opportunity to explore various aspects of distributed and cloud systems performance. My research focuses on applying machine learning to enhance distributed system performance through configuration tuning, optimize cloud resource utilization via workload modeling, and improve task scheduling efficiency. Below are some of my recent contributions:

- **Linux Configuration Tuning: Is Having a Large Dataset Enough? [59]**: This paper examines the efficacy of large datasets in tuning Linux kernel configurations. We addressed the challenge of automatically tuning system parameters to achieve better performance in distributed applications, finding that while large datasets provide value, they alone may not be sufficient without context-aware tuning strategies.
- **Modeling Batch Tasks Using Recurrent Neural Networks Recurrent Neural Networks (RNNs) in Co-Located Alibaba Workloads [60]**: In this paper, we developed a predictive model for batch tasks in co-located workloads using recurrent neural networks (RNNs) to improve capacity planning and scheduling decisions in the cloud. We specifically focused on Alibaba's large-scale cloud infrastructure. Our findings indicate that our models can forecast batch task arrivals, resource requirements and lifetimes with high accuracy.
- **Optimizing Cloud Resource Utilization with Deep Reinforcement Learning [58]**: In this paper, we developed a deep reinforcement learning (Deep RL) based approach for optimizing cloud resource utilization, focusing on batch task scheduling in cloud environments. Our results demonstrate significant improvements in cloud resource utilization compared to heuristic-based methods by dynamically adjusting resource allocation based on real-time workload demands.

Chapter 2

Linux Configuration Tuning: Is Having a Large Dataset Enough?

This is an era of big data [87]. Large volumes of data are being generated everyday in a variety of fields, from commerce to healthcare and engineering to education [77]. Businesses, research and government institutions are now regularly generating vast amounts of data and are interested in gaining insights from it to increase their organizational value and gain a competitive edge [30].

Red Hat, a provider of enterprise open source solutions, provided us with a large database of benchmark runs covering different hardware and Linux configurations, with different workload characteristics. Most of the dataset was generated on private servers inside Red Hat while part of it was generated on public cloud systems, all by Red Hat employees. Our goal was to use this data to automate the process of Linux configuration tuning, a process that typically involves running a benchmark application, monitoring the results and using educated guesses coupled with years of experience to tune the parameter values, until the performance of the application is as expected or the hardware components causing its sub-optimal performance are identified.

The data used in this study was not collected from real-world, uncontrolled environments; it was gathered in a controlled experimental setup. The dataset was

generated by Red Hat engineers studying various aspects of kernel behavior and performance. We do not have specific information regarding the cluster architecture employed during the experiments. The experiments we analyzed represents a relatively small subset of the data that was collected. Many aspects of the data collection were out of our control, including the actual architecture and potential concurrent co-location of other tests on the same test framework.

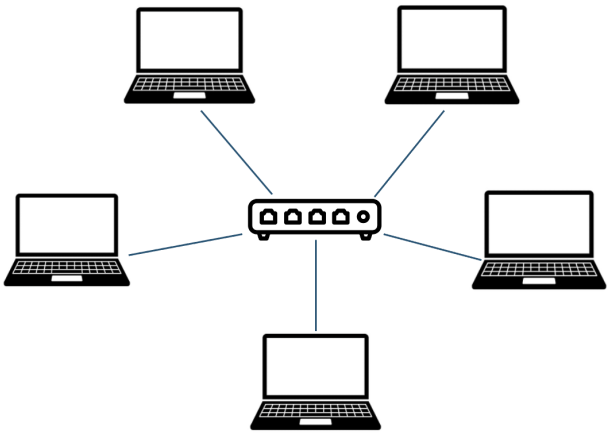

Figure 2.1: Private Servers inside Red Hat connected through a switch.

Although there were many benchmarks included with the data, we decided to initially work with the network benchmark because most large information systems are structured as distributed systems and their performance is often characterized by their network throughput and response time [86]. On the fastest networks, the performance of distributed systems is limited by the host's ability to generate, transmit, process, and receive data [20]. Since a large chunk of our dataset was generated inside Red Hat on machines connected through a network switch (as shown in Fig 2.6), where the network was stable and homogeneous, it was practical to isolate and study the impact of host configuration on network performance.

One of the most difficult problems in configuration tuning is to predict how different configurations behave with different applications and workloads. It is even more challenging with Linux, a complex system with more than $15,000$ unique configuration options [1, 2]. If each option is independent and has a binary value, this

leads to a minimum number of $2^{15,000}$ unique variants of system configuration [2]. With such a large search space, gauging the effect of all the possible settings can be extremely expensive and time-consuming.

We began working toward our goal of configuration tuning by selecting an initial set of hardware and Linux configuration parameters. Since these parameters were not necessarily the most effective in changing network performance, we used various feature selection methods to eliminate the redundant parameters and find a smaller set of parameters directly impacting network performance. Our results showed that network performance was mainly a function of hardware parameters and workload. Analyzing the dataset exposed that users who ran these network benchmarks did not typically change the operating system configuration substantively and therefore, we did not have a sufficiently diverse sample of data. Our research also revealed other limitations of the dataset that were only apparent after visualizing the data. Based on our work, we recommend a set of preliminary experiments for researchers looking to determine the worthiness of a dataset for performance tuning and share some of our findings from the dataset analysis.

The rest of the chapter is organized as follows: Section 2 describes data collection; Section 3 gives an overview of the dataset; Section 4 discusses our feature selection approach and results; Section 5 discusses visualization results from the dataset; Section 6 describes our experiments to assess data diversity; Section 7 proposes a series of experiments to evaluate the suitability of data for machine learning; Section 8 reviews the related work, and Section 9 concludes the chapter.

2.1 DATA COLLECTION

The dataset used in this work was collected using Pbench [95], an open source benchmarking and performance analysis framework developed by Red Hat. While Red Hat has been actively generating and collecting benchmark data for more than six years now, for the purpose of our work, we used recent data for eight months that were unpacked and prepared for our use. The older data did not concern sufficiently

modern kernel versions.

Pbench has built-in benchmark scripts that it can run alongside a variety of performance tools on multiple hosts within a distributed system while also collecting configuration information from all the systems involved. Pbench uses the sosreport utility [83] to collect configuration details, system and diagnostic information from hosts. The tool collects around 6,000 configuration files and the output of more than 200 commands from each specified host during each benchmark run. Pbench also provides users the flexibility to run their own benchmarks and record results in Pbench format.

The Pbench architecture consists of three major components: the Agent, Server, and Dashboard. The Pbench Agent provides convenience interfaces for users to run benchmark workloads to facilitate the collection of benchmark data, tools data (iostat, vmstat, etc.) and, system configuration information from all the hosts involved. The Pbench Server provides archival storage of data collected by the Agent and indexes data into Elasticsearch [12]. The Dashboard provides data curation and visualization of the collected data.

Pbench takes as input a benchmark type, desired workload, performance tools (e.g. sar, iostat, etc.) to run and hosts on which to execute the benchmark as shown in Figure 2.2. It outputs the benchmark results, tool results and the system configuration for all the hosts. The workflow of a Pbench run is shown in Figure 2.3. Given the input, Pbench creates a results directory on the same system, starts collection tools on all hosts, runs the workload generator and starts the benchmark. Once the benchmark run finishes, it stops the collection tools on all the hosts, and runs a postprocessing step that gathers results from all the remote hosts and executes postprocesssing tools on all of the data. This could include calculating averages, throughput and response time for various system operations. Pbench runs a test multiple times and returns the average performance results if all the runs are successful. A test fails when the standard deviation for the results of the repeated runs is more than a specified threshold.

Out of the several benchmark scripts that Pbench runs, we chose to study

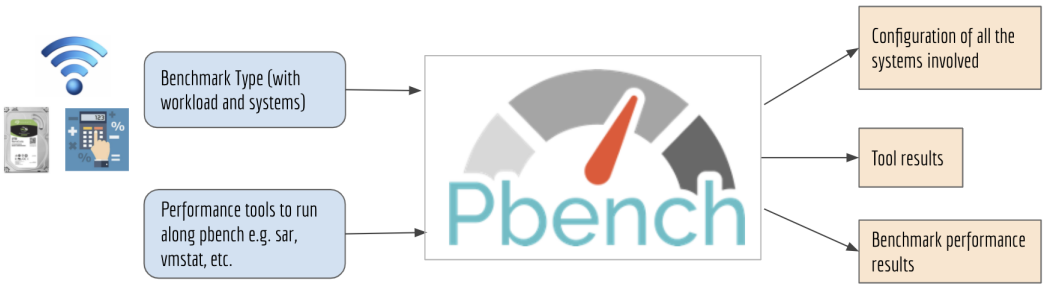

Figure 2.2: Inputs and outputs for Pbench.

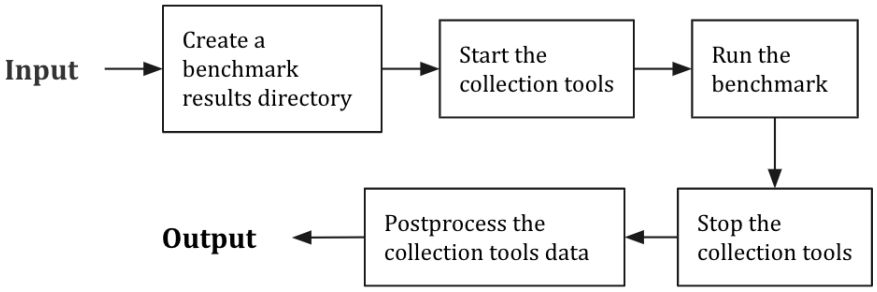

Figure 2.3: Workflow of a benchmark script in Pbench.

data collected using the uperf [75] benchmark for network performance. Other than uperf, Pbench also runs disk, CPU and user-created benchmarks.

2.1.1 Uperf - A Network Performance Tool

uperf is a network performance tool that can run in multiple settings: with single client and single server, with multiple clients and single server and, with multiple clients and multiple servers with one-to-one correspondence between them. The tool takes the description of the workload as input and generates the load accordingly to measure system performance.

uperf has seven inputs as shown in Table 2.1. The parameters *--clients* and *--servers* are used to specify the client and the server systems to run the test. *--test-type* is used to specify if the workload generated should be transactional (where response time is important) or streaming (where completion time for all tasks is important). *--message-size* tells the size of the messages in bytes, *--instances* shows the

number of open connections per host and *--protocol* represents the network protocol used for communication.

Table 2.1: uperf workload parameters and their description.

Input	Description
--clients	A list of one or more hostnames/IPs
--servers	A list of one or more hostnames/IPs
--test-type	Can be stream, maerts, bidirec, and/or rr
--message-size	Message size in bytes
--instances	Number of open connections per host
--protocol	TCP and/or UDP
--runtime	Test measurement period in seconds

2.1.2 Data Storage

Our prepared data was stored in two locations: the configuration data from the systems was stored on Red Hat servers while performance results and workload information was indexed in Elasticsearch. The mapping between Pbench runs and the corresponding configuration information was also stored in Elasticsearch. To interact with the data in Elasticsearch, we used Kibana [13].

2.2 DATASET OVERVIEW

Since the Linux configuration space is huge, we began with a smaller representative set of hardware and system configuration. We divided all the parameters of interest in the following three categories: C_{static}, $C_{dynamic}$ and P.

1. Immutable: invariants of the machine chosen. C_{static}
2. Mutable by the user. $C_{dynamic}$
3. Performance indicators. P
4. Not practically mutable by the user. "Buy a new machine." We think of these as part of C_{static}.

Based on these categories, we chose the initial set of C_{static} and $C_{dynamic}$ parameters shown in Figure 2.4. The hardware parameters were chosen based on feedback from experts at Red Hat and the SPEC benchmark [26]. Tuning parameters were chosen based on the performance tuning guide written by Red Hat [82] for RHEL. These included parameters from all major sub-components of a system including memory, disk, network, kernel and CPU that could impact network performance.

C_{static}	$C_{dynamic}$	$C_{dynamic}$
- Architecture	- Kernel	- file-max
- Model Name	- Thread(s) per core	- msgmax
- CPU (s)	- sched_rr_timeslice_ms	- msgmnb
- Core(s) per Socket	- nr_hugepages	- msgmni
- Socket (s)	- nr_overcommit_hugepages	- shmall
- L1d cache	- overcommit_memory	- shmmax
- L1i cache	- dirty_ratio	- shmmni
- L2 cache	- dirty_background_ratio	- threads-max
- L3 cache	- overcommit_ratio	- swappiness
- MemTotal	- max_map_count	- aio-max-nr
- Machine Type	- min_free_kbytes	
- NIC Port	- NIC Speed	
- NIC Supported Link Modes	- NIC Auto-negotiation	
	- NIC Duplex	

Figure 2.4: Hardware and Linux configuration parameters chosen for analysis.

We considered both transactional and streaming workloads for our work. In a transactional workload (which is sometimes called “interactive"), latency between request and response is important, while in a streaming workload (which is sometimes called “batch processing"), only throughput is important. For transactional workload, the performance metrics in the dataset are throughput in trans/sec and latency in usec. For streaming workloads, the benchmark only measured throughput in Gb/sec.

2.2.1 Testing Environments

The dataset used in this work contains benchmark runs executed in two different environments, either on private servers inside Red Hat or in public clouds like Ama-

zon EC2, Microsoft Azure, and the IBM public cloud. Table 2.2 shows the total number of network benchmark runs in out dataset executed internally at Red Hat and externally in public clouds. As the statistics show, more than 90% of the benchmark data was generated on Red Hat systems. Based on these results, we chose to work only with the data collected inside Red Hat as the public cloud data was not necessarily commensurate or comparable. Co-location of benchmarks with other unknown tenants in public clouds made that data difficult to compare with the Red Hat data.

Table 2.2: Statistics for tests conducted in different environments.

	Total		Single Client Server	
	Private Servers	**Public Clouds**	**Private Servers**	**Public Clouds**
Benchmark Runs	1989	152	1829	86
Iterations	204952	178840	166789	39302

Since the goal of our work is to tune Linux configuration, and distributed systems can have a complex mesh of hosts including multiple clients and multiple servers, we scoped down our problem to focus only on systems with a single client and server to avoid exploding the configuration space. Table 2.2 shows the counts of network benchmark runs with the simplest scenario of single client and server and the corresponding testing environment. As the results shows, 92% of the data generated internally used the single-client-server setting while runs outside Red Hat used this setup for only 56.6% of the cases. The table also shows the count for the number of iterations in each scenario. An iteration is a Pbench run with a given workload configuration. A single Pbench run can have multiple iterations depending upon the number of unique workloads that the run is tasked with testing.

2.2.2 Missing and Erroneous Data

We found $1,964$ total uperf runs with a single-client-server setup in our dataset. As we continued working with these runs and began collecting their configuration and performance information, we came across several cases where the data we required was missing, erroneous, or outside the scope of our work. The distribution of all

these cases is shown in Fig. 2.5. We eliminated 49 runs from our dataset because configuration information was missing (Table 2.2 does not show these runs) and 261 runs were eliminated because they used the same host for client and server, thus not using the network at all. 500 runs did not specify client and server IP addresses used for communication. This information was necessary for us to determine the physical network interface card (NIC) used by the hosts, but was missing for almost 25% of the runs. 21 runs had erroneous mapping of runs to configuration information, 104 runs had missing client or server configuration and 85 of the remaining runs were on external public clouds. After eliminating all these runs, our dataset contained 944 runs that we used for the rest of our work.

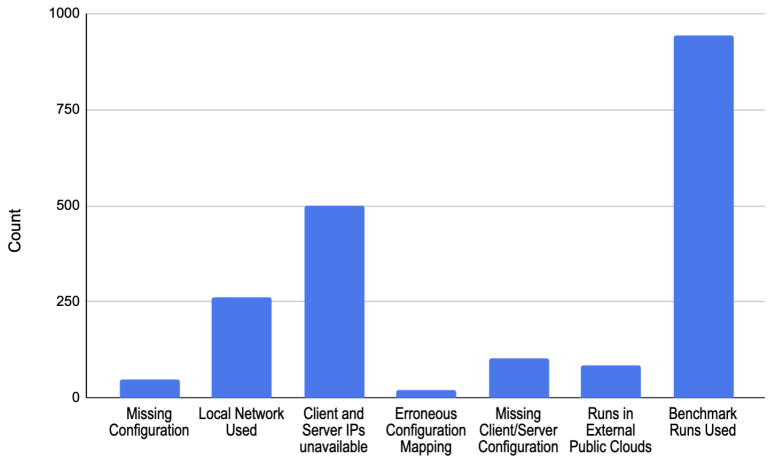

Figure 2.5: Distribution of missing and erraneous data.

2.3 CHOOSING SIGNIFICANT FEATURES

Since all the parameters that we selected for our work may not have been relevant to network performance, after selecting the preliminary set of static and dynamic features, we used common dimensionality reduction techniques to eliminate the redundant parameters. First we removed parameters with constant values. We then calculated the correlation between configuration parameters and the target variables and eliminated all the parameters with value less than $|0.1|$, a commonly used threshold to identify the uncorrelated pairs.

At the end of the above dimension reduction, we still had too many parameters, and considered further feature selection via embedded methods such as Lasso and Tree based methods [3]. We chose to work with tree-based embedded methods for their simplicity, flexibility and low computational cost compared to other methods. Within the tree based methods, we had three options: CART [68], Random Forest [103] and XGBoost [94].

Using these three feature selection decision tree methods, the final set of significant features for the client and server systems for all three types of target variables is shown in Table 2.3. The results show that most of the features that are significant in determining network performance are hardware parameters except a few configuration options for the network interface card that also obey a hardware constraint i.e. *NIC Duplex* and *NIC Speed*. The network speed can be configured up to the maximum capacity of the network card.

Table 2.3: Significant workload, hardware and configuration features.

	Throughput (trans/sec)	**Latency (usec)**	**Throughput (Gb/s)**
Workload	instances	protocol, message-size	instances, message-size
Hardware and Linux parameters	server_Speed, client_Type	server_Port, server_CPU(s), client_Duplex, server_MemTotal	server_Duplex client_Speed

Table 2.3 also shows workload parameters that are the most effective in changing network performance. The results suggest that the benchmark input *instances* is mainly responsible for throughput performance of transactional workload. For streaming workloads, *message-size* is also significant. For latency performance of transactional workload, both *protocol* and *message-size* impact the results.

These results were surprising for us since we expected at least a few linux parameters to play an important role in network performance. For example, [82] describes a tuning deamon that RHEL users can use to adapt system configuration for better performance under certain workloads. For improved network performance, the tool sets *vm.dirty_ratio* to 40%.

