Detecting Anomalies From Big Data System Logs

2019

Siyang Lu
University of Central Florida

DETECTING ANOMALIES FROM BIG DATA SYSTEM LOGS

by

SIYANG LU
M.S. Tianjin University, 2015

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2019

Major Professor: Liqiang Wang

ABSTRACT

Nowadays, big data systems (*e.g.*, Hadoop and Spark) are being widely adopted by many domains for offering effective data solutions, such as manufacturing, healthcare, education, and media. A common problem about big data systems is called anomaly, *e.g.*, a status deviated from normal execution, which decreases the performance of computation or kills running programs. It is becoming a necessity to detect anomalies and analyze their causes. An effective and economical approach is to analyze system logs. Big data systems produce numerous unstructured logs that contain buried valuable information. However manually detecting anomalies from system logs is a tedious and daunting task.

This dissertation proposes four approaches that can accurately and automatically analyze anomalies from big data system logs without extra monitoring overhead. Moreover, to detect abnormal tasks in Spark logs and analyze root causes, we design a utility to conduct fault injection and collect logs from multiple compute nodes. (1) Our first method is a statistical-based approach that can locate those abnormal tasks and calculate the weights of factors for analyzing the root causes. In the experiment, four potential root causes are considered, `i.e.`, CPU, memory, network, and disk I/O. The experimental results show that the proposed approach is accurate in detecting abnormal tasks as well as finding the root causes. (2) To give a more reasonable probability result and avoid ad-hoc factor weights calculating, we propose a neural network approach to analyze root causes of abnormal tasks. We leverage General Regression Neural Network (GRNN) to identify root causes for abnormal tasks. The likelihood of reported root causes is presented to users according to the weighted factors by GRNN. (3) To further improve anomaly detection by avoiding feature extraction, we propose a novel approach by leveraging Convolutional Neural Networks (CNN). Our proposed model can automatically learn event relationships in system logs and detect anomaly with high accuracy. Our deep neural network consists of `logkey2vec` embeddings,

three 1D convolutional layers, a dropout layer, and max pooling. According to our experiment, our CNN-based approach has better accuracy compared to other approaches using Long Short-Term Memory (LSTM) and Multilayer Perceptron (MLP) on detecting anomaly in Hadoop Distributed File System (HDFS) logs. (4) To analyze system logs more accurately, we extend our CNN-based approach with two attention schemes to detect anomalies in system logs. The proposed two attention schemes focus on different features from CNN's output. We evaluate our approaches with several benchmarks, and the attention-based CNN model shows the best performance among all state-of-the-art methods.

ACKNOWLEDGMENTS

I would like to thank my academic advisor, Dr. Liqiang Wang, for his extraordinary guidance, caring, and patience. This dissertation would not be completed without his support and tireless help. As an excellent supervisor and researcher, he will be a great example throughout my academic life.

I would like to express my deepest gratitude to the dissertation committee members, Dr. Shaojie Zhang, Dr. Wei Zhang, and Dr. Dazhong Wu, for their invaluable suggestions to improve this dissertation.

I would like to thank my parents and my lovely girl Q. You are always there for me, love me, help me, care for me. My family members are the most important part of my life.

I would like to thank all of my colleagues in the UCF HEC-213 lab. Especially, I would like to appreciate Xiang, Yandong, and Jie for their knowledge sharing, collaboration, and contribution to various projects related to this dissertation.

Last but not least, I would like to thank my friends in Wyoming and Florida, my Gracie Barra coaches, Prof. Steven Silva, and Prof. Todd Broadaway. This dissertation would not have been possible without their warm love, continued patience, and endless support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Big data system plays an increasingly important role along with the rapid growth of massive data size. Several parallel computing frameworks have been widely used in real-world applications such as Dryad [31], Hadoop [1], and Spark [2]. When these big data systems process numerous data in parallel on distributed file systems [65, 23, 11, 59], they also produce massive logs. In order to scrutinize problems in big data systems and improve their performance, these logs can be leveraged to mine crucial information for performance tuning. Log-based anomaly detection is one of common approaches to improve security and performance.

When anomalies happen, programs could be terminated or impacted, and the performance of computation will be decreased. System logs will record all execution histories of programs. However, analyzing these logs is very challenging. It is very hard to analyze their root causes by only using system logs, because the logs are numerous and various. For example, Hadoop and Spark applications often demand long execution duration, thus huge size of logs will be generated [55, 56, 47]. Furthermore, each system may employ its own logging framework such as log4j [22] and self4j [3]; hence log formats could be diverse. Human-based manual detection methods are time-consuming with low accuracy. To identify anomalies from different logging frameworks, it requires that users are very familiar with the whole systems. Moreover, some unexpected events happening during the program execution might cause big performance degradation, or failures, even some worst scenarios such as programs keep quit, and stragglers happened without error message. Those scenarios are hard to be detected manually, even for system experts. Therefore, to effectively detect anomalies from such huge and unstructured logs is a big challenge for system operators.

In this dissertation, we propose four approaches that can accurately and automatically analyze anomalies from big data system logs. Moreover, to detect abnormal tasks in Spark logs and analyze

root causes, we design a utility to conduct fault injection and collect logs from multiple compute nodes.

- Our first method is a statistical approach that can locate those abnormal tasks and calculate the weights of factors for analyzing root causes. In the experiment, four potential root causes are considered, *i.e.*, CPU, memory, network, and disk I/O. The experimental results show that the proposed approach is accurate in detecting abnormal tasks as well as finding root causes.

- To give a more reasonable probability result and avoid ad-hoc factor weight calculation, we propose a neural network approach to analyze root causes of abnormal tasks. We leverage General Regression Neural Network (GRNN) to identify root causes for abnormal tasks.

- To further improve anomaly detection by avoiding feature extraction, we propose a novel approach by leveraging Convolutional Neural Networks (CNN). Our proposed model can automatically learn event relationships in system logs and detect anomalies with high accuracy. Our deep neural network consists of logkey2vec embeddings, three 1D convolutional layers, a dropout layer, and max pooling. According to our experiment, our CNN-based approach has better accuracy compared to other approaches using Long Short-Term Memory (LSTM) and Multilayer Perceptron (MLP) on detecting anomalies in Hadoop Distributed File System (HDFS) logs.

- To analyze system logs more accurately, we extend our CNN-based approach with two attention schemes to detect anomalies in system logs. The proposed two attention schemes focus on different features from CNN's output. We evaluate our approaches with several benchmarks, the attention-based CNN model shows the best performance among all state-of-the-art methods. Additionally, we use an internal attention scheme to create adversarial examples for evaluating the robustness of neural network approaches for log analysis.

1.1 Big Data Systems and Logs

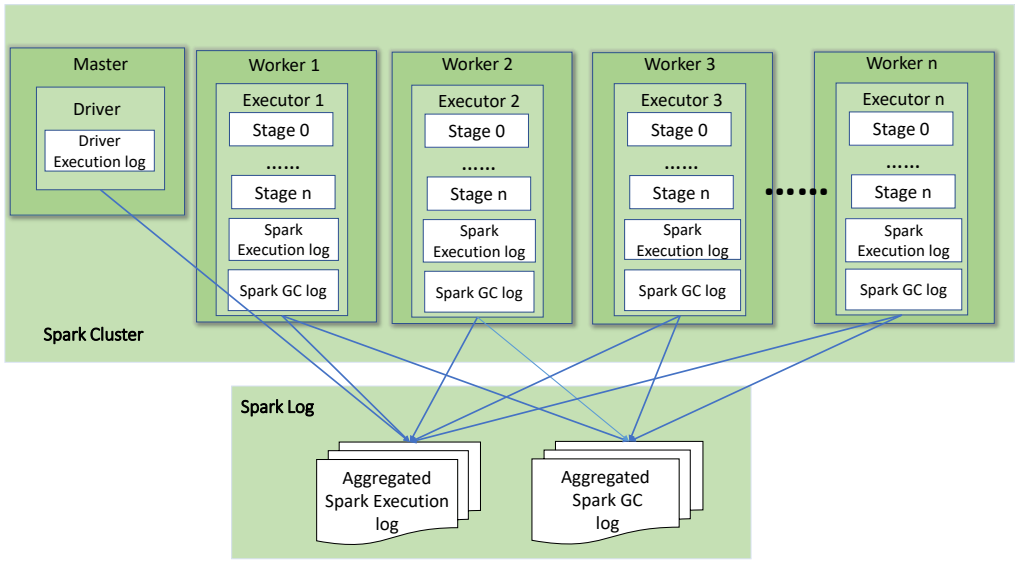

Figure 1.1: Spark framework and log files

Apache Hadoop [1] is a popular big data computing platform by leveraging MapReduce computing framework that splits input data sets into independent chunks in parallel. The framework sorts the outputs of maps, which are then fed to reduce tasks. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks. Typically, both input and output of jobs are stored on a file system called Hadoop distributed file system (HDFS) [9], which is a distributed, scalable, and portable file-system written in Java for the Hadoop framework. Hadoop ecosystem consists of a few components such as HDFS, PIG, ZOOKEEPER, and YARN.

Beside of the above, Apache Spark [2] is a fast and general engine for large-scale data processing. In order to achieve scalability and fault tolerance, Spark introduces resilient distributed dataset (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. As shown in Figure 1.1, Spark cluster consists of one master node and several slave nodes, named as workers, which may contain one or more executors. When

a Spark application is submitted, the master will request computing resources from the resource manager based on the requirement of the application. When the resource is ready, Spark scheduler distributes tasks to all executors to run in parallel. During this process, the master node will monitor the status of executors and collect results from worker nodes. When an application is submitted to Spark, the cluster manager will allocate compute resources according to the requirement of the application, then Spark scheduler distributes tasks to executors, and tasks will be executed in parallel. During this process, Spark driver node will monitor the status of executors and collect tasks results from worker nodes. In order to parallelize a job, Spark scheduler divides an application into a series of stages based on data dependence [66]. The tasks within a stage do not have data dependence and usually execute the same function.

Spark and Hadoop both use "log4j", a Java logging framework, as its logging framework. Spark users can customize "log4j" by changing configuration parameters, such as log level, log pattern, and log direction. In our experiment, we use the default configurations in "log4j". As shown in Figure 1.2, each line of Spark execution log contains four types of information: *timestamp* with ISO format, *logging level* (INFO, WARNING, or ERROR), *related class* (which class prints out this message) and *message content*. A message content contains two main kinds of information: constant keywords (`Finished task in stage TID in ms on`), and variables (`1.0 1.0 47 14075..`).

During the execution of a Spark application, JVM monitors memory usage and outputs its status to GC logs when garbage collection is invoked. GC logs report two kinds of memory usage: heap space and young generation space, where young generation space is a part of heap memory space to store new objects. Figure 1.3 shows an example of Spark JVM GC log, where "Allocation Failure" invokes this GC operation, and "PSYoungGen" shows the usage of young generation memory space. In "95744K->9080K(111616K)", the first numeric is the young space before this GC happens, the second one is the young space after this GC, and the last one is the total young

memory space. Similarly, "95744K->9088K(367104K)" illustrates heap memory instead of young generation space.

17/02/22 21:04:02.259 INFO
TaskSetManager: Starting task 12.0 in stage 1.0 (TID 58, 10.190.128.101, partition 12, ANY, 5900 bytes)
17/02/22 21:04:02.259 INFO
CoarseGrainedSchedulerBackend$DriverEndpoint: Launching task 58 on executor id: 1 hostname: 10.190.128.101.
17/02/22 21:04:02.276 INFO
TaskSetManager: Finished task 1.0 in stage 1.0 (TID 47) in 14075 ms on 10.190.128.101 (1/384)

Figure 1.2: An example of Spark execution log.

GC (Allocation Failure)
PSYoungGen: 95744K->9080K(111616K)
95744K->9088K(367104K), 0.0087250 secs] [Times: user=0.03, sys=0.01, real=0.01 secs]

Figure 1.3: An example of Spark garbage collection (GC) log.

1.2 Anomaly Tasks

Anomaly could be identified as an abnormal execution of a program. It could be a task straggler, error, and even some programming warning. For root cause analysis, we only consider four kinds of resource failures, *i.e.*, CUP, DISK, IO, and Networks.

A log entry (logline) is considered as *anomaly* if it contains abnormal key words (*e.g.* "error", "warning") or shows significant unexpected order in context, such as a Spark executor restarts repeatedly before it stops working. Classical anomaly detection has been studied for many years. Various algorithms and methods have been developed, such as basic keyword searching, regulation expression matching, traditional statistical and machine learning approaches. It may incorrectly identify anomalies and report false positives when searching anomalies with key words, or

matching with regular expression.

Hence, some techniques such as Support Vector Machine (SVM) and Principal Component Analysis (PCA) are often used to reduce the complexity of feature set to be analyzed and improve accuracy. However, the hidden relationships in extracted feature set are still very difficult to be analyzed by these aforementioned approaches, which often require more sophisticated approaches. However, those features may be produced by ad-hoc and it would mislead the approach to learn knowledge from the wrong rules. Anomaly logs are not only output with critical levels or some keywords (error, warning) but also be printed out with different execution paths without special keyword.

In recent years, deep learning approaches are leveraged in the log analysis domain to improve automation and accuracy. For instance, Long Short Term Memory (LSTM) and Recurrent Neural Network (RNN) are used by [17, 10] to detect anomalies with high accuracy to avoid ad-hoc feature extraction. Within all deep learning methods, Convolutional Neural Network (CNNs) could be the most famous and widely used approach, which has obtained great achievements in computer vision. Due to convolution layers, the CNN-based approach can learn the hidden relationships with higher accuracy than other deep learning methods.

1.3 Contributions

The main contributions of this dissertation are our four anomaly detection approaches. This dissertation mainly focuses on analyzing logs of big data systems, but the techniques can also be applied to other systems.

- Our offline approaches can accurately locate where and when abnormal tasks happen based on analyzing only Spark logs. Those approaches detect root causes of abnormal tasks ac-

cording to Spark logs without any monitoring data, thus it does not have any monitoring overhead.

- Our offline approaches provides an easy way for users to deeply understand Spark logs and tune Spark performance, and it gives a reasonable probability results for root cause analysis.
- We propose a CNN-based approach for anomaly detection of HDFS logs.
- We propose two attention schemes to improve the accuracy of log-based anomaly detection of HDFS logs.
- A new embedding method called `logkey2vec` is designed to learn how to map logkeys to vectors.
- We propose an efficient attention-based optimization method to manipulate discrete text structure according to its embedding representation.
- We investigate the robustness of a classifier trained with adversarial examples by studying its effectiveness to attack the networks.

1.4 Organization

This dissertation describes four approaches that can accurately analyze anomalies from big data system logs without extra monitoring overhead. Chapter 2 surveys the related work about anomaly detection for logs. Chapter 3 illustrates the statistical methodology to detect abnormal tasks in Spark logs and analyze root causes. Chapter 4 proposes a GRNN neural network based approach for detecting abnormal tasks and analyze root causes from Spark logs. Chapter 5 presents CNN based anomalies detection from HDFS logs. Chapter 6 applies two attention schemes on CNN based approach for log analysis from HDFS logs. Chapter 7 proposes a white-box adversarial

attack on CNN for anomalies detection. Finally, Chapter 8 summarizes this dissertation and future work.

CHAPTER 2: LITERATURE REVIEW

In this chapter, we first review related work of log processing, and then classify the related work of log-based anomaly detection into three categories: statistical approaches, traditional machine learning approaches, and neural network-based approaches.

2.1 Log Processing Approaches

```
17/02/22 21:04:02.259 INFO TaskSetManager: Starting task 12.0 in stage 1.0 (TID 58, 10.190.128.101, partition 12, ANY, 5900 bytes)
.....
17/02/22 21:04:02.276 INFO TaskSetManager: Finished task 1.0 in stage 1.0 (TID 47) in 14075 ms on 10.190.128.101 (1/384)
```

Figure 2.1: Spark system log example

Big data system logs are unstructured data printed in time sequence. Normally, each log entry (line) can be divided into two different parts: constant and variable. The constant part is the messages printed directly by statements in the source code. Log keys can be extracted from these constant parts, where log keys are the common constant messages in all similar log entries. For example, as shown in Figure 2.1, the log key is "Starting task in stage TID partition bytes" in the log entry "Starting task 12.0 in stage 1.0 (TID 58, 10.190.128.101, partition 12, ANY, 5900 bytes)". The other part is remaining after removing constant parts in log entries, which may contain variable keywords such as "12.0 1.0 58 10.190.128.101, 12, ANY, 5900".

Detecting anomalies from system logs (such as Spark logs) requires log analysis, `i.e.`, analyzing root causes [44] using effective methods. Usually, log analysis consists of four main phases:

1. Parse unstructured raw logs into structure data by log parser techniques. There are two kinds of log parsing approaches [25]: heuristic and clustering. The clustering methods first

conduct clustering based on distances result of logs, then create a log template from each cluster. The heuristic methods count every word's appearance in these log entries and select frequently appeared words to be log events according to the predefined rules.

2. Extract log related features from parsed data. Different approaches may use different feature extraction methods (such as rule-based approach or execution path approach). There are several common window-based approaches for extracting different features such as session window, sliding window, and fixed window. Specifically, a session window is used for grouping log entries with the same session ID. A sliding window is used to slide forward in a certain step in the data and extract features with some overlaps. A fixed size of window can also be used to extract features.

3. Detect anomalies with extracted features, which are introduced in details in Subsection 2.3.

4. Fix problems based on detected anomalies. There are many different ways to help fix problems based on detected anomalies, such as root cause analysis, anomalies visualization. For example, [61] leverages a decision tree to visualize the anomalies, and [44] uses linear regression to compute the probability of abnormal tasks.

2.2 Root Cause Categories

There are several categories of the root causes for the abnormal performances. Ananthanarayanan *etal.* [6] identify three categories of root causes for Map-Reduce outliers: the key role cause is machine characteristics (resource problems), the other two causes are network and data skew problem. Ibidunmoye *et al.* [30] depict that four root causes may cause bottlenecks, which are system resource, workload size, platform problems, and application (buggy codes). Garbageman *et al.* [20] analyzes around 20-day cloud center data and summarizes that the most common root cause in

cloud center of abnormal occurrence is server resource utilization, and data skew problems only take 3% of total root causes. According to the above studies on a real-world experiment, the primary root causes of abnormal tasks are machine resources, which include CPU, memory, network, and disk I/O. Moreover, the mentioned resource root causes mainly impact the performance of CPS computation layers. Therefore, in our work, we consider only the four main root causes and ignore data skew and ineffective code problems.

2.3 Anomaly Detection Approaches

.

Statistical and machine learning techniques are promising approaches in the root cause analysis and anomaly detection. in the parallel computing area, solving the performance degradation problem caused by abnormal executions is one of the critical tasks in log analysis. Basically, it is used for detecting target patterns that differ from normal execution behaviors. Existing approaches in anomaly detection mainly use three kinds of techniques: statistical method, traditional machine learning, and deep learning approach. The first two are considered to be traditional log analysis approaches.

2.3.1 Statistical Log Analysis Approaches

As one kind of common approaches for log analysis, statistical methods can analyze logs without training or learning stages, instead, they use rule-based and static analysis methods for classification such as Principal Component Analysis (PCA) and Factor Analysis (FA). Tan *et al.* [58] introduce a pure off-line state machine tool called SALSA, which simulates data flows and control flows in big data systems with a statistical method, and leverages Hadoop's historical execution

logs. Similarly, Aguilera *et al.* [5] uses two kinds of statistical methods on distributed historical logs and monitoring data to discover causal paths (workflow). Moreover, Xu *et al.* [61] extract two kinds of log variable vectors by using a syntax tree (AST) to parse the system source code, then analyze extracted patterns from the vectors by leveraging PCA. Safyallah *et al.* [53] leverage a log-based approach to analyze frequent and common sequence execution traces in order to detect anomalies. Fu *et al.* [18] use a rule-based methods to identify the log keys and detect anomalies in distributed system logs. He *et al.* [25] propose a guideline for log-based anomaly detection by evaluating six kinds of supervised and unsupervised approaches. A log-based statistical approach for detecting abnormal tasks and analyzing root causes from Spark logs is proposed in our prior work [44]. Also, statistical approaches have online detection strategy, which is invoked during the executions of applications. For example, both Spark and Hadoop provide online "speculation" [63], which is a built-in component for detecting stragglers statistically. Although it can detect stragglers during runtime, it does not offer the root causes. In addition, the speculation is often inaccurate, `i.e.`, it may raise too many false alarms [32]. Chen *et al.* [12] propose a tool called Pinpoint that monitors the execution and uses log traces to identify the fault modules in J2EE applications via standard data mining approaches. A stream-based mining algorithm for online anomalies prediction is presented by Gu *et al.* [21]. Ananthanarayanan *et al.* [6] design a task monitoring tool called Manrti, which can cut outliers and restart tasks in real time according to its monitoring strategy. Chen *et al.* [13] propose a self-adaptive tool called SAMR, which adds weights for calculating each task duration according to historical data analysis. Aguilera *et al.* [5] propose two statistical methods to discover causal paths in distributed systems by analyzing historical log and monitoring data from the traces of applications. The most closely related work to our approach is BigRoots [67], which detects stragglers by Spark speculation and analyzes the root causes by extracted features. It leverages experience rule to extract features for each task from application logs and monitoring data. However, the threshold in Spark speculation is not proper to detect abnormal tasks. In addition, BigRoots considers only the features for each individual

task, which can not capture the status change of the cluster, thus such a rule-based method is very limited.

