Predicting Task Offloading Requests and Resource Demands in IoT Edge Computing using Hybrid Deep Learning Model

Abdu Aliyi Geleta

A Thesis Submitted to the Department of Computer Science and Engineering,

College of Electrical Engineering and Computing

Office of Graduate Studies

Adama Science and Technology University

October 13, 2025

Adama, Ethiopia

Predicting Task Offloading Requests and Resource Demands in IoT Edge Computing using Hybrid Deep Learning Model

Abdu Aliyi Geleta

Advisor: Ketema Adere (PhD)

A Thesis Submitted to the Department of Computer Science and Engineering,

College of Electrical Engineering and Computing

Office of Graduate Studies

Adama Science and Technology University

October 13, 2025

Adama, Ethiopia

DECLARATION

I declare that this Master Thesis entitled **„„Predicting Task Offloading Requests and Resource Demands in IoT Edge Computing using Hybrid Deep Learning Model"'** is my own work and has not been submitted to any university for the award of any academic degree, diploma, or certificate in any other university. All sources of materials that are used for this thesis have been duly acknowledged through citation.

Abdu Aliyi Geleta		
Name of Student	Signature	Date

ADVISOR RECOMMENDATION

I, the advisor of this thesis, hereby certify that I have closely advised the student while working this thesis and read the thesis entitled **"Predicting Task Offloading Requests and Resource Demands in IoT Edge Computing using Hybrid Deep Learning Model"** prepared under my guidance by Abdu Aliyi Geleta submitted in partial fulfillment of the requirements for the degree of Masters of Science in Computer Science and Engineering (CSE). Therefore, I recommend the submission of the revised version of the thesis to the department following the applicable procedures.

_____	_____	_____
Advisor/Supervisor	Signature	Date

APPROVAL PAGE FOR ADVISOR

I, the advisor of the thesis entitled **"Predicting Task Offloading Requests and Resource Demands in IoT Edge Computing using Hybrid Deep Learning Model"** developed by Abdu Aliyi Geleta, hereby certify that the recommendation and suggestions made by the board of examiners are appropriately incorporated into the final version of the thesis.

_____	_____	_____
Advisor/Supervisor	Signature	Date

APPROVAL PAGE OF M.Sc. THESIS

I/we, the advisors of the thesis entitled **"Predicting Task Offloading Requests and Resource Demands in IoT Edge Computing using Hybrid Deep Learning Model"** developed by Abdu Aliyi Geleta, hereby certify that the recommendation and suggestions made by the board examiners are appropriately incorporated into the final version of the thesis.

Advisor/Supervisor	Signature	Date

We, the under signed, members of the Board of Reviewers of the proposal open defense by Abdu Aliyi Geleta have read and evaluated the thesis proposal entitled **"Predicting Task Offloading Requests and Resource Demands in IoT Edge Computing using Hybrid Deep Learning Model"** and assessed the understanding of the candidate about the proposed research. This is, therefore, to certify that the thesis proposal is accepted and we recommend the implementation of the proposal.

Chairperson	Signature	Date
Internal Examiner	Signature	Date
External Examiner	Signature	Date

Final approval and acceptance of the thesis is contingent upon submission of its final copy to the Office of Postgraduate Studies (OPGS) through the Department Graduate Council (DGC) and College Graduate Committee (CGC).

Department Head	Signature	Date
College Dean	Signature	Date
Office of Postgraduate Studies, Dean	Signature	Date

ACKNOWLEDGMENT

First of all, I want to thank God for giving me the strength and wisdom to complete this thesis and for guiding me throughout this journey. His constant support has been a true source of inspiration.

I am extremely grateful to my advisor, Dr. Ketema Adere, for his constant support and expert advice from start to finish. His valuable feedback and encouragement have been crucial in shaping this work, and I deeply appreciate his dedication to my success.

I also want to thank my family for their love and endless support. Their belief in me kept me going through the tough times, and I couldn't have done this without them.

Finally, I'd like to extend my thanks to the faculty and staff at the Department of Computer Science and Engineering at ASTU. Their guidance and help have been invaluable in improving my academic and research skills.

To everyone who has supported me in any way, thank you from the bottom of my heart.

Contents

CHAPTERS **PAGES**

DECLARATION i
ADVISOR RECOMMENDATION ii
APPROVAL PAGE FOR ADVISOR iii
APPROVAL PAGE OF M.Sc. THESIS iv
ACKNOWLEDGMENT v
LIST OF TABLES x
LIST OF FIGURES xi
LIST OF EQUATIONS xii
LIST OF ACRONYMS AND ABBREVIATIONS xiii
ABSTRACT xv
CHAPTER ONE 1
INTRODUCTION 1
1.1 Background of the study 1
1.2 Motivation of the Study 3
1.3. Statement of the problem 4
1.4 Research question 6
1.5 Objectives of the Study 7
1.5.1. General objective 7
1.5.2. Specific objective 7
1.6 significance of the study 7
1.7 Scope and Limitations of the Study 7
1.7.1. Scope of the Study 7
1.7.2. Limitations of the Study 8
1.8. Organization of the Study 8
CHAPTER TWO 9
LITERATURE REVIEW AND RELATED WORK 9
2.1. Literature review 9
2.1.1. Introduction to Internet of Things 9
2.1.2. Overview of Edge Computing 9
2.1.3. Task Offloading in Internet of Things 13
2.1.4. Workload Prediction Techniques 14
2.1.5. Dataset Description 21

2.2. Related Work 24
2.2.1. Summary of Related Works and Research Gaps 27
2.4 Baseline Papers 29
CHAPTER THREE 30
RESEARCH METHODOLOGY 30
3.1. Introduction 30
3.2. Research Design 30
33. Dataset Sources and Descriptions 31
3.4. Data Preprocessing Techniques 35
3.5. Feature Engineering 36
3.6. Feature Selection 37
3.7. Model Selection 37
3.8. Model Training and Testing 41
3.9. Parameter Tuning 41
3.10. Evaluation Metrics 42
3.11. Design and Development Tools 44
3.11.1. Design Tools 44
3.11.2. Development Tools 44
CHAPTER FOUR 46
PROPOSED SOLUTION ARCHITECTURE 46
4.1. Chapter Overview 46
4.2. Proposed Model Architecture 46
4.3. Proposed Model Algorithm 48
4.4. Hyper-parameter Optimization 50
4.4.1. Regularization Techniques 50
4.4.2. Learning Rate 51
4.4.3. Batch Size and Number of epochs 51
4.4.4. Activation Functions 52
4.4.5. Loss Function 52
4.4.6. Optimizer 53
CHAPTER FIVE 54
IMPLEMENTATION OF THE PROPOSED MODEL 54
5.1. Chapter Overview 54

5.2. Working Environment Setup 54
5.3. Proposed Model Implementation 54
5.3.1. Dataset Pre-processing 54
5.3.2. Feature Engineering 55
5.3.3. Feature selection 57
5.3.4. Proposed Model Building 58
5.4. Benchmark Models Implementation 60
5.4.1. Classical Machine Learning Models 60
5.4.2. 1D-CNNs 61
5.4.3. DNNs 62
5.4.4. GRU 63
5.4.5. LSTM 64
5.4.6. BiLSTM 64
5.4.7. CNN with BiLSTM 65
5.5. Model Testing and Evaluation 66
CHAPTER SIX 67
RESULT AND DESCUSSIONS 67
6.1. Chapter Overview 67
6.2. Model Experiments and hyper-parameter settings 67
6.2.1. Experiment 1: The Proposed CNN-BiLSTM-Attention Model 68
6.2.2. Experiment 2: Classical Machine Learning Benchmarks 69
6.2.3. Experiment 3: 1D-CNN Model 70
6.2.4. Experiment 4: DNN Model 70
6.2.5. Experiment 5: LSTM Model 71
6.2.6. Experiment 6: GRU Model 72
6.2.7. Experiment 7: BiLSTM Model 73
6.2.8. Experiment 8: CNN-BiLSTM Model 73
6.3. Model Performance Evaluations 74
6.3.1. Classification Tasks Analysis Result 74
6.3.2. Performance Evaluation Based on Regression Metrics 77
6.4. Proposed Model in Comparison with Other Related Work Models 87
6.5. Result Discussion on Proposed Model 83
6.6. Research Question Discussions 89

7. CONCLUSIONS AND FUTURE WORK 93
7.1. Conclusion 93
7.2. Major Contribution 94
7.3. Future Work 95
REFERENCES 96
Appendix I 103
Appendix II 106

LIST OF TABLES

Table 2-1: Comparison of Cloud and Edge Computing 12
Table 2-2: Comparison of DL Models 18
Table 2-3: Dataset Comparisons 24
Table 2-4: Summary of Related Works 27
Table 3-1: Summary of feature descriptions 34
Table 5-1: Mapping class names for task priority and scheduling class 56
Table 6-1: Hyper-parameter Configurations of Models 67
Table 6-2: Performance comparison of models on categorical targets 74
Table 6-3: MAE Results 78
Table 6-4: MSE Results 78
Table 6-5: RMSE Results 78
Table 6-6: R^2 Results 79
Table 6-7: MAPE Results 79
Table 6-8: Comparison of the proposed method with and without the attention mechanism 84
Table 6-9: Comparison table of the proposed model with the existing related works 88

LIST OF FIGURES

Figure 2-1: Three-tier edge computing architecture 11
Figure 2-2: Logic diagram of IoT task offloading in MEC 13
Figure 2-3: An illustration of a 1D-CNN model 15
Figure 2-4: An LSTM cell structure 16
Figure 2-5: BiLSTM model 17
Figure 2-6: Some Deep Learning Models and Their Hybrids 19
Figure 2-7: Attention Mechanism Model 21
Figure 3-1: General Flow of Research Methodology 31
Figure 3-2: Sample raw Cluster2019 data records 33
Figure 4-1: Proposed Model Architecture 48
Figure 5-1: Sample records of preprocessed ClusterData2019 56
Figure 5-2: Top 12 Feature Importance using Random Forest 58
Figure 5-3: Importing necessary libraries 58
Figure 5-4: Proposed CNN-BiLSTM-Attention Training 59
Figure 5-5: Logistic Regression Training 60
Figure 5-6: Decision Tree Training 61
Figure 5-7: 1D-CNN implementation 62
Figure 5-8: DNN Model Training 63
Figure 5-9: GRU Model Training 63
Figure 5-10: LSTM Model Training 64
Figure 5-11: BiLSTM Model Training 65
Figure 5-12: CNN-BiLSTM Model Training 65
Figure 6-1: Training and Validation Loss of CNN-BiLSTM-Attention model 69
Figure 6-2: Training and Testing Loss/Errors of Decision Tree. 70
Figure 6-3: Training and Validation Loss/Errors of Linear Regression 70
Figure 6-4: Training and Validation Loss of 1D-CNN. 70
Figure 6-5: Training and Validation Loss of DNN 71
Figure 6-6: Training and Validation Loss of LSTM 72
Figure 6-7: Training and Validation Loss of GRU 72
Figure 6-8: Training and Validation Loss of BiLSTM 73
Figure 6-9: Training and Validation Loss of CNN-BiLSTM 74
Figure 6-10: Models comparison on the categorical targets predictions 75
Figure 6-11: Confusion Matrix of proposed model 76
Figure 6-12: Some Benchmark Models Confusion Matrices 77
Figure 6-13: Actual vs Predicted values by proposed hybrid model 80
Figure 6-14: Actual vs Predicted values by GRU model 81
Figure 6-15: Comparison of models performances on prediction accuracy (R2) 82
Figure 6-16: Comparison of models performances on MAE 82
Figure 6-17: Comparison of models performances on RMSE 82
Figure 6-18: Comparison of models performances on MAPE 83
Figure 6-19: Attention score heat-maps from two heads of the proposed model 86

LIST OF EQUATIONS

Equation 3- 1: Mean Absolute Error Formula 42
Equation 3- 2: Mean Square Error Formula 42
Equation 3- 3: Mean Absolute Percentage Error Formula..... 43
Equation 3- 4: Root Mean Square Error Formula..... 43
Equation 3- 5: R-squared Formula 43

LIST OF ACRONYMS AND ABBREVIATIONS

1D-CNN	1D Convolutional Neural Network
AI	Artificial Intelligence
AP	Access Points
BiLSTM	Bidirectional Long-Short Term Memory
BS	Base Stations
CAP	Computational Access Point
CNN	Convolutional Neural Networks
CPU	Central Processing Unit
DDPG	Deep Deterministic Policy Gradient
DDQN	Double Deep Neural Networks
DNN	Deep Neural Networks
DQL	Deep Q-Learning
DQN	Deep Q Network
EC	Edge Computing
ETSI	European Telecommunications Standards Institute
FRS	Fine-grained Resource Scheduling
HSTM	Hierarchical Spatial-Temporal Monitoring
IoT	Internet of Things
LSTM	Long-Short Term Memory
LTE	Long Term Evolution
MAE	Mean Absolute Error
MCC	Mobile Cloud Computing
MDP	Markov Decision Process
MEC	Multi-access Edge Computing
OPO	Online Policy Optimization
QoE	Quality of Experience
QoS	Quality of Service
RAN	Radio Access Networks
RMSE	Root Mean Square Error
SVM	Support Vector Machine
TDs	Terminal Devices
UAV	Unmanned Aerial vehicles

VM	Virtual Machine
WiFi	Wireless Fidelity
WLAN	Wireless Local Area Network

ABSTRACT

The dynamic and heterogeneous nature of IoT edge environments, where billions of devices continuously generate diverse and time-varying workloads, necessitate predictive mechanisms capable of accurately forecasting both task offloading requests and resource demands. While existing prediction-based offloading studies have primarily focused on forecasting task volumes or a single resource and in some cases both CPU and memory), none have jointly predicted offloading requests with task-level features alongside their associated resource requirements. To address this gap, we propose a hybrid multi-task learning CNN-BiLSTM-Attention model. The CNN component extracts local temporal patterns, the BiLSTM captures long-range dependencies, and the Attention mechanism emphasizes the most informative time steps and features. The model jointly predicts two categorical offloading parameters (task priority and delay tolerance) and four continuous resource-demand metrics (CPU request, memory request, maximum CPU usage, and maximum memory usage). A multi-objective learning strategy was employed, with classification targets representing task-level semantics and regression targets estimating resource demands. Optimization was performed using a weighted combination of sparse categorical cross-entropy and mean squared error losses to effectively balance the heterogeneous objectives. The Google Cluster dataset was employed to train and evaluate the proposed model. Model performance was assessed using standard evaluation metrics, including MAE, MSE, RMSE, R^2, MAPE and ordinal-aware accuracy for categorical targets. Post training quantization was performed for edge compatibility. The results indicate the superior performance of the proposed model consistently achieving the best results, with a minimum values of MAE =0.00014, MSE and RMSE values as low as 0.0001, and an R^2 score of 0.99, alongside a minimum MAPE of 2.30%. For classification tasks, the model attained the highest accuracy of 0.98. Additionally, we have benchmarked deep learning and classical machine learning models. Furthermore, when compared with prior state-of-the-art studies on workload prediction using the same dataset, the proposed model improved prediction accuracy by 6.4% in terms of R^2, thereby demonstrating its outperformance and advancement over existing approaches.