2.3.1 Hidden Parameters

Since it was not guaranteed that we included every significant feature in our initial set of parameters, we devised a strategy to determine the hidden parameters. We formed groups of data based on specific values of the significant features chosen in the previous step. For each of these groups, we calculated normalized standard deviation for performance results. We compared the configuration data from certain runs in these groups; substantial standard deviation indicated that there were likely to be unobserved variables affecting performance in that group. The results revealed *kernel version* to be a hidden variable. Apart from it, the amount of memory and cache used also varied in these runs. While including kernel version as a significant feature reduced the normalized standard deviation in groups, it did not lower it to an acceptable level because our dataset did not contain all configuration parameters and some system variables, such as free memory and cache, were not included in our feature set.

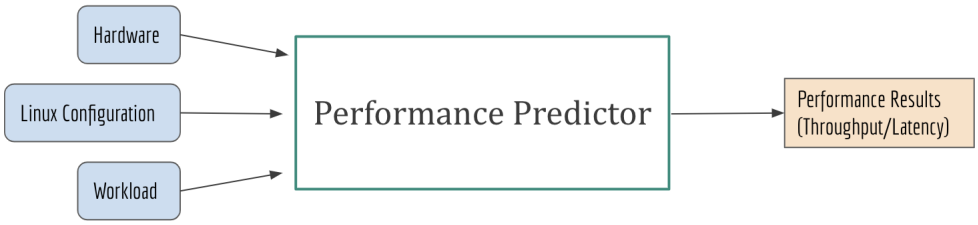

Figure 2.6: Predictive Model for Network Performance.

2.3.2 Predictive Model

As part of a system that recommends configuration, we wanted to inform the user of performance gains that can be achieved by optimizing the configuration. To do so, we developed a predictive model for the transactional workload to show how the user's current and new configuration perform. We developed the Random Forest (RF) [103] prediction model since it is known to perform better then CART [68] and is also easier to visualize. It also performed better than multiple linear regression with our dataset. The model takes significant workload, hardware and Linux

configuration parameters (chosen in the previous step) as input and outputs the network performance gains. We constructed the model independently for throughput and latency since the significant features for the models were different. To evaluate our predictive model, we used repeated k-fold [53] cross validation technique. The accuracy of the model was calculated using the two metrics given below.

1. **R2 score**: It is the proportion of the variance in the dependent variable that is predictable from the independent variable(s). Its value varies between -1 and 1 where 1 is the best possible score and negative values indicate that the model is performing arbitrarily worse.
2. **Root mean squared error (RMSE)**: It measures the average squared difference between the estimated values and the actual values and returns its square root. The smaller the value of RMSE, the better the model.

Based on the above two criteria, the results for the predictive model with throughput and latency as target variables are shown in Table 2.4. The results show that the RF models for the two target variables fit the observed data pretty well.

Target Variable	R2 score	RMSE
Throughput (trans/sec)	0.984	0.012
Latency (usec)	0.930	0.027

Table 2.4: Predictive Model Results.

These results improved our faith in the dataset. In the next step, we visualized the data to study the space of latency and throughput results in relation to each other. The goal was to recommend to the user an ideal balance for latency and throughput based on their requirements and not only improve one performance metric at the cost of the other.

2.4 Visualization and Trends in the Dataset

Pbench measures both throughput and latency for a transactional workload. In this section, we discuss some of the visualizations created by combining distinct records of throughput and latency results for the same hardware, configuration and workload. The purpose of creating these visualizations is to study *the space of results* for various configurations because most of the users are interested in a balance between latency and throughput. Figure 2.7 shows the log scale distribution of throughput and latency results in our dataset for the transactional workload. We were surprised by the multiple populations in these plots. Further investigation showed that one of the workload parameters, instances per host, was responsible for clusters in the results as shown in Figure 2.8.

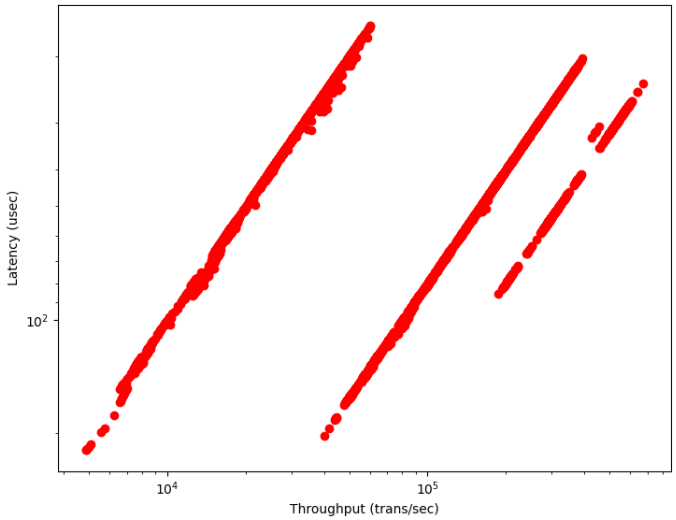

Figure 2.7: Throughput vs Latency for transactional workload.

The different clusters formed due to change in number of instances in Figure 2.7 have a linear correspondence in log space. If a user uses the same hardware and dynamic configuration with the same workload, only changing the number of instances will move the results from one cluster to another. This information could in principle be used to extrapolate predictions for workload combinations with varying numbers of instances that are not already measured.

Since we receieved a mix of expected and unexpected results in the feature

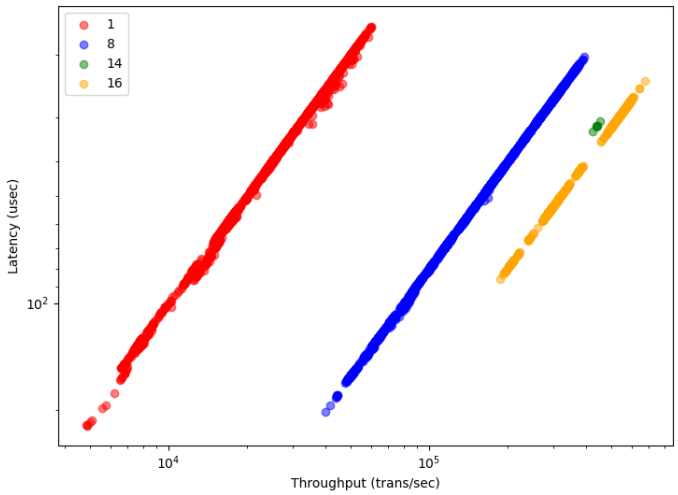

Figure 2.8: Number of instances causes clusters.

selection and the visualization phases of our work, we decided to analyse the dataset to investigate the cause of this.

2.5 TESTING THE DATASET FOR DIVERSITY

The hidden problem in our dataset was the lack of diversity of hardware, Linux configurations, and workloads. After eliminating missing and incorrect data, our dataset consisted of 944 unique network benchmark runs and 133555 total iterations within these runs. It contained separate iterations based on the result type, for example, we found two iterations for the same configuration and workload but different performance results i.e. throughput and latency.

2.5.1 Distribution of Hardware Specifications

The parameters that we considered static for the purpose of our experiments are shown in Figure 2.4. One of these parameters is "Machine Type" that can have two possible values, *physical* or *virtual*. Some of the static parameters in our set are mutable in virtual machines but not in physical machines. For the purpose of simplicity, we did not create further categories for virtual and physical systems that could have been used for a more accurate distinction between immutable and

dynamic parameters.

Table 2.5 shows the number of unique values for each of the static parameters in our dataset. As shown, the only static parameter with more than 10 unique values is *MemTotal*. Some parameters have missing values in the dataset. NIC parameters have 7.1% missing values whereas *L3_ cache* has only 3% missing values. The values for all other parameters including sizes for all other caches in the system are available.

Table 2.5: Counts of unique values for static parameters.

C_{static}	Client	Server
MemTotal	80	77
Model Name	8	10
Supported Link Modes	8	7
CPU (s)	8	7
Socket (s)	6	6
Cores per socket	4	5
Port	5	5
L2 cache	4	5
L3 cache	4	5
L1d cache	1	2
L1i cache	1	1
Architecture	1	1

2.5.2 Linux Configurations

To make useful configuration recommendations, we required a rich dataset comprising of several combinations of configuration values. To quantify the diversity of Linux configurations in our dataset, we counted the number of unique values for each of the configuration parameters that we shortlisted for our tuning recommendations. As the results show in Table 2.6, there are only three configuration parameters for the client and the server with more than 10 unique values. These are *file-max*, *kernel* and *threads-max*. For brevity, we did not include counts in the table for all the parameters with 1 or 2 unique values. None of the dynamic parameters have any missing values in the dataset except parameters associated with the network

interface card i.e. *NIC speed*, *NIC Duplex* and *NIC Auto-negotiation*. For client, each one of them has 7.1% missing values while for the server, they only have 6.5% missing values.

Table 2.6: Counts of unique values for dynamic parameters.

$C_{dynamic}$	Client	Server
file-max	123	86
kernel	73	71
threads-max	56	55
NIC speed	7	7
min_free_kbytes	4	5
dirty-ratio	3	3

2.5.3 Workload

While uperf takes in seven fields as input as shown in Table 2.1, some of these parameters like client or server are only there to identify the end hosts for communication. Similarly, the *runtime* field specifies for how long the workload should run. The workload is mainly generated based on four fields: *test_type*, *message_size*, *instances* (open connections per host) and the type of network *protocol*. The unique values for these fields and their percentage frequency in the dataset is shown in Tables 2.7, 2.8 and 2.9. As shown in Table 2.7, *rr* (request-response traffic) is the most frequently occurring test type. For this particular test type, we have separate records for throughput and latency performance measurements.

Table 2.7: Distribution of different test types in the dataset.

Test Type	% Frequency
rr	54.43%
stream	29.28%
bidirec	8.46%
maerts	7.84%

For message size, there are only four values in bytes that have been tested frequently in the dataset, 64, 256, 1024 and 8192. The message size of 1 byte was only tested once and may have been used for a sanity check. Our results for

Table 2.8: Distribution of different message sizes in the dataset.

Message Size in bytes	%Frequency
1	0.03%
64	25.79%
128	0.35%
256	24.27%
512	0.35%
1024	25.06%
8192	23.42%
16384	0.72%

Table 2.9: Distribution of different number of instances in the dataset.

Number of Instances	%Frequency
1	47.08%
2	0.00%
4	0.18%
8	43.70%
14	0.11%
16	8.87%
43	0.04%

instances show that a large number of tests only used 1, 8 or 16 open connections for communication with 1 being the most frequently occurring value. All the remaining values other than these three have been used $< 1\%$ in the dataset. For the network protocol, *UDP* was only used in 2% of the tests. In all the remaining tests, *TCP* was used for communication.

2.5.4 Diversity in Complete Dataset

It may seem based on the diversity results that we may have a large number of unique static and dynamic configurations in our dataset. Fig 2.9 shows the total number of unique static and dynamic configurations for the clients and the servers included in our dataset. We found only about 300 unqiue static configurations for the client and the server in our dataset of $133,555$ records. The number of dynamic configurations are almost double the number of static configurations, but they are still not diverse enough to make meaningful recommendations.

For further analysis, we also looked into workload diversity. We found 131

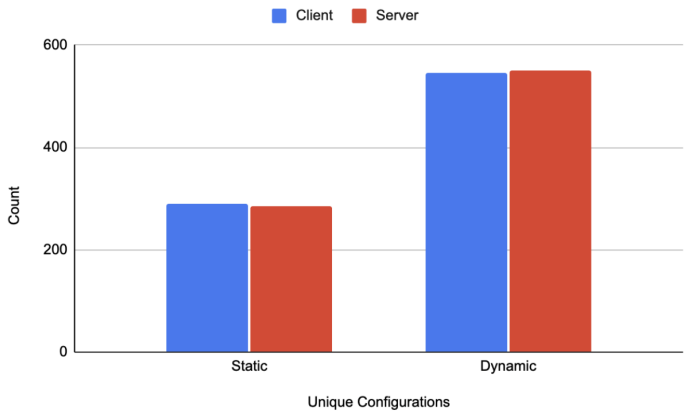

Figure 2.9: Count of unique static and dynamic configurations in the dataset.

different workload combinations part of the dataset. These were tested with a total of 126 unique client and server pairs. When we looked at the unique machine IDs in our dataset, they came out to be 23 for both client and server. This indicated that a large number of these tests were conducted on the same physical hosts but with a few changes in static and dynamic configurations. Our results also showed that the dataset contained 63 and 65 unique NICs for client and server respectively. Since a large number of the machines in our dataset have more than one NIC, our results show that the users used different NICs for different tests.

Based on this analysis, we can conclude that there is insufficient diversity of hardware and Linux parameters in our dataset that might affect workload response, as well as insufficient variety of workloads that could judge the values of those parameters.

2.5.5 Diversity in Disk Benchmark Runs

We wondered if this was only true for the network benchmark data, and our preliminary results showed that the disk benchmark similarly lacked diversity. We picked runs with different kernel versions and listed all of the system parameter values different from the default. The results showed that users kept all the parameter values default except *file-max*, *threads-max*, *aio-max-nr*, *vm.dirty_ratio*, *vm.swappiness*

and *vm.overcommit_memory*. Since studies have shown other storage configuration parameters to also have an impact on performance [17], and most of these tuned parameters had less than 5 unique values, we concluded that the disk dataset is also not sufficiently diverse for machine learning.

2.6 Recommended Preliminary Tests for Machine Learning

Based on our experience, we recommend the following set of preliminary tests for anyone looking to apply machine learning on a large complex dataset.

1. For sanity testing, import or use only samples of the dataset. With complex datasets, importing the entire database and running preliminary experiments can be costly as well as time consuming. For example, data preparation for a sample is often considerably faster than for a large dataset. Moreover, running experiments on a large database can sometimes take hours to days. Improving program efficiency to run with such a large dataset can be another contributing factor in the slowdown. Because one sample may not be sufficient, the user must use multiple samples to verify the results.
2. Filter out any incomplete, erroneous or irrelavent data for the problem and calculate the average percentage of the data that is filtered out. This information could be useful to determine the size of the final dataset that the machine learning algorithms will use for learning.
3. Identify suitable dimensionality reduction and feature selection algorithms for the dataset and its use case and implement these.
4. Visualize the dataset to identify trends and patterns. This step may also reveal some of the limitations or constraints of the dataset that may not be apparent without visualization.
5. Determine diversity of the significant features. This step could be straightforward or complex depending upon how many different kinds of variables are

present. For example, in our case, we used workload, system configuration and performance parameters. It is necessary to look at all the different groups separately as well as together. It may also be useful to look at the distribution of values for the significant features since it is possible that a variable has diverse values but this diversity is too infrequent to be useful.

2.7 RELATED WORK

To achieve optimal performance for distributed applications, it is necessary to use suitable hardware and settings for different configuration parameters of the systems involved. It is an extremely challenging task to tune these settings because of the large parameter space [2] and the complex interactions between them. While there has been much work done in the area of tuning configurations [1, 20, 21, 101, 110], the efforts have generally been focused on the application or the transport layer of the system. Reference [78] uses a set of hardware parameters to predict performance for system design alternatives of single and multi-processor systems, but it does not take system configuration parameters into account or recommend hardware based on performance requirements. Reference [16] did some work in identifying important tuning parameters for the storage system and [18] applied several optimization algorithms to tune the storage system. However, we have not come across any work that focuses on tuning Linux system configuration to improve network performance and that uses a large previously collected dataset to do so instead of traditional optimization methods.

2.8 CONCLUSION AND FUTURE WORK

We began this work with the goal of tuning system configuration for improved network performance. We began with a large dataset of network benchmark runs provided by Red Hat. To recommend settings for Linux configuration, we selected a subset of hardware and Linux parameters based on feedback from experts and performance tuning guide by Red Hat. We used various tree-based feature selection

methods to identify the parameters that impact network performance significantly. Our results showed that all significant parameters are part of the hardware configuration, and none of them concern Linux configuration. Investigating these results, we found that our dataset lacked in diversity for Linux configurations. Visualizing the data revealed a few other trends and limitations of our dataset. Based on these experiments and their outcomes, we concluded that the users of pbench did not alter the system configurations substantively and that one should not take data diversity, even of huge datasets, for granted. If we had attempted machine learning with this dataset, it would have failed. We also recommend a set of preliminary tests for anyone working on similar complex datasets and planning to use machine learning.

One limitation of our work is that we only looked at Pbench data, available to us through Red Hat. There might be other datasets out there that might be a better fit for Linux configuration tuning, but these are not easily available due to privacy concerns. If a sufficiently diverse dataset becomes available, we would like to attempt tuning with that data as future work. Also, our analysis of the dataset filters features by correlation and thus might ignore non-linear relationships. These could be studied in the future as well.

Chapter 3

Modeling Batch Tasks Using Recurrent Neural Networks in Co-located Alibaba Workloads

Businesses today are routinely required to perform resource-intensive computations but often lack sufficient on-site resources. As a consequence, many computational jobs are offloaded to the "cloud". The cloud refers to off-site resources that may be accessed via the Internet. Such cloud services run on shared clusters within data centers to lower costs and improve resource utilization [106]. Jobs from different parties are co-located on the same machines. While co-location improves machine utilization, it poses a number of challenges to the data center, including security (isolation between different services), scheduling and performance interference [55], [102]. Additionally, different jobs or services may contend for the same resources causing service delays that affect Quality of Service (QoS) of applications [22].

To address these challenges and improve cloud operation, efficient planning and optimization is required [41]. For example, through better planning of which resources to provision and when, capacity planners can proactively support future workloads while trying to avoid resource shortage and contention issues [8]. Contention can negatively effect performance and efficiency of co-located workloads. It

leads to increased pressure on memory resources due to increased paging and swapping activities, all of which ultimately lead to QoS degradation and unpredictable application behavior. By understanding the properties and behavior of co-located workloads from real production environments, we can improve decision making in the cloud. [64] characterized a trace of co-located workloads from Alibaba's production cluster to study some of these properties such as the heterogeneity of clouds. We, on the other hand, propose the development of a workload prediction model to provide better estimates of future workloads for improved scheduling and capacity planning decisions.

Accurate cloud workload models are valuable for improved decision-making and planning within cloud management systems. However, the task of accurately modeling these workloads is inherently challenging due to the "heterogeneous" and "imbalanced" nature of the cloud with respect to resource allocation and lifespan [98]. Modeling co-located jobs presents an even greater challenge due to additional factors such as interference, resource contention, complex inter-job dependencies, varying resource demands and isolation requirements, all of which render simplistic modeling techniques inadequate.

Addressing this gap, this chapter proposes a Machine Learning (ML) based approach to workload modeling using real-world cloud data. While this method is expected to be accurate and realistic, the availability of such data is a challenge. Cloud providers are generally reluctant to publicly release their data [15]. Even when data is available, it is often limited, making it challenging for reliable training of ML algorithms. In this chapter, we work with one such dataset from Alibaba [5]. A workload model derived from such a dataset can not only be used for better planning decisions in cloud environments, but also for generating realistic synthetic workloads, which, in turn, can proactively support predicting needed increases in system capacity without large downtime or data gathering [8].