2.3.2 Classical Machine Learning Approaches

In order to avoid the ad-hoc feature extraction in statistical methods for anomaly detection from logs, machine learning approaches have been investigated. Support Vector Machine (SVM) and Hidden Markov Model (HMM) are common and effective supervised machine learning approaches for anomaly detection and failure prediction. Fulp *et al.* [19] parse system logs by using a sliding window and use SVM to predict anomaly. Liang *et al.* [41] leverage SVM, RIPPER (a rule-based classifier), and a customized Nearest Neighbor to build up to three classifiers for failure prediction. Lou *et al.* [43] apply a Bayesian learning approach on system logs to extract a constructed graph. Although classical machine learning methods could avoid ad-hoc feature extraction with better performance, they are more time-consuming when handling large training sets. Xu *et al.* [61] use an automatic log parser to parse the source code and combine PCA to detect an anomaly, which is based on the abstract syntax tree (AST) to analyze source code and uses machine learning to train data. Qi *et al.* [51] leverage Classification and Regression Tree (CART) to analyze straggler root causes by using Spark event logs and monitoring data (hardware metrics such as CPU status, disk read/write rate and network send/receive rate) which collected by synchronous sampling tool.

Some machine learning approaches are also leveraged in predicting system faults using logs and monitoring data, which are similar to the root cause analysis problem. Fulp*et al.* [19] leverage a sliding window to parse system logs and predict failures using SVM. Yadwadkar*et al.* [62] propose an offline approach that works with resource usage data collected from the monitoring tool Ganglia [49]. It leverages Hidden Markov Models (HMM), which is a linear machine learning approach. Moreover, there are some off-line approaches that analyze both log files and monitoring

data to identify abnormal events.

2.3.3 Deep Learning Approaches

Deep learning [15] approaches have achieved great success in various fields especially computer vision. Basically, In a regular fully connected network (FC), all neurons in the current layer are in fully connected with those in the previous layer, and back-propagation [52] is used for computing the error gradient. Nevertheless, the FC is not suitable for calculating high-dimension datasets such as images. To solve this problem, CNN can capture local semantic relationships instead of global information. Log analysis also benefits from deep neural network models. As a special recurrent neural network (RNN), Long Short Term Memory LSTM is widely used in the NLP domain. Recently, researchers have begun to leverage LSTM to analyze logs. Brown *et al.* [10] present an unsupervised LSTM and attention-based LSTM to discover hidden relationships in system logs. Du *et al.* [17] propose an LSTM-based approach named DeepLog, which uses LSTM as its training model to detect anomalies among log execution paths. In previous work [46], we detect abnormal tasks and analyze root causes from Spark logs by using a General Regression Neural Network (GRNN), which is a simple and fast neural network to avoid the ad-hoc weight calculation for classifications. To Compare with our statistical approach [44], our GRNN-based approach achieves better accuracy and offers reasonable portability results.

CNN has been widely employed in computer vision and derives many famous networks such as AlexNet [37], GoogLeNet [57], and many others. Recently, Deep Resnet [24] is proposed for image classification and achieves comparable results with manually labeled recognition performance. In NLP domain, Kim *et al.* [35] present an effective CNN model that can directly classify distributed embedding of words. Jason *et al.* [34] apply a CNN model on discrete embeddings with good performance.

Inspired by the human recognition system, attention mechanism is leveraged to continually improve performance of DNN by focusing on relevant features. When humans try to recognize objects, they usually first focus on partial information (relevant features), then entire events could be recognized and processed. The attention model is first proposed in NLP to cope with tasks and visual captioning. In the NLP area, attention-based DNN could pay attention to relevant words or sentences instead of using whole feature sets. For example, Bahdanau *et al.* [7] propose soft attention aiming at automatically capturing soft alignments between source words and target words in machine translation.

In computer vision, attention could extract proposal regions in each picture by calculating the weighted average of each feature. Li *et al.* [40] propose an end-to-end video LSTM that leverages the soft attention for video classification. Long *et al.* [42] propose an attention mechanism named Keyless Attention with better effectiveness and efficiency. In log analysis, Das *et al.* [14] propose an unsupervised attention-based LSTM approach for anomaly detection on network log.

CHAPTER 3: ABNORMAL DETECTION AND ROOT CAUSE ANALYSIS FOR SPARK

3.1 Introduction

With rapid growth of data size and diversification of workload types, big data computing platforms increasingly play more important roles in solving real-world problems [27, 26]. Several widely used frameworks include Hadoop [1], Spark [2], Storm and Flink. Among them, Apache Spark might be the popular one due to its fast and general programming model for large-scale data processing, where Resilient Distributed Dataset (RDD) [64] are used to hold input and intermediate data generated during the computation stages. RDDs are divided into different blocks, called partitions, with almost equal size among different compute nodes. Apache Spark uses pipeline to distribute various operations that work on a single partition of RDD. In order to serialize the execution of tasks, Spark introduces stage. All tasks in the same stage execute the same operation in parallel.

Compute nodes may suffer from severe interferences from other software (such as operating systems or other processes) or hardware, which leads to abnormal problems. For instance, a task could become an abnormal task or straggler when encountering a significant delay in comparison with other tasks in the same stage. In Spark, there is a mechanism named speculation to detect this scenario, where slow tasks will be re-submitted. Spark performs speculative execution of tasks till a specified fraction (defined by `spark.speculation.quantile`, which is 75% by default) of tasks must be completed, then it checks whether or not the running tasks run slower than the median of all successfully completed tasks in a stage. A task is a straggler if its current execution time is slower than the median by a given ratio (which is defined by `speculation.multiplier`,

1.5x by default). This chapter proposes a new approach compared with Spark Speculation. In our method, we consider whole Spark stages and abnormal tasks happening in all life span could be detected. In addition, Spark's report could be inaccurate because Spark uses only fixed amount of finished task durations to speculate the unfinished tasks.

When abnormal tasks (including stragglers) happen, the performance of Spark applications could be degraded. However, it is very difficult for users to detect and analyze the root causes. First, Spark log files are tedious and difficult to read, and there is no straight-forward way to tell whether abnormal tasks happen or not, even though stragglers can be reported when speculation is enabled. Second, when an abnormal scenario happens, there is not enough information about the error in log files so that it is difficult for users to see the concreted reasons that lead to the straggler problem. Third, even online tools can monitor the usage and status of system resource such as CPU, memory, disk, and network, these tools do not directly cooperate with Spark, and users still need many efforts to scrutinize root causes based on their reporting. In addition, these monitoring tools usually carry overhead and may slow down Spark's performance. Abnormal tasks could be caused by many reasons, where most of them are resource contentions [20] by CPU, memory, disk, and network. Our motivation is to help users find the root causes of abnormal tasks by analyzing only Spark logs.

This chapter proposes an off-line approach to detect abnormal tasks and analyze the root causes [44]. Our method is based on a statistical spatial-temporal analysis for Spark logs, which consists of Spark execution logs and Spark garbage collection logs. There are four steps to detect the root causes. (1) We parse Spark log files according to keywords, such as task duration, data location, timestamp, task finish time, and generate a structured log data file. This step will eliminate all irrelevant messages and values. (2) We extract the related feature set directly from structured log files based on our experimental study. (3) We detect abnormal tasks from the log data by analyzing all relevant features. Specifically, we calculate the mean and standard deviation of all tasks in each

stage, then determine abnormal tasks for each stage. (4) We generate factor combination criteria for each potential root cause based on analyzing their weighted factor in training datasets. Thus, our approach can effectively determine the proper root causes for given abnormal tasks.

3.2 Methodology

Spark log does not show abnormal tasks directly, thus users cannot locate abnormal tasks by simply searching keywords. This motivates us to design an automatic approach to help users detect the abnormals and analyze the root causes.

3.2.1 Approach Overview

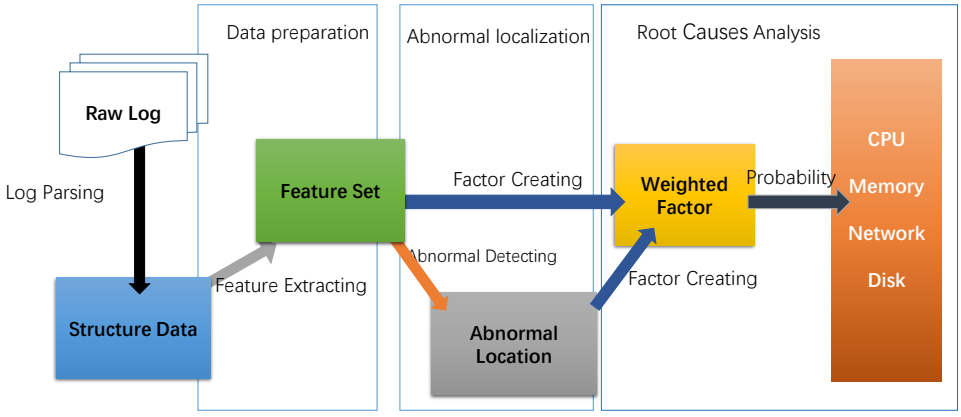

Figure 3.1: Workflow of abnormal detection and root cause analysis.

The workflow of our approach for abnormal detection and root cause analysis is shown in Figure 4.6.

1. *Log preprocessing*: We collect all Spark logs, including execution logs and Spark GC logs,

from the driver node and all worker nodes. Then, we eliminate noisy data and reformat logs into more structured data.

2. *Feature extraction*: Based on Spark scheduling and potential abnormal task happening conditions, we screen execution-related, memory-related, CPU-related data to generate two matrices: execution log matrix and GC matrix. The details are illustrated in Section 3.2.2.

3. *Abnormal detection*: We implement a statistical analysis approach based on the analysis of four kinds of features, including task duration, timestamps, GC time, and other task-related features, to determine the degree of abnormal tasks and locate their happening. The details are discussed in Section 3.2.3.

4. *Root cause detection*: Instead of qualitatively deciding the exact root causes that lead to the abnormals, we quantitatively measure the degree of abnormals by a weighted combination of certain specific cause-related factors. The details are sh in Section 3.2.5.

3.2.2 Feature Execution

According to Spark scheduling strategy, we define and classify all features into three categories, namely, execution-related, memory-related, and CPU-related, which are shown in Table 4.4. For example, the execution-related features can be extracted from Spark execution logs, including task ID, task duration, task finished time, task started time, stage ID, and job's duration. Spark GC log records all JVM memory usage, from which we can extract memory-related features such as heap usage, young space usage, as well as features related system CPU usage such as system time and user time. These feature sets extracted from Spark execution log and GC log are shown in Table 4.4.

3.2.3 Abnormal Detection

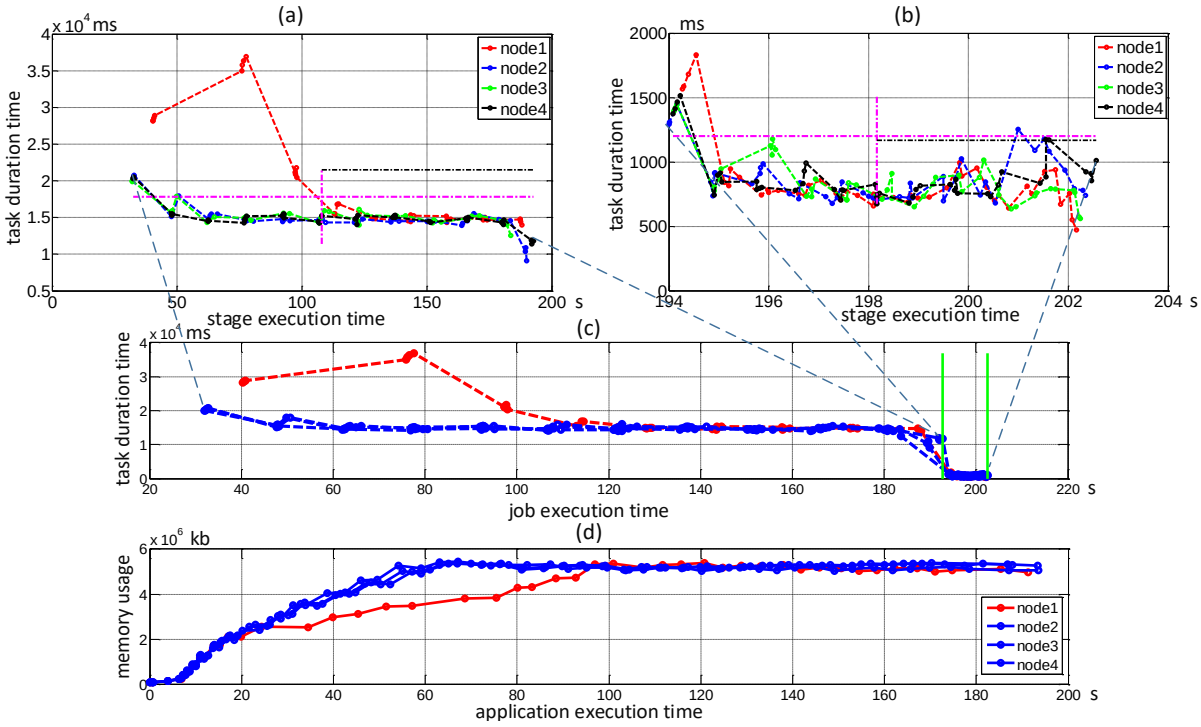

Figure 3.2: Abnormal detection under CPU interference in the experiment of WordCount: (a) Abnormal detection result in Stage-1. (b) Abnormal detection result in Stage-2. (c) Spark execution log features for abnormal detection in the whole execution. (d) Spark GC log features for abnormal detection in the whole execution.

Adopting Spark speculation may bring false negatives in the process of abnormal detection. Hence, we provide a more robust approach to locate where stragglers happen and how long they take. We will also consider about special scenarios, for example, different stages are executed in sequence or in parallel.

One basic justification of abnormal tasks is that the running time of abnormal tasks is relatively longer than the normal ones. [20] uses “mean” and “median” to decide the threshold. However, in order to seek a more reasonable anomaly detection strategy, we consider not only the mean

or median task running time, but also the distribution of the whole data, namely the standard deviation. In this way, we can get a macro-awareness on the task's execution time, and then based on the distribution of data, a more reasonable threshold can be set to differentiate abnormals from the normal ones. The abnormal detection mainly includes the following two issues.

1. *Comparing task running time on different nodes*

We compare task execution time on different nodes in the same stage. Let $T\_task_{i,j,k}$ denote the execution time of task k in stage i on node j. Let avg_stage_i denote the average execution time of all tasks, which belong to different nodes but in the same stage i.

$$avg\_stage_i = \frac{1}{\sum_{j=1}^{J} K_j}(\sum_{j=1}^{J}\sum_{k=1}^{K_j} T\_task_{i,j,k}) \tag{3.1}$$

where J and K_j are the total number of nodes and the number of tasks in node j, respectively.

Similarly, the standard deviation of task execution in stage j of all nodes is denoted as std_stage_i. Abnormal tasks are determined by the following conditions:

$$GBKsongtask_k \begin{cases} abnormal & T\_task_k > avg\_stage_i + k * std\_stage_i \\ normal & \text{otherwise} \end{cases} \tag{3.2}$$

where k is a factor that controls the threshold for abnormal detection. In this work, we set it to 1.5 by default for fair compare with Spark provided speculator.

Figure 6.3 (c) shows abnormal detection process in Wordcount under CPU interference. Figure 6.3 (a) and (b) are two stages inside the whole application. Moreover, inside each of the stage,

purple-dot line is the abnormal threshold determined by Eq. (3.1), and the black dot-line indicates the threshold calculated by Spark speculation. For all tasks within a certain stage, the execution time above that threshold are detected as abnormals; otherwise, they are normals. Figure 6.3 (d) displays memory occupation along the execution of its corresponding working stages.

2. *Locating abnormal happening*

After all tasks are properly classified into "normal" and "abnormal", the whole time line are labeled as a vector with binary number (*e.g.*, 0 or 1, which denote normal and abnormal, respectively). To smooth the outliers (for example, 1 appears in many continuous 0) inside each vector, which could be an abrupt change but not consistent abnormal base, we then empirically set a sliding window with size of 5 to flit this vector. If the sum of numbers inside the window is larger than 2, the number in the center of the window will be set to 1, otherwise 0.

The next step is to locate the start and end time of this abnormal task. Note that, as Spark logs record the task finishing time but not the start time, so we locate the abnormal task's start time as the recorded task finishing time minus its execution time. Moreover, for abnormal detection in each stage, the tasks are classified into two sets. One is for the initial tasks whose start time stamps are the begin of each stage, as these tasks often have more overhead (such as loading code and Java jar packets), and the execution time usually operates much longer than its followings. Another set consists of the rest tasks. Our experiments show that this classification inside each stage can lead to a much accurate abnormal threshold. In this way, our abnormal detection method can not only detect whether abnormals happen, but also locate where and when they happen.

Table 3.1: Extracted Spark Feature Sets

Related	Name	Meaning
	Time stamp	Event happening time
	Task duration	A task's running duration time
	Stage ID	The ID number of each stage
	Host ID	The Node ID number
	Executor ID	The ID number of each executor running in per-worker
	Task ID	The unique ID number of each task
	Job duration	A job duration(an application has many jobs)
	Stage execution time	A stage running duration time
	Application duration	An application running duration time (after submitted)
	Data require location	The location of task required data
	Heap space	Total Heap memory usage
	Before GC Young space	Young space memory usage before clearing Young space
	After Young GC space	Young space memory usage after clearing Young space
	Before Heap GC space	Total Heap memory usage before GC
	After Heap GC space	Total Heap memory usage after GC
	Full GC time	Full GC execution time
	GC time	Minor GC execution time
	GC category	The time spend on one full GC operation
	user time	CPU time spent outside kernel execution
	sys time	CPU time spent insides kernel execution
	real time	Total elapsed time of the GC operation

3.2.4 Factors Used for Root Cause Analysis

After abnormals are located, we analyze their root causes inside that certain area. For different root causes, we use different features in Spark log matrix and GC matrix to determine criteria to decide the root causes. Specifically, for each root cause analysis, we use the combination of weighted factors to define the degree of probability of each root cause. In all normal cases the factor should equal to 1, and if an abnormal tends to any root cause, the factor will become much bigger than 1. The factors are denoted as a,b,c,d,e,f,g for weight calculating. All of the indexes which are used in our factors' definition are listed here: j,J,i,I,k,K,n,N, inside which, j indicates the jth node, J is set of nodes; i is the index of stage, I is a set of stages; k denotes a task, K is a task set; n

stands for a GC record, N is GC records set. All factors used to determine root causes are listed as below.