***Keywords**: CNN-BiLSTM-Attention, Google Cluster dataset, hybrid deep learning, IoT edge computing, multi-task learning, task offloading requests, resource demand prediction*

CHAPTER ONE

1. INTRODUCTION

1.1 Background of the study

Internet of Things (IoT) edge computing has become a new approach that acts like a distributed processing center, unlike the existing traditional cloud-centric architecture, providing flexible management by bringing computation and storage closer to end devices from a logically decentralized edge layer (Deepak et al., 2023; Shi et al., 2016). IoT edge computing's ability to separate the device, edge, and cloud layers gives developers more flexibility and makes application deployment and maintenance easier in these kinds of situations (Mao et al., 2017). There are three main parts of IoT edge computing: the device layer, the edge layer, and the cloud layer. The edge layer holds configurations and executes offloaded tasks based on scheduling and resource allocation policies; the device layer comprises IoT devices such as sensors, wearables, and smart appliances that generate computational tasks; the cloud layer provides large-scale computation and storage for tasks that cannot be processed at the edge (Chiang & Zhang, 2016).

The key part of IoT edge computing is task offload, a way that allows resource-constrained IoT devices pass intensive computational tasks to nearby edge hosts (Ullah et al., 2023). Offloading reduces both the processing burden and use of power for terminal devices and it makes them last long. However, task offloading is not without drawbacks. When too many devices try to offload tasks at once, edge servers can quickly become overloaded (Dean & Barroso, 2013; Zhang et al., 2021). This can introduce latency, missed deadlines, and denial of service, particularly when edge servers have limited computational resources (Shi et al., 2016; Taleb et al., 2017). In busy times, many requests can overload the system. This can make it hard to do tasks fast and within expected timeframes. It leads to waits and missed deadlines (Gautam & Malhotra, 2024). This kind of congestion affects both users and service providers in distinct but interconnected ways. On the user side, when CPU, memory, and network bandwidth are all too full, applications may slow down or not work well. This hurts the flow of work, access to services and overall user experience (Walia et al., 2024). For vendors, the consequences are equally serious. Slow runs can cause less work, financial penalties and losses in business chances (Lan et al., 2024). These risks forces organizations to plan ahead, not just act when things get bad. In particular, strong predictive mechanisms are essential. Since, IoT traffic and offload ways act in ways

tied to time, these can be effectively modeled using time series approaches (Zhang et al., 2021).

Traditionally, time series workloads were predicted using ARIMA, but in IoT edge computing, where rapid adaptation is mandatory, in literature, these have been replaced by Machine Learning (ML) for capturing complex patterns and scaling to large datasets, though ML requires more data, risks overfitting, and is computationally costly (Islam et al., 2012). Deep learning has since become the preferred approach, offering feature extraction and multi-target predicting, even from incomplete data (Casolaro et al., 2023). Compared to ML, Deep Learning methods are excellent for detecting complex patterns and scalability, but require careful hyper-parameter tuning and extensive training data (Patel & Bedi, 2023).

One of major research challenge in predicting offloading requests and workload behaviors in IoT edge computing is lack of suitable training data (Liu & Wang, 2024). Many existing datasets were collected from small, non-edge environments, and synthetic data leading to models that perform poorly in real-world IoT edge deployments due to differences in workload scale and behavior. For example, the study (Nugroho & Kim, 2024) used synthetic data gathered from simulated IoT edge environments.

The study (Tu et al., 2022) attempted to predict task volumes for mobile devices and server loads. However, this work primarily focuses on task volume, without considering other crucial task-level features such as priority and delay tolerance. This omission limits the scheduler's ability to differentiate between critical and non-critical tasks, leading to suboptimal resource allocation, QoS violations, and potential SLA breaches. Ignoring these lead to misallocation of scarce edge resources, deadline violations for latency-sensitive tasks, and unfair treatment of critical workloads. In IoT environments where applications range from mission-critical healthcare monitoring to delay-tolerant data aggregation, overlooking these features undermines both user experience and provider reliability.

While other studies including (Chen et al., 2022; E. Karim et al., 2021; Lackinger, 2023; Yadav et al., 2021), and (Nugroho & Kim, 2024) mainly limits their methodology in observing and predicting a single resource such as CPU. In case of multivariate contributions predicting both CPU and memory usages, several works have been done including (Devi & Valli, 2022; Jeong et al., 2023; E. Karim et al., 2021; Patel & Bedi,

2023; Xu et al., 2022). However, we have seen that none of these works simultaneously considered both task level features and the associated resource needs. Therefore, reviewing literatures and to the best of our knowledge, none of the existing studies on IoT edge computing have addressed joint prediction of task offloading requests with their corresponding resource demands.

1.2 Motivation of the Study

Edge-assisted IoT deployment has emerged as new approach bringing computation and storage closer to the end devices. An interesting feature of this computing paradigm is task offloading; a way that resource-limited IoT devices offload their heavy processing tasks to the nearby edge hosts. This could significantly reduce latency, privacy issues and energy consumptions. But, when many concurrent user tasks arrives a certain edge server simultaneously, the server will not be able to allocate the required resources for all user tasks. This unexpected spike in offload requests leads to long wait times, missed deadlines, and performance degradation. Many examples of such workloads' sudden rises and its outage effects have been reported in real-world IoT deployments by the authors (Anandayuvaraj & Davis, 2022). Since, users offload requests are related time, it is possible to predict before they actually arrive. Therefore, workload predictions become interesting research area. Embedding such predictions into the scheduling loop could enable early task scheduling and resource allocations, load balancing, reduced waiting times and admission control.

Therefore, considering the serious impact of sudden workload surges and the diverse, constantly changing nature of user requests in IoT edge computing, it is essential to address these challenges with advanced solutions. Hybrid deep learning models are well-suited for this task because they employ strengths of more than one model and can learn complex patterns in workload data and give accurate predictions of user offloading requests and resource demands. To make these predictions truly effective, it will be imperative to integrate attention mechanisms as they enable the model to concentrate on the most influential workload features, thereby enhancing both the accuracy and interpretability of multivariate predictions. Furthermore, the effectiveness of such joint and multivariate predictions critically depends on effective feature extraction techniques and the use of real-world server logs containing essential workload characteristics.

1.3. Statement of the problem

IoT edge computing brings compute and storage closer to sensing devices, enabling low-latency processing of data generated by billions of endpoints. Each offloading request typically includes metadata such as Task/Device ID, task size, deadline, delay tolerance, and resource requirements. Once these requests reach the edge servers, they are queued and scheduled for execution using different scheduling algorithms. While these methods help to manage task execution order, they also introduce waiting times and queuing delays between the moment tasks are submitted and the time results are returned. But, when a large number of heterogeneous IoT devices send many computation requests to the nearest edge servers, it causes overload of the edge layer, where the edge orchestrator is not able to allocate resources, meet deadlines and balance load across nodes. So, it causes service degradation and deadline violations.

Particularly, in high-demand scenarios, this can affect both users and service providers in distinct but interconnected ways (Gautam & Malhotra, 2024). On the user side, when CPU, memory, and network bandwidth are all too full, applications may slow down or not work well. This hurts the flow of work, access to services and overall user experience (Walia et al., 2024). For vendors, the consequences are equally serious. Slow runs can cause less work, financial penalties and losses in business chances (Lan et al., 2024).

In addition to service breaks and financial losses, sharp rises in processing requests impose significant management expenses on IoT service providers. Mitigating the effects of these sudden increase often needs adding more edge nodes, and bringing IT teams to troubleshoot and fix the system (Guim et al., 2022). Also, fast rise in user requests can lead to big fails, like too much web traffic or cloud crash from too much load. This harms the whole system's work (Hong et al., 2023). Since, IoT traffic and offload ways act in ways tied to time, these can be effectively modeled using time series approaches (Zhang et al., 2021). Consequently, implementing strong predictive mechanisms based on deep learning is essential to enable IoT edge systems to manage tasks efficiently, prevent service disruptions, and decline in performance.

Many researchers proposed different methods for time series predictions using DL learning approaches. For example, researchers Tu et al., (2022), Wang et al., (2024), Pandiyan, G, Sasikala, (2023), Nugroho & Kim, (2024), Ai et al., (2023) and others we summarized in table 2-1 have explored predictive frameworks to predict offloading requests and server

workloads using historical logs. While their contributions represent an important step forward, current prediction mechanisms still have notable gaps that need to be addressed.

One major limitation is that most existing approaches focus on a single resource usages. For instance, some methods classify tasks by priority but fail to predict CPU or memory requirements. Other works attempted to predict resource usage but ignored task features such as delay tolerance and priority. This approach, as highlighted by Morichetta et al., (2023), this narrow approach has two main drawbacks. First, a new model must be trained for different groups of workloads or individual workloads, which involves both time and resource costs. Secondly, this uni-variate prediction ignores the interdependencies among multiple task and resource features, such as how delay tolerance, priority, and arrival times affect both CPU and memory usage.

Another issue is many existing studies often use synthetic datasets or small-scale experimental traces that do not reflect the complexity and variability of real-world IoT environments. As a result, models trained on such data do not generalize well when deployed in production environments. Although some studies used Google cluster traces from real-world data, they did not conduct comprehensive feature engineering to derive influential features associated with sudden increases in user requests and resource demand spikes.

Furthermore, a lot of earlier methods use separate deep learning architectures, usually CNNs for local spatial correlations or LSTMs for temporal sequence modeling, but they rarely combine the two. Their ability to recognize both short-term workload variations and long-term temporal dependencies that naturally arise in dynamic IoT edge environments is limited by this separation. Furthermore, attention mechanisms which are essential for determining and weighing the most significant features and time steps, improving interpretability and adaptability under workloads that change quickly have not been included in many studies.

Moreover, multi-task learning techniques that simultaneously optimize the prediction of several correlated targets like offloading requests, CPU load, and memory usage in a single, cohesive framework have not been investigated in earlier research. In addition to limiting model efficiency, this exclusion ignores the possible performance improvements

that could be attained through shared representation learning across related resource dimensions.

To overcome these challenges, we propose a hybrid multi-task deep learning model that combines 1D-CNN, BiLSTM, and attention mechanisms to jointly anticipate resource demands and task offloading requests in future network time slots before these tasks actually arrive. In the proposed model, 1D-CNN was employed to extract short-term patterns from this large scale dataset and BiLSTM to track complex temporal patterns and long-range connections. The Google Cluster Trace dataset was selected as the experimental basis for this study because of its large scale, temporal continuity, and operational diversity. Millions of task-level records gathered from production-scale distributed systems are included in it, illustrating realistic fluctuations in workload intensity, task arrival patterns, and resource consumption patterns over time. Despite coming from cloud environments, the dataset's time-dependent patterns, heterogeneity, and multi-resource characteristics strongly resemble those of extensive IoT edge infrastructures.

Our novelty lies in addressing key methodological gaps observed in previous workload prediction research through the combination of semantically rich modeling approach, multivariate feature learning**,** hybrid temporal modeling**,** multi-task optimization, and usage of real-world large-scale dataset. Additionally, we used post-training quantization to minimize model size and computational complexity without significantly sacrificing accuracy in order to make sure that the proposed hybrid model is appropriate for resource-constrained edge environments. Thus, we aim to develop a predictive, multivariate, and lightweight multi-task deep learning framework that can anticipate IoT task offloading requests and resource demands at the edge, enabling proactive scheduling, reduced wait times, and better load balancing.

1.4 Research question

To address the aforementioned issues, this study seeks to answer the following research questions:

1. What preprocessing methods for Google cluster dataset best give multivariate inputs for time series predictions of offloading requests and resource needs?
2. How can a hybrid multi-task deep learning architecture combining 1D-CNN, BiLSTM, and attention mechanisms be designed and trained to jointly predict task offloading requests and resource demands in IoT edge computing?

3. In what ways does adding an attention mechanism into the proposed hybrid model improve its prediction accuracy and interpretability?
4. How does the proposed hybrid model perform in comparison with baseline deep learning models in terms of prediction accuracy, and feasibility for edge deployment?

1.5 Objectives of the Study

1.5.1. General objective

The general objective of this research was to predict task offloading requests and the expected resource demands in IoT edge computing using hybrid deep learning model.

1.5.2. Specific objective

1. To preprocess and perform feature engineering on the Google Cluster dataset to generate multivariate inputs for time series predictions.
2. To develop a hybrid multi-task deep learning CNN-BiLSTM-Attention model to jointly predict task offloading requests and multivariate resource demands.
3. To evaluate the effect of the Attention mechanism in the proposed hybrid model on prediction accuracy, and interpretability.
4. To compare the performance of the proposed hybrid model with baseline models in terms of prediction accuracy and feasibility for edge deployment.

1.6 significance of the study

This research is strongly useful in predicting the incoming task offloading requests and how much computing resources those tasks will need. The proposed solution is intended to be embedded to the edge servers' scheduler component that schedules IoT offloading requests and associated resource required ahead before they arrive. This early scheduling and resource allocations is helpful for the IoT service providers, edge computing infrastructure managers, and industrial users to prevent system overloading and outages which increase QoS, wisely use of edge resources and mitigate deadline violations and long-waiting time in edge services under high-traffic scenarios. Hence, it increases the service accessibility of IoT edge services which maintains customer trust and SLAs.

1.7 Scope and Limitations of the Study

1.7.1. Scope of the Study

Nowadays, there are many advanced task offloading optimizations in IoT edge computing environments. However, this study focuses on predicting task offloading requests and the expected resource demands. Several tools, techniques and datasets are to be applied, but our study focused on hybrid 1D CNN-BiLSTM-Attention model to jointly predict the

incoming offloading requests and the corresponding resource consumptions. We evaluate our model only on one cluster of Google cluster dataset which was traces recorded over 31 days, in May 2019.

1.7.2. Limitations of the Study

Although, studying edge server selection techniques would provide additional knowledge for the edge server schedulers, it was beyond the scope of this research due to time and resource constraints.

1.8. Organization of the Study

This thesis is organized into seven chapters. The first chapter outlines the background, the motivation, the problem, objectives, the significance, and scope of the study. The second chapter provides a review of the existing literature on IoT edge computing, task offloading strategies, time series prediction techniques, and critical evaluations of related works. In chapter three, we present the research methodology we used including an explanation of dataset sources, preprocessing of data, feature engineering, features selection techniques, the model selection, evaluation metrics, and tools. In chapter four we discuss the proposed model architecture. In chapter five we discuss implementation of the proposed model including the environment setup, implementation steps, and optimization methods. In chapter six we discuss the results and discussions, including evaluating the performance of the proposed model against baseline models. In the final chapter we discuss the contributions we made, and recommendations for future research directions.