The Alibaba dataset considered in this work consists of traces of co-located workloads over an eight day period [5]. This set of workloads includes both online services and batch workloads. We focus on modeling batch workloads because online

services are guaranteed resources due to their high priority, while batch jobs are executed on the remaining resources left on the servers. We hope that modeling batch workloads can lead to their improved resource utilization, performance and efficiency. Batch jobs in Alibaba's dataset are divided into tasks, where task executions are subject to dependency constraints. These tasks are further divided into instances that have the same binary code and resource requests but differing input data. We model batch tasks in our work as they are the smallest unit of batch jobs for which we have information about resource requirements and completion times. This low-level model can be readily used to model batch jobs if needed.

Our model, illustrated in Figure 3.2, uses the Alibaba dataset to predict arrivals, associated resource requirements, and lifetimes/completion times for batch tasks. For modeling arrivals, we use the Autoregressive Integrated Moving Average (ARIMA) model [35]. This popular time series forecasting technique is particularly effective at capturing trends and seasonal patterns in data. By using ARIMA, we can account for the cyclical nature of task arrivals, allowing us to provide accurate forecasts based on historical trends. Furthermore, ARIMA models are interpretable, offering insights into the underlying data structure that can inform decision-making. In contrast, we use a Long Short-Term Memory (LSTM) based neural network to predict resource requirements and completion times. LSTMs are well-suited for handling sequential data and capturing long-term dependencies [89]. This capability is crucial for predicting resource demands, which often rely on historical usage patterns. LSTMs excel at modeling complex nonlinear relationships, making them ideal for dynamic cloud environments where resource requirements and task lifetimes can fluctuate unpredictably.

By leveraging both ARIMA and LSTM in our framework, we capitalize on their complementary strengths. The ARIMA model effectively addresses seasonal patterns in arrivals, while the LSTM handles the complexities of resource demand and lifetime predictions. This dual approach enhances the accuracy of our predictions and ensures adaptability in the face of evolving cloud workloads.

Our model can reproduce the Alibaba dataset with very high accuracy. In

order to make practical predictions, our model must be able to generate random yet realistic workloads. This can be very easily realized by tuning parameters of the ARIMA model or by modeling the probability distributions over the resource requirements and lifetime through the use of Bayesian Machine Learning methods such as Gaussian Process Regression. Additionally, we can model task arrivals as a Poisson process, an approach adopted in [8] when modeling Virtual Machine (VM) arrivals in Microsoft Azure [27].

The remaining sections of the chapter are organized as follows: Section 2 discusses background and related research; Section 3 describes the Alibaba dataset and our approach to model the batch tasks; Section 4 explains the setup used for training models; Section 5 presents the results from the experiments; and finally, Section 6 concludes the chapter.

3.1 BACKGROUND AND RELATED WORK

Bergsma et al. [8] modeled the production virtual machine workload from two real-world cloud providers, Microsoft and Huawei, and demonstrated its applications in scheduling and capacity planning. While we found their work inspiring, it did not account for co-located workloads. Co-located workloads have become increasingly prevalent in modern cloud environments, with leading cloud providers like Google and Alibaba adopting the technique to enhance cost efficiency and optimize resource utilization [97]. Costa et al. [28] modeled Google's co-located traces using statistical methods and clustering techniques, however, their work does not address our specific problem. Google's cluster management system operates on a monolithic architecture, utilizing a centralized resource scheduler for resource allocation and management [24], whereas our focus is on online and batch services managed by separate schedulers. Moreover, Google's dataset does not contain workload (online and batch) specific information, making it challenging to characterize different workloads when co-located.

Acquiring realistic workload data for modeling is challenging as most cloud

providers, apart from a few exceptions such as Google [84], Alibaba [5], and Microsoft [27], are reluctant to disclose their data. Additionally, scholarly papers rarely provide information about their data collection methods or even release it for reproducibility. For this reason, we selected Alibaba's publicly available dataset, which offers distinct information for batch and online services, enabling us to delve deeper into the characteristics of co-located workloads.

Within the Alibaba dataset, we opted to initially focus on modeling batch services. This choice stems from the observation that batch services generally utilize more CPU resources than online services [64]. Furthermore, due to the prioritization of online services, they are executed within containers that receive a dedicated allocation of resources, leaving only a limited set of resources available for batch services. This allocation strategy ensures the availability of resources for online services at all times. Therefore, by gaining insights into batch workloads, we aim to enhance job scheduling for batch services and optimize resource provisioning for co-located workloads. To the best of our knowledge, we have not come across any existing work focused on modeling co-located workloads or exclusively batch services.

We now briefly discuss some of the past research in cloud workload modeling. Moreno et al. [74] have previously modeled arrival rates, resource requirements, and job duration for specific users in a Google cloud trace. In contrast, our approach does not rely on user-specific information, allowing us to apply it more broadly to model large-scale future workloads. Similarly, a workload generator is presented by Bahga and Madisetti [7] to evaluate cloud applications. They simulate user behavior with inter-session times and session duration. A number of papers focus solely on modeling job arrival rates [57, 61], whereas our work models task arrivals, resource requirements, and task completion times within batch services. In addition, there has been a lot more work on VM scheduling/co-location rather than workloads in clusters [27], where the challenges as well as the solutions are not necessarily applicable to our problem.

One of the most popular stochastic models in time series forecasting is the ARIMA model developed by Box and Jenkin [35]. It can capture noise, trend as

well as the seasonal component in the dataset [48]. Rodrigo et al. [14] used it to successfully predict hourly web requests to English Wikipedia resources. Due to its simplicity and flexibility, it has also been used to predict cloud coverage (weather) [104], tourist arrivals [25] and short-term resource usage in the cloud [52] with high accuracy. We used it to model the batch-task arrival counts in our dataset. To model the resource requirements and lifetimes, we used LSTM, a type of recurrent neural network which excels at capturing long-term dependencies. It has been used to model VM resource requirements and lifetime in the Microsoft dataset [27] by Shane et al [8].

3.2 OUR APPROACH

The Alibaba dataset contains more than 14 million data points. It captures co-located online and batch jobs from a cluster of 4034 machines over an eight-day period. The dataset is described in detail below.

3.2.1 Alibaba Dataset and Batch Task Modeling

Alibaba's cluster management system oversees resources for two different kinds of workloads: online and batch. It uses two different schedulers, namely Sigma and Fuxi, each of which operates with its own dedicated resource pool. Sigma is responsible for user-facing, long-running online services executed within containers, while Fuxi handles batch jobs executed directly on physical hosts as shown in Figure 3.1. To facilitate improved scheduling decisions, Sigma and Fuxi share cluster state information. The dataset collected from this system contains information about server metadata, server usage, container metadata, container usage, and batch tasks and batch instances, including information about status, resource usage, resource requirements, and arrival and completion times of submitted jobs.

Batch-processing applications utilize a predefined and limited amount of resources, with a low priority. When there are insufficient resources for a newly arrived online job, some or all of the batch jobs are preempted to free up resources.

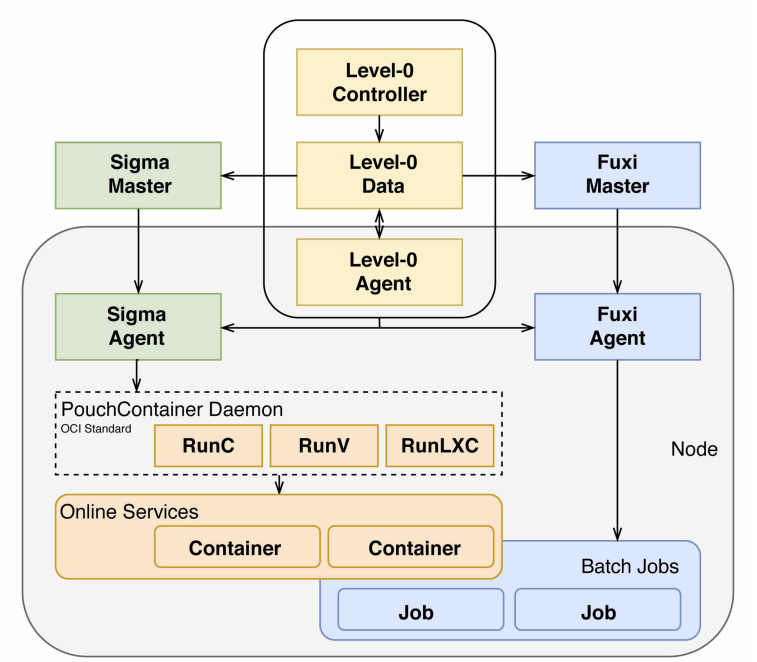

Figure 3.1: The architecture of Alibaba cluster management system [5].

While batch jobs are not latency critical, preempting them leads to overhead due to pre-emption and rescheduling. To avoid rescheduling, batch workloads are typically scheduled in windows when the arrival rate of online jobs is lower, e.g, late at night. Such policies clearly give latency-critical applications preference over batch-processing applications [44]. Modeling these processes (both online and batch jobs) is imperative for better analysis and better utilization of available resources. In this chapter, we focus on modeling batch jobs.

Each batch job consists of one or more tasks. These tasks can have dependencies, where the completion of one task predicates the completion of others, represented as a directed acyclic graph. Further, each task may create one or more instances with the same binary code and resource requests but with different input data. Such a task instance is the basic scheduling unit in Alibaba Cluster Management System. The duration of a job is the sum of its task durations. The duration of a task is the sum of the execution time of all its instances. The Alibaba dataset contains the following information with respect to the tasks (from some batch job):

1. Start and End times.

2. Requested CPU and memory resources

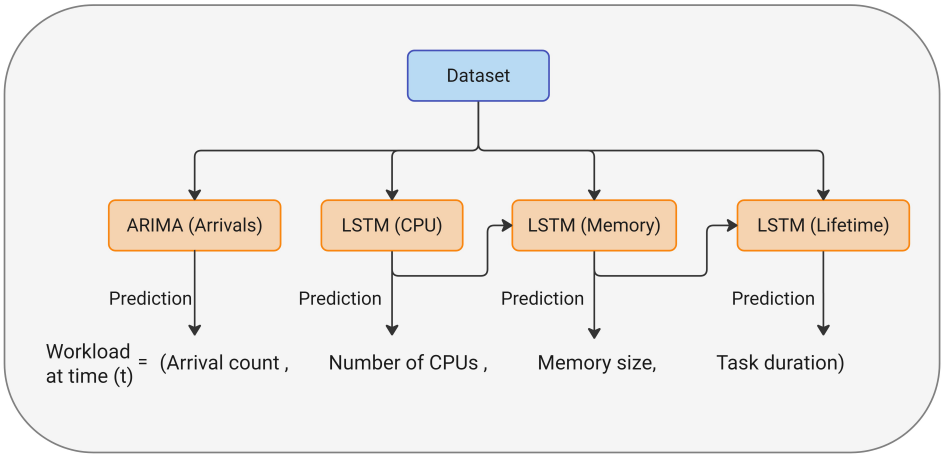

Figure 3.2: Illustration of the Workload Prediction Model.

3.2.2 Workload Prediction Model

Figure 3.2 illustrates our workload prediction model. Using the Alibaba dataset from Section 3.2.1, this model predicts the following:

1. The number of batch tasks that arrive within the t^{th} 30 minute window.
2. The number of CPU, memory requirements for each arrival within the t^{th} 30 minute window.
3. The lifetime of each arrival within the t^{th} 30 minute window.

The dataset is used to train four different ML models. Specifically, the Autoregressive Integrated Moving Average (ARIMA) model is trained to forecast arrival counts, while three Long Short-Term Memory (LSTM) networks are trained to predict the CPU and memory requirements as well as lifetimes. In order to predict memory requirements, we use the predicted CPU requirements as input. Conversely, for the lifetime model, we use memory requirements as input. In Section 3.3, we delve into the qualitative impact of using CPU requirements to predict memory and memory to predict lifetimes. Now, we provide an overview of ARIMA and LSTM networks, along with an explanation for our choice of these models.

3.2.3 Challenges in Modeling Arrivals

Initially, we attempted to model batch task arrivals using Poisson Regression [29], as used in [8]. We generated arrival counts for every 5-minute window in our data and used Day, Hour and time interval as predictors. Our dataset did not contain explicit information for these, but since it recorded time as elapsed seconds from the start of trace collection, we generated synthetic day and hour information. The time-interval for each entry was represented by an upper bound (in seconds) for every 5-minute window within an hour. The data for the 9th day was incomplete and thus excluded from the analysis. This approach of generating data was consistent with the methodology used by Shane et al. in [8], allowing us to apply Poisson regression effectively. They used timestamps provided along with the VM data [27] to extract information for the day and the hour, and used 5-minute windows along with synthetic user IDs to group arrivals into clusters. The results for our Poisson Regression model are illustrated below.

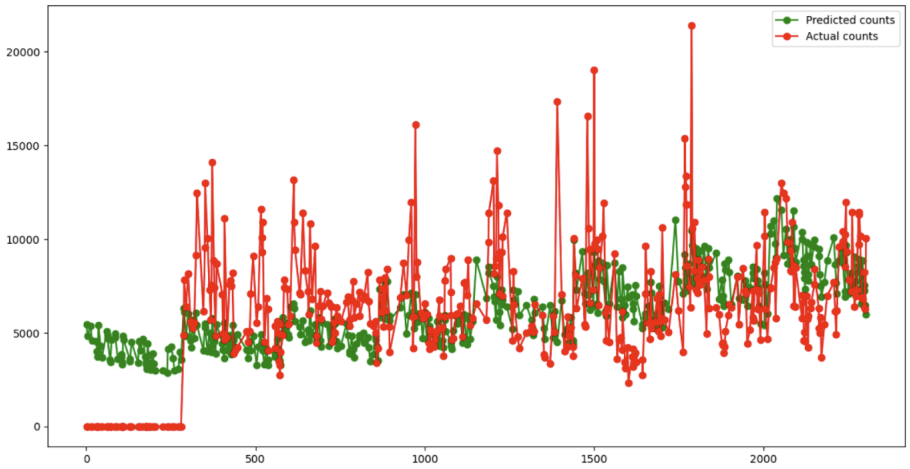

Figure 3.3: Poisson Regression with predictor variables: day, hour and time-interval.

While the results appear visually reasonable, the model had large Deviance and Pearson Chi-squared values that indicate poor model fit, as suggested by [29, 80]. This implies that the dataset does not align well with the assumed Poisson model. To explore alternatives, we used a Negative Binomial Regression model [29], which

produced similar results.

The data exhibits bursty behavior that results in deviations from the Poisson process. This burstiness likely arises from tasks that follow a Poisson arrival pattern but subsequently trigger additional tasks that do not. In a true Poisson process, arrivals are independent and do not cause or interact with others. More than 75% of batch tasks are dependent on at least one other task [5], violating the Poisson assumption.

In [8], group arrivals were modeled as Poisson, while the individual arrivals within each group were not. They focusing solely on the first event in each group and fitted these events to a Poisson distribution to achieve a close match between generated and observed data. This suggests that their group arrivals are memoryless, meaning that tasks arrive independently of one another, without coordinated behavior between users. In our dataset, we hypothesized that using "job id" could be the key to define group arrivals, identified when the first event in a job occurs. This might offer behavior closer to obeying the Poisson criteria, as batch tasks are mainly dependent on other tasks that belong to the same job.

The load generation framework based upon task groups had two phases:

- **Batch Arrival Generation:** Predict group arrivals using a Poisson model.
- **Batch Content Generation:** Populate the group with arrival events that are clustered closely together.

We tested models based on independent tasks using both Poisson and Negative Binomial regression, defining groups based solely on 5-minute time intervals. The results suggested that neither model is sufficient to fully capture the bursty nature of the arrivals, as exemplified in Figure 3.4: .

Next, we refined the group definition based upon batch `job_id`. We noticed the same `job_id` can reflect periods of activity followed by periods of relative inactivity. To capture the bursty periods without the inactivity, a group is defined as "a set of tasks from the same job within the same 5-minute period." After excluding data from Days 1 and 9 due to instability, the results were as follows:

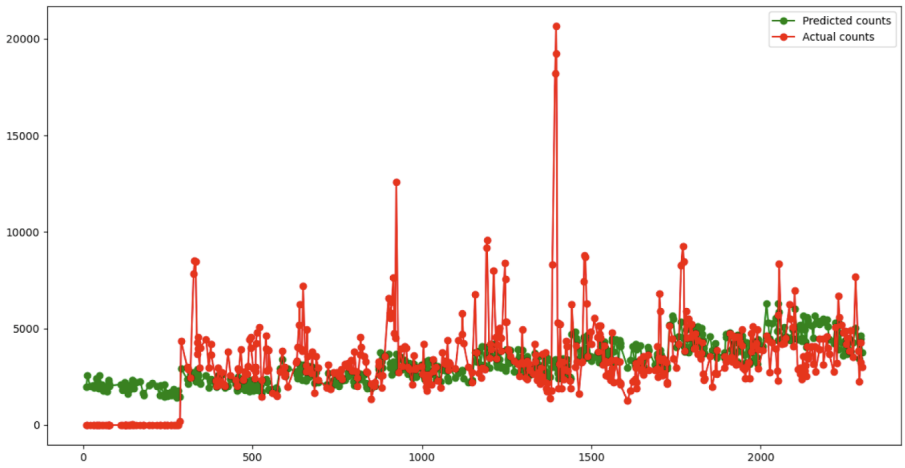

Figure 3.4: Poisson Regresson for Independent tasks.

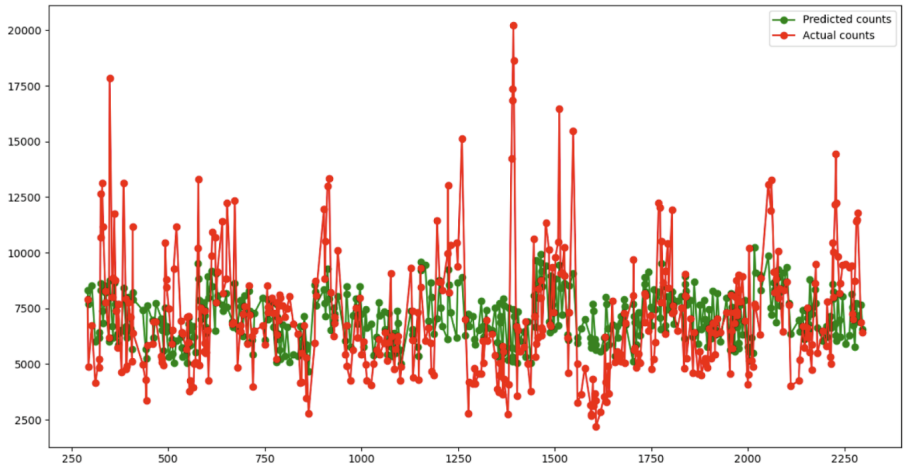

Figure 3.5: Poisson Regression using Job Ids.