1. *Degree of Abnormal Ratio (DAR)*

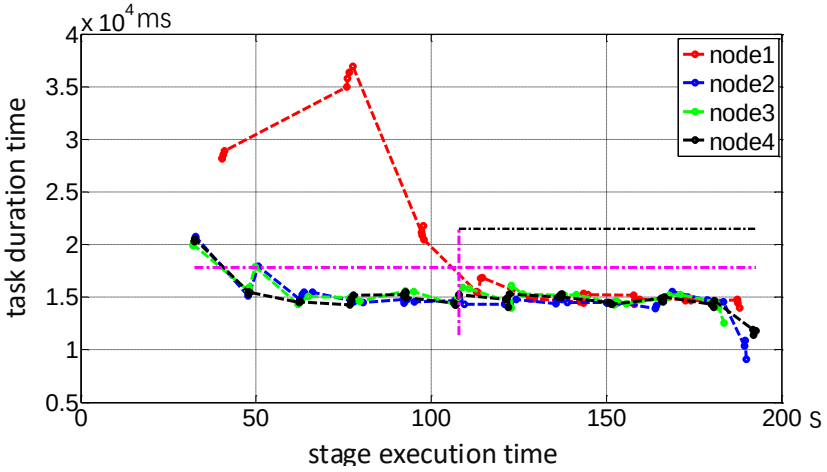

Figure 3.3: CPU interference injected after 20s application was submitted, and continuously impacts 80s

Eq. (3.3) indicates the degree of abnormal ratio in a certain stage.

$$a = \frac{k_{j'}}{\frac{1}{J-1}((\sum_{j=1}^{J} k_j) - k_{j'})} \tag{3.3}$$

where k_j indicates the number of tasks in node j, and J is the total number of nodes in the cluster. Here, we assume that node j' is abnormal.

2. *Degree of Abnormal Duration (DAD)*

The average task running time should also be considered, as the abnormal nodes often record

longer task running time.

$$b = \frac{avg\_node_{j'}}{\frac{1}{J-1}((\sum_{j=1}^{J} avg\_node_j) - avg\_node_{j'})} \tag{3.4}$$

where avg_node_j is defined as:

$$avg\_node_j = \frac{1}{K_j}(\sum_{k=1}^{K_j} T\_task_{i,j,k}) \tag{3.5}$$

3. *Degree of CPU Occupation (DCO)*

This factor c shown in Eq. (3.6) is used for expressing the ratio between the wall-clock time and the real CPU time. In the normal multiple-core environment, "realTime" is often less than "sysTime+UserTime", because GC is usually invoked in multi-threading way. However, if the "realTime" is bigger than "sysTime+UserTime", it may indicate that the system is very busy. We choose a Max value across nodes as the final factor.

$$c = \max_{j \in J}(avg_{j \in J}(\frac{realTime_{i,j}}{sysTime_{i,j} + userTime_{i,j}})) \tag{3.6}$$

4. *Memory Changing Rate (MCR)*

Eq. (3.7) indicates the gradient of GC curve. Under CPU, memory, and Disk interference, the interfered node's GC curve will change slower than the normal nodes' GC curve, as shown in Figure 3.4. k_stable and k_end are the gradients of the connected lines between start position (the corresponding memory usage at abnormal starting time) to the stable memory usage position and the start position to the abnormal memory end position (both the abnormal start and end time

are obtained in the previous section) respectively. The reason we conduct this equation is that the interfered node uses less memory than normal nodes under interference. In this way, we use the maximum value of k_stable in the whole cluster (k_stable of normal node) to divide the minimums k_end in the whole cluster (interfered node) to get the value of this factor.

$$d = \frac{\underset{j \in J}{max}(k\_stable_j)}{\underset{j' \in J}{min}(k\_end_{j'})} \tag{3.7}$$

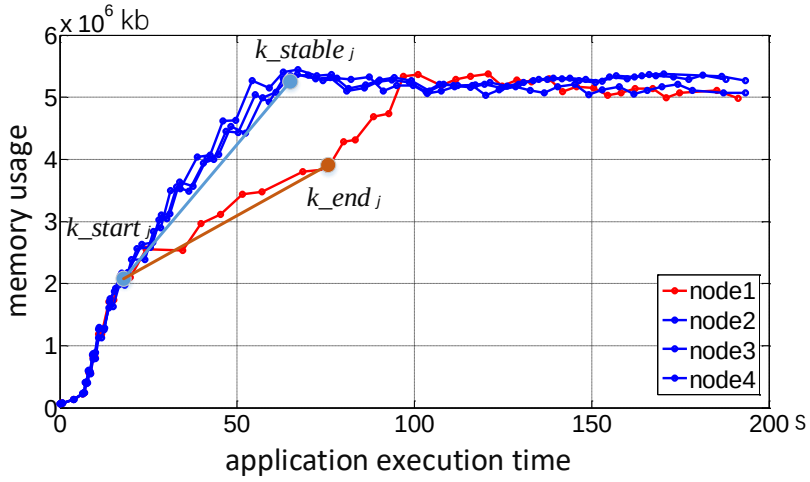

Figure 3.4: CPU interference is injected after WordCount has run for 30s, and continuously impacts 120s.

5. *Degree of Task Delay (DTD)*

For network interference, the task execution time will be affected when data transmission is delayed. Moreover, a Spark node often accesses data from other nodes, which leads to network interference propagation. Based on these facts, if network interference happens inside the cluster, the whole nodes will be affected, as shown in Figure 3.5, which is the location of our detected

interference. Let a be a factor that describes the degree of interference.

$$e = \exp(J * \prod_{j=1}^{J} abn\_prob_j) \tag{3.8}$$

Where abn_prob_j indicates the ratio of abnormal that we detect for each node j inside that area. The reason that we use the product of abnormal ratio other than the sum of them is that only when all nodes are with a portion of abnormal should we identify them with a potential of network interference, or if sum is used, we cannot detect this joint probability. Meanwhile, the exponential is to make sure that the final factor e is no less than 1. In this way, the phenomenon of error propagation will be detected and quantified, which can only be shown in the cluster with network interference injection.

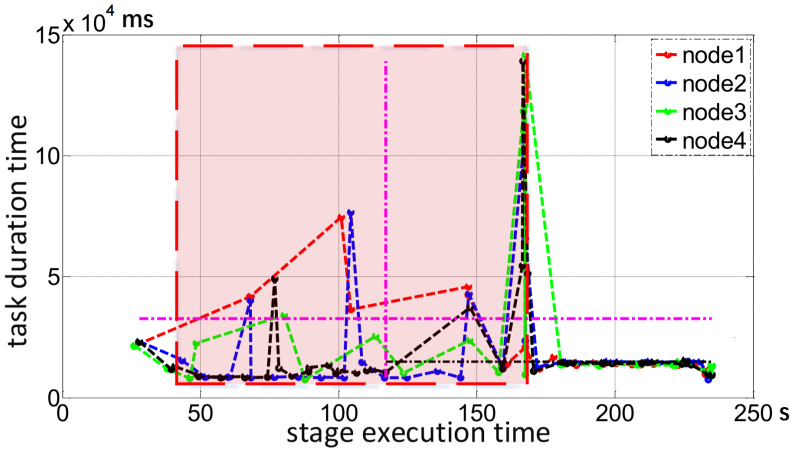

Figure 3.5: Network interference is injected after WordCount has been executed for 30s, and continuously impacts for 160s.

6. *Degree of Memory Changing (DMC)*

As network bandwidth is limited or the network speed slows down, when one node get affected by

that interference, the task will wait for their data transformation from other nodes. Hence, CPU will wait, and data transfer rate becomes low. As shown in Figure 3.6.

$$f = \frac{\underset{j \in J}{max}\{\underset{n \in N}{max}[e^{-|m_{j,n}|} * (x_{j,n} - x_{j,n-1})]\}}{\underset{j \in J}{min}\{\underset{n \in N}{max}[e^{-|m_{j,n}|} * (x_{j,n} - x_{j,n-1})]\}}$$
$$\text{where}, m_{j,n} = \frac{y_{j,n} - y_{j,n-1}}{x_{j,n} - x_{j,n-1}} \tag{3.9}$$

where $m_{j,n}$ indicates the gradient of memory changing in nth task on node j. Eq. (3.9) is to find the longest horizontal line that presents the conditions under which tasks' progress become tardy (, CPU is relatively idle and memory is kept the same). We first calculate the max value of gradient for each GC point, denoted as m. To identify the longest horizontal line in each node, we make a trade-off between its gradient and the corresponding horizontal length. To determine a relative value that presents the degree of abnormal out of normal, we finally compare the $\max$ and $\min$ among nodes with their $\max$ "horizontal factor" ($e^{-|m_{j,n}|} * (x_{j,n} - x_{j,n})$), where e is to ensure that the whole factor of b not less than 1).

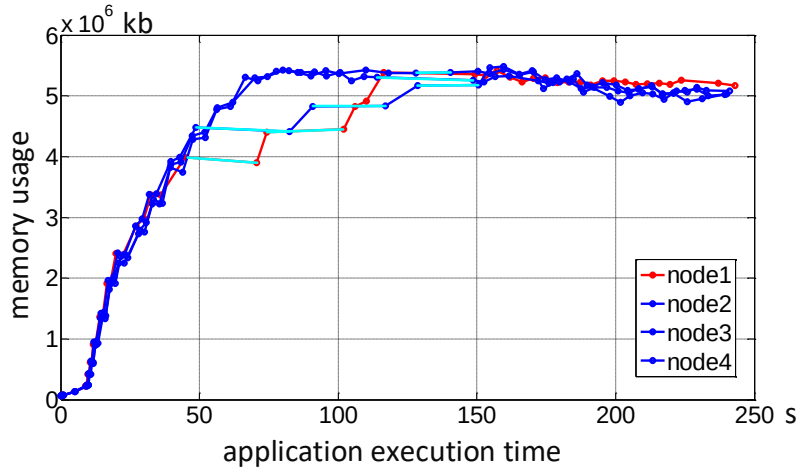

Figure 3.6: Network interference is injected after WordCount has been executed for 30s, and continuously impacts for 120s.

7. *Degree of Loading Delay (DLD)*

Considering that the initial task at the beginning of each stage always have a higher overhead to load data blocks compared to the rest tasks. As shown in Figure 3.7. To only focus on that area, the factor of g is proposed to measure its abnormality. Similar to factor f, instead of taking all the tasks inside the detected stage into consideration, here, the first task of each node is used to replace the "avg_node_j"in Eq. (3.4). Formally, the equation is modified as Eq. (3.10) shows.

$$g = \frac{T\_task_{i,j',1}}{\underset{j \in J}{avg}(T\_task_{i,j,1})}, where\ j' \notin J \tag{3.10}$$

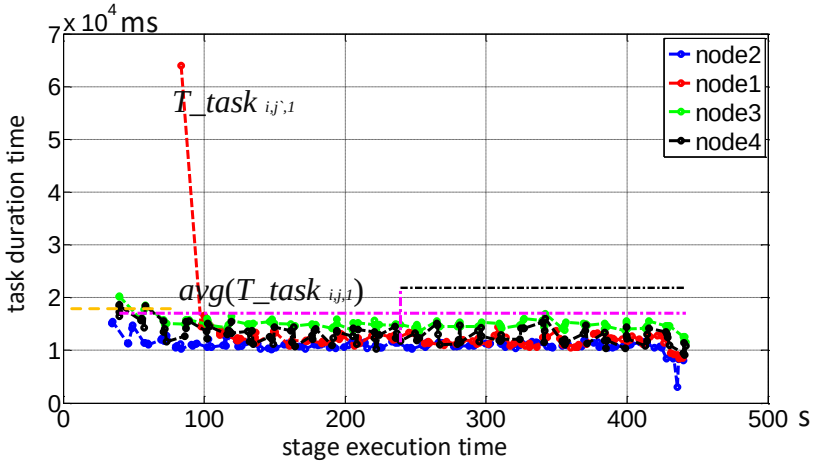

Figure 3.7: Disk interference is injected after WordCount has run for 20s, and continuously impacts 80s

3.2.5 Root Cause Analysis

As shown in Table 4.2, each root cause is determined by a combination of factors with specific weights.

The nodes with CPU interference often have a relatively lower computation capacity, which leads to less tasks allocated and longer execution time for tasks on it. Factors a and b are used to test if the interference is CPU or not, because CPU interference can reduce the number of scheduled tasks and increase the abnormal tasks' execution time. Factor c indicates the degree of CPU occupation, and CPU interference will slow down of the performance compared to normal cases. Factor d is used to measure memory changing rate, because CPU interference may lead memory change to become slowly than other regular nodes.

For the network-related interferences, because of its propagation, the original interfered node will often recover earlier. So our approach is to detect the first recovered node as the initial network-interfered node, and the degree b quantitatively describes the interference. When network interference occurs, tasks are usually waiting for data delivery (factor e), the memory monitored by GC log f is usually unchanged.

For the memory-related interferences, when memory interference is injected into the cluster, we can even detect a relatively lower CPU usage than other normal nodes. Considering this, the task numbers (factor a) and task duration (factor b) are also added to determine such root causes with certain weights. Moreover, the memory interference will impact memory usage, and the factor d should be considered for this root cause detection.

To determine disk interferences, we introduce the factor g to measure the degree of disk interference. The task set scheduled at the beginning of each stage could be affected by disk I/O. Therefore, these initial tasks on disk I/O interfered nodes behave differently from other nodes' initial tasks beginning tasks (factor g), CPU will become busy, and memory usage is different with other nodes'. Therefore, The memory changing rate (factor c) and CPU Occupation (factor d) are also used to determine such root causes.

After deciding the combination of factors for each root cause, we give them weights to determine

root causes accurately as Eq. (3.11) shows. Here, all weights are between 0 and 1, and the sum of them for each root cause is 1. To decide the values of weights, we use classical liner regression on training sets that we obtained from experiments on WordCount, Kmeans, and PageRank, which are discussed in more details in Section 4.5.

$$\begin{aligned} CPU &= 0.3 * a + 0.3 * b + 0.2 * c + 0.2 * d \\ Memory &= 0.25 * a + 0.25 * b + 0.5 * d \\ Network &= 0.1 * b + 0.4 * e + 0.5 * f \\ Disk &= 0.2 * c + 0.2 * d + 0.6 * g \end{aligned} \tag{3.11}$$

Then, Eq. (3.12) is proposed to calculate the final probability that the abnormal belongs to each of the root causes.

$$probability = 1 - \frac{1}{factor} \tag{3.12}$$

Table 3.2: Factor for each root causes

Factor type		CPU	Mem	Network	Disk
a	DAR	$\surd$	$\surd$		
b	DAD	$\surd$	$\surd$	$\surd$	
c	DCO	$\surd$			$\surd$
d	MCR	$\surd$	$\surd$		$\surd$
e	DTD			$\surd$	
f	DMC			$\surd$	
g	DLD				$\surd$

3.3 Experiments

In this section, we present the experimental results on our abnormal detection and the root cause analysis in three Spark applications, *e.g.*, WordCount, Kmeans, and PageRank which are provided by sparkbench [38].

3.3.1 Experimental Setup

To evaluate the performance of our proposed approach, we build an Apache Spark Standalone Cluster with four compute nodes, in which each compute node has a hardware configuration with Intel Xeon CPU E5-2620 v3 @ 2.40GHz, 16GB main memory, 1 Gbps Ethernet, and CentOS 6.6 with kernel 2.6. Apache Spark is v2.0.2.

3.3.2 Interference Injection

1. *CPU*: We spawn a bunch of processes to compete with Apache Spark jobs for computing resources, which triggers straggler problems in consequence of limited CPU resource.

2. *Memory*: We run a program that requests a significant amount of memory to compete with Apache Spark jobs. Thus, Garbage Collection will be frequently invoked to reclaim free space.

3. *Disk*: We simulate disk I/O contention using "dd" command to conduct massive disk I/O operations to compete with Apache Spark jobs.

4. *Network*: We simulate a scenario where network latency has a great impact on Spark. Specifically, we use "tc" command to limit bandwidth between two computing nodes.

3.3.3 Experimental Result Analysis & Evaluation

Table 3.3: Root causes diagnosis result

Benchmark	Interference	CPU	Memory	Network	Disk
Wordcount	CPU	**86.5**	35.0	20.0	60.0
	Memory	61.2	**62.6**	20.4	36.0
	Network	51.5	32.5	**85.0**	32.4
	Disk	60.2	40.5	26.2	**82.5**
	Normal	8.5	3.5	5.2	10.3
Kmeans	CPU	**86.0**	53.1	24.5	42.3
	Memory	**60.5**	53.5	35.6	30.5
	Network	43.5	35.2	**87.2**	42.5
	Disk	76.5	53.2	46.2	**82.3**
	Normal	8.6	2.3	3.6	9.6
PageRank	CPU	**83.2**	43.3	24.3	52.5
	Memory	65.4	**67.6**	26.5	45.0
	Network	53.5	46.8	**85.8**	51.0
	Disk	60.3	53.6	25.5	**75.6**
	Normal	9.1	4.5	3.6	10.2

We conduct experiments on three benchmarks, WordCount in Spark package, Kmeans and Page Rank in SparkBench [38]. We run each of the benchmarks 20 times with simulated interference injection.

Table 3.3 summarizes the probability results of our root cause detection approach. For the first step, totally 320 abnormal cases are created, out of which 38 are detected as normal (accuracy: 88.125%). Among these mis-classified cases, 29 are from memory fault injection and the rest 9 are from disk IO. Meanwhile, additional 60 normal cases are also put into our approach for root cause detection, and no one is reported as abnormal. We also check the normal cases' abnormal factors to demonstrate the effectiveness of our approach. In all three benchmarks, the impact of CPU interference is significant, and tasks under CPU interference can be detected as abnormal with high probability. For memory interference, its probability is not significant because memory

interference has less direct effect on Spark tasks, not like root causes. Injecting significant memory interference into one node will cause the whole application crash because the executers of Spark will fail if without enough memory. For network interference, the results show that the proposed approach gives a high probability. Lastly, disk interference shows a high probability in disk root causes. Worth mentioning here, for all different root causes, the detected probability of CPU are always high, because all root causes will eventually affect the efficiency of CPU.

3.3.4 Discussion

Our approach is only tested on clusters with injecting interference on a single node. In order to show considerable effect, the interference will last a while. Additionally, as our approach is based on the task analysis inside each stage, it requires the target application with a certain amount of task partitions for each stage. Furthermore, our approach would be less suitable for analyzing user's log with different Garbage Collectors such as G1, CMS, and the new version of Spark log with different Spark schedulers.

3.4 Conclusion

This chapter proposes a novel statistical-based approach for Spark log analysis, and it identifies abnormal tasks by combining both Spark log and Spark GC log, and then analyze the root causes by weighted factors without using additional system monitoring information. Different with other Spark related methods, our approach is a pure off-line method and only leverage Spark log to analyze abnormal tasks. Furthermore, we prefer using probabilistic output to determine the degree and category of abnormality, rather than considering the problem of classifications of positive and negative samples that CART did. Our approach achieves a reasonable probabilistic results than the

speculated straggler detection in Spark. Moreover, our approach can also identify the root causes of abnormal tasks with probability. Experimental results demonstrate that the proposed approach can accurately locate abnormals and find their root causes in different applications.

CHAPTER 4: GRNN-BASED NEURAL NETWORK APPROACH

4.1 Introduction

This chapter extends our previous work [44], which presents a statistical rule-based approach for log analysis and offers a reasonable result to explain its root causes probabilities [46]. However, it can not give a satisfying result with higher precision for its classifying. Since the relationship between factors is not simply linearly correlated, and we also changed old factor MCR to a new factor MCS with AUC calculation instead of gradients calculation and add it to our factor sets. From this point, a GRNN-based approach is proposed for root cause analysis to consider non-linearly correlated relationship of new factor set, and avoid human ad-hoc choosing and classification.

In this chapter, we leverage General Regression Neural Network (GRNN) to identify root causes for abnormal tasks. The likelihoods of reported root causes are presented to users according to the weighted factors by GRNN. We named our detection tool as LADRA, which means Log-based Abnormal Detection and Root causes analysis. LADRA is an off-line tool that can accurately analyze abnormality without extra monitoring overhead. Four potential root causes, *e.g.*, CPU, memory, network, and disk I/O, are considered. We have tested LADRA atop of three Spark benchmarks by injecting some aforementioned root causes. Experimental results show that our proposed approach is more accurate in the root cause analysis than other existing methods.

GRNN is a simple and efficient network with fast computing speed, because GRNN's transfer function (pattern layer) is a kind of Gaussian function, and it could achieve local approximation with fast speed without any backpropagation training operations. Due to the fact that classical neural networks, especially deep neural networks, require much more effort to tune hyperparameters, which has been proved to be not proper to fit small datasets, just like our Spark log. Hence,

we choose GRNN in our design. Thanks to its flexible structure, which can automatically set the number of nerve cells in the pattern layer.