CHAPTER TWO

2. LITERATURE REVIEW AND RELATED WORK

This chapter presents a review of the existing literatures relevant to the proposed study. The review is structured into several sections that discuss overview of edge computing, task offloading strategies, algorithms/approaches and ML/DL models in task offloading requests and the anticipated resource requirements, offloading prediction models and related works on offloading approaches in IoT edge computing. This section provides a foundation for understanding the research gaps and the need for the proposed solution.

2.1. Literature review

2.1.1. Introduction to Internet of Things

The Internet of Things (IoT) is a computing paradigm in which any low-power device connects to the Internet, such as a smartphone, wearable, sensor, and so on. IoT provides interconnection of various heterogeneous devices which can transfer data via Internet. Today various people are connected with internet via several types of communication devices. Connecting all electronic gadgets to the internet is the basis of the Internet of Things. These devices have the ability to send and receive information when they are connected to the internet (Rayes & Salam, 2022).

IoT is now integral to sectors like smart cities, agriculture, healthcare, smart homes, and industries. Even though it has wide applications, IoT faces challenges such as scalability due to the rapid growth of smart devices, security and privacy concerns, network infrastructure issues affecting performance, connectivity limitations with centralized systems, and high energy consumption. Task offloading has emerged as a solution to improve efficiency and address these challenges (Heidari et al., 2020).

2.1.2. Overview of Edge Computing

As data flows in from various sources worldwide, edge technology is transforming how it is stored, processed, and distributed. The rise of Internet of Things (IoT) and new applications demanding real-time computing power are driving the development of edge computing platforms. Systems like 5G wireless are enabling real-time applications such as video processing and analytics, self-driving cars, artificial intelligence, and robotics to be developed and supported more efficiently (Carvalho et al., 2021).

Edge computing brings computation and data storage closer to the point of collection, rather than relying on remote locations. This reduces latency issues, especially for real-time data processing, ensuring better application performance. The amount of data sent to centralized or cloud-based systems is minimized, leading to cost savings for businesses by handling processing locally. The proliferation of internet-connected IoT devices, which either receive information from the cloud or send data back, has contributed to this shift (Hasan & Idrees, 2024).

2.1.2.1. Architecture of Edge Computing-Based IoT

The three-tier architecture, shown in figure 2-1, was used by author Vijayasekaran, (2023) and consists of Thing, edge, and cloud layers. The roles and interactions of each layer in the architecture are described below:

(i) Thing Layer

Also known as the user layer consists of end devices such as sensors, surveillance cameras, and smart devices. These devices generate and collect diverse types of data, which may be processed locally within the same layer, at the edge layer, or in the cloud, depending on Quality of Service (QoS) and Quality of Experience (QoE) requirements. End devices may be either static or dynamic, depending on the application. Managing resources for dynamic devices presents greater challenges compared to static ones due to their varying conditions (Vijayasekaran, 2023).

(ii) Edge layer

The edge layer, serving as the middle tier in the three-tier architecture, acts as a bridge between the thing layer and the cloud layer. It includes components like edge servers, controllers, cellular towers, and gateways. Devices in the thing-layer connect to the edge layer through various communication mediums such as Wi-Fi, LTE, and dedicated short-range networks. The edge layer provides robust computing and storage capabilities for the thing layer. It also features an edge management facility for services such as orchestration, resource scheduling, and task execution. Data streams from thing-layer devices are received, processed, and forwarded by the edge layer. Additional features include privacy protection, process optimization, real-time control, intelligent sensing, and data analysis. It enhances load balancing, reduces energy consumption, and optimizes bandwidth utilization, communication, and computational overhead by offloading some cloud services to the edge layer (Vijayasekaran, 2023).

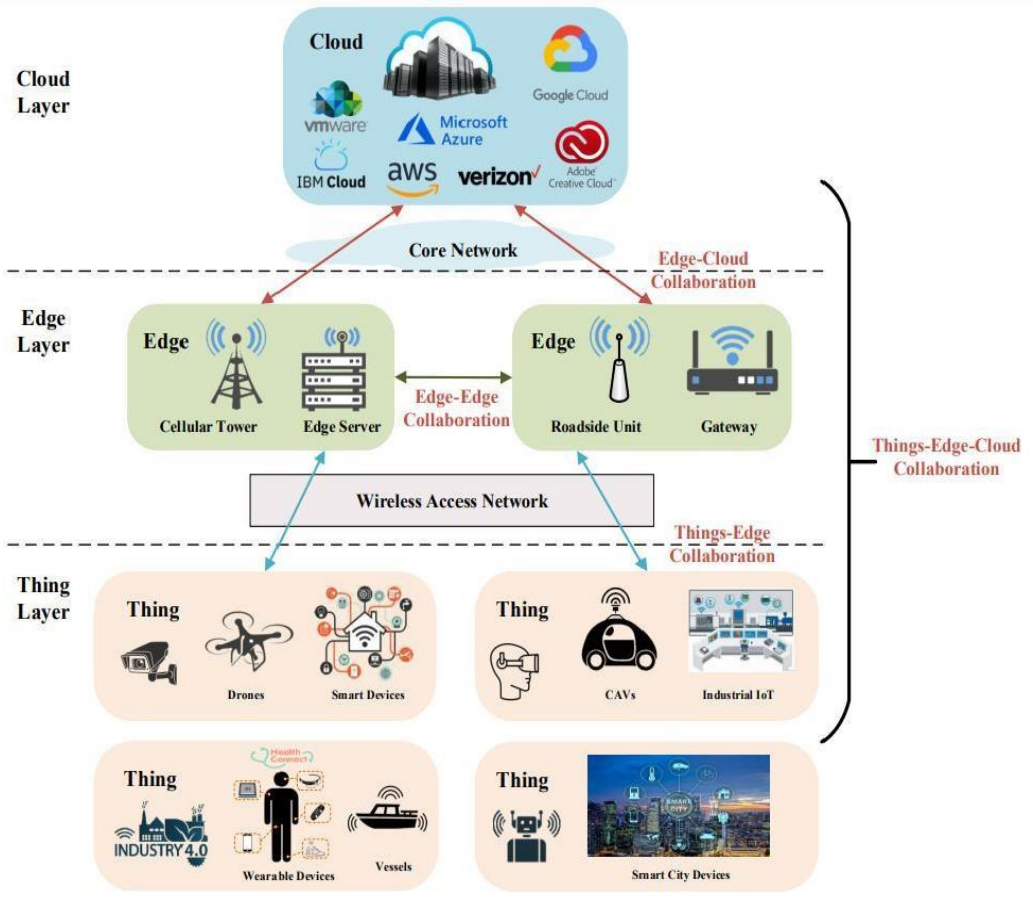


Figure 2-1: Three-tier edge computing architecture
(Source adapted from: (Vijayasekaran, 2023))

(iii) Cloud layer

In the architecture it is placed at the topmost level and provides advanced computing resources, including storage units, data centers, and processing infrastructure, to support the edge layer. While edge servers can handle significant data processing and reduce latency and energy consumption, the cloud is indispensable for managing complex tasks requiring high computational power. The cloud layer delivers a wide range of services, including decision support systems, networking collaboration, personalized services, intelligent production, and service extensions. These services enhance the overall functionality of the edge and thing-layers, ensuring efficient performance and scalability (Vijayasekaran, 2023).

2.1.2.2. Cloud vs edge Computing

The evolution from cloud computing to edge computing represents a significant shift in the way data processing and storage are handled. Cloud computing involves centralized data centers that offer vast computational resources, facilitating data storage and processing over the internet. In contrast, edge computing decentralizes the processing power by bringing computational resources closer to the data source. This reduces latency, enhances data privacy, and optimizes bandwidth usage. Edge computing is particularly beneficial for IoT applications that demand low latency and real-time analytics, offering a complementary solution to cloud computing rather than a replacement. Table 2-1 presents comparison of these focusing on some important aspects.

Table 2-1: Comparison of Cloud and Edge Computing

Aspect	**Cloud Computing**	**Edge Computing**
Architecture	centrally located data centers	Decentralized, with resources closer to the data source.
Resource Availability	Offers vast computational resources for large-scale processing and storage.	Limited resources localized near the edge devices.
Latency	Higher latency due to the physical distance between data centers and end-users.	Low latency as happens near to the data source.
Bandwidth Usage	Requires significant bandwidth to transmit large volumes of data to centralized data centers.	Optimizes bandwidth by processing data locally before transferring only essential information.
Privacy	Data is often transmitted over the internet, raising potential privacy concerns.	Enhanced privacy by keeping sensitive data closer to its source.
Suitability	Ideal for applications requiring extensive computational resources but not latency-sensitive tasks.	Suited for real-time applications such as IoT, autonomous vehicles, and smart cities.
Role in IoT	Acts as a back-end for extensive analytics and storage.	Provides real-time analytics and processing for latency-sensitive IoT applications.
Limitations	Prone to latency and bandwidth	Limited computational capacity;

	constraints; dependent on stable internet connectivity.	may require collaboration with cloud for complex tasks.
Complementary Nature	Provides large-scale processing capabilities to support edge computing when local resources are insufficient.	Complements cloud computing by handling latency-sensitive tasks locally.

2.1.3. Task Offloading in Internet of Things

Computation or task offloading enables IoT devices to send resource-intensive workloads to edge servers, reducing device energy consumption and latency as well as extending operational lifetime. Numerous Internet of Things devices, including sensors and tiny embedded systems, are limited by their energy and processing capabilities. These restrictions make it more difficult for them to process data locally and carry out intricate computations (Heidari et al., 2020).

As IoT applications become more complex, tasks requiring real-time analytics, such as augmented reality, image recognition, self-driving cars, industrial automation and other computationally intensive tasks, the capabilities of resource-constrained IoT devices are often exceeded (Heidari et al., 2020).

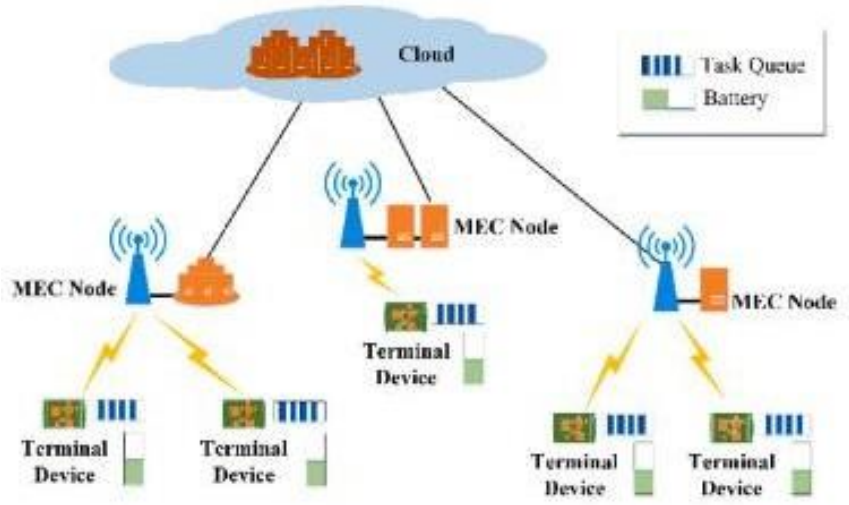


Figure 2-2: Logic diagram of IoT task offloading in MEC
(Source adapted from: (Heidari et al., 2020))

Although, latency can be greatly decreased by offloading jobs to more potent edge or cloud servers, wireless network connectivity and Multi-access Edge Computing (MEC) have limited processing capacity. There are still issues with terminal devices (TDs) while offloading tasks (Filali et al., 2020).

2.1.4. Workload Prediction Techniques

Time series prediction methods can be broadly classified into three major categories: statistical approaches, machine learning (ML) approaches, and deep learning (DL) approaches (Gao et al., 2020). Traditional statistical approaches, such as the Autoregressive Integrated Moving Average (ARIMA) and Vector Autoregression (VAR), have long been used for time series prediction tasks. However, these techniques frequently have limitations in their capacity to capture nonlinear relationships and long-term dependencies that are common in real-world applications (Devi & Valli, 2022). The purpose of this section is to provide an overview of the latter two methods and to highlight their respective strengths and limitations. In this section we focus on some ML and DL techniques and their comparison.

2.1.4.1. Machine learning based methods

Machine learning has gained significant attention among researchers for addressing time series prediction challenges (Taheri-abed et al., 2023). Machine learning techniques are particularly effective for developing offloading predictive models, as they analyze these variables to predict optimal task delegation for better performance and resource utilization (Acheampong et al., 2022). For example, Workload estimates have made extensive use of machine learning-based techniques like k-nearest neighbors, random forests, decision trees, and support vector machines. These techniques have ability to capture complex patterns, handling high-dimensional data, adapting to changing trends and scaling to large datasets. However, ML approaches also present certain challenges: they typically need huge volumes of training data, are prone to overfitting, demand significant computational resources, and often involve lengthy training processes (Jin et al., 2022).

2.1.4.2. Deep Learning based methods

Deep learning architectures learn hierarchical temporal features directly from raw sequences, improving generalization to nonlinear dynamics and long-range dependencies. They are particularly useful to process sequential or time-dependent data. Some of these are discussed as follows.

a. 1D Convolutional Neural Network (1D-CNN): 1D-CNN works well at capturing local patterns and extracting salient features, and while widely used in image and object classification; they have also been employed for analyzing time series data. It processes sequential data by applying convolutional operations through filters (or kernels) that slide along the time dimension, uncovering meaningful local patterns and features. A 1D-CNN's main advantages are: (i) To extract features: The convolutional layers automatically extract important features from the time series data, with deeper layers capturing more complex structures and initial layers concentrating on simple patterns; (ii) Temporal dependencies: By performing convolutions along the temporal axis, the model efficiently detects relationships and correlations over time; (iii) Down-sampling: Pooling layers condense the data representation, summarizing important features and bolstering the model's resilience. Architecture of this model is depicted in figure 2-3 below.