Ultimately, none of these approaches proved effective, indicating a need to explore alternative techniques to model the bursty behavior of arrivals.

3.2.3.1 ARIMA for Arrivals

In the statistical parlance, the sequence of arrival counts within successive 30 minute windows constitutes non-stationary time series data. This data may exhibit variations, such as higher workload arrivals during the day compared to night. This trend may vary across days of the week, e.g., the weekends may be quieter. The ARIMA model is a popular statistical method that is often used to fit non-stationary time series data. It can also account for seasonal patterns in the data. In summary, the

ARIMA model has three key components:

- **"AR" (Autoregressive)**: This component accounts for temporal dependencies by regressing over the past values of the evolving variable - arrival counts in our case.
- **"I" (Integrated)**: It involves differencing the data to achieve stationarity, enabling more accurate predictions.
- **"MA" (Moving Average)**: This component considers moving averages to capture the average changes in values, which helps in understanding the evolving patterns of arrival counts over time.

These three components are defined by the primary model parameters: p, d and q, for the non-seasonal aspects of the data, and P, D, Q for their seasonal counterparts. Additionally, the model can be parameterized by the number of periods within every season, denoted as s. It is trained using the Box-Jenkins method [35], [89]. Recall that the Alibaba dataset contains the start times for batch tasks over an eight day period. As a preprocessing step, these start times are used to generate the time series dataset that is the number of arrivals within each 30-minute window. The ARIMA model is trained using this transformed dataset for prediction and analysis. The dataset is divided into 30-minute intervals, as using shorter intervals would result in longer seasonal periods, which can pose challenges for ARIMA modeling [51].

3.2.3.2 LSTM for CPU, Memory and Lifetimes

We used LSTM networks to predict the CPU and memory requirements, and the lifetime of a batch task. Unlike a regular feed-forward network, LSTMs are artificial neural networks capable of processing feedback [49]. They are composed of special long short-term memory units designed to capture temporal information effectively. LSTMs are particularly well-suited for time series forecasting applications, especially when datasets contain relevant events separated in time. In the Alibaba dataset, the task executions are governed by a dependency graph. Specifically, a task may only

be executed provided that predecessor tasks have already been executed. When predicting the resource requirements (CPU, memory) or lifetime, it is important to consider other related tasks that have been submitted for execution.

The Alibaba dataset features 16 distinct CPU and over 300 unique memory values. To address these different prediction tasks, we use two separate LSTMs. In particular, we train the CPU-LSTM as a 16-class classifier whereas the memory-LSTM is trained using regression. Additionally, we include requested CPU as an input feature when predicting memory. However, our findings in Section 3.3 show that the inclusion of CPU does not significantly enhance the accuracy of memory predictions; memory can be predicted accurately without explicitly considering CPU. Lastly, we consider the problem of predicting task lifetimes. In the dataset, each task is associated with one of four states: terminated, running, waiting or failed. Our analysis focuses exclusively on predicting successfully terminated tasks, which account for over 98% of the dataset. As in the case of memory, we employ an LSTM model trained through regression to predict the lifetime of a task. The LSTM takes the (predicted) memory requirement as an additional input.

3.3 EXPERIMENTAL SETUP

In order to present our numerical results, we need to first specify the setup used to conduct the various experiments. We used `Python 3.10` for all of our experiments. Our models are ML based, and Python provides sufficient libraries to implement them.

3.3.1 Data Preprocessing

Since we wish to predict the number of batch arrivals in a given 30-minute window, we begin by preprocessing the dataset to generate time series data containing task arrival counts in consecutive 30-minute windows. As each task is associated with a start and an end time, this preprocessing step is fairly straightforward. We use ARIMA to fit the resulting time series data. We use Python's `pmdarima` package

[11], and call on the function `auto-arima` for training on the arrival count time series.

The CPU and memory requirements, and the completion times are fitted using LSTM networks. As there are only 16 unique values for CPU, we solve the CPU prediction as a classification problem. For memory and lifetime predictions, we adopt a regression approach. We use the `keras` API, which is developed by Google and is a popular choice to train neural networks, to train our LSTMs.

3.3.2 Selecting Hyperparameters

We begin by noting the hyperparameters for the seasonal ARIMA model. In traditional ARIMA models, three key values must be specified: p, the number of autoregressive terms; d, the degree of differencing applied to make the data stationary; and q, the number of moving average terms. Seasonal ARIMA (SARIMA) models extend this by also requiring the specification of seasonal parameters: P (seasonal autoregressive order), D (seasonal differencing order), Q (seasonal moving average order), and s (the length of the seasonal cycle). The parameter s varies depending on the recurrent periodicity in the data. For instance, a daily periodicity corresponds to a value of 7, while weekly and monthly periodicities have values of 52 and 12, respectively. In our case, we are modeling day-over-day seasonality in an 8-day dataset, with arrival counts aggregated over 30-minute periods each day. Therefore, the value of s is set to 48, which corresponds to the number of 30-minute buckets in a 24-hour day. The ARIMA hyperparameters are tuned using the Alibaba dataset to increase prediction accuracy by the `auto-arima` function. The recommended model uses $p = 4$, $d = 0$, $q = 3$ in the non-seasonal part and $P = 2$, $D = 0$ and $Q = 1$ to model the non-seasonal components of the data.

Now, we look at the hyperparameters tuned for the LSTMs used to predict the resources and the lifetime. We use the same LSTM network for the three prediction tasks. In particular, all our LSTMs are single layered with 32 hidden LSTM activations. The classification-LSTM uses a soft-max output layer, while the other two LSTMs use a linear output layer. Since LSTMs are trained using the Back-

Propagation Through Time (BPTT) algorithm, we need to specify the number of steps in time that the BPTT algorithm must look back. Our models look 10 steps back in time. The optimizer used is the Stochastic Gradient Descent (SGD) with momentum algorithm. We use a decaying learning rate starting from 0.001 and a momentum value of 0.9. The learning rate decays as a function of the epochs. When it comes to the loss functions, the classification problem uses the cross-entropy loss, and the regression problems use the mean-squared loss.

3.4 EXPERIMENTAL RESULTS

We present the results from various experiments in this section. We start by looking at the task arrivals prediction model.

3.4.1 ARIMA - Arrivals

It is essential to eliminate non-stationarities in data for ARIMA to have a high prediction accuracy. There is an "initial differencing" step in ARIMA that is repeated a few times in order to eliminate non-stationarities. The number of repetitions are determined by the parameter d. In order to find the optimal d, we used the Augmented Dickey-Fuller (ADF) statistical test [36]. The general guideline for the ADF test is that if the p-value is less than the critical value of 0.05, the d differencing steps have eliminated trends. In our case, for the chosen d parameter value of 1, the p-value was $4.5e-07$ which is less than 0.05.

All time series data have four components: average value, trend (i.e. an increasing or decreasing mean), seasonality (i.e. a repeating cyclical pattern), and residual (random noise) [70]. Trends and seasonality are not always present in time dependent data, so we performed decomposition to identify any underlying seasonal patterns. Figure 3.6 illustrates the decomposition of the arrival counts data, where it clearly displays daily seasonality. As a result of this analysis, we decided to use SARIMA instead of ARIMA to model arrival counts.

Figure 3.7 shows the modeling results for the number of task arrivals per 30-

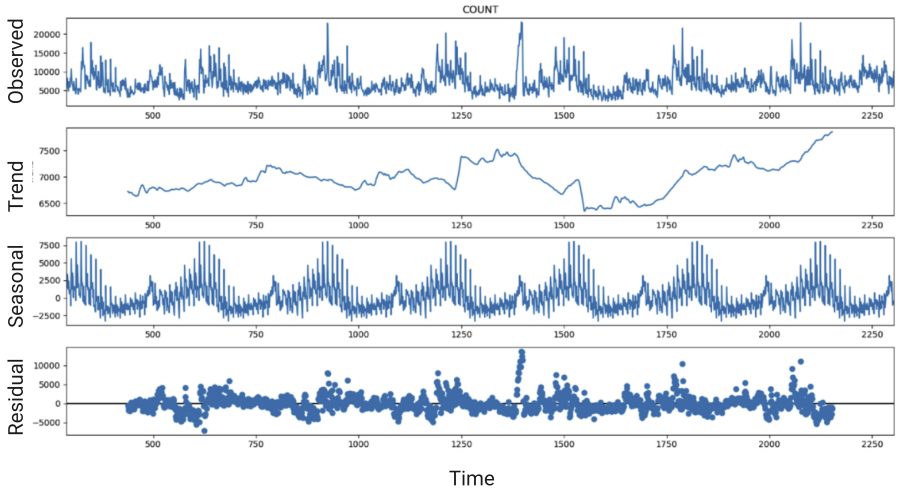

Figure 3.6: Time series decomposition of arrival counts.

minute time intervals using SARIMA. The model uses 70% of the data for training and 30% for testing. As shown, the predicted values effectively capture seasonality as well as the bursts in the data.

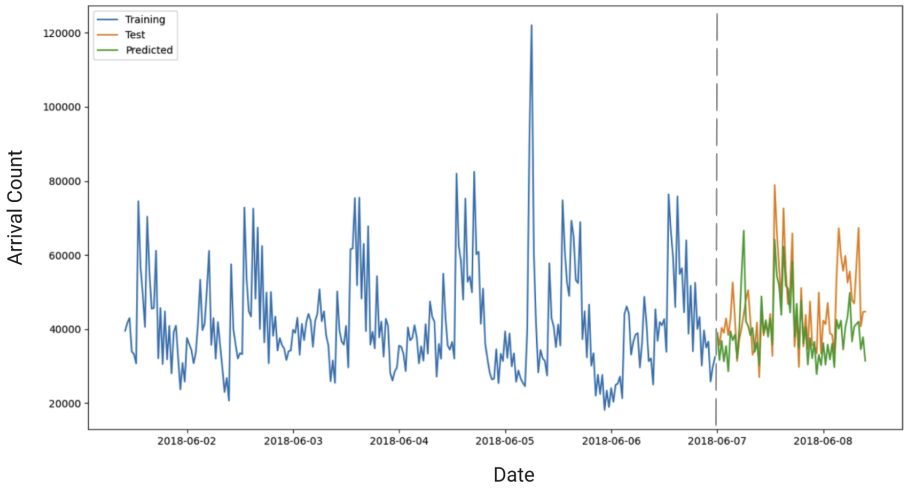

Figure 3.7: Prediction results for ARIMA model.

We use prediction intervals to evaluate our model using Root Mean Squared Forecasting Error (RMSFE) [10]. The validity of this approach relies on the assumption that the residuals of our validation (or test) predictions are normally distributed. To test this assumption, we used a Probability-to-Probability (PP) plot, and tested the normality of our prediction errors using the Anderson-Darling,

Kolmogorov-Smirnov, and D'Agostino K-squared [69] tests. The PP-plot compares the data sample with the plot of a normal distribution. Ideally, when the data follows a normal distribution, the data points align to form a straight line. The three normality tests use p-values to determine how likely it is that a data comes from a population that follows a normal distribution. If the p-values for all tests are greater than a chosen α threshold, there is evidence to suggest that the data comes from a normal distribution. Figure 3.8 shows that all three tests returned a p-value larger than the $\alpha = 0.01$, therefore, indicating that our data points come from a normal distribution.

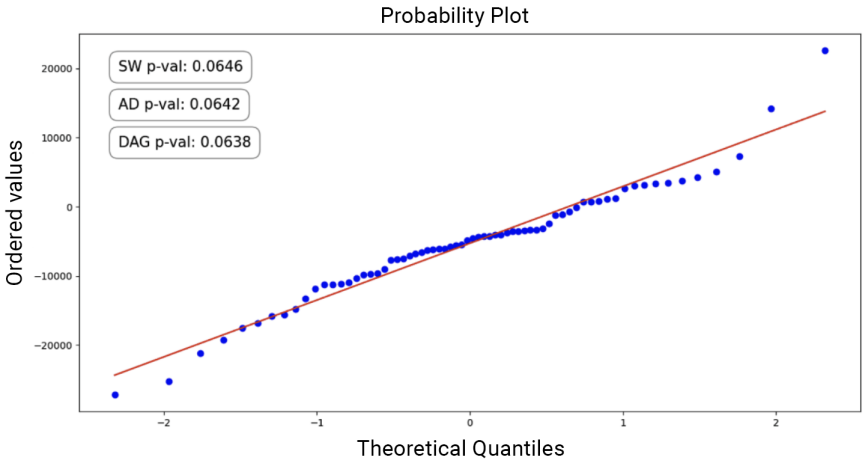

Figure 3.8: Probability-to-Probability (PP) Plot and Normality Tests for prediction errors in ARIMA model.

In a normal distribution, approximately 95% of the data points lie within 1.96 standard deviations of the mean. Hence, to determine the size of our prediction interval, we multiplied 1.96 by the RMSFE for our arrival counts model. The results, as shown in Figure 3.9, indicate that our model captures over 94% of the data points within the 95% prediction interval. The line in the figure represents the mean of predictions. Our model tends to slightly overestimate the arrivals in approximately 90% of the cases. This indicates that while our model can be utilized for an informed planning of the future through forecasting, it also exhibits the ability to accommodate small estimation errors and operate under small variations in

expected load.

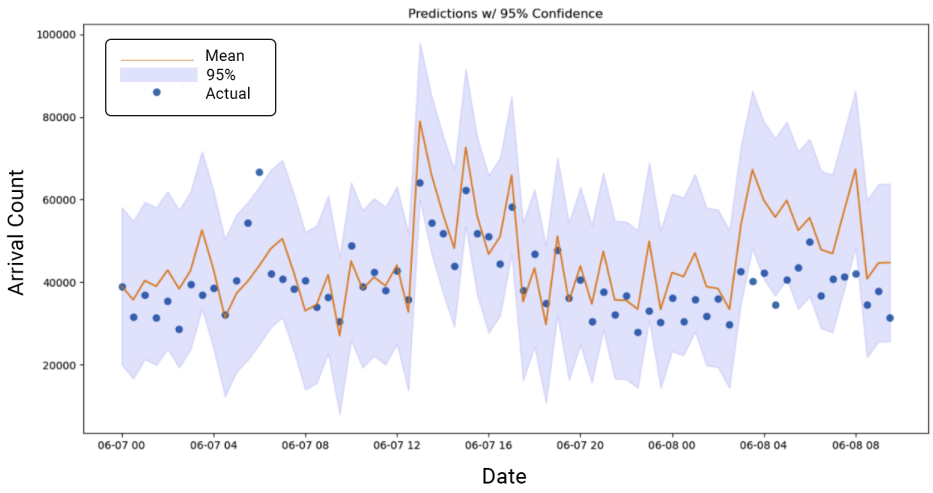

Figure 3.9: Actual and generated (with mean and 95% prediction intervals) arrival counts.

We modeled individual arrivals within each 30-minute period by a Poisson process. Since a randomly distributed set of arrival times will have subsequent times exponentially-distributed, we modeled individual arrivals in any given 30-minute period by sampling its arrival count from a uniform distribution. The actual and generated results for one time period are shown in Figure 3.10.

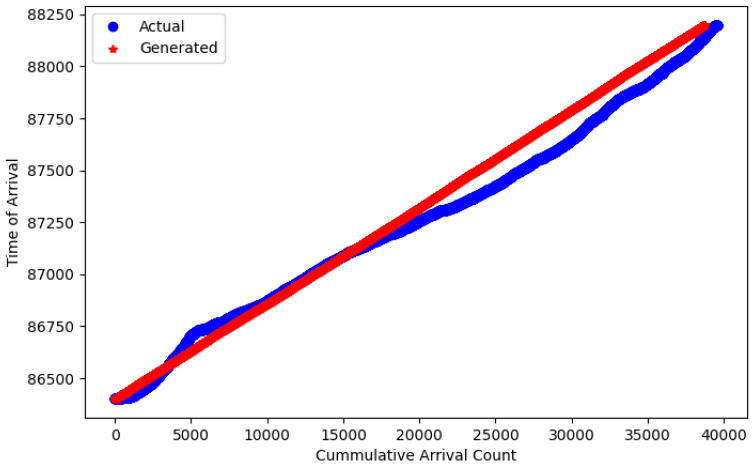

Figure 3.10: Actual and generated individual arrivals over one time period.

3.4.2 LSTM - CPU, Memory, Lifetime

As stated earlier, we model both resource requirements and task lifetime in our dataset using LSTMs. Note that the resource requirements include both CPU and memory resources.

3.4.2.1 CPU

Considering that our dataset contains only 16 unique values for CPU, we opted for a classification approach to model CPU. The dataset was divided into three subsets: training, validation, and testing, with 75% of the data allocated for training and validation, and the remaining 25% for testing. The input data was one-hot encoded before feeding it to LSTM. To train our model, we used time series cross validation with $k = 10$. Our LSTM comprised of a single layer with 32 hidden nodes and used the SGD optimizer with a decaying learning rate of 0.001. The loss was calculated using the 'cross-entropy' function. The cross-entropy for different epochs and time series cross-validation splits for the training and validation sets can be observed in Figure 3.11 and Figure 3.12, respectively. The loss for the test data was 0.795.

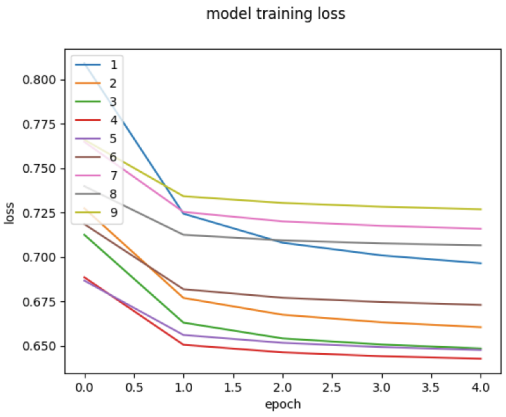

Figure 3.11: Cross entropy training loss for CPU.

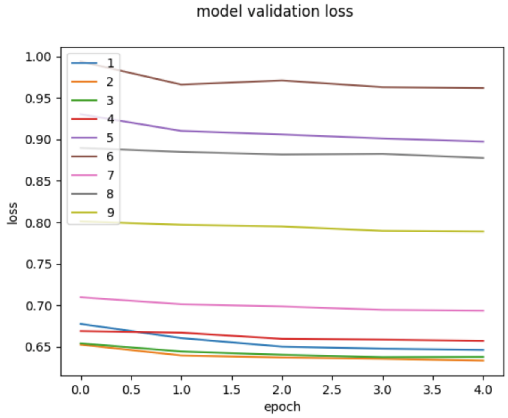

Figure 3.12: Cross entropy validation loss for CPU.