In brief, the BP (Back Propagation) based deep learning algorithms may be vulnerable to the over-fitting problem especially when the dataset is small, which is just the characteristic of our dataset. Traditional data fitting algorithms usually assume that the data obey a certain distribution in advance, which can drastically affect the final result. As a non-parameter neural network model for data fitting, with its high efficiency and accuracy, GRNN is fully capable of dealing with our current problem. In addition, the experimental results demonstrate the effectiveness of GRNN compared with other attempts we have tried.

As a non-parameter neural network model for data fitting, with its high efficiency and accuracy, GRNN is fully capable of dealing with our current problem. In addition, the experimental results demonstrate the effectiveness of GRNN compared with other attempts we have tried. A representation of the GRNN architecture for our implementation of root cause identification is shown in Figure 4.6. Our model consists of four layers: input layer, pattern layer, summation layer, and output layer. According to our data structure, the input layer consists of 7 neurons, which indicates the dimension of our extracted input feature vector (x_a, x_b... x_g). The pattern layer is a fully connected layer, which consists of neurons with the same size as input data, and followed by the summation layer. At the end, the output layer of GRNN gives a prediction result on the probability for each root cause. We use softmax function to convert the output into a normalized one for more intuitive comparison.

This chapter proposes a new neural network-based model to automatically calculate the probability of each root cause. We use a one-pass training neural network, GRNN, to create a smooth transition and more accurate results.

Although Spark logs are informative, they lack direct information about the root cause of ab-

normal tasks. Thus, simple keyword-based log search is ineffective for diagnosing the abnormal tasks, which motivates us to design an automatic approach to help users detect abnormal tasks and analyze their root causes. An overview of our tool is depicted in Figure 4.1, which contains five primary components: log preprocessing, feature extraction, abnormal task detection, factor extraction, and root cause analysis.

1. *Log preprocessing*: Spark log contains a large amount of information. In order to extract useful information for analysis, we first collect all Spark logs, including execution logs and JVM GC logs, from the driver node and all worker nodes. Then, we use a parser to eliminate noisy and trivial logs, and convert them into structured data.

2. *Feature extraction*: Based on the Spark scheduling and abnormal task occurring conditions, we quantify the data locality feature with a binary number format. Then, we screen structured logs and select three kinds of feature datasets: execution-related, memory-related, and system-related. Finally, we store them into two numerical matrices: execution log matrix and GC matrix.

3. *Abnormal detection*: We implement a statistical abnormal detection algorithm to detect where and when the abnormal tasks happen based on the analysis of execution-related feature sets. This detection method determines the threshold by calculating the standard deviation of task duration and uses it to detect abnormal tasks in each stage from Spark logs, which is introduced in Section 4.2.

4. *Abnormal factor extraction*: According to our empirical case study, we combine special features to synthesize two kinds of factors, the speed factor and the degree factor, which describe the status of each node in the whole cluster. Section 4.3 introduced these factors used by our root cause analysis method.

5. *Root cause analysis*: We propose a General Regression Neural Network (GRNN) based approach for our root-cause analysis, in which probability results can be calculated more accurately than our previous statistical work. Our experiments show that the GRNN-based approach has more accurate results than existing approaches, which are introduced in detail in Section 4.4.

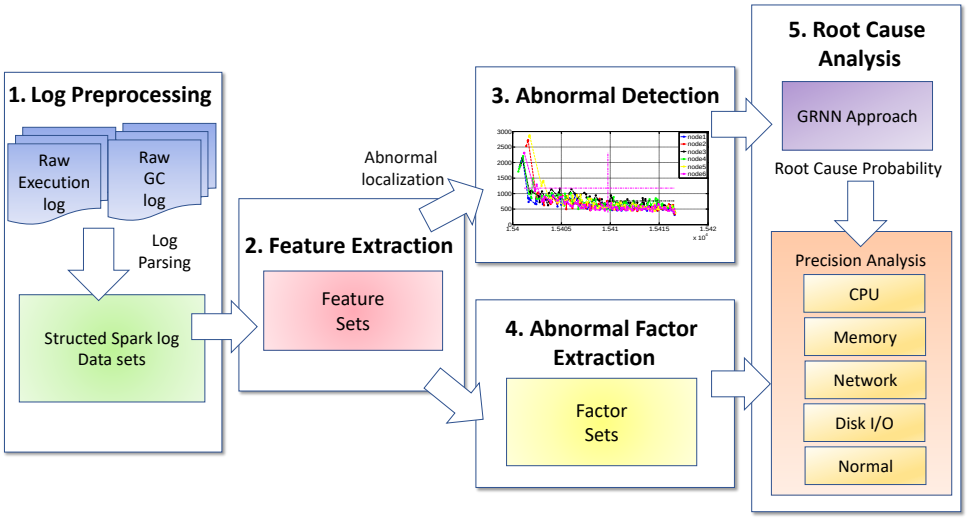

Figure 4.1: The workflow of LADRA

4.2 Log Feature Extraction and Abnormal Task detection

4.2.1 Log Feature Extraction

When an abnormal task happens, it usually does not cast any warnings or error messages. As Spark does not directly reveal any information about abnormal tasks, it is a very challenging problem to detect these problems. Our approach starts from understanding the Spark scheduling strategy, then extracts features associated with CPU, memory, network, and disk I/O to build a feature matrix,

which reflects the whole cluster's status. These features can be classified into three categories: execution-related, memory-related, and system-related, as shown in Table 4.4.

The execution-related features are extracted from Spark execution logs, including (1) the ID number of each task, stage, executor, job, and host, (2) the duration of each task, stage, and job, (3) the whole application execution time, (4) the timestamp for each event, and (5) data locality. Spark GC logs represent JVM memory usage of each executor in workers, from which we can extract memory-related features such as heap usage, young space usage before GC, young space usage after GC. In addition, system related features can be also extracted from GC logs, such as real time, system time, and user time.

Table 4.1: Extracted features for abnormal task detection

Feature Category	Feature Name		
Execution related	Task ID	Job ID	Task duration
	Stage ID	Job duration	Data locality
	Host ID	Stage duration	Timestamp
	Executor ID	Application execution time	
Memory related	GC time	After young GC	After Heap GC
	Full GC time	Before young GC	Before Heap GC
	Heap space	GC category	
System related	Real time	CPU time	User time

4.3 Factor Extraction

To look for the root causes of abnormal tasks, we introduce abnormal factors, which are the synthesis of features based on the empirical study on the 22 features in Spark log matrix and GC matrix. Those factors are normalized features that present status change of the whole cluster, not only for assessing individual components, such as task and stage, but also a series of abnormal tasks, which may be generated by continuous interference affecting the cluster.

In normal cases, each factor should be close to 1; otherwise, it implies an abnormal case. In our factors' definition, j denotes the jth node, J presents a set of nodes; i indicates the index of stage, I is a set of stages; k denotes a task, K is a task set; n stands for a GC record, N is a GC record set. All factors used to determine root causes are listed as below.

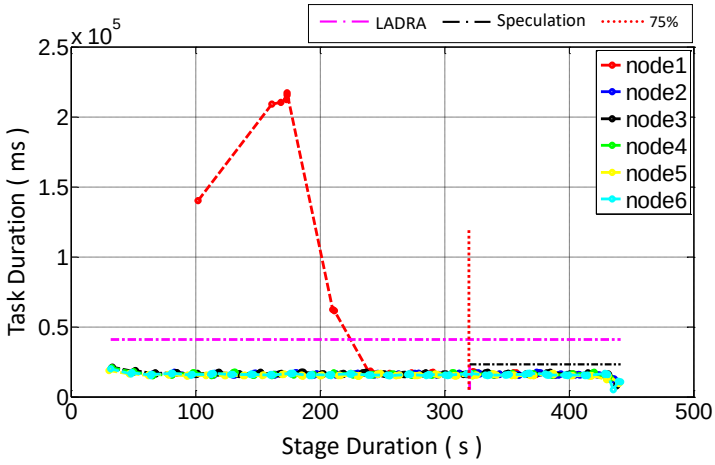

Figure 4.2: Task duration variation in CPU interference injected after Sorting application has been submitted for 60s, and continuously impacts for 120s.

Degree of Abnormal Ratio (DAR) describes the degree of imbalanced scheduling of victim nodes, due to the fact that the victim nodes will be scheduled with fewer tasks than other normal nodes. For example, as shown in Figure 4.2, CPU interference can cause fewer tasks (red dots) to be scheduled at a victim node (node1) than normal nodes. Eq. (4.1) illustrates the degree of abnormal ratio in a certain stage. Therefore, the factor DAR implies that the number of tasks in intra-node on a certain stage can be used for abnormal detection.

$$DAR = \frac{\frac{1}{J-1}((\sum_{j=1}^{J} k_j) - k_{j'})}{k_{j'}} \tag{4.1}$$

where k_j denotes the number of tasks on node j, and J is the total number of nodes in the cluster.

Here, we assume that node j' is abnormal.

Degree of Abnormal Duration (DAD) is used to measure the average task duration, as the abnormal nodes often record longer task duration.

$$DAD = \frac{avg\_node_{j'}}{\frac{1}{J-1}((\sum_{j=1}^{J} avg\_node_j) - avg\_node_{j'})} \tag{4.2}$$

where avg_node_j is defined as:

$$avg\_node_j = \frac{1}{K_j}(\sum_{k=1}^{K_j} T\_task_{i,j,k}) \tag{4.3}$$

Degree of CPU Occupation (DCO) describes the degree of CPU occupation by calculating the ratio between the wall-clock time and the real CPU time. In the normal multiple-core environment, "realTime" is often less than "sysTime+userTime", because GC is usually invoked in a multi-threading way. However, if the "realTime" is bigger than "sysTime+userTime", it may indicate that the system is quite busy due to CPU or disk I/O contentions. We choose a max value across nodes as the final factor.

$$DCO = max(\underset{j \in J}{avg}(\frac{realTime_{i,j}}{sysTime_{i,j} + userTime_{i,j}})) \tag{4.4}$$

Memory Change Speed (MCS) indicates the speed of memory usage change according to GC curve. Due to the fact that under CPU, memory, and disk I/O interference, the victim node's GC curve will vary slower than the normal nodes' GC curve, as shown in Figure 4.3. $start_a$ and $stable_a$ are the points of the start position (the corresponding memory usage at abnormal starting time) and the stable memory usage position, respectively. $start_b$ and $stable_a$ are the start and

end positions of abnormal memory, respectively, which are obtained by analyzing logs introduced before. The intuition is that the interfered node gradually uses less memory than normal nodes under interference, as shown in Figure 4.3. Hence, we use the area under GC curve a in the whole cluster ($start_a$ of normal node) to calculate this factor, as shown in Eq. (4.5).

$$MCS = \frac{\int_{start}^{stable_a} f(x_a)dx_a}{\int_{start}^{stable_b} f(x_b)dx_b} \tag{4.5}$$

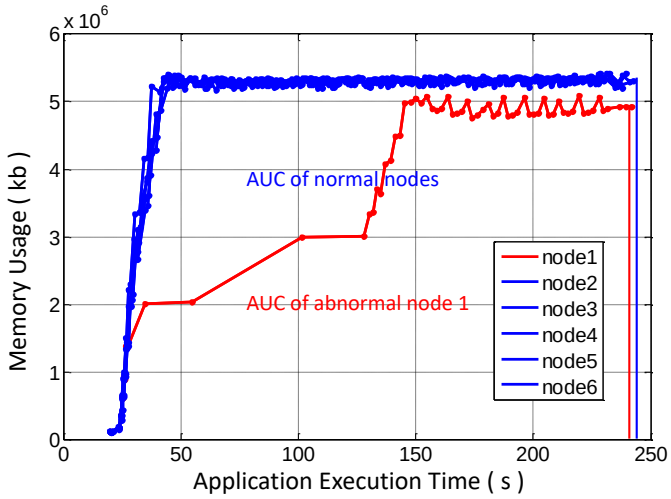

Figure 4.3: Memory usage variation in CPU interference injected after WordCount application has been submitted for 20s, and continuously impacts for 120s.

Abnormal Recovery Speed (ARS) measures the speed of abnormal task's recovery. Since one Spark node often accesses data from other nodes, it can leads to network interference propagation. It is both inter-node and intra-node problem. We can detect network interference happening inside cluster, as shown in Figure 4.4, which is the location of our detected interference and shows that task duration will be affected by delayed data transmission. We leverage Eq. (4.6) to calculate this factor, where abn_prob_j indicates the ratio of the abnormals that we detect for each node j inside that area. The reason that we use the product of abnormal ratio other than the sum of them

is that only when all nodes are with a portion of abnormal, we identify them with a potential of network interference; if their sum is used, we cannot detect this joint probability. Meanwhile, the exponential is to make sure this factor is no less than 1. Hence, the phenomenon of error propagation will be detected and quantified by calculating this factor.

$$ARS = \exp(J * \prod_{j=1}^{J} abn\_prob_j) \tag{4.6}$$

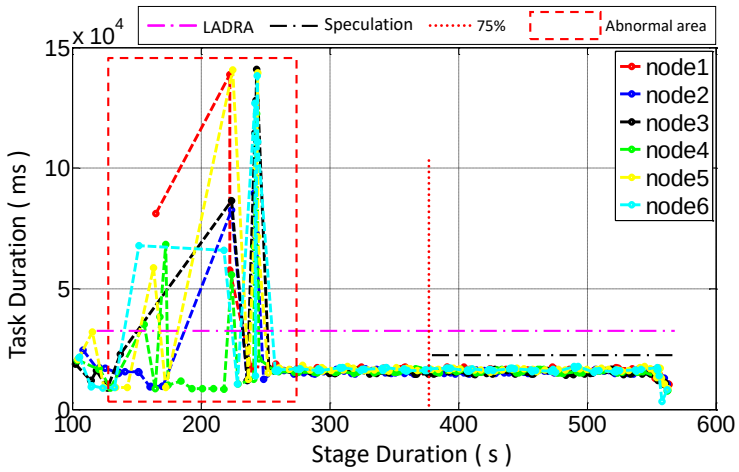

Figure 4.4: Task duration variation in Network interference injected after WordCount has been executed for 100s, and continuously impacts for 160s.

Degree of Memory Change (DMC) describes how much of memory usage changed during the execution in each node. In fact, when network bandwidth is limited, or the network speed slows down, the victim node gets affected by that interference, and tasks will wait for their data transformation from other nodes. Hence, the tasks will pause or work very slowly, and data transfer rate becomes low, as shown in Figure 4.5. We leverage Eq. (4.7) to find the longest horizontal line that presents the conditions under which tasks' progress become tardy (`e.g.`, CPU is relatively idle

and memory remains the same). In Eq. (4.7), $m_{j,n}$ indicates the gradient of memory changing in the nth task on node j. First, the max value of gradient is calculated for each GC point, denoted as m. Second, we make a trade-off between its gradient and the corresponding horizontal length to identify the longest horizontal line in each node. Then, to determine a relative value that presents the degree of abnormal out of normal, we finally compare the $\max$ and $\min$ among nodes with their $\max$ "horizontal factor" ($e^{-|m_{j,n}|} * (x_{j,n} - x_{j,n-1})$), where e is to ensure that the whole factor of b not less than 1.

$$DMC = \frac{\underset{j \in J}{max}\{\underset{n \in N}{max}[e^{-|m_{j,n}|} * (x_{j,n} - x_{j,n-1})]\}}{\underset{j \in J}{min}\{\underset{n \in N}{max}[e^{-|m_{j,n}|} * (x_{j,n} - x_{j,n-1})]\}} \tag{4.7}$$

where $m_{j,n} = \frac{y_{j,n} - y_{j,n-1}}{x_{j,n} - x_{j,n-1}}$.

Degree of Loading Delay (DLD) measures how much difference of loading duration on cluster nodes. Note that the initial task at the beginning of each stage always has a higher overhead to load data compared with the rest tasks. Similar to the factor DMC, instead of taking all tasks inside the detected stage into consideration, here, the first task of each node is used to replace the "avg_node_j".

Instead of taking all the tasks inside the detected stage into consideration, here, the first task of each node is used to replace the "avg_node_j" in Eq. (4.2). Formally, the equation is modified as Eq. (4.8) shows.

$$DLD = \frac{T\_task_{i,j',1}}{\underset{j \in J}{avg}(T\_task_{i,j,1})} where,\ j' \notin J \tag{4.8}$$

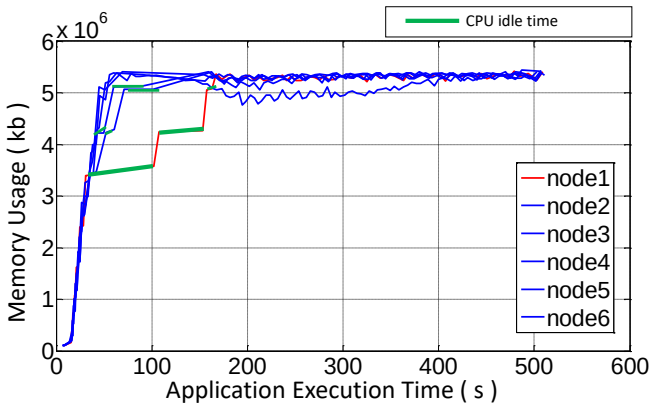

Figure 4.5: Memory usage variation in Network interference injected after Wordcount has been executed for 30s, and continuously impacts for 160s.

4.4 Root Cause Analysis

The statistical rule based approach offers a reasonable result to explain its root causes probabilities. However it can not give a satisfied result with higher precision for its classifying. Since the relationship between factors is not simply linearly correlated, and we also changed old factor MCR to a new factor MCS with AUC calculation instead of gradients calculation and add it to our factor sets. From this point, a GRNN-based approach is proposed for root cause analysis to consider non-linearly correlated relationship of new factor set, and avoid human ad-hoc choosing and classification.

4.4.1 GRNN Approach

We propose a new neural network based model to automatically calculate the probability of each root cause. We use a one-pass training neural network, GRNN, to create a smooth transition and more accurate results.

GRNN is a simple and efficient network with fast computing speed, because GRNNs transfer function (pattern layer) is a kind of Gaussian function, and it could achieve local approximation with fast speed without any back propagation training operations. As due to the fact that classical neural networks, especially deep neural networks, require much more efforts to tune hyper-parameters, which has been proved to be not proper to fit small datasets, just like our Spark log. Hence, we choose GRNN in our design. Thanks to its flexible structure, which can automatically set the number of nerve cells in the pattern layer. In brief, the BP (Back Propagation) based deep learning algorithms may be vulnerable to the over-fitting problem especially when the dataset is small, which is just the characteristic of our dataset. Traditional data fitting algorithms usually assumes that the data obey a certain distribution in advance, which can drastically affect the final result. As a non-parameter neural network model for data fitting, with its high efficiency and accuracy, GRNN is fully capable of dealing with our current problem. In addition, the experimental results demonstrate the effectiveness of GRNN compared with other attempts we have tried.

As a non-parameter neural network model for data fitting, with its high efficiency and accuracy, GRNN is fully capable of dealing with our current problem. In addition, the experimental results demonstrate the effectiveness of GRNN compared with other attempts we have tried. A representation of the GRNN architecture for our implementation of root cause identification is shown in Figure 4.6. Our model consists of four layers: input layer, pattern layer, summation layer, and output layer. According to our data structure, the input layer consists of 7 neurons, which indicates the dimension of our extracted input feature vector (x_a,x_b...x_g). The pattern layer is a fully connected

layer, which consists of neurons with the same size as input data, and followed by the summation layer. At the end, the output layer of GRNN gives a prediction result on the probability for each root cause. We use softmax function to convert the output into a normalized one for more intuitive comparison.