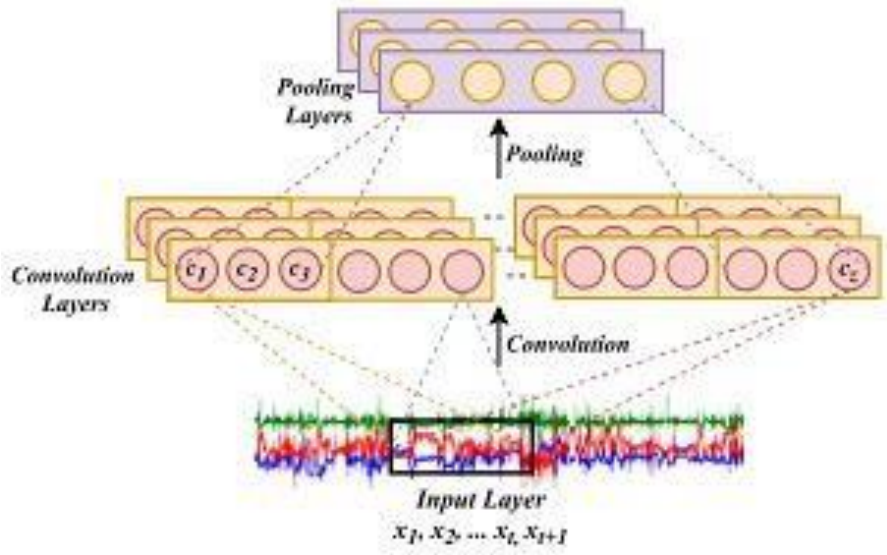


Figure 2-3: An illustration of a 1D-CNN model
(Source adapted from: (Ihianle et al., 2020))

b. Recurrent Neural Networks (RNNs): RNNs are designed for modeling sequential data by using internal memory states, where each output depends on the current input and prior computations. This structure allows RNNs to retain context and predict outcomes by analyzing patterns in input sequences. Commonly used in speech recognition, predicting, and virtual assistants, RNNs rely on the tanh activation function and integrate inputs with hidden states to refine predictions. However, they face challenges like vanishing and exploding gradients during extended training, which can hinder performance on long sequences. While techniques like gradient clipping address exploding gradients, the vanishing gradient issue and reliance on tanh and ReLU activation functions make processing long inputs slow and less efficient (Pandiyan, G, Sasikala, 2023).

c. Long Short-Term Memory (LSTM): LSTM networks are a specialized type of RNN designed to effectively learn temporal sequences and manage long-range dependencies. They achieve this through three core features: memory cells, gated mechanisms (forget, input, and output gates), and a recurrent structure shown in the figure 2-4. LSTM offers several advantages: (i) Long-term dependency handling: It retains information over extended time periods, making it ideal for context-sensitive tasks; (ii) Vanishing gradient mitigation: Its gate architecture ensures smooth gradient flow, overcoming the limitations of traditional RNNs; (iii) Stateful processing: LSTM can retain states across sequences, making it well-suited for applications like speech recognition and video analysis (Hochreiter, S., & Schmidhuber, 1997).

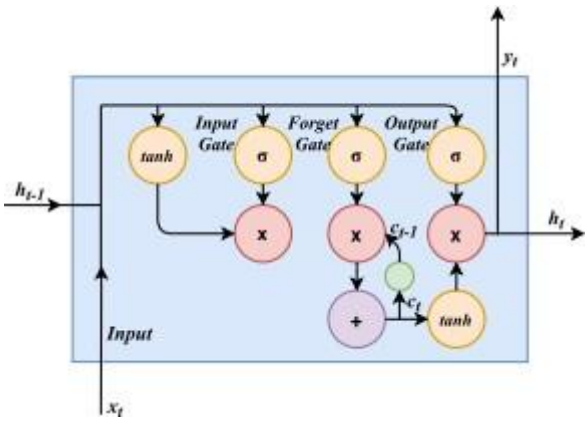


Figure 2-4: An LSTM cell structure
(Source adapted from: (Ihianle et al., 2020))

d. Bidirectional LSTM (BiLSTM): Another deep learning model called BiLSTM was created to find patterns in sequential data by utilizing two hidden layers. This model processes both past and future data, allowing for more precise predictions of future outcomes. It has been used in offload prediction by researchers including Pandiyan, G, Sasikala, (2023). Bidirectional LSTM enhances the standard LSTM by incorporating two parallel LSTM layers that process data in both forward and backward directions. This dual-loop architecture, depicted in Figure 5, allows the model to use information from the past and the future to make predictions. By doing so, it establishes dependencies between the current data point, prior inputs, and future inputs. In Figure 2-5, the forward sequence ($h^{\rightarrow}$) and the backward sequence ($h^{\leftarrow}$) are visually represented by red and green arrows, respectively (Ihianle et al., 2020).

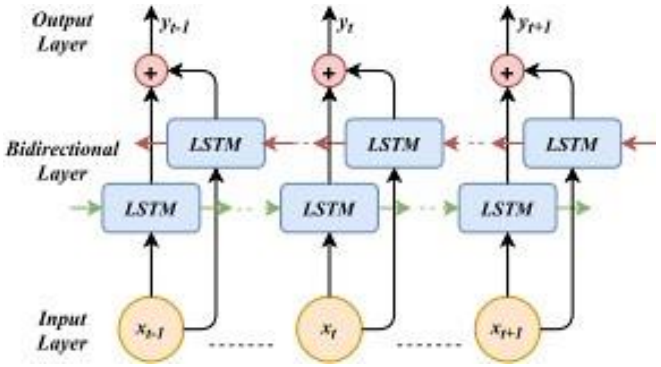


Figure 2-5: BiLSTM model
(Source adapted from: (Ihianle et al., 2020))

e. Deep Neural Network (DNN): DNNs have become a cornerstone of artificial intelligence, gaining significant advances in areas such as computer vision, natural language processing, and intelligent control systems. Drawing inspiration from the way the human brain is structured, DNNs comprise several levels of interrelated computational units (neurons) that progressively learn to identify complex patterns and construct hierarchical representations of data. Their transformative success has been made possible by the convergence of several factors: the availability of large-scale datasets, the exponential growth of computational power, and continual algorithmic innovations. Collectively, these developments have enabled DNNs not only to tackle highly complex tasks but also to expand the boundaries of what machines can achieve in real-world applications (Singh & Charankar, 2024).

f. Gated Recurrent Unit (GRU): LSTM networks, which were created to overcome the drawbacks of conventional RNNs, especially the vanishing gradient issue, are simplified into GRUs. Because of their simplified architecture, GRUs are computationally more efficient while yet retaining the capacity to identify long-term dependencies in sequential data. The update gate (z), which determines how much of the previous concealed state should be kept for the next time step and the reset gate, which determines how much of the previous hidden state should be forgotten, are the two main gates that GRUs utilize to regulate the information flow GRUs are widely used in various applications involving sequential data, including natural language processing, speech recognition and time-series analysis (Kirori & Ireri, 2020).

Table 2-2: Comparison of DL Models

Categories	**CNN**	**DNN**	**LSTM**	**GRU**	**BiLSTM**
Input/Output Structure	Fixed input/output sizes	Input size may vary, output fixed by final layer	Input/output sizes may vary depending on time steps	Input/output sizes may vary depending on time steps	Similar to LSTM, but processes sequences bidirectionally
Model Architecture	Convolutional and pooling layers	Fully connected layers (non-sequential)	Memory cells with input, forget, and output gates	Update and reset gates; simpler than LSTM	Dual LSTM layers: one forward, one backward
Data Preprocessing Needs	Moderate to high (sequence shaping)	Lower; mainly depends on network depth	Requires sequence preparation and padding	Requires sequence preparation; slightly simpler than LSTM	Higher; requires sequence shaping in both directions
Use	To learn local spatial/temporal patterns	Non-sequential data or post-CNN input	for long-term dependencies	Efficient modeling of sequential data with fewer parameters	When both past and future context are critical for prediction

2.1.4.3. Hybrid DL in Time Series Predictions

Adapting to a fixed workload sequence is usually manageable for forecasting framework built on a single predictive model. However, in real-world scenarios, where workload patterns fluctuate rapidly over time, this becomes much more challenging. Such variability often persists even under conditions of over-provisioning or resource scarcity. Hybrid deep learning architectures combining CNN and RNN have emerged as powerful solutions in such multivariate time series predictions. These models use the complementary strengths of different neural components, enabling them to simultaneously capture local temporal features, long-range dependencies, and structural trends within complex datasets (Wang et al., 2020).

The taxonomy in Figure 2-7, adapted from Sarker, (2021), categorizes deep learning models into three major groups: (i) Discriminative models such as MLPs, CNNs, and RNNs with variants like LSTM, BiLSTM, and GRU, (ii) Generative models like GANs, and VAEs, and (iii) Hybrid models that combine convolutional or temporal modules with recurrent architectures. Within the hybrid family, two important branches are highlighted: CNN-RNN hybrids and TCN-RNN hybrids. This classification shows the growing importance of hybrid models in modern time series predicting, as they are increasingly designed to capture both short-term local patterns and long-range sequential dependencies in complex data streams (Sarker, 2021).

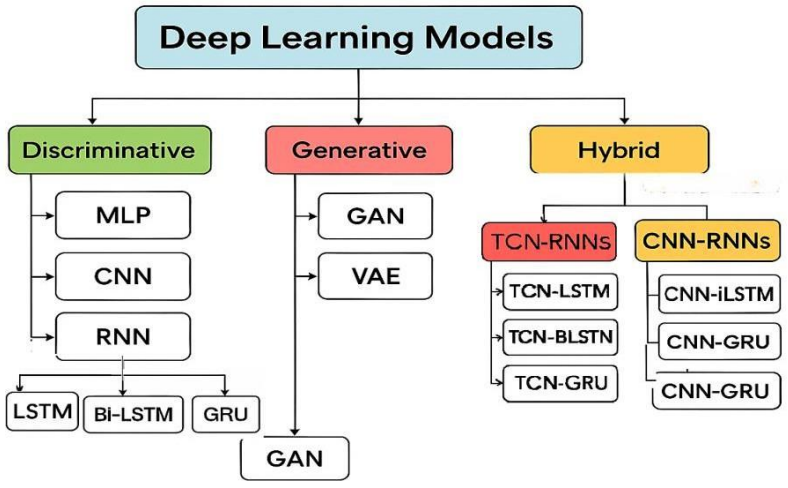


Figure 2-6: Some Deep Learning Models and Their Hybrids
(Adopted from taxonomy by author Sarker, (2021))

The first branch of hybrid models, CNN-RNN hybrids, combines the sequence modeling capability of RNNs with the feature extraction capability of CNNs. Initially, CNN layers are employed to detect transient local characteristics like bursts, fluctuations, or micro-trends, which are then passed into recurrent layers to capture longer-term dependencies. CNN-LSTM models are effective at learning long-term patterns in noisy or non-stationary time series because the CNN filters reduce noise before LSTM layers process sequential information. CNN-BiLSTM models extend this approach by processing inputs in both forward and backward directions, providing richer temporal context and stronger predictive accuracy. CNN-GRU hybrids reduce computational complexity and training cost while maintaining strong performance, making them particularly suitable for applications in finance, sensor-based activity recognition, and many more other real-world scenarios (Liu & Wang, 2024; Sarker, 2021; Wang et al., 2020).

The second branch, TCN-RNN hybrids, integrates Temporal Convolutional Networks (TCNs) with recurrent layers. TCNs use dilated causal convolutions and residual connections, which allow them to model long-range temporal dependencies while maintaining stable gradient flow in deep networks. When combined with recurrent units such as LSTM, BiLSTM, or GRU, these models can capture both short-term fluctuations and multi-scale temporal structures. For example, TCN-LSTM models are strong at handling periodicity and long-term dependencies, while TCN-BiLSTM architectures further improve modeling power by incorporating bidirectional memory. TCN-GRU hybrids are useful for large-scale applications because they strike a balance between computing efficiency and precision such as energy load predicting and traffic prediction (Dogani et al., 2023; Sarker, 2021).

Empirical studies provide strong evidence for the effectiveness of these hybrid models. Sarker, (2021), for instance, benchmarked multiple hybrid architectures on traffic volume and air quality datasets. In the paper, CNN-BiLSTM and TCN-LSTM also performed strongly, further validating the benefits of combining convolutional and recurrent layers. Overall, hybrid approaches that integrate CNNs, RNNs, and TCNs represent a significant step forward in multivariate time series predicting, consistently outperforming both single-module deep learning models and conventional statistical techniques (Liu & Wang, 2024; Sarker, 2021; X. Wang et al., 2020).

While hybrid models have made significant progress in time series predicting, they often assume that all time steps and input features are equally important, which isn't how real-world data behaves. In practice, some moments and variables carry more weight than others, and that's where attention mechanisms come in. These mechanisms help models focus on the most relevant parts of the input by assigning dynamic weights, improving both accuracy and interpretability. The model of the attention mechanism is illustrated in Figure 2-8.

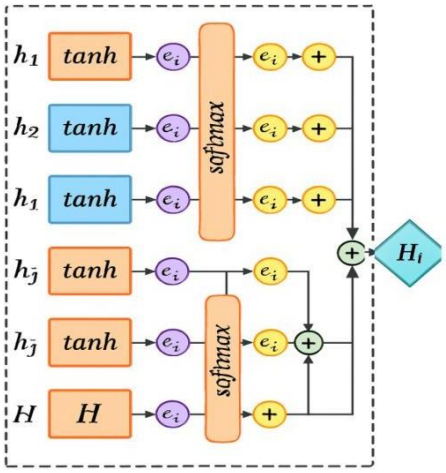


Figure 2-7: Attention Mechanism Model

In deep learning, attention techniques have become essential, especially for tasks that call for modeling contextual relevance and long-range relationships. At their core, attention mechanisms compute a weighted representation of input elements by assigning scores that reflect their relative importance to a given task or query. This dynamic weighting allows models to selectively focus on salient features while suppressing irrelevant ones, thereby enhancing interpretability and performance. Originally introduced in the Transformer architecture for NLP tasks, attention has since been adapted for time series predicting, enabling models to better capture temporal heterogeneity and cross-feature interactions in multivariate data streams (Dogani et al., 2023).

2.1.5. Dataset Description

The majority of the data utilized in the work are specifically generated based on the experimental environment conditions, and there aren't many publicly available datasets, according to the survey results conducted by the researchers Dayong et al., (2024). Few studies have used real-world data to assess and validate their suggested algorithms and methods. While some of these datasets offer computing tasks, others include trajectories

for motion simulation of TDs. Otherwise, the majority of studies create unique test data sets and configure their simulation parameters (Dayong et al., 2024).

There are many datasets available, such as the Azure dataset, Alibaba, Bitbrains and Google cluster dataset which are often used in modeling edge-cloud computing environments. In this section we examined these datasets and summarized in Table 2-3.