Although loss is a useful metric for assessing the performance of our CPU model, the F1 score provides a more comprehensive evaluation of its accuracy. The F1 score is an ML evaluation metric that combines precision and recall scores to

measure the class-wise performance of a classification problem. It is particularly beneficial when dealing with imbalanced class distributions within the dataset. Figure 3.13 shows the frequency of occurrence for all the CPU classes along with their prediction frequency using our LSTM model. The two classes - 50 and 100 - occur in more than 80% of the dataset and our model is able to predict them with similar frequency. Apart from that, Figure 3.14 shows the F1 scores for all the classes (except for one, i.e., 12 which was neither predicted nor was part of the test data used to generate these results). Here again, the results show that the model works well with the two most frequently occurring classes and the class 400. The remaining classes are taken as outliers.

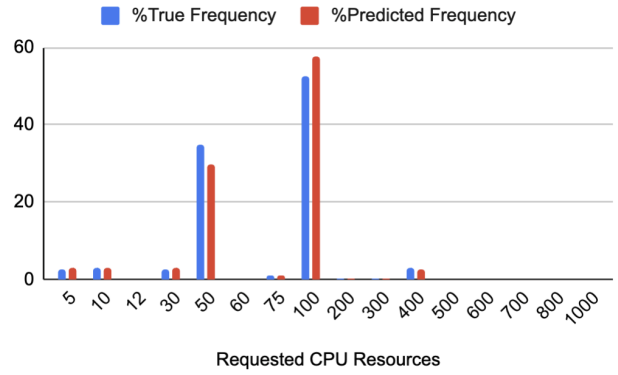

Figure 3.13: True and predicted frequency for CPU classes.

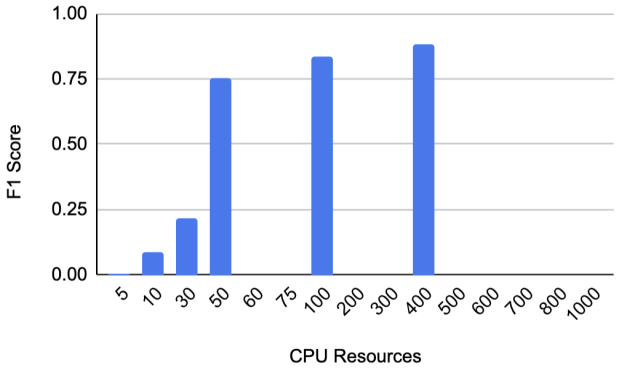

Figure 3.14: F1 score for CPU classes.

Given that our model predicts three classes well, we pool groups of CPU requirements together in order to reduce the number of classes, and also to reduce

class imbalance. To do so, we divide our dataset into three classes, with 50 and 100 remaining intact. The newly created class contains all the other less frequently occurring classes and is named 500. If we train our model with this new dataset, we get the results shown in Figure 3.15 and Figure 3.16. As expected, our F1 scores for the individual classes increased by grouping the less frequently occurring classes together.

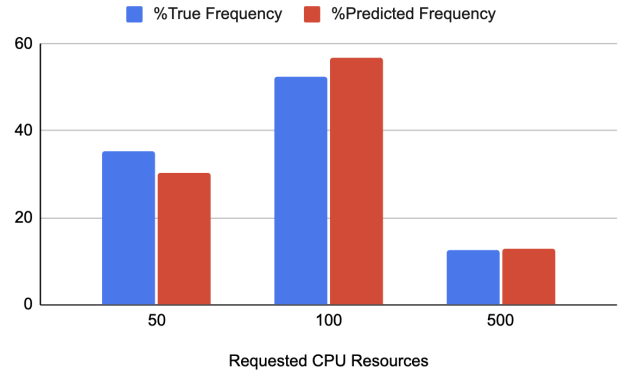

Figure 3.15: True and predicted frequency for grouped CPU classes.

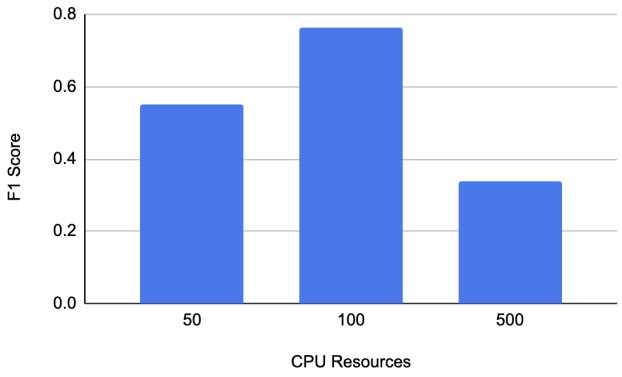

Figure 3.16: F1 score for grouped CPU classes.

Since the values in the new class vary widely, if we make a prediction of CPU resources to be "500", then that value could be as low as 5 and as high as 1000. To make a reasonable prediction within this group, we build an empirical distribution (sample frequency = no. of samples of a particular class/ total number of other classes) over these classes by using the dataset at hand. Every time our model predicts 500, we sample from this distribution. So, on an average, our model does

well.

3.4.2.2 Memory

When modeling memory resources for batch tasks, we considered two options: 1) treating memory as a standalone time series without any additional features, and 2) incorporating CPU as a feature. To assess the impact of CPU resources on improving the effectiveness of our predictive model, we calculated the importance scores for the CPU feature. To do so, we scaled the data using Min-Max between 0 and 1, fitted a linear regression model on the regression dataset and extracted the coefficients assigned to each input variable [6]. These coefficients serve as a basic measure of feature importance. The importance score obtained for CPU was 0.00392. As this value is positive, it suggests that the inclusion of CPU values does not hinder the learning process of our model. Instead, it indicates a minor positive influence of CPU on predicting memory requirements. Consequently, we decided to include CPU as a feature in our LSTM model for memory prediction.

Our memory LSTM comprised of a single layer with 32 hidden nodes and used the SGD optimizer with a decaying learning rate of 0.001. The loss was calculated using the Mean Squared Error (MSE) between the true and the predicted values. The loss for different epochs and time series cross-validation splits for the training and validation sets can be observed in Figure 3.17 and Figure 3.18, respectively.

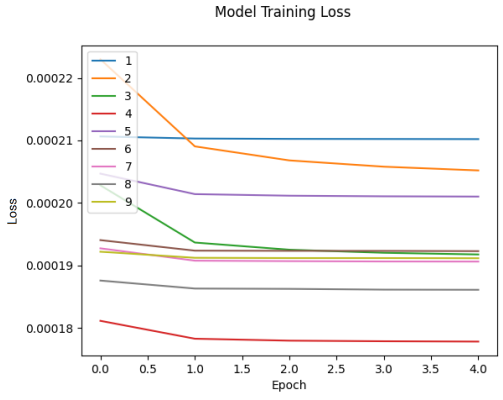

Figure 3.17: MSE training loss for memory.

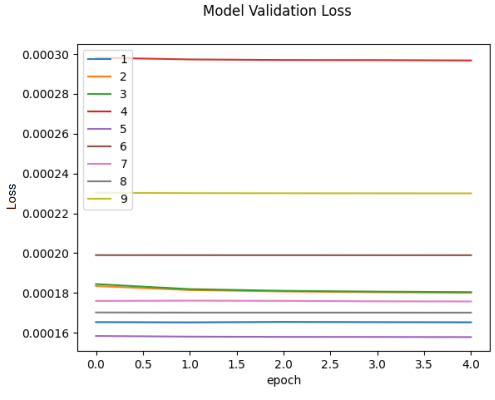

Figure 3.18: MSE validation loss for memory.

The loss for the test data is $1.88e-04$ when the values have been scaled between 0 and 1. Since the original values of memory in the dataset are also normalized and range between 0 and 100, we can simply multiply the loss by 100 to get the loss for the original data. The value is less than 1% in both the cases.

3.4.2.3 Lifetime

Task lifetimes are also predicted using an LSTM model with regression. In order to assess the significance of resource requirements in determining the duration of tasks, we once again calculated the importance scores for CPU and memory features using linear regression. Interestingly, we observed a negative importance score of -0.87 for CPU as a feature, indicating its limited impact on predicting task lifetimes. On the other hand, the memory feature exhibited a considerably higher importance score of 350.26. One possible explanation for this is the significantly lower diversity of unique values in the CPU data compared to memory. Therefore, to model task lifetimes, we included only the memory feature.

The LSTM model used for predicting task lifetimes is similar to the one employed for modeling memory. Both models utilize MSE losses during training. The results for our training and validation losses for the lifetime model are shown in Figure 3.19 and Figure 3.20, respectively. When evaluated on the test data, which accounts for 25% of the dataset, our model achieved a loss of $1.94e-06$. These results demonstrate the high accuracy of our model in predicting task lifetimes.

3.5 USE CASES

In this section, we explore multiple use cases of our model, including capacity planning, task scheduling, and stress testing. We will present empirical results for capacity planning and briefly discuss how our model can optimize task scheduling and facilitate effective stress testing.

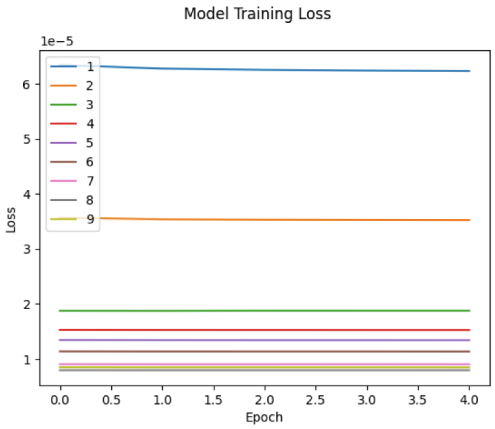

Figure 3.19: MSE training loss for lifetime.

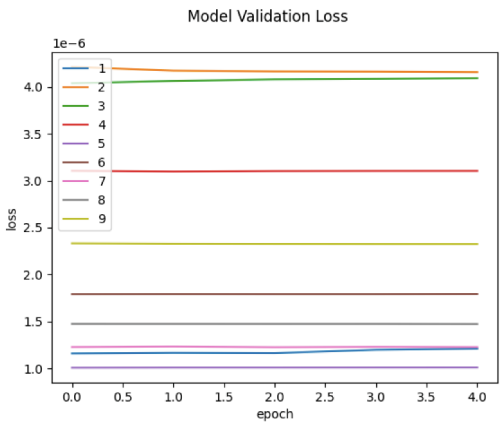

Figure 3.20: MSE validation loss for lifetime.

3.5.1 Capacity Planning

Capacity planning involves predicting future demand to ensure an adequate supply of resources. In traditional data centers, a common approach to capacity planning is to base decisions on current resource usage and load measurements. By providing an accurate forecast of variations in resource requirements, we can effectively plan and allocate the necessary resources to meet future demands. For example, we can accurately generate a load that is double the current load and determine how well existing or proposed resources handle that load.

Some of the major cloud providers [8] heavily rely on accurate forecasts of resource usage to make informed server purchasing decisions and identify any potential bottlenecks or performance issues. We evaluated how well our system can support this use case by considering how well it generates forecasts for the memory size active at each moment of the test window. The same can be done for total number of CPUs.

To evaluate the quality of our forecasts, we calculated the proportion of true data that falls within the 95% prediction interval. The results in Figure 3.21 demonstrate the exceptional accuracy of our model in predicting future memory resources. The prediction interval covers over 95% of the data points, indicating the model's high precision.

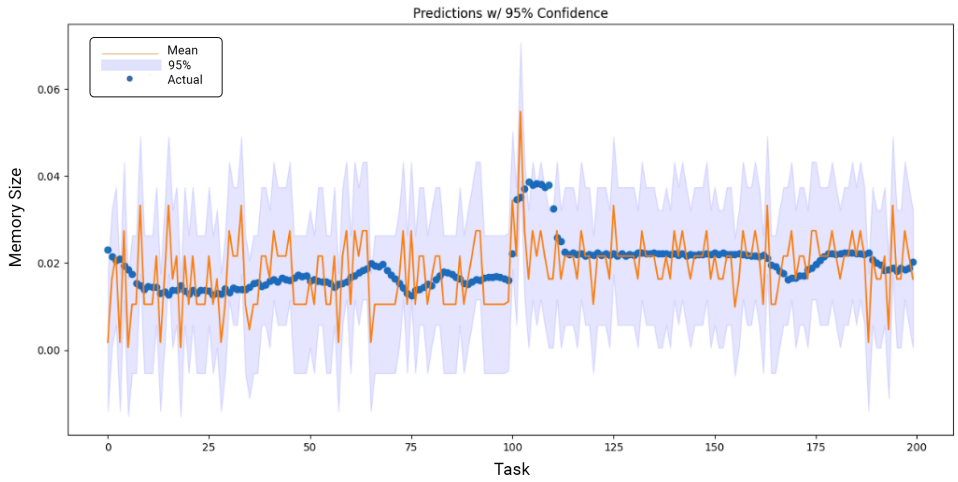

Figure 3.21: Actual and generated (with mean and 95% prediction intervals) memory size.

3.5.2 Scheduler Testing

Scheduling workload entails assigning job requests to specific physical servers. In the case of Alibaba Cloud, various types of imbalances, such as spatial and temporal imbalances, as well as imbalances in resource usage and demands, have been observed [65]. Therefore, it becomes crucial to employ intelligent scheduling techniques to balance the workload distribution and mitigate hot-spots in machine utilization. By doing so, the efficiency of the cluster can be significantly improved.

Our workload models can be used to test schedulers on Alibaba servers by generating realistic estimates of increased workloads that can be used to test scheduler performance, as well as the effect of increasing or decreasing resource availability.

3.5.3 Stress Testing

With the addition of a job generator that adds job parameters, our generative model of Alibaba Cloud workloads can simulate realistic load traces, including job calls, that accurately mimic real-world usage patterns. These synthetic traces allow us to stress test the cloud infrastructure by simulating workloads that exceed normal usage levels. This approach enables us to evaluate whether the architecture can effectively handle unexpected spikes in demand or increased user loads, ensuring

both scalability and robustness under diverse conditions.

3.6 CONCLUSION AND FUTURE WORK

In this chapter, we considered the problem of building a predictive model for co-located tasks in a cloud computing environment. We started by looking at the Alibaba dataset that contains the following data for an eight day period: (a) online and batch task arrivals (co-located) (b) CPU and memory requirements for each task (c) tasks lifetimes. We trained an ML model using this dataset to predict the number of batch tasks that arrive in a 30-minute window, the associated CPU and memory requirements, and their lifetimes. We used Seasonal ARIMA to predict the batch-task arrival counts and three different LSTM networks to predict CPU, memory and lifetime for an arriving task. Our results show that our trained models accurately forecast the number of batch task arrivals in 30-minute windows as well as their associated CPU, memory requirements, and lifetimes.

While our primary focus in this work is on generating realistic workload simulations, we acknowledge that directly measuring the impact of these predictions on system performance is still a challenge. Although we have developed the models to simulate load effectively, the current phase of our work does not involve real-time assessments of how these generated loads influence system performance. Instead, our emphasis lies in understanding the dynamics of workload generation through the ARIMA and LSTM models, setting the stage for future research that could explore the real-time implications of these simulated workloads on system behavior.

Looking ahead, we aim to generalize our prediction model through the use of probabilistic generative models. A probabilistic model, e.g., to predict the CPU resources, implies that we can sample from a distribution over a valid set of CPU values. Hence, our model will predict different sequences of CPU requirements for different runs. Training a task scheduler with this probabilistic framework will significantly enhance its robustness and adaptability.

3.6.1 Limitations of the Dataset

One of the major limitations of our dataset in its architecture diversity. Our models have primarily been trained on specific types of workloads and hardware configurations, which may not fully capture the complexities of modern cloud environments, including those utilizing GPUs or other architectures. As cloud workloads continue to evolve, it is crucial to investigate how ARIMA and LSTM techniques can adapt to these changes and perform under different conditions.

Future work will involve testing our models across a broader range of architectures and workload types to assess their generalizability. By exploring these variations, we hope to refine our approach and better prepare for the diverse demands of contemporary cloud computing environments.

Chapter 4

Optimizing Cloud Resource Utilization with Deep Reinforcement Learning

In large production clusters, multiple workloads are collocated on the same machine to achieve operational efficiency at scale [5, 84]. They share the underlying physical resources of the machine such as CPU, cache, memory, disk, and network-bandwidth. However, the challenge of resources being underutilized or over-utilized may arise as a result of improper scheduling, which in turn leads to inefficient usage of cloud resources [50]. Resource management in a cloud environment is even more challenging because cloud workloads exhibit highly dynamic variations and unexpected bursts in terms of task submission rates. As a result, the task scheduler must be designed to withstand and adapt to these dynamic changes in submitted tasks [100]. To efficiently use cloud resources, the task scheduler must target operational excellence by improving overall cluster utilization and minimizing resource fragmentation [73], while also meeting SLA requirements [81]. Finding optimum initial placement for the workloads is crucial as later migration has a high overhead and causes degraded user-experience [37].

To address the above challenges, a well-designed cloud task-scheduling ap-

proach is essential. Several solutions have been proposed for task scheduling including methods focused on scheduling problems for offline batch tasks, however they are not suitable for highly dynamic workloads [100]. For online task-scheduling methods, existing solutions use hand-crafted heuristics that cannot automatically adapt to the change of the environment and optimize for specific workloads [62, 63]. Reinforcement learning (RL) approaches are particularly well-suited for resource management systems. They can model complex systems and decision-making policies as deep neural networks analogous to the models used for game-playing agents [71]. Moreover, it is possible to train them for objectives that are difficult to optimize directly (because they lack precise models) by using reward signals related to the objective [66]. Finally, by continuing to learn, an RL agent can optimize for a specific workload [66].

In this work, we model the task scheduling problem as a deep reinforcement learning (Deep RL) task. A common practice to improve cluster resource efficiency is to maximally pack jobs on the least number of machines [107]. For example, systems like Google Borg [99] and Tetris [42] use multi-resource bin packing to minimize resource fragmentation and optimize server utilization. In this work, our primary focus is on building a self-learning scheduler that maximizes cloud resource utilization by strategically delaying jobs and consolidating them onto the fewest number of machines possible. We specifically target batch tasks as they are less time-sensitive [54], so that the introduced delays are expected to have minimal impact. Furthermore, since most batch jobs are small and short in duration [43], any delays incurred to optimize packing efficiency are hopefully insignificant.