The transfer function F_i in pattern layer is defined in (4.9), $\mathbf{X}$ denotes the input data, σ represents as a smooth parameter, which is set to 0.5 according to our experimental attempts. The hyper-parameter of σ is used to control the smoothness of the model. When the value is relatively large, it is equivalent to increasing the variance in the Gaussian density distribution, which makes the transition between different categories smoother. While the problem is that the classification boundary will be blurred. Conversely, when a smaller value is assigned to this hyper-parameter, the ability to fit real data of the model will be stronger but the generalization turns out to be relatively weak. In the following, summation layer is added, which contains two kinds of neurons: S-summation neuron (S) and D-summation neuron (SD), as defined in (4.10), respectively. SD neurons are used to calculate the arithmetic summation of pattern layer's output. The remaining S neurons weight summation for the output of pattern layer. The i denotes ith number of input data, j denotes the jth dimension of output, and S_j denotes the j^{th} S neuron output. Then, the w denotes weight in hidden layer. The label (output layer) here is a 5-dimension one-hot vector with one indicating normal log and the rest four are injections. y indicates y_j indicates the j^{th} output item the output as defined in (4.11). Due to probability representation of root cause, after the output layer of GRNN, we add a softmax layer to convert the sum of 5-dimensional output to be 1.

$$F_i = exp\,(\frac{-(\mathbf{X}-\mathbf{X_i})^{\mathbf{T}}(\mathbf{X}-\mathbf{X_i})}{2\sigma^2}), where\ \mathbf{X} = [\mathbf{x_a}, \mathbf{x_b}...\mathbf{x_g}]^{\mathbf{T}} \tag{4.9}$$

$$Summations \begin{cases} SD = \sum_{i=1}^{n}(F_i), & \text{where } i = 1:n \\ S_j = \sum_{i=1}^{n}(w_{ij}F_i), & \text{where } j = 1,2,3,4,5 \end{cases} \tag{4.10}$$

where n is equal to input data set size.

$$y_j = \frac{S_j}{SD}, \text{where } j = 1,2,3,4,5 \tag{4.11}$$

To sum up, GRNN can select a dominant weight for each of our factors, and provide the root cause probability results with high accuracy.

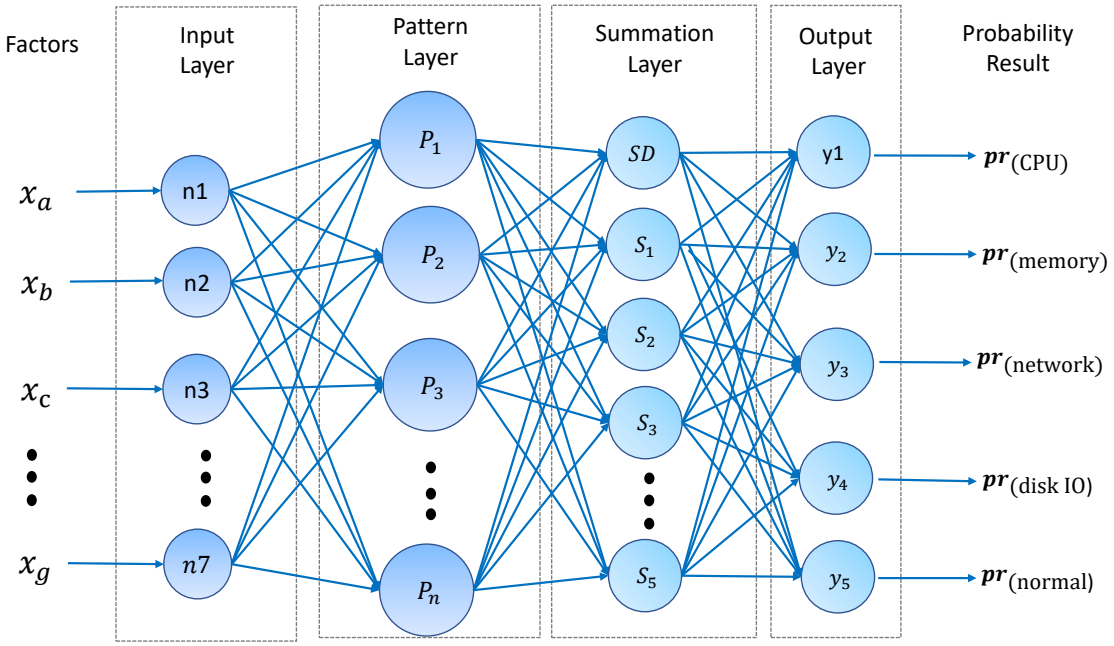

Figure 4.6: The architecture of our GRNN-based model for root-cause analysis.

4.5 Experiments

We evaluate LADRA on four widely used benchmarks and focus on the following two questions: (1) Can the abnormal tasks be detected? (2) What accuracy can LADRA's root cause analysis achieve? In the experiment, we conduct a series of interference injections to simulate various scenarios that lead to abnormal tasks.

4.5.1 Setup

Table 4.2: Related factors for each root cause

Factor	CPU	Mem	Network	Disk
DAR	√	√		
DAD	√	√	√	
DCO	√			√
MCS	√	√		√
ARS			√	
DMC			√	
DLD				√

Table 4.3: Benchmark resource intensity

	CPU	Memory	Network	Disk I/O
WordCount	√		√	√
Sorting	√		√	√
K-Means	√		√	
PageRank	√	√		

Clusters: We set up an Apache Spark standalone cluster with one master node (labeled by m1) and six slave nodes (labeled by n1,n2,n3,n4,n5,n6) based on Amazon EC2 cloud resource. Each node is configured with type of "r3.xlarge" (24 virtual cores and 30GB of memory) and Ubuntu 16.04.9.

We conduct a bunch of experiments atop of Apache Spark 2.2.0 with JDK 1.8.0, Scala-2.11.11, and Hadoop-2.7.4 packages. Given that an AWS instance is configured with EBS by default, it is difficult for us to inject disk I/O interference. Hence, we set up a 90G ephemeral disk for each instance and deploy a HDFS to store data.

Workload: In fact, some Spark applications may consume resources more intensively. According to previous studies on Spark performance [54], we choose four benchmarks built on Hibench [28] and one real-world CPS application in our experiments: WordCount, Sorting, PageRank, K-means, which cover the domain of statistical batch application, machine learning program, and iterative application. WordCount and Sorting are one-pass programs, K-means and PageRank are iterative programs. We characterize the benchmarks by resource intensive type and program type for underpinning our approach's scalability. The resource intensity of each benchmark is shown in Table 4.3. The characteristics of four benchmarks are listed as follows.

- WordCount is a one-pass program for counting how many times a word appears. We leverage RandomTextWriter in Hibench to generate 80G datasets as our workload and store it in HDFS. It is CPU-bound and disk-bound during map stage, then network-bound during reduce stage.

- Sorting is also a one-pass program that encounters heavy shuffle. The input data is generated by RandomTextWriter in Hibench. Sorting is disk-bound in sampling stage and CPU-bound in map stage, and its reduce stage is network-bound.

- K-means is an iterative clustering machine learning algorithm.The workload is generated by the k-means generator in Hibench, and is composed of 80 million points and 12 columns (dimensions). It is CPU-bound and network-bound during map stage.

- PageRank is an iterative ranking algorithm for graph computing. In order to analyze root

causes of abnormal tasks with PageRank, we use Hibench PageRank as the testing workload, and generate eighty thousand vertices by Hibench's generator as input datasets. It is CPU-bound in each iteration's map stage, and network bound in each reduce stage.

A CPS K-means is a real-world CPS application in civil engineering that we developed before. The workload data size is 18 GB and collected by sensors installed at a classroom building. Those sensors measure real time temperature and humidity from each classroom. The collected data set is leveraged for detecting outlier temperature and humidity. To solve this real-world problem with effective approaches, we implemented a K-means algorithm on Spark for pre-clustering and grouping sensor data into sub-clusters and decide the outliers.

4.5.2 LADRA Interference Framework

In order to induce abnormal tasks in the real execution for experiment, we design an interference framework that can inject four major resource (CPU, memory, disk I/O, and network) interference to mimic various abnormal scenarios. In order to simplify experiment, we apply all interference injection techniques only on node n_1 for all test cases. In addition, for each injection, it will be launched during a time interval of 10 seconds and 60 seconds after the first spark job is initiated, and continue for 120 seconds to 300 seconds. Finally, when a test case is over, we recover all involved computing nodes to normal state by terminating all interference injections. Specifically, the following interference injections are used in our experiments:

- *CPU interference*: CPU Hog is simulated via spawning a bunch of processes at the same time to compete with Apache Spark processes. This injection causes CPU resource contention in consequence of limited CPU resource.

- *Memory interference*: Memory resource scarcity is simulated via running a program that requests a significant amount of memory in a certain time to compete with Apache Spark jobs, then we hold on this certain of memory space for a while. Thus, Garbage Collection will be frequently invoked to reclaim free space.

- *Disk interference*: Disk Hog (contention) is simulated via leveraging "`dd`" command to continuously read data and write them back to the ephemeral disk to compete with Apache Spark jobs. It impacts both write and read speed. After the interference is done, we clear the generated files and system cache space.

- *Network interference*: Network scenario is simulated when network latency has a great impact on Spark. Specifically, we use "`tc`" command to limit bandwidth between two computing nodes with specific duration. In this way, the data transmission rate will be slowed down for a while.

Table 4.4: Extracted features for abnormal task detection

Feature Category	Feature Name		
Execution related	Task ID	Job ID	Task duration
	Stage ID	Job duration	Data locality
	Host ID	Stage duration	Timestamp
	Executor ID	Application execution time	
Memory related	GC time	After young GC	After Heap GC
	Full GC time	Before young GC	Before Heap GC
	Heap space	GC category	
System related	Real time	CPU time	User time

4.5.3 *Abnormal Task Detection*

Table 4.5: LADRA's abnormal task detection compares with Spark speculation's approach in four intensive benchmarks, where TPR = True Positive Rate, FPR = False Positive Rate.

	LADRA		Spark speculation	
	TPR	FPR	TPR	FPR
WordCount	0.96	0.06	0.94	0.8
Sorting	0.96	0.16	0.96	0.7
K-Means	0.7	0.1	0.2	0.7
CPS K-Means	0.7	0.1	0.2	0.7
PageRank	0.6	0.517	0.9	0.48

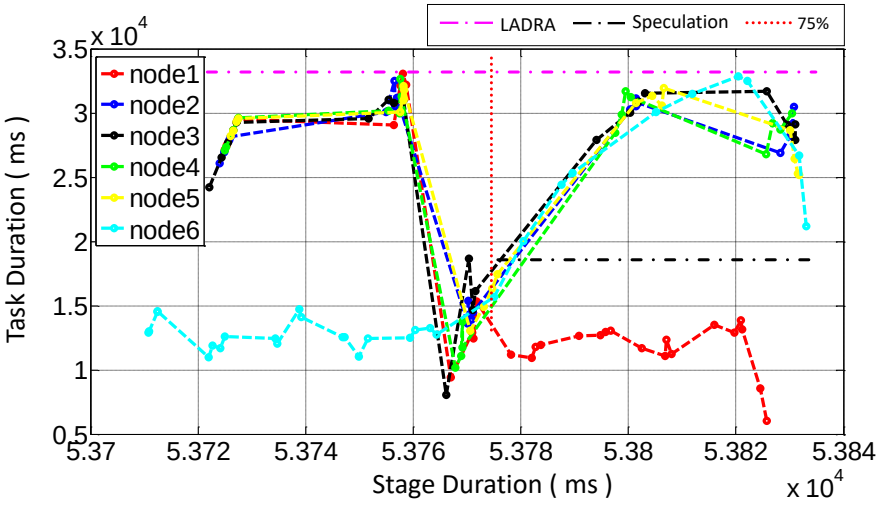

Figure 4.7: Abnormal task detection for K-means without interference injection.

To evaluate LARDA, we compare LADRA's detection with the Spark speculation. Each benchmark is executed 50 times without any interference injection, and 50 times under the circumstances of abnormal tasks. After that, we calculate the True Positive Rate (`TPR`) and False Positive Rate (`FPR`) results by counting the correct rate of each job classification as shown in Eq. (4.12) and Eq. (4.13). The comparison result is shown in Table 4.5.

As a build-in straggler detector, Spark speculation brings False Positive (FP) and True Negative (TN) problems in abnormal task detection. We compare LADRA with Spark speculation in details. For instance, Figure 4.7 shows one stage in a normal K-means execution, x-axis and y-axis present stage duration and task duration, respectively, and no abnormal tasks are detected by LADRA (purple higher horizontal dash dotted line). However, Spark speculation (black lower horizontal dash dotted line) detects stragglers (area above the speculation line and beside red dotted vertical line) after 75% tasks (red dotted vertical line) finish. In this way, Spark speculation may delay the normal execution, as it will reschedule the stragglers to other executors. Moreover, Spark speculation will cause true negative problems as shown in Figure 4.2, because it only checks the 25% slowest tasks. As shown in Table 4.5, LADRA has a better accuracy in abnormal task detection than Spark speculation for all benchmarks. However, LADRA has lower accuracy on K-Means and PageRank than WordCount and Sorting. We find that under normal execution, most tasks in the map stage or sampling stage of K-Means and PageRank have an unexpected longer duration, because these benchmarks have many iteration stages, and tasks in those stages have data skew and cross-rack traffic fetching problems. LADRA cannot detect data skew problem within normal detection results. Too many such kinds of tasks with unexpected duration will cause LADRA to report false positives.

$$TPR = \frac{TP}{(TP + FN)} \tag{4.12}$$

$$FPR = \frac{FP}{(TP + TN)} \tag{4.13}$$

Table 4.6: Root cause analysis result of LADRA's GRNN approach, TPR = True Positive Rate, P = Precision

	WordCount		Sorting		K-Means		CPS K-Means		PageRank	
	TPR	P	TPR	P	TPR	P	TPR	P	TPR	P
CPU	1.000	1.000	1.000	0.940	0.857	0.835	0.866	0.837	0.951	0.826
Disk I/O	0.450	0.420	0.679	0.894	0.423	0.692	0.533	0.666	0.540	0.847
Network	1.000	0.955	1.000	0.853	0.679	0.730	0.700	0.750	0.688	0.564
Normal	0.919	0.837	0.965	0.924	0.733	0.686	0.732	0.632	0.602	0.640

4.5.4 LADRA's Root Cause Analysis Result

To test the accuracy of LADRA's GRNN approach for root cause analysis, we use cross validation strategy with 1/3 for test data and 2/3 for train data each time. Data in normal cases is also used in our training for improving the accuracy. In order to demonstrate the effectiveness of our approach, we run the GRNN 100 times and get the final accuracy result. We calculate the Precision (`P`) and True Positive Rate (`TPR`) for each detected root cause type by Eq. (5.4) and Eq. (4.12).

$$P = \frac{TP}{(TP + FP)} \tag{4.14}$$

We abandon memory root cause analysis in our experiments for three reasons. First, injecting significant memory interference into one node may cause the whole application to crash, as executors of Spark will fail if without enough memory. For instance, injected memory interference in PageRank benchmark not only causes Out-of-Memory (OOM) failures, but also makes executor keep quitting (executors are continuously restarted and fail). Secondly, memory interference does not work for non memory-intensive benchmarks. For instance, WordCount is not a memory-intensive program, and it will not evoke abnormal tasks, even injecting significant memory interference. Thirdly, memory interference could also consume CPU resources, and may mislead

GRNN's classifying.

Table 4.6 summarizes the total P and TPR results of LADRA's root cause analysis for four benchmarks. There are two issues to be noted. (1) LADRA has the highest CPU analysis precision (1.000 in CPU root cause analysis for WordCount) and higher network analysis precision (0.9545 in network root cause analysis for WordCount) results than disk I/O (0.4200 in disk I/O root cause analysis for WordCount) for three reasons. First, all four benchmarks are CPU-intensive, and require large CPU resource for computing (map and sampling stages), and network resource to transfer data (reduce stages). Secondly, abnormal tasks have longer duration after CPU interference is injected, and the impact of network injection is significant (CPU stays idle). Thus, the synthesized factors demonstrate their effectiveness. Thirdly, as disk hog is injected by leveraging a bunch of processes to read and write disk, it consumes not only disk I/O but also a certain of CPU resources. Therefore, disk I/O injections may be wrongly classified into other root causes (, CPU, network, or normal). (2) As shown by Table 4.6, LADRA is more precise on one-pass benchmarks than iterative benchmarks, such as K-means and PageRank. The TPR of k-means and PageRank's disk I/O is lower than the other two benchmarks. It is because that PageRank and k-means are not disk I/O-intensive benchmarks, if the intermediate data is small enough to be caught in memory, it will not use disk space. Therefore, the disk interference does not impact too much for these benchmarks that have small size intermediate data. Moreover, wrong classification of other root causes in k-means and PageRank also impacts LADRA's normal root cause classification, it causes more FP problems, or less TP. So the normal cases in k-means and PageRank also have lower precision and TPR. To compare with the same approach with different data size in different domains, two K-means experiments are performed on our LADRA. One uses a generated dataset by Hibench [28], and the other one uses the dataset produced by a real-world CPS application. We keep all the hyper-parameter setting to be identical. Theoretically, due to the workload data distribution is different, the Spark platform will give a weakly different but similar result since data

itself is not a critical role, as shown in our experiment.

To sum up, LADRA can analyze root causes via Spark log with high precision and TPR for one-pass applications. However, there may be a few of limitations for LADRA to analyze root causes by only using Spark logs. Although Spark logs contain full information, but not so rich as monitoring data.

It might be not possible to analyze all kinds of root causes by only leveraging log files. Some root causes such as code failures, resource usages, and network failures, may rely on monitoring tools. LADRA's goal is to mine useful information and leverage limited log information to analyze resource root causes without extra overhead.

4.6 Conclusion

This chapter presents LADRA, an off-line log-based root cause analysis tool to accurately detect abnormal tasks for big data platforms. LADRA can identify abnormal tasks by analyzing extracted features from Spark logs, which is more accurate than Spark's speculation-based straggler detection method. In addition, LADRA is capable of analyzing the root causes precisely using a GRNN-based method without additional monitoring. The experimental results using realistic benchmarks demonstrate that the proposed approach can accurately locate abnormalities and report their root causes. According to our experiment results, we can effectively detect the resource abnormal and analyze root causes in Spark applications.

CHAPTER 5: CONVOLUTIONAL NEURAL NETWORK FOR DETECTING ANOMALIES

5.1 Introduction

A log entry (log line) is considered *anomaly* if it contains abnormal key words ("error", "warning") or shows significant unexpected order in context, for example, a Spark executor restarts repeatedly before it stops working. Classical anomaly detection has been studied for many years. Various algorithms and methods have been developed, such as basic key word searching, regulation expression matching, traditional statistical and machine learning approaches. It may incorrectly identify the anomalies and report false positives when searching anomalies with key words, or matching with regular expression.

Hence, some techniques such as Support Vector Machine (SVM) and Principal Component Analysis (PCA) [60] are often used to reduce the complexity of feature set to be analyzed and improve accuracy. However, the hidden relationships in extracted feature set are still very difficult to be analyzed by these aforementioned approaches, which often require more sophisticated approaches.

In recent years, deep learning approaches are leveraged in the log analysis domain to improve automation and accuracy. For instance, Long Short Term Memory (LSTM) and Recurrent Neural Network (RNN) are used by [10, 17] to detect anomalies with a high accuracy to avoid ad-hoc feature extraction. Within all deep learning methods, Convolutional Neural Networks (CNNs) could be the most famous and widely used approach, which has obtained great achievements in computer vision. Due to the convolution layers, CNN-based approach can learn the hidden relationships with higher accuracy than other deep learning methods.

This chapter proposes a Convolutional Neural Networks [45] approach for anomaly detection from HDFS logs.

5.2 Methodology

In this section, we present our two-fold method. We first introduce our log processing, and then detail our CNN-based approach and MLP-based approach as the baseline.

5.2.1 Log Processing

The purpose of our log processing is to generate structural input for our CNN model. As the system log consists of multiple identifiers (defined by [61]), an identifier is an object and has a certain execution path. For example, `block_ID` is an identifier token in HDFS log, and `block_a` is an actual identifier and the execution path of `block_a` is a sequence that consists of three related log keys (*e.g.*, `Receiving block`, `PacketResponder for block terminating`, `Deleting block file`). Initially, a log template parser is used to find the frequent log constants, named log key. Then, we use another parser to analyze and filter the raw logs into structured data consisting of log keys (exclude useless information like timestamp of specific logs). Next, we encode each of the parsed log key with a unique number (, HDFS log has 29 log keys mapped to 29 numbers). Specifically, we count how many unique log keys in the whole data sets, and map each unique log key (parsed log entries) into a unique number. Finally, we leverage a session windows [25] to regroup those log keys to different sessions (group). After sorting those log keys (numbers) with execution order, we get a structured sessions. Thus, each session (group) includes one unique identifier and a series of related log keys (numbers), such as a session: `5 5 11 ... 26 26` in HDFS log belongs to one block (an identifier). Considering each vector represents an execution

path which may vary based on environment settings (different orders), also each path may have different lengths. For example, some abnormal blocks in HDFS log will be killed after just being started, so this block only contains few log keys, which has a short length of vector. Hence, we pad 0 at the end of shorter vectors, and clip longer vectors to make each vector in the log files with the same length.