- Google Cluster 2011: An earlier version of the Google traces, widely used in scheduling research. However, it is now lacks the richer features as the updated workload patterns present in the 2019 dataset. Hence, according to the work by Lackinger, (2023), several researchers were using the newer version.
- A sizable, real-world dataset gathered from Alibaba's production cluster environment is the Alibaba Cluster Trace 2018 dataset (Chen et al., 2018). It is cluster-trace-v2018 of Alibaba that recording the traces in 2018. Cluster-trace-v2018 includes about 4,000 machines in a period of 8 days. The main features of the dataset include Job IDs, task IDs, start/end times, CPU and memory usage, machine IDs, and job types. While the Alibaba dataset is valuable for studying large-scale scheduling and resource allocation, it was not chosen as the primary dataset for this study because: It lacks certain categorical scheduling attributes priority and delay tolerance essential for our classification targets.
- Azure VM Traces: Contains VM allocation and usage data from Microsoft Azure. While useful for VM-level resource prediction, it does not provide task-level scheduling details or categorical priority labels, making it less suitable for our hybrid classification-regression modeling.
- A distributed data center run by Bitbrains, a managed hosting provider that specializes in enterprise-grade computing, is the source of the Bitbrains GWA-T-12 dataset, which comprises performance metrics of 1,750 virtual machines (VMs) hosted in a distributed datacenter by Bitbrains, a service provider that specializes in managed hosting and business computation for enterprises like major banks, credit card operators, and insurers. Each file in the dataset contains performance metrics of a single VM, which are categorized into two traces: fastStorage and Rnd There are 1,250 virtual machines (VMs) linked to high-performance storage area network (SAN) devices in the fastStorage trace, and 500 VMs linked to either slower Network Attached Storage (NAS) devices or fast SAN devices in the Rnd trace.

While the Rnd trace has more administration machines that need lower-performance storage and less frequent access, the fastStorage trace has a larger percentage of application servers and compute nodes. The dataset includes key features such as CPU usage (in MHz and %), memory usage (KB), disk read/write throughput (KB/s), and network I/O (KB/s), along with provisioning details such as CPU cores and memory capacity (Di et al., 2012; Leka et al., 2023; Shen et al., 2015). But, the dataset doesn't contain certain scheduling attributes priority and delay tolerance essential for our classification targets.

- Synthetic IoT Workload Datasets: Several studies have generated synthetic IoT workload traces for simulation purposes. However, these datasets often fail to capture the complexity and variability of real-world workloads, leading to models that may not generalize well to production environments.
- The Google Cluster Trace v3, often referred to as ClusterData2019, is a publicly accessible dataset that includes production-level workload traces from eight compute cells under Borg management for the month of May 2019. The dataset is structured using protocol buffers and is accessible via BigQuery (Wilkes et al., 2020). We selected this dataset because our baseline papers recommended and proved its validity. In addition to this, the following characteristics make the dataset highly relevant for research:

    - A range of heterogeneous workloads is included, including latency-sensitive, batch, and mixed-priority tasks, closely resembling the diversity of IoT edge computing scenarios (Wilkes et al., 2020).
    - The dataset contains fine-grained temporal logs of task submissions, scheduling decisions, and resource consumptions, enabling both categorical classification and continuous regression targets.
    - More than 34 features are available in JSON format, such as Task Priority, Scheduling Class, CPU Request, Memory Request, Total CPU Usage, and Total Memory Usage. The dataset contains millions of task instances over 31 days (Wilkes et al., 2020; Yildiz & Baiocchi, 2024).
    - Furthermore, the multi-tenant nature of the cluster introduces realistic contention and interference patterns (Shahmirzadi et al., 2024; Yildiz & Baiocchi, 2024).

Table 2-3: Dataset Comparisons

Dataset	**Targets / Metrics Included**	**Scale**	**Attributes**	**Format**	**Environment Type**
Google Cluster 2011	CPU, Mem usage, scheduling	Large network	30	CSV	Cloud
Google Cluster 2019	CPU, Mem usage, scheduling	Large network	34	BigQuerytables	Cloud
Alibaba Cluster Trace 2018	CPU, Mem usage, job events	Large network	28	CSV	Cloud
Bitbrains (2013-8)	CPU, memory, disk, network	Large network	12	CSV	Cloud
Azure VM Traces	VM-level CPU, Mem usage	Large network	20	CSV	Cloud
Synthetic IoT Workloads	CPU, Mem usage, latency	Small network	Varies	CSV	Simulation

2.2. Related Work

In recent years, a number of studies have contributed to the workload predictions. In this section, some of the relevant studies were reviewed and discussed as follows.

In the study (Chen et al., 2022), a sequence-guided attention-based model (SG-CBA) was proposed for predicting CPU utilization by combining CNN, BiLSTM, and an attention mechanism. This suggested method was tested on the Alibaba 2018 cluster trace dataset, achieved promising results, with MSE of 0.001, R^2 of 0.94, and MAE of 0.024. Despite these results, no task offloading request prediction was incorporated, the study primarily focused single resource utilization only, and a relatively small dataset was employed.

For effective edge computing in the IoT, Tu et al., (2022) suggested task offloading via deep reinforcement learning (DRL) combined with LSTM prediction. Their model used LSTM to anticipate future task characteristics, such as task data size and arrival time, while the DRL module dynamically decided how to offload tasks based on the predicted information. However, their work did not incorporate other crucial task-level semantics such as priority and delay tolerance, and without explicitly forecasting computational resource needs.

To forecast CPU workload in cloud data centers, Karim et al., (2021) presented BHyPreC, a hybrid deep learning model combining 1D-CNN, Bi-LSTM, stacked LSTM, and GRU layers. Tested on Bitbrain traces, the model achieved MSE of 0.00051, RMSE of 0.02, MAE of 0.002, and MAPE of 10.78. Nevertheless, the scope of the study was limited to a single resource parameter (CPU workload), and offloading request prediction was not incorporated into the proposed framework.

In the study (Pandiyan, G, Sasikala, 2023) proposed a BiLSTM-based prediction model for offloading decisions, predicting both device task requirements and server load. The framework integrates Bi-LSTM for task requirements and server load prediction, deep reinforcement learning (DRL) for adaptive decision-making, and a TOPSIS-based ranking mechanism for server selection. Despite these contributions, the study relied heavily on Bi-LSTM prediction, which introduced significant training overheads. Furthermore, the framework did not incorporate advanced feature extraction mechanisms and multi-resource prediction.

The work by Sun et al., (2023) proposed a flexible offloading and task scheduling scheme (FLOATS) designed to adaptively optimize computation offloading decisions and scheduling priority sequences for time-dependent tasks in dynamic networks. Although FLOATS adaptively optimized offloading decisions, it did not incorporate prediction task offloading requests and resource consumption.

Researchers Sohaib et al., (2024) proposed a hybrid online-offline learning framework for task offloading in mobile edge computing systems. Their approach reduced computation delay and energy consumption by adjusting offloading policies based on network dynamics and queuing status at MEC servers. While they introduced the concept of delay prediction,

the framework primarily adjusted policies rather than predicting upcoming task offloading requests and required resources in advance.

Researchers Wang et al., (2024) proposed a Two-Stage Offloading Decision-making Framework (TSODF) with request holding and dynamic eviction. Their framework integrated LSTM-based task-offloading request prediction and MEC resource release estimation to infer the probability of a request being accepted in the subsequent time slot. However, the study was evaluated solely on simulated datasets rather than real-world traces, did not explicitly predict essential task offloading parameters such as priority and delay tolerance, and omitted resource demand forecasting, providing only system-level resource utilization predictions.

A multi-phase architecture for MEC request prediction and optimization was proposed in (Nugroho & Kim, 2024). The model initially predicted multi-step task-offloading requests, and the suggested method, OPT(Nsh), then optimized network association, computing resource provisioning, and task offloading. This resulted in effective task processing with reduced battery consumption, deadline guarantees, and less reliance on cloud resources. However, the approach used simulated datasets, no explicit task requests, and resource need predictions. However, the suggested method was evaluated solely on simulated datasets instead of real-world traces, didn't incorporate explicit task request and associated resource demand predictions. Furthermore, the implementation of multi-time-step optimization introduces considerable computational overhead, which may limit its practicality in resource-constrained and large-scale network environments.

Authors Lv et al., (2024) presented two LSTM-based decision-making prediction techniques that encouraged edge devices to participate in task offloading, guaranteed completion of latency-sensitive requests, and enabled predictive decision-making. However, their work only considers the total task volume for prediction, without explicitly incorporating task-level semantics and specific resource consumption requirements.

Finally, Lackinger, (2023) employed transformer and LSTM techniques to provide precise time-series predictions of CPU utilization in Google cloud data centers. Using the Google cluster dataset, they achieved MAE of 0.03, MSE of 0.018, and R^2=0.91. However, task offloading requests prediction was not studied, focusing on single-resource prediction.

2.2.1. Summary of Related Works and Research Gaps

There has been many works dedicated to the computation offloading decision optimization problems in edge computing environments. Table 2-3 summarizes the key related works, identified research gaps, and the evaluation metrics and performance achievements reported in predicting task offloading and resource consumption in IoT edge computing.

From the table, we can see that even though many previous studies tried different deep learning models, most of them did not combine feature extraction with their prediction methods. They also didn't try to predict both task offloading requests and the need for computing resources at the same time. Even substantial portion of existing study uses univariate time series prediction focusing on single resources such as CPU usages. On top of that, many of these studies didn't test their models well using real-world data from different sources. These gaps show that there's a need for a better and more complete prediction system and that's exactly what this study is trying to develop.

Table 2-4: Summary of Related Works

Authors (Year)	Model / Method	Dataset Used	Target Resource Metrics	Evaluation Metrics & Reported Performance	Gaps or drawbacks
Karim et al., (2021)	BHyPreC (1D-CNN + Bi-LSTM, stacked LSTM, GRU)	Bitbrains (GWA-T-12) traces	CPU utilization only	MSE = 0.000507; RMSE = 0.02; MAE = 0.002508; MAPE ≈ 10.77%	• Did not predict task features (such as priority, delay sensitivity) • focused on single resource
Yadav et al., (2021)	LSTM	logs of University Madrid	CPU only	MAE=0.043 MSE=0.006 RMSE=0.075	• Univariate CPU only • no joint task-request prediction;
Chen et al., (2022)	SG-CBA (CNN + Bi-LSTM+ Attention)	Alibaba Cluster 2018	CPU utilization only	MSE=0.0012 R^2=0.9485 MAE=0.024	• Single-resource • No task request prediction • Small dataset
Xu et al., (2022)	esDNN (Revised GRU)	Alibaba and Google cluster	Multi-variate (CPU, memory)	MSE=0.00184 RMSE=0.013 MAPE=9.78	• no task request prediction • no joint task offloading-resource requests prediction

Devi & Valli, (2022)	ARIMA / SARIMA / Holt-Winters / LSTM (+ scheduling)	Google trace and BitBrain	Multi-variate (CPU, memory, disk I/O)	MAE= 0.37 RMSE=0.49 R^2=0.93	• statistical emphasis • no joint task offloading-resource requests prediction
Jeong et al., (2023)	ProHPA (Attention-BiLSTM)	Kubernetes micro-service traces	Multi-variate (CPU, memory)	MAE=0.0066 MSE=0.0001 RMSE=0.0108 R^2=0.947	• No joint task-request-resource need prediction • No feature engineering.
Patel & Bedi, (2023)	MAG-D (Multivariate Attention + GRU)	GCD	Multi-variate (CPU, memory)	MAE=0.004 RMSE=0.0054	• no joint task-request-resource need prediction • no feature engineering
Lackinger, (2023)	LSTM, Transformer	Google Cluster	CPU only	MAE= 0.03 MSE=0.018 R^2=0.91	• Single-resource prediction • No task request prediction

After reviewing the existing research on IoT task offloading and resource demand prediction, we noticed several limitations that are still not well addressed, as summarized in the above table. Many studies have used deep learning models like CNNs, LSTMs, or hybrid versions of these. However, most of these works focused on predicting just one type of outcome usually things like CPU usage or the number of incoming tasks. They didn't try to predict more detailed task-related characteristics such as priority, and delay tolerance, even though these features are essential for understanding the urgency and behavior of tasks in edge computing environments.

Moreover, only a few researchers explored multi-variate predicting, and even, they usually focused on one resource (like CPU) without looking at other key demands such as memory. Another major limitation is that many models were trained on synthetic or simulated datasets. This can lead to less reliable results, since simulated data doesn't capture the unpredictable and diverse nature of actual IoT environments. In addition to that, most studies ignored deeper time-series analysis techniques which can reveal important relationships between variables over time.

In general, joint predictions of task offloading requests with their associated resource demands were not investigated in the previous studies. Our work addresses all of these gaps by proposing a hybrid deep learning model that is trained and tested on real-world dataset from Google Cluster.

2.3 Baseline Papers

In recent years, many deep learning models have been developed to predict workloads in cloud and edge data centers. In this thesis, the works by Lackinger, (2023) and Chen et al., (2022) are considered as baseline studies. Lackinger, (2023) employed different approaches including LSTM and transformer for accurate time series predictions. They used the Google cluster dataset to predict CPU usage in Google cloud data centers and achieved MAE=0.03, MSE=0.018 and R^2=0.91. Similarly, Chen et al., (2022) introduced a SG-CBA model using CNN, Bi-LSTM, and attention mechanisms. They tested it on the Alibaba 2018 cluster dataset and also achieved good results, such as MSE of 0.0012 and R^2 of 0.94. However, both studies only focused on single resource usage prediction, CPU usage, without considering task request prediction and other necessary IoT edge resources such as memory.

CHAPTER THREE

3. RESEARCH METHODOLOGY

3.1. Introduction

The research methodology we used to accomplish our proposed objectives is presented in this chapter. It includes a discussion of the research method and approach, description of the datasets, data preprocessing, feature engineering, feature selection, model selection, and evaluation metrics we used in detail. To investigate explanatory research problems, we also attempted to describe the methodologies or models that we used in the proposed solution to predict the offloading requests and resource demands.

3.2. Research Design

This study aims to predict the features of offloading tasks and the associated resource needs in IoT edge environments. This starts from domain understanding to the final analysis stage. The overall workflow is listed below and shown in Figure 3-1.

- **Domain Understanding**

  The initial phase of our research involves understanding the background by identifying the task offloading decisions and resource usage challenges in IoT edge computing systems. In the first step we set the scope of our study within the broader field of distributed systems and edge computing. After a detailed comparison, we focused on one of these thematic areas and continued with the rest steps.

- **Literature Review**

  A detailed literature review was conducted to examine previous research related to our problem domain on IoT task offloading requests, resource demands, and workload prediction using deep learning models. We surveyed academic articles, peer-reviewed journals, and conference proceedings relevant to our study domain. This helped us assess what methods had been used before, which datasets were commonly used, and what types of prediction targets and features were considered in prior studies. Therefore, we could summarize previous methods, models, datasets, and evaluation strategies used in similar problems.

- **Research Gap Identification**

  In this step we analyzed gaps in the previous studies, and identified problems to be solved.