Deep RL techniques have been successfully applied to multi-resource cluster scheduling problems in the past [32, 66, 73, 109]. While some of this work focused on the policy gradient method REINFORCE [66, 73], others used a value-based algorithm like Deep Q-Network (DQN) [32, 109]. We investigate the use of both Policy Gradient and DQN approaches to enhance batch tasks scheduling. The differences observed between these two methods in terms of achieving optimal results, and computational time can assist in identifying the most appropriate approach for various

scenarios.

4.1 BACKGROUND

In this section, we provide a brief overview of the RL techniques that serve as the foundation for our work. Additionally, we outline the two deep reinforcement learning algorithms, policy gradient and DQN, that we applied to resource scheduling management.

4.1.1 Reinforcement Learning

Consider Figure 4.1 which shows the general structure of RL with the policy represented as a deep neural network. One of the key components of RL is an agent that learns to make decisions by performing actions in an environment to maximize some notion of cumulative reward. During its interaction with the environment and at each time step t, the agent observes a state s_t and selects an action a_t based on that state. After the action is taken, the environment transitions to a new state s_{t+1} and the agent receives a reward r_t. These state transitions and rewards are stochastic and follow the Markov property, indicating that the transition probabilities and rewards depend solely on the current state s_t and the action a_t taken by the agent.

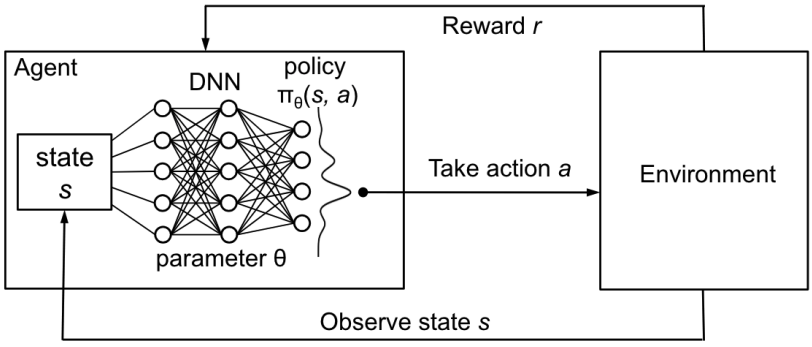

Figure 4.1: Reinforcement Learning with policy represented via DNN [66].

The objective of learning is to maximize the expected cumulative discounted

reward: $\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$, where $\gamma \in (0, 1]$ is a discount factor that determines the importance of future rewards. When γ is close to 1, future rewards are given significant weight emphasizing long-term gains. Conversely, when γ is close to 0, immediate rewards are prioritized and the agent focuses on short-term benefits.

4.1.2 Policy

In RL, a policy is a strategy that determines the behavior of an agent in an environment. It defines the mapping from states to actions, indicating what action the agent must take in each state to achieve its objectives. The policy can be deterministic: $\pi : S \to A$, where S is the set of states and A is the set of actions, or stochastic: $\pi : S \times A \to [0, 1]$, where $\pi(s, a)$ represents the probability of taking action a in state s.

In many RL problems, the state and action spaces are continuous or have a large number of possible states and actions, making it impractical to use tabular representations and leading to the use of function approximators [67]. A function approximator contains a number of adjustable parameters (e.g., weights in a neural network), denoted as θ, that the agent learns through training. The policy is represented as $\pi_\theta(s, a)$.

Deep neural networks (DNNs) [45] have recently emerged as effective function approximators for solving large-scale RL tasks [66, 105]. They can approximate complex functions, allowing them to efficiently represent policies or value functions in continuous or high-dimensional spaces. DNNs also have the ability to generalize from observed data to unseen situations.

RL can be divided into two categories: value-based and policy-based methods [105]. While policy gradient methods and DQN are both popular approaches in RL leveraging deep neural networks, they differ in their fundamental strategies for learning optimal policies.

4.1.3 Policy Gradient

Policy gradient methods optimize the policy directly. The policy is usually modeled with a function $\pi_\theta(s, a)$ parameterized by θ, and the objective is to find the optimal set of parameters that maximize the expected cumulative discounted reward:

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$$

where γ is the discount factor, r_t is the reward at time step t, and π_θ represents the policy parameterized by θ. The goal is to find θ^* such that $J(\theta^*)$ is maximized.

The REINFORCE algorithm [92] is a simple policy gradient method that updates the policy parameters based on the gradient of the expected cumulative reward. The update rule is given by:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) R_t$$

where α is the learning rate, $\nabla_\theta \log \pi_\theta(s, a)$ is the gradient of the log probability of taking action a in state s with respect to the policy parameters θ, and R_t is the return at time step t.

To estimate the gradient, the REINFORCE algorithm uses Monte Carlo sampling [47] to generate trajectories from the environment. After each trajectory, the algorithm computes the return R_t and updates policy parameters accordingly. By iteratively sampling trajectories and updating policy parameters, the REINFORCE algorithm gradually learns a policy that maximizes the expected cumulative reward.

4.1.4 Deep Q-Network (DQN)

Value-based methods in reinforcement learning aim to estimate the value function, which represents the expected cumulative rewards of following a particular policy in a given state or state-action pair. The value function can be defined as:

$$Q^\pi(s, a) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a\right]$$

where $Q^{\pi}(s, a)$ is the value of taking action a in state s under policy π, γ is the discount factor, and r_t is the reward received at time step t. The goal is to find the optimal value function $Q^*(s, a)$ that maximizes the expected cumulative reward.

The Deep Q-Network (DQN) [71] algorithm is a value-based method that uses a deep neural network to approximate the optimal value function $Q^*(s, a)$. The network takes a state s as input and outputs the estimated value of each action a in that state. The algorithm objective is to minimize the difference between the predicted values and the target values, which are updated iteratively using a Bellman equation-based update rule:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') \right)$$

where $Q(s, a)$ is the estimated value of action a in state s, α is the learning rate, r is the reward received after taking action a in state s, s' is the next state, a' is the next action, and γ is the discount factor.

The DQN algorithm uses experience replay and may use a separate target network to stabilize training and improve convergence [46]. It samples experiences from a replay buffer to train the network, and periodically updates a target network with the parameters of the main network to provide stable target values for training.

4.2 PROBLEM FORMULATION

In this section, we describe the architecture for our online multi-resource cluster scheduling formulated as a deep reinforcement learning task. The workload scheduler is represented as an agent and the scheduling policy is encoded in a neural network. For an incoming job, the trained network policy takes actions: it determines on what machine the service must be placed to optimize various metrics of operational excellence, including maximizing resource utilization and, minimizing resource fragmentation and number of machines used [73].

The environment consists of a set of N machines where the incoming jobs

are to be scheduled. Each machine has C_r amount of physical capacity for different resource types. For our model, we consider only two resource types: cpu and memory. For an incoming job j, the agent observes the resource usage of the machines along different resources and based on that schedules j on a particular machine. Along with observing the current placement map of different services running on machines, the agent also keeps track of the number of jobs waiting in the queue to be scheduled.

The jobs arrive in the cluster in discrete steps. For each job, we assume that its maximum resource usage and duration is known upon arrival. Moreover, we assume that the jobs cannot be preempted and must run to completion once they are scheduled.

4.2.1 State Space Representation

The state of the system is represented using distinct images for different resources of each machine and the backlog queue. A similar representation is used by [66] for their environment with one machine, first M jobs waiting to be scheduled and a backlog queue. The different colors in the images show the allocation of resources to different jobs. The physical units of a resource available on a machine increase along x-axis while the timesteps increase along the y-axis. At any point, the scheduler can view the current schedule of the jobs looking at T timesteps ahead into the future and based on that schedule the incoming job. An example input state representation is shown in Figure 4.2 with three machines and one backlog queue. The blue job is scheduled to begin at $t = 1$ and run until $t = 3$. It uses one unit of CPU and two units of memory.

4.2.2 Actions

At any point in time, the agent looks at the first job in the queue (if any) and assigns it to one of the machines. Therefore, the action space consists of the N machines on which jobs can be scheduled and is given by $a = \{0, 1, ..., N - 1\}$. The job is allocated to a machine only if its resources requirements are fully met starting at any $t < T$ and it can run upto completion by at most T timesteps represented in

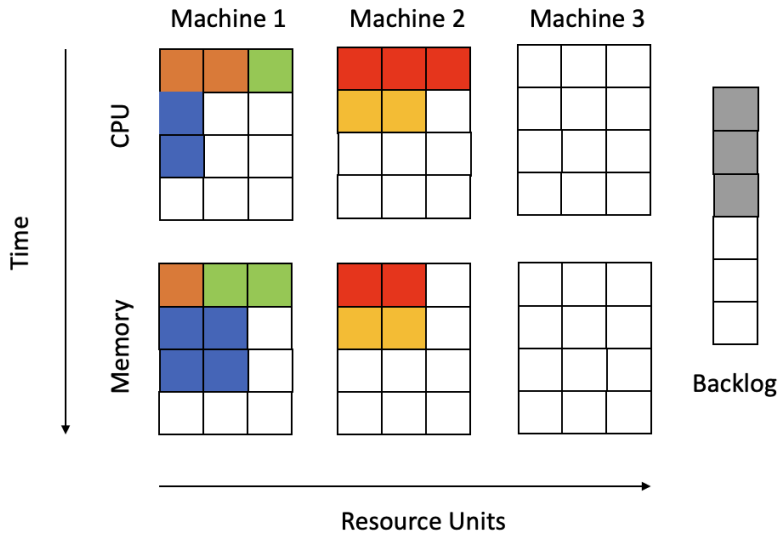

Figure 4.2: State space representation with three machines.

the state.

In order to schedule multiple jobs in one timestep, the agent keeps on scheduling jobs without proceeding in time until no more jobs are left in the queue, or it picks a machine which does not have enough resources to schedule the job [66]. When the time proceeds, the cluster images shift up by one timestep and the newly arriving jobs are added to the queue.

4.2.3 Rewards

We use negative rewards or penalty to train our RL agent. These, along with several other rewards, have been used in [73] to schedule time-varying workloads using a deep reinforcement learning based approach.

Under-Utilization Penalty: To increase the overall utilization of the cluster by enabling the scheduler to achieve tighter packing and utilize fewer machines whenever possible, we use a penalty that is proportional to the total unused resources in the active machines. An active machine is defined as one that is running at least one workload. In our state representation, empty pixels indicate the number of unused

resources at any given time.

$$P_U = -\sum_{r} \sum_{m \in M_a} |U_m(t, r)| * K_u \tag{4.1}$$

where $U_m(t, d)$ represents the unused resources for machine m at time t across resource r. The constant K_u, serves as a weighting factor and M_a denotes the set of active machines.

Wait-Time Penalty: To discourage the scheduler from delaying scheduling requests excessively while seeking a more optimal placement, we introduce a penalty proportional to the number of pending requests in the queue ($|Q_t|$). This penalty is calculated by multiplying the number of waiting requests by a constant factor (K_w).

$$P_W = -|Q_t| * K_w \tag{4.2}$$

Adjusting the weight K_w in the penalty function helps the scheduler avoid queue overflows leading to job drops.

4.2.4 Metrics for Operational Excellence

We use the following metrics [37, 73] to evaluate improvements in the cluster's operational efficiency. Let T represent the duration of the observation period until all jobs are completed.

Resource Utilization: This metric measures the average utilization of the cluster for each resource type:

$$\text{Avg Util}(r) = \frac{\sum_{t=0}^{T} \sum_{m} R(m, t, r)}{T \times \max_t A(t) \times C_r \times L} \tag{4.3}$$

Here, the maximum number of machines used at any point serves as a normalizer in the denominator. $R(m, t, r)$ denotes the resource usage of machine m at time t across resource r. L denotes the number of future timesteps considered for

each machine. Higher utilization is considered better.

Resource Fragmentation: This metric quantifies the concentration of unused resources within the cluster:

$$\text{Avg Frag}(r) = 1 - \frac{\sum_{t=0}^{T} \max_{m \in A(t)} U_m(t, r)}{T \times \sum_{m \in A(t)} U_m(t, r)} \tag{4.4}$$

Lower fragmentation indicates a higher ability of the cluster to accommodate unexpected large jobs. $U_m(t, d)$ represents the unused resources of machine m at time t across resource r.

Max Machines Used: This metric reflects the count of machines that had at least one job scheduled on them at any point in time.

Job Delay: This metric calculates the average delay experienced by jobs, measured in time units. The delay is defined as the time difference between a job's submission time and the time when it starts running.

4.3 TRAINING ALGORITHMS

Our training algorithms using the two Deep RL approaches: REINFORCE and DDQN are described below.

4.3.1 REINFORCE

Our training algorithm using REINFORCE [91] is outlined below. The neural network represents the policy and is known as the policy network. It takes a set of images as input as described in 4.2.1 and generates a probability distribution over all possible actions. The agent then samples an action from this distribution. We train the policy network in an episodic manner, as REINFORCE relies on collecting the complete trajectory of states, actions, and rewards for an entire episode before

updating the policy. Within each episode, a fixed number of jobs arrive and are scheduled. The episode concludes when all jobs have completed execution or the maximum episode length is reached. To ensure the policy generalizes effectively, we incorporate multiple instances of job arrival sequences during training. During each training iteration, we simulate N episodes. The resulting data is then leveraged to enhance the policy. Specifically, we record state, action, and reward information for all time steps within each episode. These values are then used to compute the (discounted) cumulative reward, v_t, at each time step t of each episode.

Algorithm 1: Training Algorithm using REINFORCE

Input: Policy parameterization π_θ, learning rate α, discount factor γ
Initialize policy parameters θ
for epoch $= 1$ **to** M **do**
  $\Delta\theta \leftarrow 0$
  for episode $= 1$ **to** N **do**
    Sample a trajectory $\tau = \{s_1, a_1, r_1, \ldots, s_L, a_L, r_L\}$ using policy π_θ
  end for
  For each trajectory, compute the return v_t for each time step t: $v_t^\tau = \sum_{s=t}^{L} \gamma^{s-t} r_s^\tau$
  for $t = 1$ **to** L **do**
    Compute the baseline: $b_t = \frac{1}{N} \sum_{i=1}^{N} v_t^i$
    for episode $= 1$ **to** N **do**
      Compute the policy gradient: $\Delta\theta \leftarrow \Delta\theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t)(v_t - b_t)$
    end for
  end for
  Update the policy parameters: $\theta \leftarrow \theta + \Delta\theta$
end for

In REINFORCE, calculating the policy gradient often leads to significant variance, which results in slow unstable learning. To address this, a common strategy is to subtract a baseline value from the returns v_t. We determined the baseline by computing the average of the return values v_t across all episodes within an epoch, at the same time step t. A similar approach has been used in [66] for multiple jobsets. Our results with and without baseline for the first 2500 epochs are shown in 4.3.

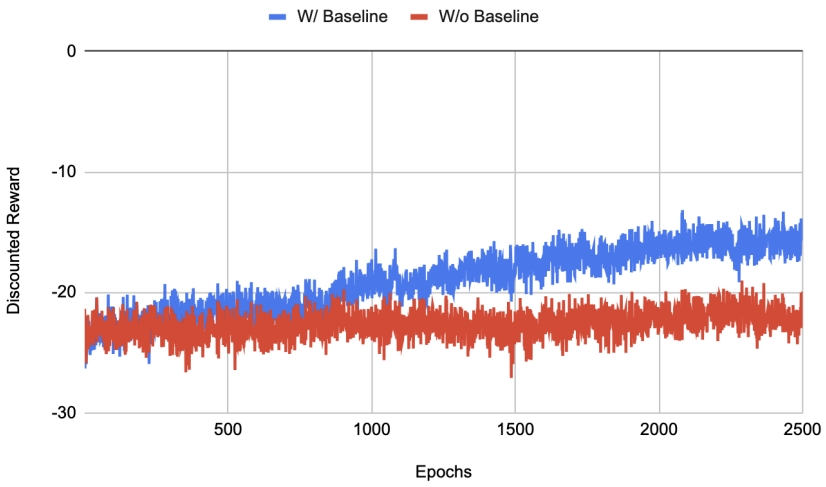

Figure 4.3: Learning curve with REINFORCE training algorithm at 50% load with and without baseline.

4.3.2 DDQN

We train the neural network using Double Deep Q-Network (DDQN), a variant of DQN that mitigates the overestimation bias observed in DQN, learns more stable Q-values and converges faster [46]. In DQN, a single neural network is used to select actions and estimate Q-values, however DDQN uses two separate neural networks: a policy network and a target network. The policy network takes a set of images as input, as described in Section 4.2.1, and outputs Q-values for all possible actions. The agent then selects actions based on these Q-values using an ϵ-greedy strategy, balancing exploration and exploitation. We also use an experience replay that stores the transition, consisting of the previous state, action, reward, and next state, in a replay buffer. The policy network is trained periodically using samples from the replay buffer. Instead of directly using the Q-values predicted by the policy network, the target network is used to estimate the value of the next state. The parameters of the target network are updated in regular intervals to match those of the policy network, ensuring a more stable estimation of the Q-values. The training algorithm using DDQN is shown below.

Once more, we train the policy network episodically. While it is possible to train DDQN per timestep, we found that adopting a finer granularity for our partic-

Algorithm 2: Training Algorithm using DDQN

Input: Replay buffer B, Q-networks Q and Q', target update frequency τ, learning rate α, discount factor γ
Initialize Q with random weights
Initialize target network weights $Q' \leftarrow Q$
Initialize empty replay buffer B
for epoch $= 1$ to M **do**
  for episode $= 1$ to N **do**
    Initialize state s
    for step $= 1$ to L **do**
      Select action a using an ϵ-greedy policy based on Q
      Execute action a, observe reward r and next state s'
      Store transition (s, a, r, s') in B
      $s \leftarrow s'$
    end for
    Sample mini-batch of transitions (s_j, a_j, r_j, s'_j) from B
    Calculate target $y_j = r_j + \gamma Q'(s'_j, \arg\max_a Q(s'_j, a; \theta); \theta^-)$
    Update Q by minimizing the loss: $\mathcal{L} = \frac{1}{N} \sum_j (Q(s_j, a_j; \theta) - y_j)^2$
  end for
  Every τ episodes, update target network: $Q' \leftarrow Q$
end for

ular scenario led to a substantially larger computation time without any significant performance gains.

4.4 EVALUATION

We perform a preliminary evaluation of the two Deep RL algorithms: REINFORCE and DDQN to answer the following questions.

- When scheduling batch services that use multiple resources, how do REINFORCE and DDQN compare with each other and other state-of-the-art mechanisms like Packer and Best-Fit?
- How long do the algorithms take to train?
- Which algorithm shows better performance and what factors contribute to this outcome?