5.2.2 CNN-based Model

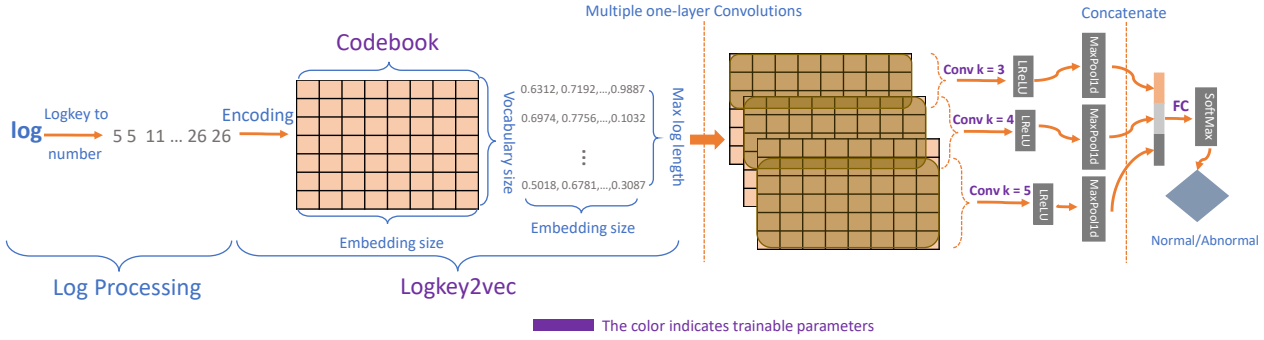

Figure 5.1: Architecture of our CNN-based anomaly detection model.

Neural network is a biologically-inspired approach for pattern recognition [8]. In regular fully connected networks, each neuron is fully connected to all neurons in the previous layer and Back-Propagation [52] is utilized to compute the error gradient [16]. However, it is not scaled well for high-dimension data such as images (, images are of size $32 \times 32 \times 3$ in CIFAR-10 [36]). Inspired by receptive fields of cat's visual cortex [29], Convolutional Neural Network (CNN) has been proposed to capture local semantic information instead of global information and defeat the over-fitting issues in regular neural networks.

Basically, convolution is the core operation applied in the convolutional layers and it extracts

features from local receptive fields on feature maps of previous layer. An activation function (, Sigmoid, ReLU (Rectified Linear Units), Tanh) is performed as a non-linear transformation. Following [33], as shown in Eq. 5.1, the value of a unit at position (m, n) in the j^{th} feature map of the i^{th} layer can be denoted as $v_{ij}^{m,n}$:

$$v_{ij}^{mn} = \sigma\left(b_{ij} + \sum_{N}\sum_{p=0}^{P_{i-1}}\sum_{q=0}^{Q_{i-1}} w_{ij}^{pq} v_{(i-1)N}^{(x+p)(y+q)}\right) \tag{5.1}$$

where b_{ij} denotes a bias function of this feature map, N indexes over the set of feature maps in the $(i-1)^{th}$ layer, P_i is the height of kernel and Q_i is the width of kernel, and w_{ij}^{pq} is the value of parameter.

As mentioned as before, Kim *et al.* [35] first propose a simple and effective CNN model based on `word2vec` [50] and vanilla CNN for sentence classification with static and non-static channels and get preeminent results in natural language processing.

Due to the fact that log file is also one special kind of text, log analysis can also benefit from the advances of NLP techniques. However, log analysis is different from the general NLP. The long span relationship widely exists in nature language context, such as a long sentence with complex structures. But logs only contain small amount of log keys. Moreover, the goal of anomaly detection is to look for unexpected execution path (log key sequences), which is a binary classification, whereas NLP tries to classify sentences into multiple categories. During the experiment result, we found that CNN can achieve better accuracy rather than other approaches for log-based anomaly detection, such as MLP and LSTM.

As shown in Figure 5.1, in the embedding layer, we create a trainable matrix, *e.g.*, 29 $\times$ 128 codebook, to map each log key in a session into a vector. For example, in embedding process, the log key `5` in session group `5 5 11 ... 26 26` will be encoded to `0.6312, 0.7192,`

`...  0.9887`, and the whole session will be encoded as a matrix. We name this embedding process as `logkey2vec`. Different from word embedding that uses word as fine-grained unit such as `word2vec`, each log key will produce log embeddings based on the 29 $\times$ 128 codebook. The `logkey2vec` is a trainable layer optimized with gradient decent during the training of Neural Network. The codebook is used for mapping 1D vector to 2D matrix as CNN input, which is a more comprehensive mapping to enhance the relationships hidden behind logs.

The next part in CNN is convolutional layers, which convolute over the embedded log vectors with three one-layer convolutions (filters) in same time. According to our experimental study, we adopt three convolutional layers in parallel for CNN training after encoding layer, with size of 3 $\times$ 128, 4$\times$128, 5$\times$128, respectively, as shown in Eq.5.1, where $P = 3, 4, 5$, and $Q = 128$. The activation function σ is Leaky Rectified Linear Unit (leaky ReLU or LReLU) shown in Eq. 5.2, due to that leaky ReLU can avoid over-fitting and solve the dead ReLU problem by setting first part of ReLU to non-zero (a small positive gradient). The dead ReLU problem means that some of neurons in the network may never be activated, hence, the parameters will never be updated. The causes of dead ReLU have two aspects. The first one is improper parameter initialization, and the second one is high learning rate setting which may lead to parameter updating too large. After three independent convolutional operations, a max-pooling layer is applied to concatenate output of the convolutional layers. While, for Leaky ReLU, as the gradient in all its domain will not be 0, it will feed back a informative update for each iteration. The max-pooling layer can also reduce over-fitting effectively by filtering out the weak related features, and leaving the strongest related features for next layer. Moreover, a dropout function is applied as a regularization in the second-to-last layer to prevent over-fitting. Finally, a softmax function is added in the output layer. The softmax function is shown in Eq. 5.3. Moreover, the parameter setting detail of CNN base model

for each layer is shown in Table 5.1.

$$\sigma(x)\begin{cases} x & if\ x \geq 0 \\ \\ 0.1x & if\ x < 0 \end{cases} \tag{5.2}$$

where x denotes the input before activator.

$$S_i = \frac{e^{a_i}}{\sum_{k=1}^{T}(e^{a_k})} \tag{5.3}$$

where a_i denotes the ith number of input, i = 1 to 2, T =2 in our implementation.

Table 5.1: network details and specific parameters in our CNN model

Layer	Output
Input: vectorized log, size: 1 x 50	--
Embedding with code book size: 29 x 128	Embedded log matrix size: 50 x 128 x1
Conv 1: [3,128,1,128], strides=[1, 1], padding="VALID" Leaky ReLU, max pool [1,48,1,1], strides=[1, 1]	48 x 1 x 128 1 x 1 x 128
Conv 2: [4,128,1,128], strides=[1, 1], padding="VALID" Leaky ReLU, max pool [1,47,1,1], strides=[1, 1]	47 x 1 x 128 1 x 1 x 128
Conv 3: [5,128,1,128], strides=[1, 1], padding="VALID" Leaky ReLU, max pool [1,46,1,1], strides=[1, 1]	46 x 1 x 128 1 x 1 x 128
Concatenate Conv 1, Conv 2, Conv 3, dropout 0.5	1 x 384
FC: [384,2]	2
softmax	

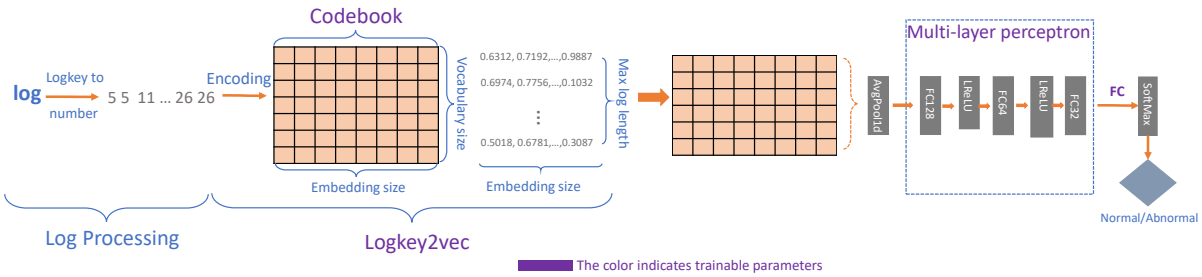

Figure 5.2: Architecture of our MLP-based anomaly detection model.

Table 5.2: Network details and parameters in our MLP model

Layer	Output
Input: vectorized log, size: 1 x 50	--
Embedding with code book size: 29 x 128	Embedded log matrix size: 50 x 128 x1
Avg pool [1,50,1,1], strides=[1, 1], flatten	
Dropout	128
FC: [128*128], leaky ReLU	
Dropout	128
FC: [128*64], leaky ReLU	64
FC: [64*32], leaky ReLU	32
FC: [32*2]	2
softmax	

5.2.3 MLP-based Model

According to our empirical study, parameter tuning is very challenging for LSTM, it is difficult to train such a complicated model because of gradient vanish/exploding issues existing in Recurrent Neural Networks like LSTM. As a result, the accuracy for anomaly detection may decrease. Hence, we decide to use a simple and clear network with easy adjustable parameters as our baseline to

compare with CNN in order to prove its efficacy. Therefore, we design a Multilayer Perceptron (MLP) as our baseline model and also train it on `logkey2vec` of HDFS logs. Before Deep Neural Network (DNN), MLP is one way feed-forward layered network which can be built up with three main layers (input layer, the hidden layers, and the output layer). Basically, the input layer sent weighted inputs to front hidden layer to learn the relationship and sent middle data to next hidden layer. Then, the classification results are sent from last hidden layers to output layer. It consists of three components: input layer, hidden layers and output layer. Each hidden layer is activated with a non-linear function. BP is often utilized to update the weights of MLP. More than one hidden layers are designed to increase/decrease the complication of models. The output layer could be different depending on the objective function.

The workflow of our MLP model for log-based anomaly detection is shown in Figure 5.2. The input embedding stage is the same as CNN's `logkey2vec`, and it encodes vectors using the same codebook. The parameters of MLP model for each layer are shown in Table 5.2, the hidden layers are three fully-connected layers without any convolutional layers, and the number of MLP hidden neurons for each fully connected (FC) layer is 128, 64, and 32, respectively. Following the CNN model, LReLU is also used as MLP's activation function. The output of FC layer is concatenated to a vector by an average-pooling layer.

5.3 Evaluation

In this section, we first introduce the experiment setup, and then evaluate the accuracy of the CNN-based approach for detecting anomalies in HDFS data sets.

5.3.1 Experiment Setup and Dataset

Our CNN-based approach is implemented in TensorFlow [4]. We compare the accuracy of our approach with other deep learning methods in log-based anomaly detection using HDFS log, a widely used benchmark dataset employed by other approaches [61, 17].

The HDFS log is a dataset generated from running over 200 days experiment in Amazon EC2. The data was first published by Xu *et al.* [61], and analyzed by many approaches such as SVM, PCA, logistic, and LSTM based anomaly detection. The raw log file is 1.55 GB and contains 11,197,954 log entries. Moreover, HDFS log records the states of each HDFS block during job execution time, and includes 29 unique log keys. Furthermore, the raw data is always parsed with session windows, and each line consists of unique blockId with related log keys in the parsed format. We leverage the parsed and labeled ground truth data, which is the same as [17]. It contains normal training set (4,855 parsed sessions), normal testing set (553,366 parsed sessions), abnormal training set (1,638 parsed sessions) and abnormal testing set (15,200 parsed sessions).

5.3.2 Results

We first assume that CNN model can achieve a better accuracy which compare to other comparisons. To prove this point, we evaluate CNN with our MLP baseline model and LSTM model on HDFS logs.

Due to the fact that both CNN and MLP are supervised approaches, and both require training before testing, Hence, we first train the CNN model with training data set, and leverage testing set to evaluate our model. For training detail, we use normal training set from [17], and select 1% of abnormal testing data set as abnormal training set. Here, the remained 99% abnormal testing set combines with normal testing set as total testing set. It is the same amount which we described in

above subsection. Finally, we compare to the CNN and MLP training results with LSTM which is presented at [17].

Those models are evaluated by the metrics listed blow: True positive (TP) represents the number of real anomalies that are correctly detected as anomalies by our approach. True negative (TN) represents the normal cases that are correctly identified as normal case. False positive (FP) presents the normal scenarios that are incorrectly identified as anomalies. False negative (FN) represents the abnormal log cases that are identified as normal. Based on the four metrics, we calculate the Precision (P), Recall, and F1-measure for each tested approaches. Precision is calculated by Eq. (5.4), which represents the correctly detected anomalies percentage in reported anomalies. Recall is calculated by Eq. (5.5), which shows the detected true anomalies in all real anomalies. F1-measure is calculated by Eq. (5.6), which represents the harmonic average of the P and recall.

$$P = \frac{TP}{(TP + FP)} \tag{5.4}$$

$$Recall = \frac{TP}{(TP + FN)} \tag{5.5}$$

$$\textit{F1-measure} = \frac{2P \cdot Recall}{(P + Recall)} \tag{5.6}$$

To compare the accuracy of our CNN model with LSTM and our MLP baseline model, we list all the evaluation metrics results in Table 6.1. Our CNN achieves better results on all the metrics than the other two models.

Figure 6.3 compares the accuracy, precision, recall, and F1-measure of our CNN and MLP based approaches in each epoch of the training using HDFS logs. The red line is our CNN-based ap-

proach, and the blue line is the fully connected MLP-based approach without convolution layers. Figure 6.3 (a) shows that CNN has the higher accuracy. Although both CNN and MLP can achieve high accuracy finally, MLP starts with lower accuracy and slowly converges to high accuracy after 85 epochs. Figure 6.3 (b) presents that the precision curves of both models have some fluctuations; however, the curve of CNN model is much more stable than MLP. Moreover, CNN could converge in high precision after few epochs, and MLP converges after 100 epochs. Figure 6.3 (c) shows the recall of both models. The recall value of MLP starts at 0 and converges to 98.7 in 20 epochs, and CNN's recall is around 0.9 at the beginning, which is much higher than MLP's recall. Figure 6.3 (d) shows F1-measure of both models, where CNN could reach to a high accuracy in few epochs, but MLP converges till 90 epochs. All the evaluation metrics show that the MLP model converges slowly and is more time-consuming than the CNN model on the training of HDFS logs.

To evaluate if the embedding layer could impact the accuracy, we design an extra experiment by eliminating the embedding layer inside our MLP model. After MLP model trains with a series of log key vector directly without embedding process, we get the results with accuracy of 0.997, precision of 0.9732, recall of 0.95044 and F1-measure of 0.961726. Compared with the results shown in Table 6.1, it demonstrates that the embedding process could cause big difference in the efficiency MLP for HDFS log classification. It is because that the embedding layer leverages codebook to encode vector into matrix, and this processing could learn comprehensive semantic representation of log.

Table 5.3: The comparison of different models on HDFS log.

Model	Accuracy (%)	Precision	Recall	F1-measure
CNN	99.9±856e-05	97.7±068e-05	99.3±0035	98.5±0014
MLP	99.89±588e-05	98.12±918e-05	98.04±0036	98.08±0018
LSTM [17]	—	95	95	96

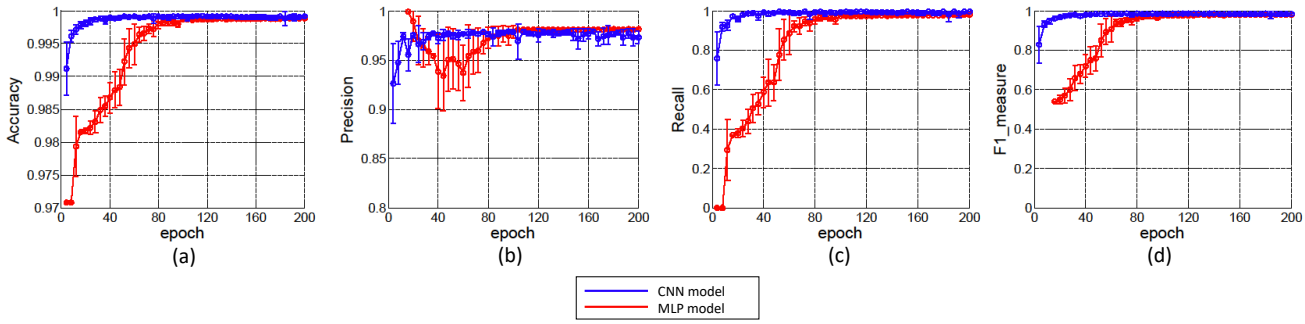

Figure 5.3: Accuracy procedure of CNN-based approach and MLP-based approach on HDFS logs (a)Accuracy. (b) Precision. (c) Recall. (d) F1-measure.

5.4 Discussion

This section discusses potential reasons why our CNN-based approach could achieve better accuracy than MLP, and LSTM. Furthermore, the reasons of embedding layers using is discussed.

5.4.1 CNN vs. MLP

The experiments show that MLP and CNN models have different accuracy because of two reasons. First, due to the fact that after the first embedding layer, the related log keys arer sorted, and our CNN model could use multiple filters to mine more buried relationships which hided among log keys. Secondly, the learning process of weights is a two-dimensional convolution operation, hence it considers the correlation between the horizontal embedding, and the longitudinal entries in logs. Although the MLP approach is quite effective, semantic information can not be leveraged by training stage.

5.4.2 CNN vs. LSTM

LSTM has more advantages than other neural network methods for solving NLP sequence classification problems because LSTM has one kind of units called memory cells to store context information. The context stored in the previous cells could be used for next memory cells. LSTM performance will be impacted by three factors when it is employed for anomaly detection. First of all, `word2vec` embedding is required in NLP task for word separation, and the log analysis needs an embedding for log key separation, such as `logkey2vec`. Secondly, log can be considered as one kind of execution flows consisting of many log entries (log keys). Moreover, those log entries could present short time sequence relationship. However, one log entry may have weak relationships with far distanced log entry from it. For example, there are two states in HDFS log, the one is a start state called Receiving block, and the other one is an end state called Deleting block file. Thirdly, due to the fact that the network structure of LSTM is more complex and thus the tuning work for LSTM's parameters is more difficult. Hence, to achieve a good performance is still a challenging task for LSTM.

5.5 Conclusion

This chapter presents a novel Neural Network based approach to detect anomaly from system logs. A CNN-based approach is implemented with different filters for convoluing with embedded log vectors. The width of filter is equal to the length of a group of log entries. A max-overtime pooling is applied for picking up the maximum value. Multiple convolutions layers are employed for computing. Then, we add a fully connected softmax layer to produce the probability distribution results. We also implement a MLP-based model that consists of three hidden layers without any convolutional kernels. Our experimental results demonstrate that the CNN-based method can achieve a higher and faster detection accuracy than MLP and LSTM on big data system logs

(HDFS logs). Moreover, our CNN model is a general method that can parse log directly and does not require any system or application specific knowledge.

CHAPTER 6: DETECTING ANOMALIES WITH ATTENTION-BASED CONVOLUTINGAL NEURAL NETWORK

6.1 Introduction

This chapter extends our previous work [45], which presents a novel approach to detect anomalies from system logs by using vanilla CNN to explore complex latent relationships effectively. In the proposed vanilla CNN model, firstly, we design an embedding method named `logkey2vec` to embed log keys into feature vectors, which are then fed into convolutional layers, where the width of filter equals to the length of embedding. Secondly, we apply a max-overtime pooling layer to select the maximum value of all the features. Finally, a fully connected softmax layer is applied to calculate the probability distribution results. We train the CNN model with labeled HDFS logs.