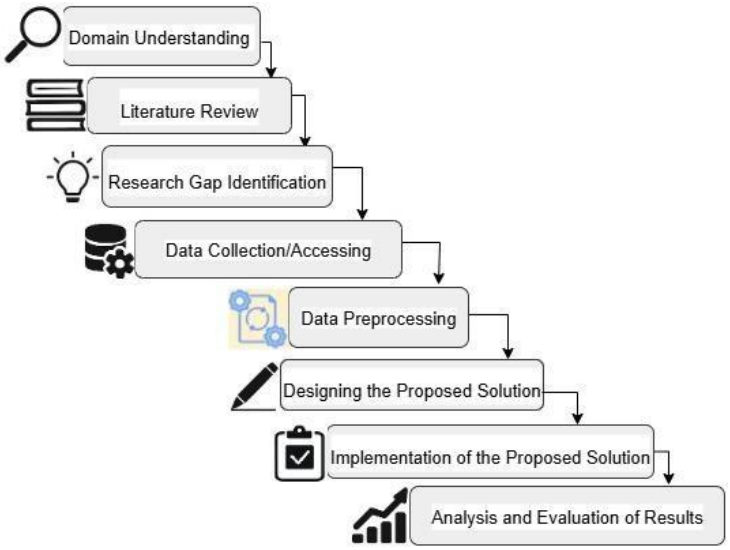


Figure 3-1: General Flow of Research Methodology

- **Designing the Proposed Solution**

  In this step, we designed a solution to address the gaps we identified during our review of related works. The design of our model was supported by data preprocessing pipeline, which involves cleaning raw time-series data and normalizations. Feature engineering and selection were also considered in this stage.

- **Implementation of the Proposed Solution**

  At this stage, we used different tools and methods to convert design to model. The proposed model training, testing, and evaluation processes were carried out using Google Colab, which provides a cloud-based environment with access to GPU acceleration and facilitated easy integration of datasets and model components. Python served as the primary programming language. Libraries such as TensorFlow, Keras, Scikit-learn, Pandas, and NumPy were also used for building the model, preprocessing the datasets, and analyzing prediction results.

- **Analysis and Evaluation of Results**

  After implementing the model, we evaluated its performance using appropriate metrics. At this stage, we also benchmarked and compared the performance of the proposed model with the baseline models.

33. Dataset Sources and Descriptions

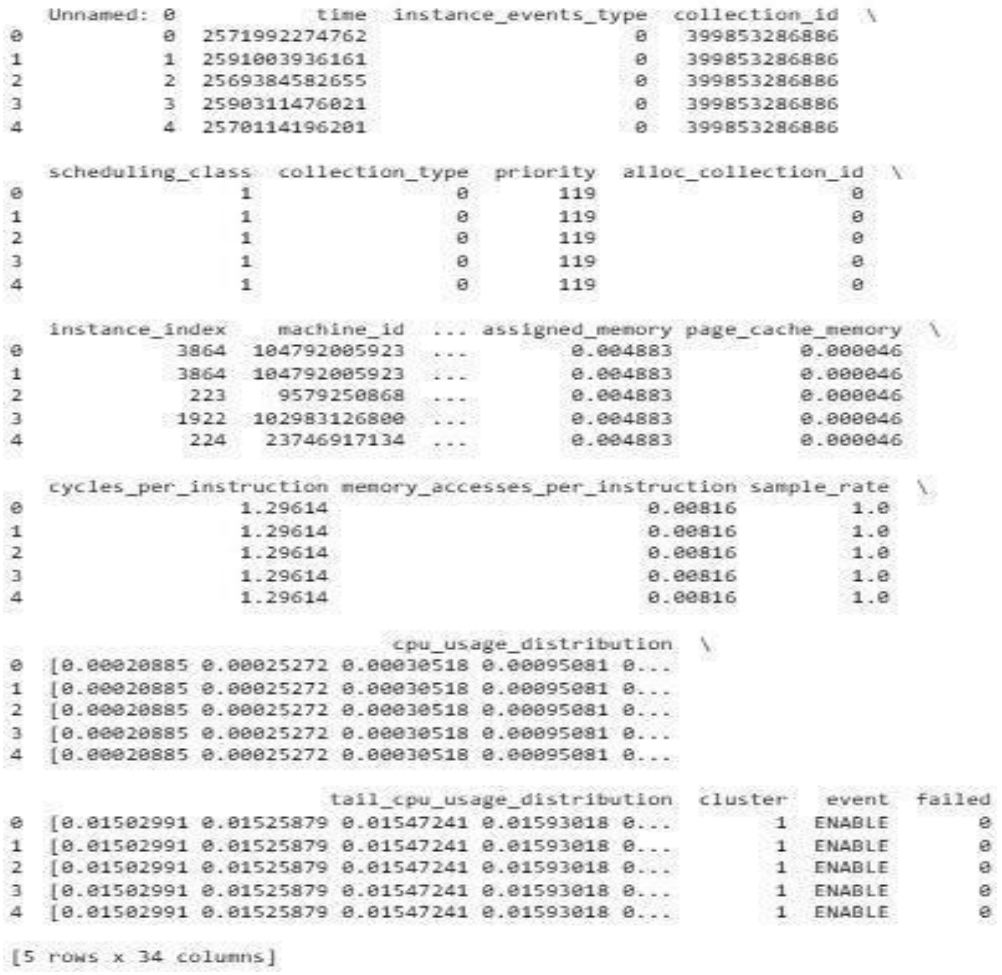
For evaluating the proposed method, selecting an appropriate dataset is necessary. Such datasets may either be generated from scratch or obtained from publicly available sources. Key considerations include the dataset's content, size, and scientific relevance. There are

several datasets available such as Alibaba, Bitbrains, Azure and Google cluster datasets. We reviewed the features and schema of each datasets to decide which dataset best suitable for our work. After that we found that the first three datasets were not relevant for our work since they do not contain some features we considered such as priority and delay tolerance. Regarding the Google Cluster dataset, only the 2019 trace was considered relevant, since this version contains significantly the required data for this study. In addition to this, the dataset was also used and proved as valid in our baseline papers. The document (Wilkes, 2020) describes the semantics, data format, and schema of usage traces of describes version 3 of the trace of few Google compute cells.

Google Cluster dataset: A Google cluster is a group of machines organized into racks and connected through a high-bandwidth network. Within a cluster, a cell refers to a collection of machines that operate under the same cluster-management system, which is responsible for assigning work across machines. In Borg, resource requests can take two forms: a job (made up of one or more tasks) or an alloc set (made up of one or more allocs, also called alloc instances). A task is basically a Linux program possibly with multiple processes that runs on a single machine. Jobs define the computations a user wants to execute, while alloc sets define resource reservations where jobs can be executed. Cluster users are typically Google engineers and services, with tasks and jobs scheduled onto machines according to their respective lifecycles. Each trace consists of multiple tables, usually indexed by a primary key that includes a timestamp (Wilkes, 2020).

In order to allow outside academics to study scheduling in massive compute clusters, Google made available a one-month trace from its Borg cluster management system in 2011. Since then, this dataset has been used by several hundred researchers to investigate a variety of phenomena (van Loo et al., 2022). An open research question, however, concerns how workloads in such systems evolve over time and how advancements in cluster management have influenced scheduling decisions. In order to support additional research into these problems, Google released a new trace (Google Cluster, 2019) that offers comprehensive Borg task scheduling data from eight distinct Google compute clusters (Borg cells) over the whole month of May 2019. The contents of this new trace extend and enrich the data originally provided in the 2011 release. The data is available through Google BigQuery due to its large size approximately 2.4 TiB compressed (Wilkes et al., 2020).

(Dataset available at: https://github.com/google/cluster-data).



	Unnamed: 0	time	instance_events_type	collection_id	\
0	0	2571992274762	0	399853286886	
1	1	2591003936161	0	399853286886	
2	2	2569384582655	0	399853286886	
3	3	2590311476021	0	399853286886	
4	4	2570114196201	0	399853286886	

	scheduling_class	collection_type	priority	alloc_collection_id	\
0	1	0	119	0	
1	1	0	119	0	
2	1	0	119	0	
3	1	0	119	0	
4	1	0	119	0	

	instance_index	machine_id	...	assigned_memory	page_cache_memory	\
0	3864	104792005923	...	0.004883	0.000046	
1	3864	104792005923	...	0.004883	0.000046	
2	223	9579250868	...	0.004883	0.000046	
3	1922	102983126800	...	0.004883	0.000046	
4	224	23746917134	...	0.004883	0.000046	

	cycles_per_instruction	memory_accesses_per_instruction	sample_rate	\
0	1.29614	0.00816	1.0	
1	1.29614	0.00816	1.0	
2	1.29614	0.00816	1.0	
3	1.29614	0.00816	1.0	
4	1.29614	0.00816	1.0	

	cpu_usage_distribution	\
0	[0.00020885 0.00025272 0.00030518 0.00095081 0...	
1	[0.00020885 0.00025272 0.00030518 0.00095081 0...	
2	[0.00020885 0.00025272 0.00030518 0.00095081 0...	
3	[0.00020885 0.00025272 0.00030518 0.00095081 0...	
4	[0.00020885 0.00025272 0.00030518 0.00095081 0...	

	tail_cpu_usage_distribution	cluster	event	failed
0	[0.01502991 0.01525879 0.01547241 0.01593018 0...	1	ENABLE	0
1	[0.01502991 0.01525879 0.01547241 0.01593018 0...	1	ENABLE	0
2	[0.01502991 0.01525879 0.01547241 0.01593018 0...	1	ENABLE	0
3	[0.01502991 0.01525879 0.01547241 0.01593018 0...	1	ENABLE	0
4	[0.01502991 0.01525879 0.01547241 0.01593018 0...	1	ENABLE	0

[5 rows x 34 columns]

Figure 3-2: Sample raw Cluster2019 data records

The data is structured across several main tables defined in the protocol buffer format[1]. For our study, we accessed the dataset through BigQuery, specifically from cluster ‘a’. Figure 3-2 illustrates a sample of the raw records obtained from the cluster ‘a’ trace using BigQuery we provided in appendix II.

To summarize the features and the corresponding data types are presented in below table to show all features, types, and their description:

[1] Google, *Cluster Data Trace Format v3*, GitHub repository, available at: https://github.com/google/cluster-data/blob/master/clusterdata_trace_format_v3.proto (Accessed: 23 February 2024).

Table 3-1: Summary of feature descriptions

S. No	Feature Names	Data type	Description
1	time	Int64	Timestamp of the event (in microseconds since epoch).
2	instance_events_type	Int64	Type of event affecting an instance (e.g., START, FINISH, EVICT).
3	scheduling_class	Int64	Classification that influences how the job is scheduled.
4	collection_type	Int64	Indicates if the collection is a job (0) or an allocation set (1).
5	Priority	Int64	Priority level assigned to the collection (higher values = higher priority).
6	instance_index	Int64	Index of the instance within its collection.
7	resource_request	Object	Requested resources (e.g., CPU, memory) for the instance.
8	collections_events_type	Int64	Type of event for the collection (e.g., SUBMIT, FINISH).
9	start_time	Object	Start time of instance/resource usage measurement window.
10	end_time	Int64	End time of instance/resource usage measurement window.
11	average_usage	object	Average CPU/memory usage during the measurement window.
12	maximum_usage	object	Maximum resource usage observed in the window.
13	random_sample_usage	object	Resource usage measured at a randomly sampled point in time.
14	assigned_memory	Int64	Average memory limit assigned to the instance.
15	page_cache_memory	float64	Average memory used for file page caching.
16	cycles_per_instruction	float64	Average number of CPU cycles per instruction executed.
17	memory_accesses_per_instruction	float64	Average number of memory accesses per instruction.
18	sample_rate	float64	Number of samples taken per second during the usage window.
19	cpu_usage_distribution	object	11 percentiles of observed CPU usage (0% to 100%).
20	tail_cpu_usage_distribution	object	9 percentiles of high-end CPU usage (from 91% to 99%).

3.4. Data Preprocessing Techniques

Effective dataset preprocessing is a critical step in preparing raw data for machine learning and deep learning applications. These techniques enhance data quality and consistency, thereby facilitating model training and improving predictive accuracy. The preprocessing pipeline aims to eliminate irrelevant features, standardize input formats, and optimize learning efficiency.

In this study, we used ClusterData2019 dataset, which is available in Google BigTables. We accessed the data using Google BigQuery (appendix II) to extract feasible data size and relevant features from cluster 'a'. Then before model training, we preprocessed the data to make it consistent, clean, and usable for effective model performance and it involves:

- **Data Cleaning:** This involves finding and eliminating mistakes and inconsistencies from the data. This may include filling in missing values, removing features such as such as identifiers and converting data to a consistent format. Many resource-related columns such as CPU and memory usage were stored as dictionary-like strings. These were parsed into structured formats to extract meaningful numerical statistics. Missing values were handled using imputation to avoid future data leakage. The time information in the dataset was originally stored in microseconds, and it was offset by 600 seconds before the actual trace started. After that we converted it to datetime in minutes and segmented the data into consistent time intervals.
- **Input Normalization:** Some columns are normalized by providers. But**,** some features have varied numerical values. Training models directly on unscaled data can lead to prediction errors and extend training durations. To mitigate this, all features were normalized using Min-Max scaling to a common range. This transformation prevents dominant features from skewing the learning process and ensures numerical stability during training.
- **Label Encoding:** Google provided priority values in the ranges of 0 to 450. So, these values were mapped into four tiers to reflect the priority nature in our domain, and scheduling classes were similarly grouped into four levels. Then, the categorical fields encoded using LabelEncoder.
- **Feature Aggregation and Selection:** Job-level features were computed by aggregating task-level metrics. CPU and memory requests were summed across all constituent tasks to reflect total resource demand. Average and maximum usages were calculated using

mean and max functions, respectively. From the total 25 preprocessed columns, we used Random Forest to identify and rank the most relevant features for our prediction task. As a result, 12 features were selected.

- **Handling Data Imbalance:** imbalanced class distributions occur when the distribution of data instances is not equal in the classification model. A model trained on imbalanced datasets is biased toward the dominant class and may exhibit poor performance in the minority class. To address this, we employed data balancing techniques. Specifically, Synthetic Minority Over-sampling Technique (SMOTE) was applied to augment minority class instances and achieve a more equitable class distribution.

3.5. Feature Engineering

To improve predictive performance, we performed feature engineering based on the information provided in the dataset documentations. This process involves restructuring identifiers, extracting time-based features, and aggregating resource metrics at both task and job levels. To support task grouping and job-level aggregation, we created a unique Task_ID by combining each task's collection_id and instance_index. Then, we extracted the Job_ID from this new field. We also cleaned and standardized the IDs for jobs and users using simple number-based labels like J_1, J_2,…, J_n and u_1, u_2, ,…, u_n, so they were easier to work with.

Time-related features were extracted to capture temporal dependency. Specifically, we computed ArrivalTime to represent the time a task entered the system, and Duration to quantify its execution span. These features were derived from timestamp fields originally stored in microseconds, which were offset and converted into datetime format in minutes. The data was then segmented into consistent time intervals to preserve temporal causality.