4.4.1 Workload

We model the incoming jobs to the cluster as a Poisson process with three different average cluster load scenarios: 30%, 50%, and 70%. Each incoming service is characterized by two resources: CPU and memory. We use the batch tasks production workload from Alibaba traces [5], with approximately equal number of tasks categorized as CPU-intensive and memory-intensive. For each trace, our training and test sets consist of 100 and 30 distinct job sequences, respectively [73]. The resource utilization and duration values of jobs obtained from the traces are mapped to our state-space dimensions. The job durations are bounded by the time horizon shown in the state space representation as in [66]. This limitation is primarily to compute baseline used in the REINFORCE algorithm 1.

4.4.2 Methodology

Our evaluation runs on a cluster with 10 machines. The neural networks are implemented using *Keras*, and comprise of two hidden layers of 128 neurons each, followed by an output layer with neurons equal to the number of machines in the cluster. The hidden layers use the ReLU activation function. For the output layer, REINFORCE employs softmax activation [40], while DDQN uses linear activation [71]. Both algorithms use the Adam optimizer with a learning rate (η) of 0.001 and a discount factor (γ) of 0.95. In each episode, a fixed number of jobs arrive based on the cluster load and are scheduled by the agent. The state space parameters are $M = 10$, $T = 20$, $r = 2$, $C1 = C2 = 8$. The penalty parameters are set to $K_u = 0.003125$ and $K_w = 0.05$ for REINFORCE, while for DDQN, $K_w = 0.06$ yielded the highest performance gains. Training and testing are conducted on Nvidia A100 GPUs.

When training using the REINFORCE algorithm 1, the number of trajectories (N) is set to 20 for a total of 3000 iterations, with maximum episode length $L = 200$. With DDQN algorithm 2, training comprises 12000 episodes, with the target network updated every 10 episodes. The replay memory size is set to $100,000$ and the batch size is 64. The loss used is Mean Squared Error (MSE).

4.4.3 Baselines

We compare the performance of REINFORCE and DDQN with baseline heuristics including Best-Fit [9] and Packer [42]. The Best Fit heuristic operates by selecting jobs in a first-in-first-out (FIFO) manner and assigning them to the machine with the least units of the dominant resource of the job available at that time. The packing heuristic projects both job requirements (j_r) and machine resources (m_r) into a Euclidean space, and selects the task-machine pair with the highest dot product value. The dot product favors larger tasks and those that utilize resources in proportions similar to what is available on the machine. Both of these agents are greedy, meaning that they aim to utilize all available resources by allocating as many jobs as possible within each time step.

4.4.4 Results

The REINFORCE algorithm outperforms baseline algorithms by strategically delaying jobs to achieve a more efficient utilization of resources. These delays have minimal impact on batch jobs, which are typically short, and less time-critical [43]. Additionally, the delays can be managed by adjusting queue sizes and incorporating such variations while training.

REINFORCE provides a $125 - 200\%$ increase in average CPU and memory utilization compared to Best-Fit and Packer over different cluster-load conditions, as shown in Figures 4.4 and 4.5. This is primarily due to efficient packing that requires significantly fewer machines as shown in Figure 4.6. The benefit is more apparent under low-load conditions as REINFORCE avoids excessive delays that might lead to incoming jobs being dropped due to queue overflow. As a result, under high loads, its resource utilization is nearly comparable to that of Packer and Best-Fit, and in some cases, it is marginally inferior. Both Best-Fit and Packer greedily allocate as many jobs as can be accommodated with the available resources. Best-Fit slightly outperforms Packer as it just packs jobs onto the machines while Packer also takes into account the alignment between job demands and resource availability

[66]. REINFORCE provides $5-30\%$ reduction in resource fragmentation compared to Best-Fit and Packer as shown in Figure 4.7 and 4.8, however, it suffers from large delays as shown in Figure 4.9.

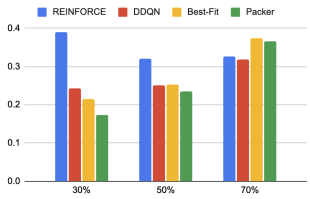

Figure 4.4: CPU utilization

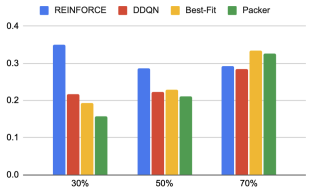

Figure 4.5: Memory utilization

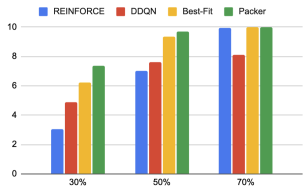

Figure 4.6: Machines used.

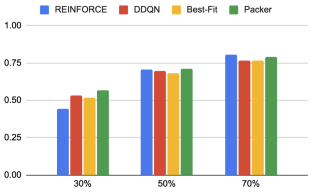

Figure 4.7: CPU fragmentation.

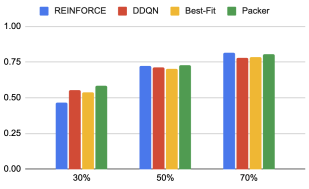

Figure 4.8: Memory fragmentation.

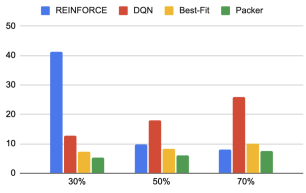

Figure 4.9: Average job delay.

DDQN achieves job packing on fewer machines compared to Best-Fit and Packer as illustrated in Figure 4.6, and it also incurs less delays than REINFORCE. However, it shows similar resource utilization and fragmentation outcomes as Best-Fit and Packer by frequently delaying a substantial number of jobs, which reduces cluster utilization. At a load of 70%, DDQN begins dropping jobs, which is undesirable and results in lower resource utilization and machine usage compared to REINFORCE, Packer, and Best-Fit. The sub-optimal performance of DDQN could stem from several factors:

1. DDQN operates in a partially observable environment where it lacks complete information about the state, e.g, the resource requirements of incoming jobs. This setup results in rewards dependent on information not available in the state representation, also known as a Partially Observed Markov Decision Process (POMDP) [72]. In contrast, REINFORCE has shown efficacy in similar scenarios [66].

2. Given our environment where delaying jobs strategically can lead to greater benefits, DDQN faces a challenge due to its reliance on a static Q-value function. This setup complicates its ability to execute long-term plans effectively in environments with partial observability [34, 76].
3. DDQN uses ϵ-greedy exploration strategy to discover new states and actions, but in environments where the agent cannot observe the entire state space, it may struggle to explore effectively. This can lead to the agent getting stuck in local optima or failing to find optimal policies [34, 76].

In terms of computation time, DDQN takes approximately $31 - 35$ hours to train depending upon the cluster load, whereas REINFORCE takes only $10 - 13$ hours. DDQN also requires more memory than REINFORCE as it stores experiences in a replay buffer and it neural network architecture includes two separate networks (online and target networks), which also contribute to its increased space complexity. REINFORCE only uses one policy network and updates the policy directly without using experience replay. DQN, however, is more sample efficient and only requires 12000 episodes for training compared to 60000 episodes required by REINFORCE.

4.4.4.1 Scalibility of REINFORCE

When applying the REINFORCE algorithm for scheduling decisions in a large cluster of machines, several performance characteristics must be considered:

1. **Sample Efficiency:** REINFORCE relies on collecting many samples to estimate the policy gradient accurately. In a scheduling context, where each decision affects resource allocation across potentially thousands of machines, this demand for samples can be significant. To address this challenge, extensive training on simulated or historical data may be necessary, resulting in increased computational overhead.
2. **Variance in Gradient Estimates:** The Monte Carlo approach used in REINFORCE introduces high variance in gradient estimates, which can lead to unstable training and oscillating performance metrics. This instability makes

it challenging to deploy the learned scheduler in real-time systems, where consistent performance is critical. To mitigate this, subtracting a baseline (such as the value function or a simple average reward) from the returns can help reduce the variance of the gradient estimates without introducing bias, thereby enhancing stability [91].

3. **Action Space Complexity:** In large clusters, the action space (the set of possible machines) can be enormous, complicating the learning process. A larger action space may slow down convergence and make it harder for the algorithm to learn effectively.

In Alibaba cloud's architecture, the scheduler is typically a separate component from the worker nodes. This separation allows the scheduler to focus solely on decision-making without being hindered by job execution or data processing tasks. Such a design facilitates more efficient resource management and improved scalability. The scheduler can train large neural network for REINFORCE and process vast amounts of scheduling data by utilizing more resources, such as GPUs, that can significantly speed up training times and enhance responsiveness in real-time scheduling scenarios.

4.4.4.2 Limitations of REINFORCE

Despite its potential, REINFORCE has several limitations compared to traditional scheduling methods:

1. **Sample Efficiency:** REINFORCE is typically less sample-efficient than traditional methods, such as heuristic-based approaches, meaning it requires more interactions with the environment to learn effectively. In scenarios where data collection is costly (e.g., real-time job scheduling), traditional methods may be preferable. To mitigate this limitation, techniques like experience replay or combining REINFORCE with other reinforcement learning algorithms (e.g., Advantage Actor-Critic) can help improve sample efficiency.

2. **Real-time Decision Making:** Scheduling often requires real-time decisions. The inherently high latency of REINFORCE due to the need for multiple rollouts can be detrimental. Implementing a model-based approach or leveraging a hybrid method that incorporates heuristics for quick decisions while using REINFORCE for periodic updates can balance efficiency with performance.
3. **Environment Dynamics:** REINFORCE assumes a relatively stable environment during training. In a dynamic setting like a cloud computing cluster where job arrivals and departures can be unpredictable, this assumption may not hold. Regularly retraining the model or using online learning techniques can help adapt to changes in the environment. Additionally, incorporating a form of transfer learning can allow the model to adapt quickly to new workloads.

4.5 RELATED WORK

There has been past research in multi-dimensional resource packing. Parkes et al. [79], Joe-Wong et al. [56], Ghodsi et al. [38], and Grandl et al. [42] have explored techniques for efficiently packing jobs that require multiple resources onto available machines. Similarly, Delimitrou and Kozyrakis [31] and Gog et al. [39] have contributed to the development of placement strategies aiming to optimize resource allocation for improved system efficiency and responsiveness. However, these solutions rely on hand-crafted heuristics that cannot automatically adapt to the change of the environment or optimize for specific workloads [63]. Our approach utilizes Deep RL methodologies that can learn to optimize scheduling for dynamic environments. DeepRM [66] uses Deep RL to schedule jobs, but its state space uses a single monolithic machine and its final objective is to reduce job slowdown or completion time. In contrast, we focus on scheduling batch jobs on multiple machines with the final objective of efficiently packing them across multiple machines. Shanka et al. [73] applied Deep RL to improve packing efficiency for jobs with varying resource usage, including both online and batch services. Unlike online services, which

are time-sensitive and require careful consideration to avoid compromising response times, our approach optimizes packing efficiency for batch services with constant resource requirements. Alibaba uses separate schedulers to schedule online and batch services, further emphasizing the distinct focus of our investigation. Moreover, our primary goal is to compare the performance of policy-based and value-based deep RL approaches in scheduling batch tasks. Both the above papers only use policy-based Deep RL methodology.

Value-based methods such as Deep Q-Network (DQN) have successfully addressed task-scheduling challenges in prior research [23, 33, 63, 93]. However, these approaches typically target reducing task response times or computational costs, whereas our emphasis lies in optimizing resource utilization by strategically delaying batch services, which are less time-sensitive. To our knowledge, no existing work has explored the application of DQN or its variants specifically for this purpose.

4.6 CONCLUSION

In conclusion, our research models the task scheduling problem as a Deep RL task, specifically targeting the optimization of cloud resource utilization in shared clusters. By strategically delaying jobs and consolidating them onto fewer machines, our approach focuses on improving the efficiency of batch task scheduling, which are generally less time-sensitive and shorter in duration. We use both Policy Gradient and DQN approaches to improve batch task scheduling. Our findings indicate that REINFORCE significantly outperforms baseline algorithms such as Best-Fit and Packer. Conversely, while the DDQN approach achieves job packing on fewer machines, its performance in terms of resource utilization and fragmentation is comparable to Best-Fit and Packer. DDQN's frequent delays under higher load conditions result in undesirable job drops, leading to lower overall resource utilization. The sub-optimal performance of DDQN can be attributed to its operation in a partially observable environment, challenges in executing long-term plans, and exploration inefficiencies. These results establish a strong foundation for further exploration and refinement of

deep RL techniques in dynamic and complex scheduling environments.

Our work has certain limitations. Notably, we use a bounded time horizon in our approach, whereas the underlying optimization problem ideally considers an infinite time horizon. The bounded horizon is necessary for computing the baseline, as discussed in 4.3.1. Additionally, our job model is based on constant resource requirements, which may not accurately reflect the variability in real-world workloads. Furthermore, the resource profile of a job may not be known in advance, and the scheduler might only obtain an accurate view as the job progresses. We aim to address these limitations in future by using a value network [91] to compute baseline and exploring alternative job models to fit more realistic situations. Furthermore, we plan to evaluate DDQN in an environment with complete information to better understand its performance under these conditions.

Chapter 5

Conclusion

Our work highlights the potential of machine learning and deep reinforcement learning techniques in enhancing the performance and efficiency of distributed systems and cloud environments.

Chapter 2 of our thesis discusses the significance of having a large dataset for Linux configuration tuning. Our research aimed to optimize system configuration for improved network performance in a distributed setting, starting with a large dataset of network benchmark runs provided by Red Hat. We selected a subset of hardware and Linux parameters based on expert feedback and performance tuning guides from Red Hat. By using tree-based feature selection methods, we identified parameters that significantly impact network performance. Our findings revealed that all significant parameters were part of the hardware configuration, and none were related to Linux configuration except the kernel version. Further investigation showed that our dataset lacked diversity in Linux configurations, highlighting that even large datasets should not be assumed to be diverse. We recommend preliminary tests for anyone working with similar complex datasets and planning to use machine learning.

To further strengthen our research, we acknowledge a key limitation: our analysis was confined to pbench data provided by Red Hat. While this dataset offered valuable insights, there may be other, potentially more suitable datasets for Linux configuration tuning that are not readily accessible due to privacy concerns.

If a sufficiently diverse dataset becomes available, tuning based on that data could be explored in future work. Additionally, our current feature selection relied on correlation-based methods, which may overlook important non-linear relationships. Addressing this gap could yield further insights and enhance our understanding of how different Linux configurations impact performance.

Future efforts could focus on implementing advanced feature selection techniques and expanding our dataset to include a broader range of configurations. By adopting principles from experimental design theory, we could move away from exhaustive testing toward more structured testing that identifies essential parameters with a minimal number of new experiments. This approach would not only streamline our research process but also enhance the performance and reliability of machine learning models for configuration tuning. By systematically determining which configurations are most impactful, we could ensure that our experiments are both efficient and informative.

Chapter 3 of our work addresses the challenge of accurately modeling batch tasks using recurrent neural networks in co-located Alibaba workloads. The Alibaba dataset spanning eight days includes the following information: (a) arrival times for online and batch tasks that are co-located, (b) CPU and memory demands for each task, and (c) task lifetimes. We developed a predictive model by training a machine learning model on this dataset to forecast several aspects within a 30-minute timeframe: the count of batch task arrivals, along with their CPU and memory requirements and lifetimes. To achieve this, we used Seasonal ARIMA for predicting batch task arrivals and utilized three distinct LSTM networks to forecast CPU requirements, memory needs, and task lifetimes for each arriving task. Our findings show that our models effectively predict the number of batch task arrivals within 30-minute intervals, as well as accurately anticipate their associated CPU and memory demands and lifetimes.

Looking ahead, we could refine our predictive models to accommodate more complex and varied workloads while integrating them with real-time cloud management systems for dynamic optimization. This could enhance the generalizability of

our models through the application of probabilistic generative frameworks. This approach would enable us to generate samples from a distribution covering valid sets of values, allowing our model to predict diverse sequences of tasks. By training task schedulers with our probabilistic framework, we could significantly enhance their robustness and adaptability in real-world scenarios. Additionally, we could directly assess how our predicted workloads impact system behavior, thereby establishing a clearer connection between simulation and actual performance.

A significant limitation in our dataset is its lack of architectural diversity. Our models have primarily been trained for specific workloads and hardware configurations, which may not fully reflect the complexities of modern cloud environments, particularly those utilizing GPUs and other advanced architectures. To address this, future work could include testing our models across a broader array of architectures and workload types. By exploring these variations, we could refine our approach and ensure that our models are better equipped to meet the evolving demands of contemporary cloud computing environments.

Chapter 4 focuses on optimizing cloud resource utilization with deep reinforcement learning (DRL). Our approach strategically delays and consolidates jobs onto fewer machines, prioritizing scheduling of batch tasks, which are typically less time-sensitive and shorter in duration. We use both Policy Gradient and Deep Q-Network (DQN) approaches to improve batch task scheduling. Our findings demonstrate that the REINFORCE algorithm significantly outperforms traditional algorithms like Best-Fit and Packer. However, our DDQN approach, while effective in consolidating jobs onto fewer machines, shows comparable performance in resource utilization and fragmentation to Best-Fit and Packer. Under higher loads, DDQN's tendency to delay jobs leads to job drops and reduced overall resource efficiency. This sub-optimal performance can be attributed to its challenges in handling partially observable environments.

Despite our advancements, our research has some limitations. We employ a bounded time horizon in our approach, while the underlying optimization problem ideally considers an infinite time horizon. This bounded horizon is necessary for

computing the baseline, as discussed in 4.3.1. Additionally, our job model relies on constant resource requirements, which may not accurately capture the variability of real-world workloads. Also, in realistic settings, the resource profile of a job might remain unclear initially, with the scheduler gaining a more accurate view only as the job progresses. To address these limitations, our future work would use a value network [91] to compute the baseline and explore alternative job models that better align with realistic scenarios. We could also evaluate DDQN in environments with complete information to gain a deeper understanding of its performance under these conditions, and explore hybrid approaches that combine the strengths of multiple DRL techniques.

These three facets – system and network performance, workload generation and performance prediction, and workload scheduling – are all contributions to the general problem of understanding and optimizing cloud performance. In this research, we identified several connections that need further exploration. For example, integrating a workload generator with a workload scheduler could enhance our ability to predict resource requirements and inform the design of future architectures, ultimately optimizing throughput in innovative ways. The challenges we addressed are ongoing, and there is still much work to be done in this field. The insights gained from our research pave the way for future investigations that could lead to more efficient and adaptable cloud environments.

Bibliography

[1] M. Acher, H. Martin, J. Alves Pereira, A. Blouin, D. Eddine Khelladi, and J.-M. Jézéquel. Learning From Thousands of Build Failures of Linux Kernel Configurations. Technical report, Inria ; IRISA, June 2019.

[2] M. Acher, H. Martin, J. A. Pereira, A. Blouin, J.-M. Jézéquel, D. E. Khelladi, L. Lesoil, and O. Barais. Learning Very Large Configuration Spaces: What Matters for Linux Kernel Sizes. Research report, Inria Rennes - Bretagne Atlantique, Oct. 2019.