In this chapter, we implement a novel attention-based CNN, where the attention mechanism is applied to the vanilla CNN to improve the accuracy. Due to the fact that partial important log keys or CNN filter-extracted features may have more relevant impact, our attention mechanism focuses on using those features instead of whole CNN output. For log analysis, the relevant log patterns need more attention than the unimportant log entries. We propose two attention schemes that focus on different features from CNNs output. We compare our attention-based CNN approach with several other deep learning approaches, the attention-based CNN model shows the best performance among comparison methods.

6.2 Attention-based CNN Approaches

Our previous approach detects anomaly by using `logkey2vec` and Convolutional Neural Network, but due to the fact that complex relationships exist between log lines, the attention-based model is able to find more reasonable features. As we stated before, the anomaly detection with system log should learn non ad-hoc features, hence, the most significant extension in this paper is that two proposed attention schemes that can learn the hidden relationships among log lines.

To further improve accuracy for abnormal detection through CNN model, we adopt attention mechanism to filter out less relevant features among inner output of the model.

Our attention model employs three attention schemes, namely *logkey attention*, and *filter attention*. This model is similar to the CNN model but extended with different attention layers to filter more related features by modifying the position of attention mechanism in the whole network structure. All of the attention schemes have the same embedding and convolution parts as our vanilla CNN models.

6.2.1 Logkey Attention Scheme

In our vanilla CNN model, max pooling is added to the output of convolutional layer to obtain the maximum value features. However, the related logkey features directly produced by CNN may be ignored by max-pooling, and those logkeys could bring more accurate results. The logkey attention scheme consists of three attention layers. Let A^{m*n} denote the matrix after convolutional operation, where m and n are the dimensions of A. Formally, given A, we first segment it into a sequence of vectors $a_1, a_2...a_n$, where a_i indicates the i^{th} row of A, our implementation of attention mechanism computes an output vector c given by a weighted scaling of vectors a_i.

The attention input vector a_i and output vector c are presented by:

$$c = \sum_{i=1}^{N} \lambda_i a_i \tag{6.1}$$

The attention weight of each input i is computed by:

$$e_i = w^T a_i \tag{6.2}$$

$$\lambda_i = \frac{exp(e_i)}{\sum_{j=1}^{n} exp(e_j)} \tag{6.3}$$

where w is a learnable parameter using back propagation by comparing the ground truth and the predicted results. Let Λ^{1*n} denote the matrix of attention weights λ, where 1 and n are the dimensions of Λ. Let E^{1*n} denote the set of e, where 1 and n are the dimensions of E.

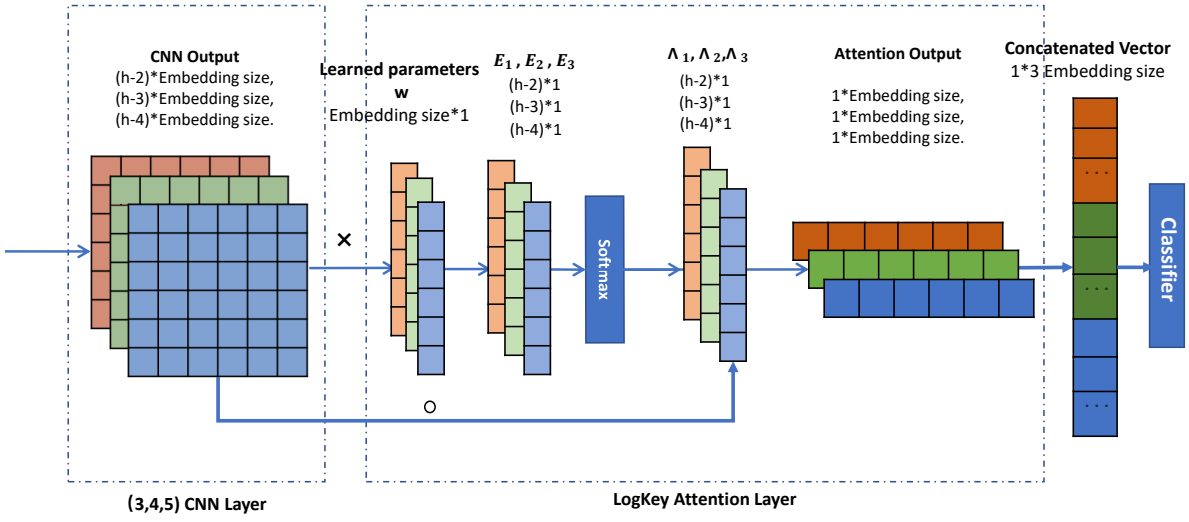

Figure 6.1: Architecture of our logkey attention scheme

6.2.2 *CNN Filter Attention Scheme*

Similar to our first attention mechanism, we further propose the second attention CNN model named CNN Filter attention scheme. The difference from our first model is that we perform attention operation after maxpooling operation other than replacing it, and the workflow of our CNN filter attention is shown in Figure 6.2. Actually, for our previous simple embedding-based CNN model, three independent CNN layers after maxpooling (three vectors with length of 128) are concatenated into a single vector, and then operate attention operation through attentionOutput,attentionOutput2,attention. Typically, for this kind of attention mechanism, we try to find the relative importance among these three extracted features.

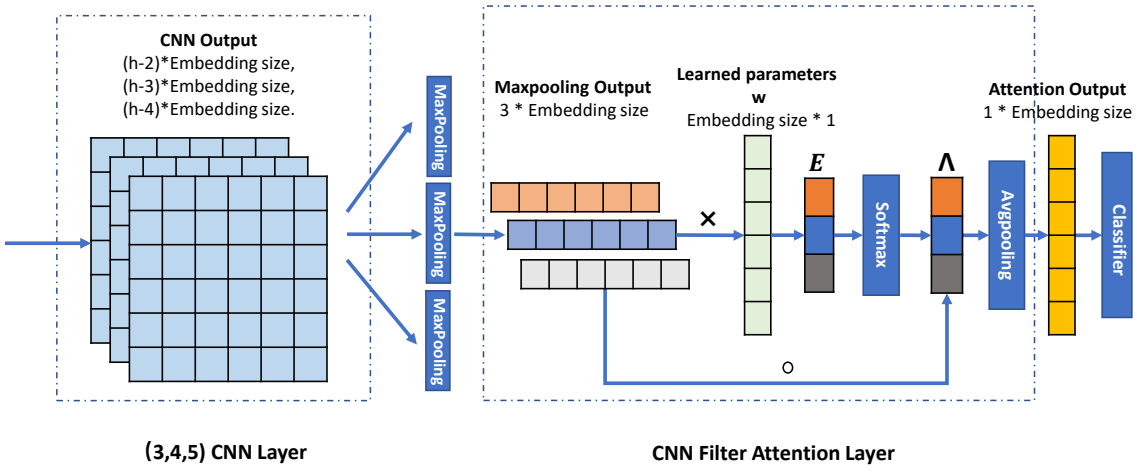

Figure 6.2: Architecture of our CNN filter attention scheme

6.3 Evaluation

In this section, firstly, we show the setup experiment and dataset. Secondly, we evaluate the accuracy of proposed attention-based CNN method for HDFS log anomaly detection.

6.3.1 Experimental Setup and Dataset

We implement our proposed approaches with TensorFlow [4]. The dataset we used for evaluation of the accuracy of proposed attention-based approach is HDFS log, a widely used public dataset [61, 17]. This dataset is first published by Xu *et al.* [61], and it is generated from an over 200 days running experiment on Amazon EC2, which mainly records the states of each HDFS block during job execution time. The size of raw data is 1.55 GB and it contains 11,197,954 log entries, 29 unique log keys. It has been analyzed by few statistical approaches and deep learning method including offline PCA-based method [61], online PCA-based method [60], Invariant Mining-based method [43], and LSTM-based anomaly detection method [17].

The preprocessing of the HDFS raw data is usually parsed with session windows, and each line consists of unique blockId with related log keys in the parsed format. The parsed and labeled ground truth data is leveraged in our experiment as same as [17]. This dataset contains abnormal training set (1,638 parsed sessions), abnormal testing set (15,200 parsed sessions), normal training set (4,855 parsed sessions), and normal testing set (553,366 parsed sessions).

6.3.2 *Results*

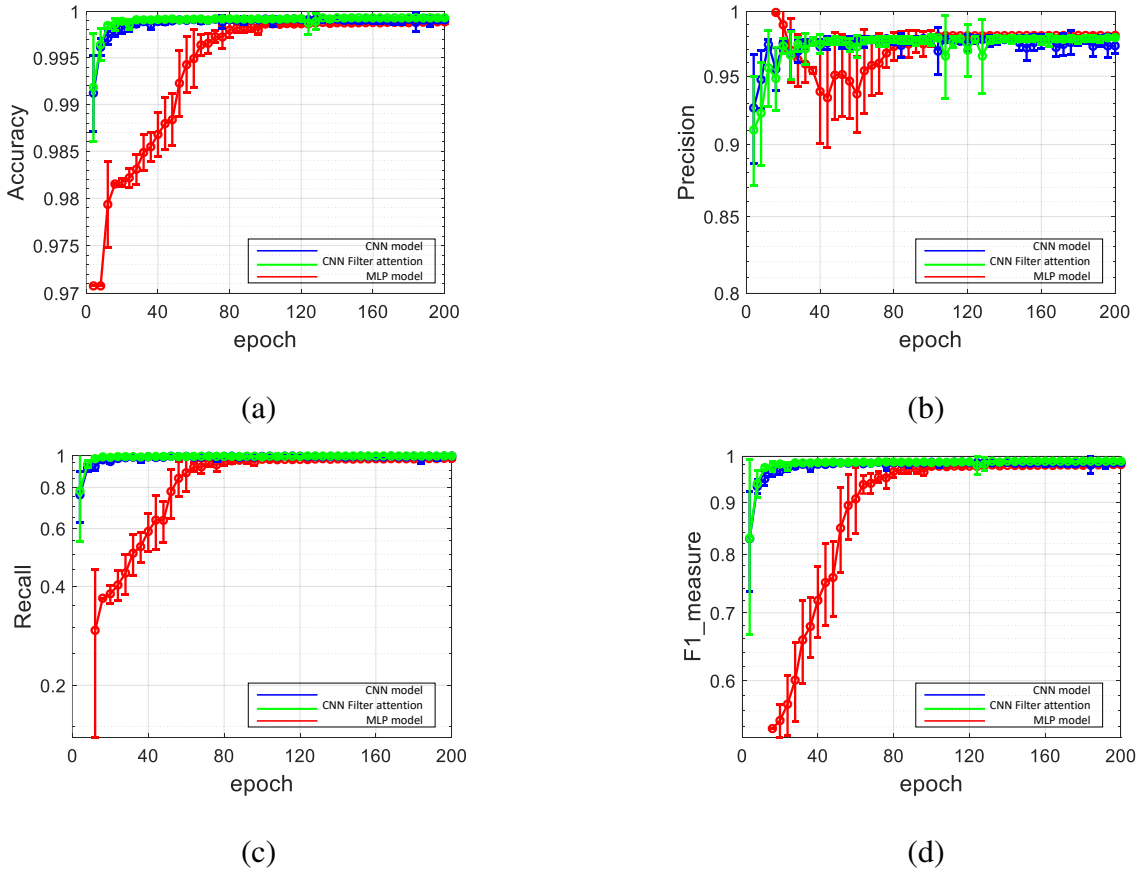

Figure 6.3: Performance Comparison of the attention-based CNN approach, CNN-based approach and MLP-based approach on HDFS logs with respect to (a) accuracy, (b) precision, (c) recall, and (d) F1-measure.

We compare our proposed attention-based models with six representative approaches including three deep learning methods (CNN model, our MLP model [45], LSTM [17]), and three non deep learning approaches (Principal Component Analysis (PCA) [17], Invariant Mining (IM) [17], N-gram [17]). As shown in our previous work [45], a Multilayer Perceptron (MLP) is designed as our baseline model which uses a simple and clear network with easily adjustable parameters to

compare with CNN and LSTM in order to prove its efficacy. The MLP model is also trained on `logkey2vec` of HDFS logs, it uses the same input embedding stage as CNN's and the same codebook for vector encoding.

Table 6.1 shows the evaluation results with respect to "accuracy", "precision (P)", "recall", and "F1-measure (F1)". As shown in Table 6.1, our attention-based CNN model (CNN filter attention scheme) outperforms the LSTM [17] by 2.89% in F1-measure, 4.76% in recall, 3.04% in precision. In addition, it outperforms the best non deep learning approach N-gram [17] by 4.89% in F1-measure, 3.76% in recall, 5.70% in precision.

Figure 6.3 presents the comparison results of the precision, accuracy, recall, and F1-measure of our proposed attention-based CNN model, basic CNN model and MLP-based approach in each epoch of the training using HDFS logs. As the CNN filter attention scheme has the best performance, it is used to compare with others.

Figure 6.3 (a) shows that the attention-based CNN has the best performance in accuracy measurement, and both attention-based and vanilla CNN models achieve good accuracy in a few epochs. Although all three models can achieve high accuracy finally, MLP converges much slower. Figure 6.3 (b) shows that the precision curves of those models have some fluctuations; however, the attention-based and vanilla CNN models are more stable than MLP. Moreover, both CNN-based models converge in high precision after a few epochs, while the attention-based CNN achieves the highest precision. MLP converges after 100 epochs. Figure 6.3 (c) shows the recall measurements of all models. The recall value of MLP converges to 98.04% in 70 epochs. The average recalls of the attention-based and vanilla CNN is around 0.8 at the beginning, which is much higher than MLP. Moreover, the attention-based CNN model (CNN filter attention scheme) gets higher recall than the vanilla CNN. Figure 6.3 (d) shows the F1-measure measurements of all models, where the attention-based and vanilla CNN models reach to high accuracy in the first few epochs, but MLP

converges till 90 epochs. All the evaluation metrics show that the attention-based CNN model (CNN filter attention scheme) achieves better performance than vanilla CNN and MLP, where the MLP model converges slower than the others.

To access whether the embedding layer (`logkey2vec`) can improve the accuracy or not, an additional experiment is designed by excluding the embedding layer in our MLP model. The accuracy, precision, recall, and F-measure without embedding is 99.7%, 97.32%, 95.04%, and 96.17%, respectively. The comparable results are shown in Table 6.1, the embedding process causes a big difference in the performance. Due to the fact that codebook is used by embedding layer to encode vector into matrix, we believe that comprehensive semantic representation among HDFS logs can be learned by embedding processing.

Table 6.1: The comparison of different models on HDFS log.

Approach		acc(%)	P(%)	Recall(%)	F1(%)
BaseLine	LSTM [17]	- -	95	95	96
	PCA [17]	- -	98	67	79
	IM [17]	- -	88	95	91
	N-gram [17]	- -	92	96	94
	MLP [45]	99.89	**98.12**	98.04	98.08
	CNN	99.9	97.7	99.3	98.5
Attention	Logkey	99.8	97.91	99.33	98.62
	CNN Filter	**99.93**	98.04	**99.76**	**98.89**

6.4 Discussion

This section discusses potential reasons why the attention-based CNN model could achieve better accuracy, and two different attention schemes and the significance of `logkey2vec` embedding are analyzed.

6.4.1 Logkey Attention vs. CNN Filter Attention

Since max-pooling layer will ignore some related features, the proposed log key attention is designed to extract logkey related features (codebook) on CNN output directly. In other words, the logkey attention will identify important logkeys in the whole codebook. The CNN filter attention considers which CNN tunnel (3,4,5) will produce more valuable features. In HDFS dataset, the CNN filter attention filters better features and achieves more accurate results.

6.4.2 Attention-based CNN vs. Vanilla CNN

The major difference between the attention-based CNN approach and our prior CNN model [45] is that the attention model focuses on more reasonable relevant features rather than the features extracted by max pooling. Moreover, the attention features belong to a subset of whole features. Hence, the attention-based approach reduces the timing and complexity of learning relationships between features and improves accuracy.

Considering about deploying CNN to log analysis, the network training includes `logkey2vec` embedding layer, and it coverts the vectors inside the two-D matrix. Each row of the matrix indicates a log key encoding, and each column of the matrix presents the number of log keys that exists in the log file. The CNN model will think over the contents of each log, then leverages multiple CNN filters to control the "memory length" with more flexible. This training operation is more effective than recurrent-based neural networks. In the proposed CNN-based detection model, there are only 29 unique log keys, also the length of the log is quite stable. In this way, the embedding is more effective but without dropping useful information. Compare with LSTM, our CNN-based model is a suitable approach for anomaly detection from system logs.

6.5 Conclusion

In this chapter, an attention-based deep neural network approach is proposed to detect anomalies from HDFS logs. Our approach includes the `logkey2vec` embedding, an attention-based CNN model with two different attention schemes. Our model does not require any expert knowledge and could provide a high accuracy without the overhead. An MLP-based model and a vanilla CNN-based model are implemented as the experimental baselines. The experimental results demonstrate that the CNN filter attention scheme achieves higher accuracy than the basic CNN, MLP, and LSTM on HDFS logs.

CHAPTER 7: WHITE BOX ADVERSARIAL ATTACKS ON NEURAL NETWORKS FOR ANOMALY DETECTION

7.1 Introduction

Deep Neural Network (DNN) is being adopted in the log analysis domain for higher accuracy and better automation. For instance, Long Short Term Memory (LSTM) and Recurrent Neural Network (RNN) are used by [10, 17] to detect anomalies with high accuracy to avoid ad-hoc feature extraction. Within all deep learning methods, Convolutional Neural Network (CNN) is one of the most famous and widely used approaches. Due to its convolutional layers, the CNN-based approach can learn spatial relationships with high accuracy. Recently, another important technique, attention mechanism, has achieved dramatic improvements in many applications. Moreover, those methods could reach high accurate results for anomaly detection. However, the network's results may not be as stable as our thoughts. To the best of our knowledge, there is no work to test the robustness of neural networks for log analysis so far.

Various adversarial examples could be leveraged to attack target networks [39]. The white-box and black-box are two popular approaches for adversarial attacking. Due to the fact that a white-box attacker knows all information of the target networks, which include input data, output data, parameters, even activation function, and loss function. Fast Gradient Sign Method (FGSM) is one common white-box approach that was first proposed in the computer vision domain [48]. It leverages the gradients of testing data to generate adversarial examples. In this chapter, a white-box adversary attacking is proposed to against our previous attention-based CNN anomaly detector, and this work is in submission. We first generate adversarial examples according to the attention weights, and vulnerable logkeys. Then, we leverage three different attack strategies to attack CNN-

based neural networks for anomaly detection.

7.2 Target Models and Dataset

HDFS dataset and CNN-based model described in Chapter 4 are our target dataset and model. The network is trained with HDFS dataset, which contains 200 days Hadoop logs on Amazon EC2. The data was first published by Xu *et al.* [61], and analyzed by many approaches such as SVM [19], PCA [61], and LSTM [17] based anomaly detection. The raw log file is 1.55 GB and contains 11,197,954 log entries. Moreover, HDFS log records the states of each HDFS block during job execution time and includes 29 unique log keys. Furthermore, the raw data is always parsed with session windows, and each line consists of unique blockIDs with related log keys in the parsed format. We leverage the parsed and labeled ground truth data, which is the same as [17]. It contains normal training set (4,855 parsed sessions), a normal testing set (553,366 parsed sessions), an abnormal training set (1,638 parsed sessions) and an abnormal testing set (15,200 parsed sessions).

7.3 Methodology

As mentioned before, white-box attacking will leverage all information of the target network. By using the inner attention weights, network could check whether the model actually focuses on relevant logkeys. Inspired by this, our adversarial attack examples are generated by leveraging the internal attention mechanism of target neural networks. We perform a logkey level white-box adversarial attack by using the models' internal attention weight distributions to locate vulnerable logkeys and attack them. Our attacking approach is twofold. Firstly, we calculate maximum attention weights and identify the related logkeys. Secondly, according to the proposed strategies,

we can modify the vulnerable logkeys and create test set for attacking.