Resource usage features were engineered by parsing dictionary-encoded columns and aggregating task-level metrics. For each job, we calculated total CPU and memory requests by summing across constituent tasks. Additionally, we derived statistical summaries such as average and maximum usage values using mean and max functions, respectively.

3.6. Feature Selection

Feature selection is a dimensionality reduction technique and way of selecting relevant features from a dataset which involves identifying features that have a high relation or score to dependent variables. It can lead to improved accuracy, faster training, reduced model complexity and computational cost. Therefore, selecting the most important features was a critical step to reduce dimensionality, improve model performance, and keep the predicting pipeline interpretable. Previous studies such as (Khalid et al., 2014; Mumuni & Mumuni, 2025; Theng & Bhoyar, 2024) have shown that effective feature selection can be achieved through different methods, such as Random Forest, Principal Component Analysis (PCA), decision trees, and statistical tests. Therefore, in this study, we employed Random Forest as it was used in our baseline works. It is a versatile ensemble learning method that can be used for both classification and regression tasks Random Forests assess feature importance during training and features that contribute more in increasing information gain are considered more important (Reza et al., 2001).

3.7. Model Selection

To capture the most accurate analysis of time series prediction for IoT offloading requests and resource needs, it is essential to include the most relevant approaches. In particular, we inspect relevant methods belonging to the two main categories we discussed in literature review section, namely ML algorithms, and DL techniques. Concerning ML class, we deliberately chose to use Logistic Regression and Decision Tree as classical baselines rather than a larger set of high performance ensemble classifiers for three interconnected methodological reasons related to the nature of this study: (1) The dataset and experimental design prioritize sequence modeling (temporal windows and sliding window features) over static tabular prediction; (2) our problem is fundamentally multitask (simultaneous classification and regression) with joint loss weighting and shared representation learning; and (3) our goal was to create minimal, interpretable baselines that reveal when temporal/deep architectures are required. Thus, Decision Tree and Logistic Regression operate as interpretable, lightweight lower bounds: Decision Tree for categorical targets (transparent baseline for class boundaries) and Logistic Regression for continuous targets (linear baseline for resource estimates) (Tantri & Bhat, 2025).

Because they work under different assumptions and need more extensive feature engineering or architectural adaptation to be fairly compared in a multi-task, sequence forecasting setting, stronger ML models like SVM, RF, or XGBoost are not irrelevant, but their inclusion necessitates careful, separate treatment. It is necessary to design and tune time series feature pipelines, extend them to multi-output prediction, and perform extensive hyperparameter/ensemble tuning in order to properly evaluate off-the-shelf RF/XGBoost/SVM, which in practice expect flattened tabular features. These procedures add more variables (feature design decisions, tuning budgets) that may make it difficult to determine whether performance variations are due to better feature engineering and tuning or better sequence modeling (Ahmed et al., 2010).

For transparency and experimental control we therefore held the baseline set minimal and interpretable while benchmarking a full family of sequence and hybrid deep models like 1D-CNN, DNN, LSTM, GRU, BiLSTM, and CNN-BiLSTM that operate on the same sequential inputs without bespoke feature engineering.

1D-CNN: basic CNNs are primarily designed to process two-dimensional data, such as images. As a result, standard 2D-CNN architectures are not inherently compatible with one-dimensional time series inputs. To address this mismatch, some researchers have transformed 1D signal into 2D representations to leverage existing 2D-CNN frameworks. While this strategy can be effective in certain specialized domains, it often introduces additional computational overhead and may reduce overall efficiency.

To overcome these limitations, one-dimensional CNNs (1D-CNNs) have been developed specifically for sequential data. These models accept raw 1D inputs directly, eliminating the need for dimensional conversion. Compared to their 2D counterparts, 1D-CNNs offer reduced computational complexity and faster training times, making them well-suited for real-time and resource-constrained applications. 1D-CNNs have shown strong performance across a range of domains, including fault diagnosis in rotating machinery, structural health monitoring, and real-time signal analysis. Moreover, they can effectively capture temporal dependencies and correlations in multivariate time series without requiring extensive manual feature engineering (Seba et al., 2024;Singh et al., 2021).

BiLSTM: The BiLSTM networks extends the traditional LSTM by incorporating two LSTM layers one processing the sequence forward and the other backward. This bidirectional processing enables the model to learn from both past and future contexts (Yang & Wang, 2022). In our case models, we employed BiLSTM to benchmark the prediction performance in predicting task features and resource usage trends. Recent studies have shown that BiLSTM architectures outperform traditional LSTM models and conventional machine learning techniques in various time series prediction tasks. Their ability to learn bidirectional dependencies makes them particularly valuable in applications requiring high temporal sensitivity (Benidis et al., 2022).

Hybrid Deep Learning Models: The core of our proposed solution is a hybrid deep learning model that integrates **1D-CNN** with BiLSTM. These hybrid models offer multiple benefits:

- **Improved Predicting Performance**: the hybrid model achieved highest prediction accuracy by combining local feature extraction and temporal dependency learning.
- **Handling Complex Dependencies**: The hybrid model captures both local burst patterns and long-range dependencies more effectively than single deep learning models.
- **Improved Generalization**: The model adapt well to heterogeneous features extracted from the Google Cluster dataset, allowing broader applicability.
- **Mitigating Model-Specific Weaknesses**: CNNs alone cannot handle sequence memory, while DNNs lack inherent memory for sequential modeling. The combination balances these limitations for comprehensive time-series modeling.

Combining 1D-CNN with BiLSTM: This hybrid model employs 1D-CNN to extract short-term patterns and BiLSTM layers to process the sequences both forward and backward in time. The extracted features are then processed by two LSTM layers: one moving forward through the data and the other moving backward. This bidirectional structure allows the model to gain information from both past and upcoming trends.

Proposed Hybrid Model: 1D-CNN-BiLSTM-Attention: Building on the strengths of combining 1D-CNN with BiLSTM, our proposed hybrid model further incorporates an attention mechanism to address a critical limitation in conventional time series predictions. The front-end element, the 1D-CNN layer, is in responsible of identifying short-term temporal and spatial dependencies in input sequences. In order to identify sudden shifts or

peaks patterns in offloading requests and resource consumption which are typical in edge workloads it carries out localized filtering across time steps. The extracted feature maps minimize noise and computational load for the next recurrent layer while maintaining vital high-frequency dynamics.

The BiLSTM layer then processes the CNN-derived sequence representations both forward and backward in order to capture long-range temporal dependencies. This makes it possible for the model to properly understand how historical workload patterns affect present and future task arrivals and resource needs. For time series with irregular patterns and non-linear dependencies, the LSTM's bidirectional nature ensure that temporal correlations are learned effectively. In real-world edge computing environments, this is rarely the case, despite the aforementioned models' frequent assumption that every second and every feature in a sequence matters equally.

The aforementioned models often assume that every moment and every feature in a sequence matters equally, but that’s rarely the case in real-world edge computing environments. According to the survey done by Zheng et al., (2020), offloading requests and resource usages can spike suddenly, tasks may shift in priority, and certain time intervals carry more weight than others. Therefore, taking this into account, in the proposed model attention mechanism was applied on top of the BiLSTM outputs. This module computes contextual weights for each time step and feature dimension, allowing the model to prioritize the most relevant segments of the sequence while minimizing the influence of less informative ones. This selective focus not only improves accuracy and effectiveness under dynamic workloads but also enhances model interpretability, revealing which temporal and feature patterns drive the network’s decisions. Finally, the shared attention-enhanced feature representations are passed to two parallel output branches within a Multi-Task Learning setup: one for classification of task offloading requests and another for regression of resource demands. These two branches are jointly optimized using weighted loss functions, allowing the model to balance both prediction objectives within a single training pipeline.

3.8. Model Training and Testing

After preprocessing the data and selecting relevant features, the Google cluster trace dataset, which contains 31 days of workload data from May 2019, was divided into training, validation, and test sets using a 70:10:20 ratio. This ratio was selected because it provides an optimal balance between effective model learning, hyper-parameter tuning, and unbiased performance evaluation ensuring sufficient samples for training while retaining enough unseen data for reliable validation and testing, as recommended in deep learning-based time series forecasting research (Ng et al., 2021; Tan et al., 2021).

To preserve the temporal dependencies inherent in real workload sequences, the data was split chronologically rather than randomly. Accordingly, the first 22 days (≈70%) of traces were used for training to enable the model to learn underlying workload patterns and temporal correlations. The next 3 days (≈10%) were allocated for validation, allowing fine-tuning of hyper-parameters and early stopping based on unseen yet temporally adjacent data. The last 6 days (≈20%) were reserved for testing, ensuring that model evaluation was conducted on future time intervals unseen during training or validation.

The proposed hybrid Multi-Task deep learning model, supporting multivariate time series inputs, was implemented using the Keras framework. The architecture consisted of input, convolutional, recurrent, attention, and dense layers, each equipped with suitable activation functions and regularization mechanisms. To jointly optimize the heterogeneous learning objectives classification of task offloading requests and regression of resource demands the model was trained using weighted loss functions, ensuring that each task contributed proportionally to the overall optimization process. Model training was conducted on a Tensor Processing Unit (TPU) via Google Colab, with validation performance continuously monitored to adjust learning rates and regularization parameters. Evaluation metrics were pre-defined prior to retraining to ensure consistency, fairness, and reproducibility across all experiments.

3.9. Parameter Tuning

Parameters are configuration settings that are set before the training of a deep learning model begins (Yu & Zhu, 2020) such as the learning rate, batch size, number of layers, the number of neurons in every layer, activation function, and regularization techniques such as dropout. Tuning hyper-parameters is essential because it can lead to significant improvements in model accuracy, training speed, and generalization to new data. Poorly

chosen hyper-parameters can result in under-fitting, over-fitting, or training instability, which can severely impact model performance.

3.10. Evaluation Metrics

After training, the model's performance is evaluated using key metrics including Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE), Mean Square Error (MSE), Root Mean Squared Error (RMSE), and R-squared (R^2). These metrics are listed and discussed below.

A. **MAE** quantifies the average absolute difference between the predicted and actual values. It provides a simple way to interpret the predict accuracy, indicating how far the predictions deviate from the actual observations, on average. MAE is useful for evaluating the accuracy of regression models, as it treats all errors equally without emphasizing larger errors. It is simple to interpret and is often used when we need a straightforward understanding of model accuracy (Kuhn & Johnson, 2013). Mathematically represented by the below formula

$$\mathrm{MAE} = (1/\mathrm{n})\sum |\mathrm{y}; - \hat{\mathrm{y}};|$$

Equation 3- 1: Mean Absolute Error Formula

Where:

y_i is the actual value,

$\hat{y}_i$ is the predicted value,

n is the number of data points.

B. **MSE** is a metric that calculates the average of the squared differences between predicted and actual values. It is widely used for measuring the performance of regression models. The MSE gives a larger penalty to larger errors due to the squaring of the differences. This is beneficial when large deviations in predictions are more undesirable and need to be penalized (Hastie et al., 2009). Mathematically it is represented as follows:

$$MSE = (1/n)\,\Sigma\,(y; - \hat{y};)^2$$

Equation 3- 2: Mean Square Error Formula

Where:

y_i is the actual value,

$\hat{y}_i$ is the predicted value,

n is the number of data points.

C. **MAPE** expresses the average absolute percentage difference between the predicted and actual values. It allows for a relative comparison of errors, which can be especially useful when comparing models across datasets with different scales. MAPE is a scale-independent metric, which makes it particularly useful for comparing performance across models when datasets have varying ranges or units (Makridakis et al., 1983). It is expressed mathematically by the following formula:

$$MAPE = (1/n)\,\Sigma\,|(y; - \hat{y};)\,/\,y;|\ *\ 100$$

Equation 3- 3: Mean Absolute Percentage Error Formula

Where:

y_i is the actual value,

$\hat{y}_i$ is the predicted value,

n is the number of data points.

D. **RMSE** measures the average magnitude of error by taking the square root of the mean of squared differences between predicted and actual values. It emphasizes larger errors and penalizes them more significantly, making it valuable when large deviations in predictions are highly undesirable. RMSE is particularly sensitive to large errors, making it a preferred choice when large deviations need to be penalized more heavily than smaller errors (Horváth & Rice, 2024). It is represented by the following mathematical formula.

$$RMSE = \sqrt{(1/n\,\Sigma\,(y; - \hat{y};)^2)}$$

Equation 3- 4: Root Mean Square Error Formula

Where:

y_i is the actual value,

$\hat{y}_i$ is the predicted value,

n is the number of data points.

E. **R^2** score measures the proportion of variance in the target variable that is explained by the model. It ranges from 0 to 1, where higher values indicate that the model explains more of the variability in the data. Perfect predictions are shown by a value of 1, whereas no variability is explained by the model when the value is 0. R^2 provides insight into how well the model's predictions match the actual data and whether the model captures the data's inherent patterns (Chatterjee & Hadi, 2013). Mathematically it can be calculated as follows:

$$R^2 = 1 - (\Sigma\,(y; - \hat{y};)^2\,/\,\Sigma\,(y; - \bar{y})^2)$$

Equation 3- 5: R-squared Formula

Where:

y_i is the actual value,

$\hat{y}_i$ is the predicted value,

$\bar{y}$ is the mean of the actual values,

n is the number of data points.

3.11. Design and Development Tools

3.11.1. Design Tools

In this thesis, design tools were important for visually presenting the structure and workflow of the proposed models. Two main tools EdrawMax and Canva were used to create diagrams, process flowcharts, and the overall architecture of the proposed solution.

3.11.2. Development Tools

Development tools include the software and hardware components that were used in implementation, documentation and overall achievement of the proposed solution. These are listed and discussed as follows

Software tools: In general we used the following software tools:

- **Google Colab**: Google Colab is a cloud-based, free Jupyter notebook environment that eliminates the need for setup. Users can create, upload, and share notebooks easily. It allows importing and exporting notebooks from Google Drive, as well as importing and publishing notebooks from GitHub.
- **TensorFlow:** An open-source machine learning framework developed by Google that simplifies the development and deployment of deep learning models. It provides a comprehensive ecosystem for model training, evaluation, and optimization, including features like TensorBoard for visualization, custom loss functions, and integration with Keras.
- **Python:** A widely adopted programming language known for its simplicity, readability, and extensive libraries. Python's ecosystem includes libraries like Pandas for data manipulation, NumPy for numerical computations, Scikit-learn for machine learning, and Matplotlib for visualization. These frameworks enable efficient development and experimentation with large datasets.
- **Mendeley Desktop:** A reference management and citation tool designed for researchers and students. Mendeley allows users to organize academic literature,

insert citations in research documents, and generate bibliographies seamlessly within word processors.