[3] R. Agarwal. The 5 feature selection algorithms every data scientist should know. https://towardsdatascience.com/the-5-/feature-selection-algorithms-every-data-scientist-need-to/-know-3a6b566efd2. [Accessed 12-11-2021].

[4] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 74–89, New York, NY, USA, 2003. Association for Computing Machinery.

[5] Alibaba. Alibaba/clusterdata: Cluster data collected from production clusters in alibaba for cluster management research, 2018.

[6] B. Artley. Time Series Forecasting: Prediction Intervals. https://machinelearningmastery.com/

calculate-feature-importance-with-python/, 2022. [Online; accessed 23-February-2023].

[7] A. Bahga, V. K. Madisetti, et al. Synthetic workload generation for cloud computing applications. *Journal of Software Engineering and Applications*, 4(07):396, 2011.

[8] S. Bergsma, T. Zeyl, A. Senderovich, and J. C. Beck. Generating complex, realistic cloud workloads using recurrent neural networks. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 376–391, New York, NY, USA, 2021. Association for Computing Machinery.

[9] J. O. Berkey and P. Y. Wang. Two-dimensional finite bin-packing algorithms. *The Journal of the Operational Research Society*, 38(5):423–429, 1987.

[10] J. Brownlee. How to Calculate Feature Importance With Python. https://towardsdatascience.com/time-series-forecasting-prediction-intervals-360b1bf4b085, 2020. [Online; accessed 5-March-2023].

[11] J. Brownlee, D. K. Arima, C. R. McLeod, et al. pmdarima: A library for seasonal arima modeling in python, 2024. Accessed: 2023-04-30.

[12] E. B.V. The heart of the free and open elastic stack. https://www.elastic.co/elasticsearch/. [Accessed 03-08-2020].

[13] E. B.V. Your window into the elastic stack. https://www.elastic.co/kibana. [Accessed 05-01-2020].

[14] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya. Workload prediction using arima model and its impact on cloud applications' qos. *IEEE Transactions on Cloud Computing*, 3(4):449–458, 2015.

[15] M. C. Calzarossa, L. Massari, and D. Tessera. Workload characterization: A survey revisited. *ACM Comput. Surv.*, 48(3), feb 2016.

[16] Z. Cao, G. Kuenning, and E. Zadok. Carver: Finding important parameters for storage system tuning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 43–57, Santa Clara, CA, Feb. 2020. USENIX Association.

[17] Z. Cao, V. Tarasov, S. Tiwari, and E. Zadok. Towards better understanding of black-box Auto-Tuning: A comparative analysis for storage systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 893–907, Boston, MA, July 2018. USENIX Association.

[18] Z. Cao, V. Tarasov, S. Tiwari, and E. Zadok. Towards better understanding of black-box Auto-Tuning: A comparative analysis for storage systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 893–907, Boston, MA, July 2018. USENIX Association.

[19] J. Chase, A. Gallatin, and K. Yocum. End system optimizations for high-speed tcp. *IEEE Communications Magazine*, 39(4):68–74, 2001.

[20] J. Chase, A. Gallatin, and K. Yocum. End system optimizations for high-speed tcp. *IEEE Communications Magazine*, 39(4):68–74, 2001.

[21] H. Chen, G. Jiang, H. Zhang, and K. Yoshihira. Boosting the performance of computing systems through adaptive configuration tuning. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, page 1045–1049, New York, NY, USA, 2009. Association for Computing Machinery.

[22] W. Chen, K. Ye, Y. Wang, G. Xu, and C.-Z. Xu. How does the workload look like in production cloud? analysis and clustering of workloads on alibaba cluster trace. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 102–109, 2018.

[23] M. Cheng, J. Li, and S. Nazarian. Drl-cloud: Deep reinforcement learning-based resource provisioning and task scheduling for cloud service providers. pages 129–134, 01 2018.

[24] Y. Cheng, Z. Chai, and A. Anwar. Characterizing co-located datacenter workloads: An alibaba case study. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, APSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[25] Y.-H. C. Ching-Fu Chen and Y.-W. Chang. Seasonal arima forecasting of inbound air travel arrivals to taiwan. *Transportmetrica*, 5(2):125–140, 2009.

[26] S. P. E. Corporation. Spec's benchmarks and tools. https://www.spec.org/benchmarks.html. [Accessed 03-10-2021].

[27] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, 2017.

[28] G. Da Costa, L. Grange, and I. de Courchelle. Modeling, classifying and generating large-scale google-like workload. *Sustainable Computing: Informatics and Systems*, 19:305–314, 2018.

[29] Dan Becker. Poisson regression model. https://timeseriesreasoning.com/contents/poisson-regression-model/, 2022. Accessed: 2022-12-25.

[30] T. H. Davenport, P. Barth, and R. Bean. How 'big data' is different. *MIT Sloan Management Review*, pages 22–24, 2012.

[31] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *SIGPLAN Not.*, 48(4):77–88, mar 2013.

[32] T. Dong, F. Xue, C. Xiao, and J. Li. Task scheduling based on deep reinforcement learning in a cloud manufacturing environment. *Concurrency and Computation: Practice and Experience*, 32, 01 2020.

[33] T. Dong, F. Xue, C. Xiao, and J. Li. Task scheduling based on deep reinforcement learning in a cloud manufacturing environment. *Concurr. Comput.*, 32(11), June 2020.

[34] A. Dosovitskiy and V. Koltun. Learning to act by predicting the future. *ArXiv*, abs/1611.01779, 2016.

[35] G. E. P. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung. *Time Series Analysis: Forecasting and Control*. Holden-Day, San Francisco, 1970.

[36] J. Fattah, L. Ezzine, Z. Aman, H. E. Moussami, and A. Lachhab. Forecasting of demand using arima model. *International Journal of Engineering Business Management*, 10:1847979018808673, 2018.

[37] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, and S. Rao. Medea: scheduling of long running applications in shared production clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[38] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 323–336, USA, 2011. USENIX Association.

[39] I. Gog, M. Schwarzkopf, A. Gleave, R. N. M. Watson, and S. Hand. Firmament: fast, centralized cluster scheduling at scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 99–115, USA, 2016. USENIX Association.

[40] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[41] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM Con-*

ference on SIGCOMM, SIGCOMM '14, page 455–466, New York, NY, USA, 2014. Association for Computing Machinery.

[42] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 455–466, New York, NY, USA, 2014. Association for Computing Machinery.

[43] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*, pages 1–10, 2019.

[44] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*, pages 1–10, 2019.

[45] M. T. Hagan, H. B. Demuth, M. H. Beale, and O. De Jesús. *Neural network design. PWS publishing company.* Boston, 1996.

[46] H. v. Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, page 2094–2100. AAAI Press, 2016.

[47] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[48] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn. Self-adaptive workload classification and forecasting for proactive resource provisioning. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, page 187–198, New York, NY, USA, 2013. Association for Computing Machinery.

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

[50] E. H. Houssein, A. G. Gad, Y. M. Wazery, and P. N. Suganthan. Task scheduling in cloud computing based on meta-heuristics: Review, taxonomy, open challenges, and future trends. *Swarm and Evolutionary Computation*, 62:100841, 2021.

[51] R. J. Hyndman. Forecasting with long seasonal periods. https://robjhyndman.com/hyndsight/longseasonality/, 2010. [Online; accessed 15-January-2023].

[52] D. Janardhanan and E. Barrett. Cpu workload forecasting of machines in data centers using lstm recurrent neural networks and arima models. In *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 55–60, 2017.

[53] Jason Brownlee. Repeated k-fold cross-validation for model evaluation in python. https://machinelearningmastery.com/repeated-k-fold-cross-validation-with-python/.

[54] C. Jiang, G. Han, J. Lin, G. Jia, W. Shi, and J. Wan. Characteristics of co-allocated online services and batch jobs in internet data centers: A case study from alibaba cloud. *IEEE Access*, 7:22495–22508, 2019.

[55] C. Jiang, Y. Qiu, W. Shi, Z. Ge, J. Wang, S. Chen, C. Cérin, Z. Ren, G. Xu, and J. Lin. Characterizing co-located workloads in alibaba cloud datacenters. *IEEE Transactions on Cloud Computing*, 10(4):2381–2397, 2022.

[56] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework. In *2012 Proceedings IEEE INFOCOM*, pages 1206–1214, 2012.

[57] D.-C. Juan, L. Li, H.-K. Peng, D. Marculescu, and C. Faloutsos. Beyond poisson: Modeling inter-arrival time of requests in a datacenter. In V. S. Tseng,

T. B. Ho, Z.-H. Zhou, A. L. P. Chen, and H.-Y. Kao, editors, *Advances in Knowledge Discovery and Data Mining*, pages 198–209, Cham, 2014. Springer International Publishing.

[58] H. Khalid and A. Couch. Optimizing cloud resource utilization with deep reinforcement learning. In *Proceedings of the INFORMS Workshop on Data Science*, 2024.

[59] H. Khalid, P. Portante, and A. Couch. Linux configuration tuning: Is having a large dataset enough? In *Proceedings of the International Conference on Pattern Recognition Applications and Methods (ICPRAM)*, 2024.

[60] H. Khalid, A. Ramaswamy, S. Ferlin, and A. Couch. Modeling batch tasks using recurrent neural networks in co-located alibaba workloads. In *Proceedings of the 13th International Conference on Pattern Recognition Applications*, 2024.

[61] F. Koltuk and E. G. Schmidt. A novel method for the synthetic generation of non-i.i.d workloads for cloud data centers. In *2020 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, 2020.

[62] F. Li and B. Hu. Deepjs: Job scheduling based on deep reinforcement learning in cloud data center. In *Proceedings of the 4th International Conference on Big Data and Computing*, ICBDC '19, page 48–53, New York, NY, USA, 2019. Association for Computing Machinery.

[63] H. Li, L. Lu, W. Shi, G. Tan, and H. Luo. Energy-aware scheduling for spark job based on deep reinforcement learning in cloud. *Computing*, 105(8):1717–1743, mar 2023.

[64] Q. Liu and Z. Yu. The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from alibaba trace. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 347–360, New York, NY, USA, 2018. Association for Computing Machinery.

[65] C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai. Imbalance in the cloud: An analysis on alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2884–2892, 2017.

[66] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, page 50–56, New York, NY, USA, 2016. Association for Computing Machinery.

[67] I. Menache, S. Mannor, and N. Shimkin. Basis function adaptation in temporal difference reinforcement learning. *Ann. Oper. Res.*, 134(1):215–238, Feb. 2005.

[68] M. P. Mesnier, M. Wachs, R. R. Sambasivan, A. X. Zheng, and G. R. Ganger. Modeling the relative fitness of storage. In *Measurement and Modeling of Computer Systems*, 2007.

[69] S. Mishra. Methods for Normality Test with Application in Python. https://towardsdatascience.com/methods-for/-normality-test-with-application-in-python-bb91b49ed0f5, 2020. [Online; accessed 15-February-2023].

[70] A. Mitrani. Time Series Decomposition and Statsmodels Parameters. https://towardsdatascience.com/time-series/-decomposition-and-statsmodels-parameters-69e54d035453, 2020. [Online; accessed 19-January-2023].

[71] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015.

[72] G. E. Monahan. State of the art—a survey of partially observable markov decision processes: Theory, models, and algorithms. *Manage. Sci.*, 28(1):1–16, Jan. 1982.

[73] S. S. Mondal, N. Sheoran, and S. Mitra. Scheduling of time-varying workloads using reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(10):9000–9008, May 2021.

[74] I. S. Moreno, P. Garraghan, P. Townend, and J. Xu. Analysis, modeling and simulation of workload patterns in a large-scale utility cloud. *IEEE Transactions on Cloud Computing*, 2(2):208–221, 2014.

[75] N. Nadgir, M. Tuxen, and V. Kononenko. uperf. https://github.com/uperf/uperf, 2009. [Accessed 08-25-2019].

[76] A. Nandy, S. Seshathri, and A. Sarkar. Maze solving using deep q-network. In *Proceedings of the 2023 6th International Conference on Advances in Robotics*, AIR '23, New York, NY, USA, 2023. Association for Computing Machinery.

[77] I. K. Nti, J. A. Quarcoo, J. Aning, and G. K. Fosu. A mini-review of machine learning in big data analytics: Applications, challenges, and prospects. *Big Data Mining and Analytics*, 5(2):81–97, 2022.

[78] B. Ozisikyilmaz, G. Memik, and A. Choudhary. Machine learning models to predict performance of computer system design alternatives. In *2008 37th International Conference on Parallel Processing*, pages 495–502, 2008.

[79] D. C. Parkes, A. D. Procaccia, and N. Shah. Beyond dominant resource fairness: Extensions, limitations, and indivisibilities. *ACM Trans. Econ. Comput.*, 3(1), mar 2015.

[80] Penn State University. Statistical modeling: Lesson 2.4. https://online.stat.psu.edu/stat504/lesson/2/2.4, 2022. Accessed: 2023-02-23.

[81] L. Ran, X. Shi, and M. Shang. Slas-aware online task scheduling based on deep reinforcement learning method in cloud environment. In *2019 IEEE 21st*

International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pages 1518–1525, 2019.

[82] RedHat. Performance tuning guide for rhel 7. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/performance_tuning_guide/index, 2018. [Accessed 07-16-2019].

[83] B. M. Reeves, J. Hunsaker, P. Moravec, and J. Castillo. Sos. https://github.com/sosreport/sos, 2014. [Accessed 04-02-2020].

[84] C. Reiss, J. Wilkes, and J. L. Hellerstein. Google cluster-usage traces: format+ schema. *Google Inc., White Paper*, 1:1–14, 2011.

[85] A. Saboori, G. Jiang, and H. Chen. Autotuning configurations in distributed systems for performance improvements using evolutionary strategies. In *2008 The 28th International Conference on Distributed Computing Systems*, pages 769–776, 2008.

[86] A. Saboori, G. Jiang, and H. Chen. Autotuning configurations in distributed systems for performance improvements using evolutionary strategies. In *2008 The 28th International Conference on Distributed Computing Systems*, pages 769–776, 2008.

[87] S. Sagiroglu and D. Sinanc. Big data: A review. In *2013 International Conference on Collaboration Technologies and Systems (CTS)*, pages 42–47, 2013.

[88] ShapeHost. Networking features in the linux kernel and how to configure them. https://shape.host/resources/networking-features-in-the-linux-kernel-/and-how-to-configure-them, n.d. Accessed: 2023-09-27.

[89] S. Siami-Namini, N. Tavakoli, and A. Siami Namin. A comparison of arima and lstm in forecasting time series. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1394–1401, 2018.

[90] SpeedGuide.net. Network adapter optimization. https://www.speedguide.net/articles/network-adapter-optimization-3449, n.d. Accessed: 2024-01-27.

[91] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[92] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999.

[93] S. Swarup, E. M. Shakshuki, and A. Yasar. Task scheduling in cloud using deep reinforcement learning. *Procedia Computer Science*, 184:42–51, 2021. The 12th International Conference on Ambient Systems, Networks and Technologies (ANT) / The 4th International Conference on Emerging Data and Industry 4.0 (EDI40) / Affiliated Workshops.

[94] G. L. Team. Understanding xgboost algorithm — what is xgboost algorithm? https://www.mygreatlearning.com/blog/xgboost-algorithm/. [Accessed 06-15-2019].

[95] A. Theurer, P. Portante, N. Dokos, and K. Rister. pbench. https://github.com/distributed-system-analysis/pbench, 2015. [Accessed 06-09-2019].

[96] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter. The netlogger methodology for high performance distributed systems performance

analysis. In *Proceedings. The Seventh International Symposium on High Performance Distributed Computing (Cat. No.98TB100244)*, pages 260–267, 1998.

[97] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. Borg: the next generation. In *Proceedings of the fifteenth European conference on computer systems*, pages 1–14, 2020.

[98] A. Verma, M. Korupolu, and J. Wilkes. Evaluating job packing in warehouse-scale computing. In *2014 IEEE International Conference On Cluster Computing (CLUSTER)*, pages 48–56, Los Alamitos, CA, USA, sep 2014. IEEE Computer Society.

[99] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[100] Y. Wei, D. Kudenko, S. Liu, L. Pan, L. Wu, and X. Meng. A reinforcement learning based workflow application scheduling approach in dynamic cloud environment. In I. Romdhani, L. Shu, H. Takahiro, Z. Zhou, T. Gordon, and D. Zeng, editors, *Collaborative Computing: Networking, Applications and Worksharing*, pages 120–131, Cham, 2018. Springer International Publishing.

[101] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, page 287–296, New York, NY, USA, 2004. Association for Computing Machinery.

[102] R. Xu, S. Mitra, J. Rahman, P. Bai, B. Zhou, G. Bronevetsky, and S. Bagchi. Pythia: Improving datacenter utilization via precise contention prediction for multiple co-located workloads. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, page 146–160, New York, NY, USA, 2018. Association for Computing Machinery.

[103] T. Yiu. Understanding random forest: how the algorithm works and why it is so effective. https://towardsdatascience.com/understanding-random-forest-58381e0602d2. [Accessed 08-10-2019].

[104] C. S. Yu Wang, Chunheng Wang and B. Xiao. Short-term cloud coverage prediction using the arima time series model. *Remote Sensing Letters*, 9(3):274–283, 2018.

[105] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, M. Ran, and Z. Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 415–432, New York, NY, USA, 2019. Association for Computing Machinery.

[106] Y. Zhang, Y. Yu, W. Wang, Q. Chen, J. Wu, Z. Zhang, J. Zhong, T. Ding, Q. Weng, L. Yang, C. Wang, J. He, G. Yang, and L. Zhang. Workload consolidation in alibaba clusters: The good, the bad, and the ugly. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 210–225, New York, NY, USA, 2022. Association for Computing Machinery.

[107] X. Zhao and C. Wu. Large-scale machine learning cluster scheduling via multi-agent graph reinforcement learning. *IEEE Transactions on Network and Service Management*, PP:1–1, 12 2021.

[108] T. Zheng, W. Cai, Y. Xu, H. Wang, K.-C. Ma, and X. Sun. An online control framework for resource efficiency in virtualized cloud environments. *Cluster Computing*, 23(3):1879–1891, 2020.

[109] T. Zheng, J. Wan, J. Zhang, and C. Jiang. Deep reinforcement learning-based workload scheduling for edge computing. *J. Cloud Comput. Adv. Syst. Appl.*, 11(1), Dec. 2022.

[110] Y. Zhu, J. Liu, M. Guo, W. Ma, and Y. Bao. Acts in need: Automatic configuration tuning with scalability guarantees. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, New York, NY, USA, 2017. Association for Computing Machinery.