7.3.1 Identifying the Vulnerable Logkeys

To attack the target neural network, we need to identify which and how many log key will contribute sinificantly to the classification. Hence, we mainly reuse the attention calculation of logkey attention scheme inside the target network to locate vulnerable logkeys. The attention weight calculation is described in Chapter 4. Inside the target model, the CNN outputs are used for attention weight calculation. An attention input vector a_i and output vector c is presented by:

$$c = \sum_{i=1}^{N} \lambda_i a_i \tag{7.1}$$

The attention weight of each input i is computed by:

$$e_i = w^T a_i \tag{7.2}$$

$$\lambda_i = \frac{exp(e_i)}{\sum_{j=1}^{n} exp(e_j)} \tag{7.3}$$

where w is a learnable parameter using back propagation by comparing the ground truth and the predicted results. Let Λ^{1*n} denote the matrix of attention weights λ, where 1 and n are the dimensions of Λ. Let E^{1*n} denote the set of e, where 1 and n are the dimensions of E.

Hence, to identify the vulnerable logkey inputs, we calculate the maximum value of attention weights λ in testing each batch and locate the vulnerable logkey inputs by analyzing different CNN filter sizes. However, three CNN filters have different sizes (3,4,5), there should be three

groups of vulnerable logkeys. To identify which filter could locate better valuable logkeys, we leverage statistical analysis to calculate the mode, average, and standard deviation of the biases (distances) between different groups of maximum attention weights position, which are produced by different sizes of CNN filters (3, 4, 5). We define the position of filter 3 is center, and collect data by running ten times of the attention-based CNN, and the average result is shown in Table 7.1. Then, we find that the area scanned by a small size filter (size = 3) is the average public area under three different filters. Hence, we leverage the suitable CNN filter (size = 3) to locate three vulnerable logkeys which is shown as Figure 7.1.

Table 7.1: Biases between different filters (filter 3 is center)

Measurement	filter 3 and filter 4	filter 3 and filter 5	filter 4 to filter 5
Mod	2	-3	-5
Avg	2	-2	-4
Stdev	1.38	1.92	2.19

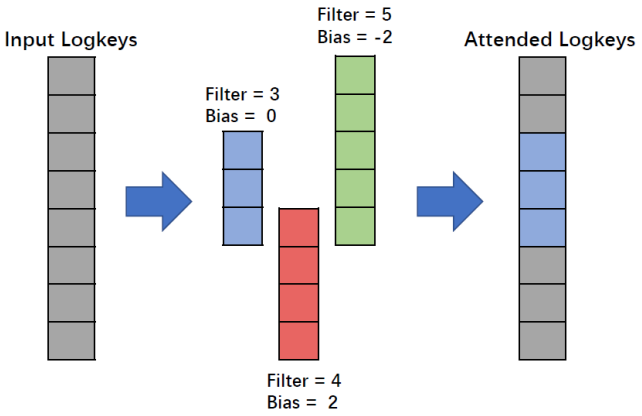

Figure 7.1: Suitable filter decision

7.3.2 *Attacking Strategies*

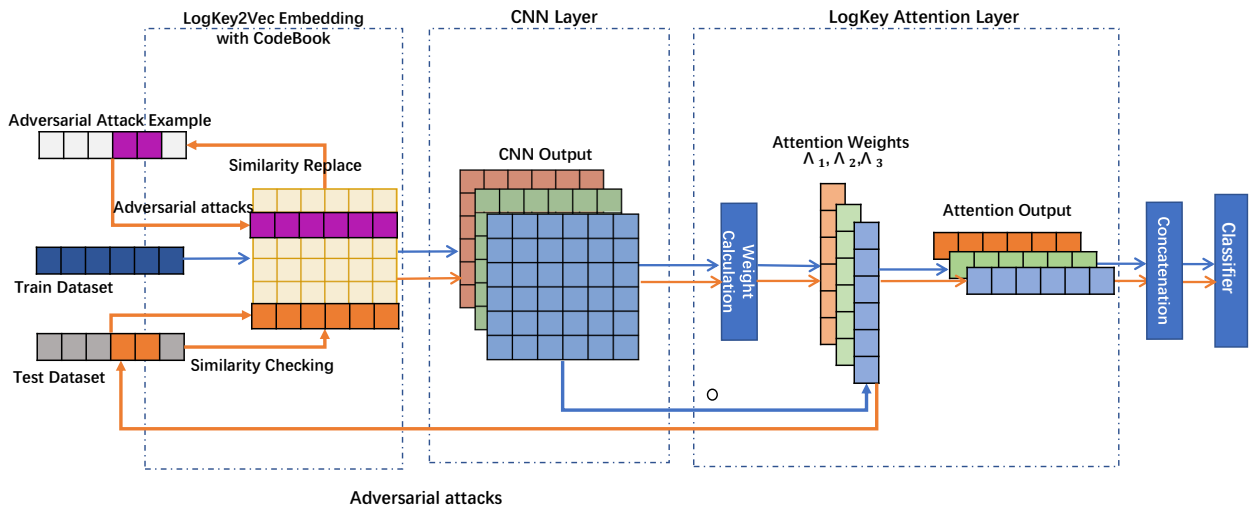

Figure 7.2: Architecture of similarity replacement strategy

After identifying three vulnerable logkeys, we design and apply three different systematic adversarial attack strategies for anomaly detection neural network, `e.g.`, exchanging positions, similarity replacement, and removing. For exchanging positions, we exchange those three positions of three logkeys to make a new sequence. For removing strategy, we just simply delete the vulnerable logkeys. For similarity replacement, we calculate the similarities of vulnerable logkeys with other logkeys by measuring the Euclidean distances and replace them with the closest one. The similarity replacement strategy is meaningful. The similarity value function of logkey replacement is shown as Eq. 7.4.

$$\min_{\Delta x_i}(\max_{\lambda_i}(\lambda_i) f(\Delta x_i)) \tag{7.4}$$

where the Δx presents the distance of vulnerable logkey x and replaced logkey x'.

The workflow of our adversarial attack approach with similarity replacement strategy is shown in Figure 7.2. The distance between each vulnerable logkey and other logkey will be calculated in each logline of test set. Then, the adversarial attack examples will be created by using the shortest Euclid distance logkeys instead of original vulnerable logkeys.

7.4 Experiments

This section presents a series of experiments of the proposed three attacking strategies. Our experiment is conducted on a sever with 4 GPUs, and our code is implemented in TensorFlow [4]. We generate our adversarial examples based on our attention-based CNN model.

To evaluate whether the three strategies decrease accuracy, three different strategies are evaluated by their test set on the same attention-based neural network model. The full normal test set is leveraged for this experiment as a normal situation without any attacks. After the first normal test finishes, we run different test sets produced by three strategies. We show our experimental evaluation results with respect to accuracy, precision (P), recall, and F1-measure (F1). For no attack model (normal cases), we get the state-of-art results, the accuracy is 99.8688, precision is 96.86, recall is 98.72, F1-measure is 97.78. Then, we first remove the fault classification test dataset and evaluate our three attack strategies based on the correct classification test dataset. Table 7.3 shows that the removing strategy brings some decreases than exchanging positions, and a similarity replacement attacking strategy shows a huge decrease by 34% in F1-measure, 60% in recall, 71% in precision, 66% in accuracy. Moreover, we evaluate our attacking with attack success rate (ASR) which presents the percentage of adversarial examples that leveraged to be classified as the target correct class. The highest ASR of our similarity replacement strategy is 66% for correct classification.

Table 7.2: The comparison of different attack strategies for correct classification

Strategy	Accuracy(%)	Precision (%)	F1 (%)	Recall (%)
Exchanging Positions	99.9	99.99	99.99	99.97
Removing	88.946	1	66.41	79.81
Similarity Replacement	33.98	28.37	65.97	39.67

Table 7.3: ASR of three attack strategies

Strategy	ASR (%)
Exchanging Positions	0.1
Removing	11
Similarity Replacement	66

7.5 Discussion

This section discusses potential reasons why the similarity replacement strategy could achieve a more effective result, all three strategies and the significance of vulnerable logkeys are analyzed. For exchanging position strategy, it seems not working at all. Because the logging framework records different log events in time sequence, usually in normal execution, the same log events could be printed in a different order, such as $2, 4, 9, 12$ and $12, 4, 2, 9$, also logkey 2 and logkey 4 always be printed in pair. Even the order of those logkeys changes, those patterns always remain normal cases. Hence, the position exchanging may not decrease accuracy effectively. For removing strategy, it achieves some attacking success rate. During this attacking, vulnerable logkeys will be removed from the normal test set. In other word, this operation will create a new sequence of inputs without vulnerable logkeys. Because of that, the location of the lesion may have been cleared, the negative classification will be misled by this strategy. Moreover, the precision does not decrease, it means the negative data could be classified as correct data because some of the vulnerable logkeys in negative data are deleted. In this way, we propose a similarity replacement

strategy for attacking target networks more effective. For similarity replacement strategy, the major difference between similarity replacement and the other two strategies is that the replaced logkeys could be highly similar in semantic meaning, which means the replacement may replace the most likely two logkeys. It will not change too much information such as removing. According to our data analysis, logkey 6 and logkey 26 are calculated as two similar logkeys, logkey 6 presents "Served block (*) to (*)" and logkey 12 presents "BLOK NameSystemåddSTOREBlock:block update" in the original dataset. Therefore, vulnerable logkeys are replaced, and the classification will be fooled. Moreover, it is hard to detect the meaning difference by users. As we know, the attention-based log analysis neural networks leverage vulnerable features to detect anomalies instead of using whole feature sets. Our similarity replacement already modifies the most important vulnerable logkeys in the whole test set. Hence, it could reach the highest ASR, and has a large effect on the average performance of target neural networks.

7.6 Conclusion

In this chapter, a white-box adversarial attack approach is proposed to attack attention-based neural networks for anomaly detection. Our approach leverages the internal network attention to create adversarial examples and applies three adversarial attack strategies on target neural networks. To prove the brittleness of attention-based models, we find that vulnerable logkeys are sensitive to our adversarial perturbation by using similarity replacement strategy.

CHAPTER 8: CONCLUSION

This dissertation presents four novel approaches to detect anomalies from big data system logs. First, a statistical rule-based approach is applied to Spark logs for detecting abnormal tasks and analyzing root causes. Secondly, a GRNN-based approach is leveraged to learn a set of features weights and avoid ad-hoc weight calculations for abnormal task detection. Thirdly, we implement a CNN-based approach with different filters for convoluting with embedded log vectors. The width of the filter is equal to the length of a group of log entries. A max-overtime pooling is applied for picking up the maximum value. Moreover, multiple convolutions layers are employed for computing. Then, we add a fully connected softmax layer to produce the probability distribution results. We also implement an MLP-based model that consists of three hidden layers without any convolutional kernels. Our experimental results demonstrate that the CNN-based method can achieve higher and faster detection accuracy than MLP and LSTM on big data system logs (HDFS logs). Moreover, our CNN model is a general method that can parse log directly and does not require any system or application-specific knowledge. Fourthly, we add two attention schemes on our previous CNN-based approach to improving accuracy for anomaly detection on the HDFS dataset. Finally, we leverage the internal attention mechanism to generate adversarial attack examples to prove the robustness of our CNN-based approaches for anomalies detection.

For future work, more complex system logs will be considered for training and testing. Furthermore, we plan to design an automatic log analyzer that can leverage deep learning approaches to detect anomalies and classify root causes into multiple classes.

Our research has the following impacts on research and society. First, our research could automatically address detection issues and save a lot of human resources by analyzing logs. Secondly, we can help people better understand their log files and make the logs more valuable. Thirdly,

our approach can offer high accuracy to detect the anomalies with the lower misclassification rate. Fourthly, we use AI-powered approaches to make detection methods smarter.

APPENDIX : COPYRIGHT MATERIALS

Copyright Clearance Center RightsLink®

Home | Create Account | Help | LIVE CHAT

IEEE Requesting permission to reuse content from an IEEE publication

Title: Log-based Abnormal Task Detection and Root Cause Analysis for Spark

Conference Proceedings: 2017 IEEE International Conference on Web Services (ICWS)

Author: Siyang Lu

Publisher: IEEE

Date: June 2017

Copyright © 2017, IEEE

LOGIN

If you're a copyright.com user, you can login to RightsLink using your copyright.com credentials.

Already a RightsLink user or want to learn more?

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

BACK | CLOSE WINDOW

LIST OF REFERENCES

[1] Apache Hadoop website. http://hadoop.apache.org/.

[2] Apache Spark website. http://Spark.apache.org/.

[3] Self4j. https://www.slf4j.org/.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[5] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review*, 37(5):74–89, 2003.

[6] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, volume 10, page 24, 2010.

[7] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[8] C. Bishop, C. M. Bishop, et al. *Neural networks for pattern recognition*. Oxford university press, 1995.

[9] D. Borthakur et al. Hdfs architecture guide. *Hadoop Apache Project*, 53, 2008.

[10] A. Brown, A. Tuor, B. Hutchinson, and N. Nichols. Recurrent neural network attention mechanisms for interpretable system log anomaly detection. *arXiv preprint arXiv:1803.04967*, 2018.

[11] L. Chen, W. Lu, E. Bao, L. Wang, W. Xing, and Y. Cai. Naive bayes classifier based partitioner for mapreduce. *Ieice Transactions on Fundamentals of Electronics Communications Computer Sciences*, 101(5):778–786, 2018.

[12] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 595–604. IEEE, 2002.

[13] Q. Chen, D. Zhang, M. Guo, Q. Deng, and S. Guo. Samr: A self-adaptive mapreduce scheduling algorithm in heterogeneous environment. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 2736–2743. IEEE, 2010.

[14] A. Das, F. Mueller, C. Siegel, and A. Vishnu. Desh: deep learning for system health prediction of lead times to failure in hpc. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 40–51. ACM, 2018.

[15] L. Deng, D. Yu, et al. Deep learning: methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387, 2014.

[16] Y. Ding, L. Wang, D. Fan, and B. Gong. A semi-supervised two-stage approach to learning from noisy labels. In *IEEE Winter Conf. on Applications of Computer Vision*. IEEE, 2018.

[17] M. Du, F. Li, G. Zheng, and V. Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298. ACM, 2017.

[18] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 149–158. IEEE, 2009.

[19] E. W. Fulp, G. A. Fink, and J. N. Haack. Predicting computer system failures using support vector machines. *WASL*, 8:5–5, 2008.

[20] P. Garraghan, X. Ouyang, R. Yang, D. McKee, and J. Xu. Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters. *IEEE Transactions on Services Computing*, 2016.

[21] X. Gu and H. Wang. Online anomaly prediction for robust cluster systems. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 1000–1011. IEEE, 2009.

[22] C. Gülcü. *The complete log4j manual*. QOS. ch, 2003.

[23] P. Guo, H. Huang, Q. Chen, L. Wang, E.-J. Lee, and P. Chen. A model-driven partitioning and auto-tuning integrated framework for sparse matrix-vector multiplication on gpus. In *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, page 2. ACM, 2011.

[24] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[25] S. He, J. Zhu, P. He, and M. R. Lyu. Experience report: system log analysis for anomaly detection. In *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*, pages 207–218. IEEE, 2016.

[26] H. Huang, L. Wang, E.-J. Lee, and P. Chen. An mpi-cuda implementation and optimization for parallel sparse equations and least squares (lsqr). *Procedia Computer Science*, 9:76–85, 2012.

[27] H. Huang, L. Wang, B. C. Tak, L. Wang, and C. Tang. Cap3: A cloud auto-provisioning framework for parallel processing using on-demand and spot instances. In *IEEE Intl. Conference on Cloud Computing*. IEEE, 2013.

[28] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 41–51. IEEE, 2010.

[29] D. H. Hubel and T. N. Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154, 1962.

[30] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth. Performance anomaly detection and bottleneck identification. *ACM Computing Surveys (CSUR)*, 48(1):4, 2015.

[31] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, volume 41, pages 59–72. ACM, 2007.

[32] H. Jayathilaka, C. Krintz, and R. Wolski. Performance monitoring and root cause analysis for cloud-hosted web applications. In *Proceedings of the 26th International Conference on World Wide Web*, pages 469–478. International World Wide Web Conferences Steering Committee, 2017.

[33] S. Ji, W. Xu, M. Yang, and K. Yu. 3d convolutional neural networks for human action recognition. *IEEE transactions on pattern analysis and machine intelligence*, 35(1):221–231, 2013.

[34] R. Johnson and T. Zhang. Effective use of word order for text categorization with convolutional neural networks. *arXiv preprint arXiv:1412.1058*, 2014.

[35] Y. Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[36] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. 2009.

[37] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[38] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In *The 12th ACM International Conference on Computing Frontiers*, page 53. ACM, 2015.

[39] Y. Li, L. Li, L. Wang, T. Zhang, and B. Gong. Nattack: Learning the distributions of adversarial examples for an improved black-box attack on deep neural networks. In *International Conference on Machine Learning (ICML)*, 2019.

[40] Z. Li, K. Gavrilyuk, E. Gavves, M. Jain, and C. G. Snoek. Videolstm convolves, attends and flows for action recognition. *Computer Vision and Image Understanding*, 166:41–50, 2018.

[41] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo. Failure prediction in ibm bluegene/l event logs. In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*, pages 583–588. IEEE, 2007.

[42] X. Long, C. Gan, G. de Melo, X. Liu, Y. Li, F. Li, and S. Wen. Multimodal keyless attention fusion for video classification. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[43] J.-G. Lou, Q. Fu, Y. Wang, and J. Li. Mining dependency in distributed systems through unstructured logs analysis. *ACM SIGOPS Operating Systems Review*, 44(1):91–96, 2010.

[44] S. Lu, B. Rao, X. Wei, B. Tak, L. Wang, and L. Wang. Log-based abnormal task detection and root cause analysis for spark. In *Web Services (ICWS), 2017 IEEE International Conference on*, pages 389–396. IEEE, 2017.

[45] S. Lu, X. Wei, Y. Li, and L. Wang. Detecting anomaly in big data system logs using convolutional neural network. In *2018 4th Intl Conf on Cyber Science and Technology Congress (CyberSciTech)*, pages 151–158. IEEE, 2018.

[46] S. Lu, X. Wei, B. Rao, B. Tak, L. Wang, and L. Wang. Ladra: Log-based abnormal task detection and root-cause analysis in big data processing with spark. *Future Generation Computer Systems*, 95:392–403, 2019.

[47] H. Ma, L. Wang, and K. Krishnamoorthy. Detecting thread-safety violations in hybrid openmp/mpi programs. In *2015 IEEE International Conference on Cluster Computing*, pages 460–463. IEEE, 2015.

[48] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[49] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[50] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[51] W. Qi, Y. Li, H. Zhou, W. Li, and H. Yang. Data mining based root-cause analysis of performance bottleneck for big data workload. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Con-*

ference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pages 254–261. IEEE, 2017.

[52] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[53] H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 84–88. IEEE, 2006.

[54] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proceedings of the VLDB Endowment*, 8 (13):2110–2121, 2015.

[55] V. Subramanian, L. Wang, E.-J. Lee, and P. Chen. Rapid processing of synthetic seismograms using windows azure cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 193–200. IEEE, 2010.

[56] V. Subramanian, H. Ma, L. Wang, E.-J. Lee, and P. Chen. Rapid 3d seismic source inversion using windows azure and amazon ec2. In *Services (SERVICES), 2011 IEEE World Congress on*, pages 602–606. IEEE, 2011.

[57] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, et al. Going deeper with convolutions. Cvpr, 2015.

[58] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. Salsa: Analyzing logs as state machines. *WASL*, 8:6–6, 2008.

[59] L. Wang, S. Lu, X. Fei, A. Chebotko, H. V. Bryant, and J. L. Ram. Atomicity and provenance

support for pipelined scientific workflows. *Future Generation Computer Systems*, 25(5): 568–576, 2009.

[60] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Online system problem detection by mining patterns of console logs. In *2009 Ninth IEEE International Conference on Data Mining*, pages 588–597. IEEE, 2009.

[61] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP*. ACM, 2009.

[62] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz. Wrangler: Predictable and faster jobs using fewer resources. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[63] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Osdi*, volume 8, page 7, 2008.

[64] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. USENIX Association, 2012.

[65] H. Zhang, H. Huang, and L. Wang. Mrapid: An efficient short job optimizer on hadoop. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 459–468. IEEE, 2017.

[66] H. Zhang, H. Huang, and L. Wang. Meteor: Optimizing spark-on-yarn for short applications. *Future Generation Computer Systems*, 2019.

[67] H. Zhou, Y. Li, H. Yang, J. Jia, and W. Li. Bigroots: An effective approach for root-cause analysis of stragglers in big data system. *arXiv preprint arXiv:1801.03314*, 2018.