- **Microsoft Office:** Essential for document preparation, presentation, and data management. Word is used for writing reports, PowerPoint for creating presentations, and Excel for dataset visualization, preprocessing, and statistical analysis.

Hardware tools: To make our research process smooth and efficient especially considering the large size of the datasets, and complexity of the model we used the following hardware resources:

- **Hard Disk**: was used to store all relevant materials, including datasets, source code, and documents. The dataset alone consumed approximately 10.89 GB of storage space.
- **GPU**: was used to accelerate the model training process and efficiently manage parallel computations. We accessed GPU capabilities via Google Colab.
- **RAM**: worked alongside the GPU to further speed up training performance. Our system was equipped with 8 GB of RAM.

CHAPTER FOUR

4. PROPOSED SOLUTION ARCHITECTURE

4.1. Chapter Overview

In this chapter, we presented a solution to the aforementioned current gaps reported in the literature sections. This section endeavors to intricately outline the architecture of our proposed solution to predict task offloading requests and resource demands in IoT edge computing environments address the identified gaps and answer the raised research inquiries with better accuracy. The main objectives of this chapter to provide a comprehensive depiction of the model employed. Additionally, we attempt to discuss the rationale behind the algorithm (how the model works).

4.2. Proposed Model Architecture

The proposed solution for predicting IoT task offloading behavior and the associated resource demands is based on a hybrid deep learning model assisted with attention mechanism. The goal is to accurately predict task-level characteristics and resource usage indicators like CPU demand, memory usage, and network bandwidth. Three-stage prediction was used for enhanced prediction performance and lower complexity for IoT edge servers. The first part was to extract spatial patterns and second learning temporal dependencies, and lastly focus on the most important parts of the input sequence. From our baseline works, (Chen et al., 2022) and (Lackinger, 2023) only CPU usage was predicted. This univariate and narrow method ignores important interdependencies between multiple task and resource features, such as priority and delay tolerance might influence both CPU and memory demand. Another critical gap is that many prior studies relied on either synthetic datasets or small-scale real-world samples. These datasets do not capture the real complexity or scale of IoT systems. As a result, models trained on such data may not perform well when applied to large, diverse workloads. To overcome this, in our proposed solution we employed multivariate time series prediction, where all relevant features are processed and used for multi-output predictions using large-scale and real-world traces from Google Cluster.

While there are many different kinds of deep learning, we selected a hybrid deep learning model by combining 1D-CNN, BiLSTM, further improved by an attention mechanism for our prediction task. This combination is widely used in time series prediction in many studies including (Chen et al., 2022), (Sathi et al., 2023) and many others due to its ability

to extract local patterns, learn long-term dependencies, and focus on the most important parts of the input sequence. These works proved the outperformances of this architecture in handling temporal short-term and long-range dependencies. Considering the heterogeneous and dynamic nature of IoT edge computing environments, we decided to employ this architecture.

Overall the proposed model involves six main phases. The first step was accessing the dataset through BigQuery and loading into our working environment. Second, we carried out extensive data preprocessing, including cleaning missing values, detecting and removing outliers using statistical methods like z-score and IQR, and normalizing continuous variables. We also encoded categorical fields and used Random Forest importance rankings to select the most relevant features for prediction. We preferred these preprocessing steps, as they have been employed in our baseline works.

Third, following preprocessing step, we applied a sliding window to structure the data into fixed-length sequences suitable for time series modeling. We also performed feature engineering to derive necessary features such as Arrival_Time and duration. Fourth, the data was then split into training (70%), validation (10%), and testing (20%) sets in chronological order to avoid data leakage and reflect real-time deployment scenarios. Fifth, model training was performed on Google Colab using GPU acceleration to efficiently handle the computational demands of our work. During training, the input sequences were first processed by the 1D-CNN layer to extract localized temporal and spatial features, capturing short-term workload variations. These feature maps were then fed into the BiLSTM layer, which learned long-range bidirectional dependencies to model evolving workload behaviors over time. The Attention mechanism was subsequently applied to the BiLSTM outputs to emphasize the most informative time steps and feature dimensions, enhancing both prediction accuracy and interpretability. The attention-enhanced representations were finally passed to two parallel output branches under the Multi-Task Learning framework one for classifying task offloading requests and another for regressing resource demands.

Hyper-parameter tuning was performed using a random search strategy, which efficiently explores the hyper-parameter space without exhaustively evaluating all possible combinations. The tuning process involved iteratively sampling candidate configurations for key parameters, including the learning rate (tested in the range of 0.0001-0.01), batch

size (16, 32, 64), dropout rate (0.2-0.5), number of CNN filters (32-128), kernel size (3-7), and number of BiLSTM units (64-256). Each sampled configuration was trained on the training set and evaluated on the validation set using the MAE and RMSE as the primary performance metrics. The random search continued until performance convergence was observed, after which the best-performing configuration was selected based on the lowest validation loss and the highest prediction stability. Early stopping was applied to prevent overfitting, and learning rate scheduling was used to refine training near convergence. This systematic tuning ensured that the final hybrid model achieved an optimal balance between accuracy, generalization, and computational efficiency.

Finally, we evaluated the model performance using key metrics such as MAE, RMSE, MAPE and R^2, along with ordinal-aware accuracy for categorical predictions. The trained hybrid model was then used for predicting both task-level metadata and resource metrics. The complete architecture of the proposed model is shown in Figure 4-1 below.

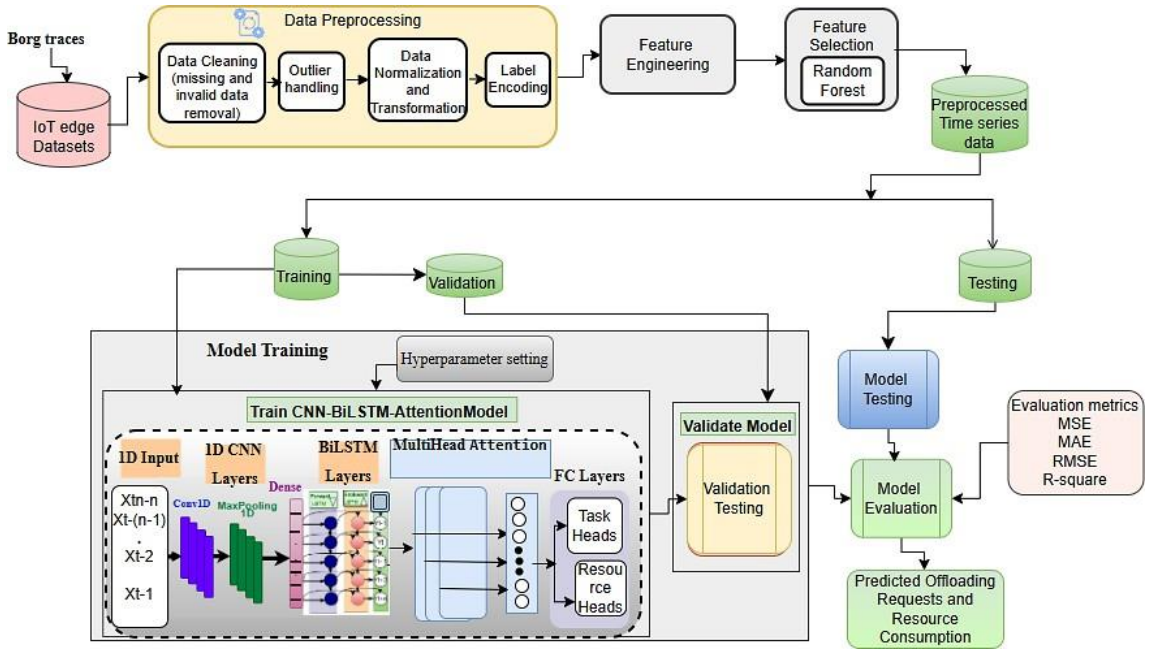


Figure 4-1: Proposed Model Architecture

4.3. Proposed Model Algorithm

To better indicate the structure of proposed model, we presented algorithm flow which involves descriptions of the sequential steps and operation of the model from input to outputs. Since our task involves multi-target predictions, the model is designed to handle both regression and classification tasks simultaneously. For regression outputs, we employed the MSE loss function to minimize prediction variance. For classification

targets, we utilized the Sparse Categorical Cross-Entropy loss function to effectively train the model on discrete class assignments. The following algorithm represents the step-by-step building and training of the proposed model:

Algorithm 1: CNN-BiLSTM-Attention model

Input**:** X, y ← IoT task offloading logs (features: CPU, memory, metadata).
Output**:** Multi-target predictions:

- Task_Offloading (*task_priority* and *scheduling_class*)
- Resource_Demand (CPU, memory, max CPU usage, and max memory usage)

Begin ()

Define deep learning hybrid architecture:

- Build a sequential model.
- Add an input layer with shape = (timesteps, features).
- Add a 1D-CNN block:
  - Conv1D layer: 32 filters, kernel size = 3, activation = ReLU, padding = same.
  - MaxPooling1D layer with pool size = 2.
- Add a BiLSTM block with 64 units, return_sequences = True.
- Add an Attention mechanism:
  - Add Permute input dimensions to transform
  - Dense layer with 64 units, activation = tanh to compute importance weights.
  - Perform element-wise multiplication with BiLSTM features.
  - Generate an attention-enhanced feature map.

Define multi-head output layers:

- Head 1 (Task Offloading): Apply dense layer(s) for categorical prediction
- Head 2 (Regression): Apply Dense layer(s) for numerical prediction

Compile the defined model:

- Optimizer = Adam.
- Loss functions: Task_Offloading → accuracy, Resource_Demand → MSE.

Training the defined model:

- Fit the model on (X_train, {y_task, y_resource}).
- Use validation data (X_val, {y_task_val, y_resource_val}).
- Apply EarlyStopping callback (monitor = val_loss, patience = 5).
- Set epochs = 30, Batch size = 64.

Testing the defined model:

- Predict using X_test.
- Return {y_pred_task, y_pred_resource}.

Return prediction_map{}

End ()

In general, the overall workflow of the proposed model in the above algorithm can be fully outlined as follows:

- Access and import the Google Cluster dataset
- Read and clean the datasets
- Combine task-related and resource-related features
- Preprocess the data (handling missing values, outliers, and normalization)
- Apply feature selection (random forest importance)
- Prepare data sequences using sliding windows and lag features
- Split data chronologically into train, validation, and test sets
- Build and compile the hybrid model
- Train the model using selected hyper-parameters
- Evaluate predictions on unseen data using appropriate metrics

4.4. Hyper-parameter Optimization

Hyper-parameter optimization involves searching for the optimal set of hyper-parameters that result in the best model performance (Yu & Zhu, 2020). Unlike model parameters, which are learned during training, hyper-parameters govern how the model learns and can significantly influence its performance. Tuning hyper-parameters is essential because it can lead to significant improvements in model accuracy, training speed, and generalization to new data. Poorly chosen hyper-parameters can result in under-fitting, over-fitting, or training instability, which can severely impact model performance. It is an external configuration setting for a model that is not learned from the data but needs to be specified before training. Finding the optimal hyper-parameters is a crucial step in training the model, as it directly impacts the model's ability to generalize well to new, unseen data (Kumar et al., 2024).

4.4.1. Regularization Techniques

Regularization is a technique used in deep learning used to prevent overfitting and improve a model's ability to generalize to unseen data. Overfitting occurs when a model learns the noise and details of the training data too well, leading to poor performance on new data. Regularization introduces a penalty to the loss function, discouraging overly complex

models and large parameter values (Srivastava et al., 2014). Common regularization techniques the followings:

L1 and L2 Regularization: L1 (Lasso) and L2 (Ridge) regularization add penalty terms to the loss function to control the magnitude of weights in the model. L1 regularization encourages sparsity by driving some weights to zero, effectively performing feature selection. L2 regularization, on the other hand, reduces the magnitude of all weights without setting them to zero, promoting smaller but non-zero weights (Srivastava et al., 2014).

Early Stopping: Early stopping monitors the model's performance on a validation set during training. If the validation performance stops improving for a specified number of epochs, training is halted to prevent overfitting (Srivastava et al., 2014).

Dropout: Dropout is a widely used regularization technique where, during training, a random subset of neurons is "dropped out" (set to zero) in each iteration. This prevents the network from relying too heavily on specific neurons and encourages learning more robust features (Srivastava et al., 2014).

4.4.2. Learning Rate

The learning rate is a critical hyperparameter in deep learning that determines the size of the steps taken during the optimization process to minimize the loss function. It controls how much the model's weights are updated in response to the error gradient during training. A well-chosen learning rate ensures efficient and stable convergence, while an inappropriate value can lead to slow training or instability (Kumar et al., 2024).

4.4.3. Batch Size and Number of epochs

Batch size is number of training samples processed before the model's internal parameters (weights) are updated. It helps us to accelerate the learning process by partitioning the training set into many subsets which are known as batch. In each batch, there is a number of epochs for training the model. In each batch, there is a number of epochs for training the model (Kumar et al., 2024).

Number of Epochs: The number of complete passes through the entire training dataset. The number of epochs is increased until there is a reduction in a validation error. During each epoch, the model processes the entire dataset, adjusts its weight and biases, and

updates the internal setting based on the optimization algorithm used. If there is a reduction error for succeeding epochs not improving, it indicates stopping to increase the epochs for better error reductions (Kumar et al., 2024).

4.4.4. Activation Functions

Activation functions are essential component in deep learning models, enabling them to capture complex, non-linear relationships in data. They add non-linearity to the network, allowing it to learn intricate patterns beyond simple linear transformations. Examples of common activation functions include Sigmoid, Tanh, ReLU, Leaky ReLU, PReLU, and ELU. Among these, ReLU and its variants are widely used in deep neural networks due to their computational efficiency and ability to mitigate issues such as vanishing gradients (Kumar et al., 2024).

In our model which combines 1D-CNN-BiLSTM-Attention, ReLU was selected as the primary activation function in the convolutional and dense layers. It was used in our baseline works as it provides fast convergence and effective learning needed in time series prediction tasks involving high-dimensional server logs. In addition to this, since our study involves multi-target prediction of both categorical and regression features, we also utilized two more activation functions, namely softmax and linear in the output layers. The softmax function generalizes the logistic function to handle multiple classes, converting a vector of values into a probability distribution. We employed softmax in the classification output layer, due to the multi-class classification nature of our categorical targets. Meanwhile, a linear activation function was applied in the output layer to support multi-output regression tasks.

4.4.5. Loss Function

The loss function also referred to as the cost or objective function, measures the discrepancy between the predicted outputs of a model and the actual ground truth values in the training data. During model training, the goal is to minimize this loss function, thereby improving the model's accuracy and its ability to generalize well on unseen data. The choice of loss function depends on the type of task being performed whether it is classification, regression, or a more specialized application. Common loss functions include Mean squared error, Mean absolute error, binary cross-entropy loss, categorical cross-entropy loss, hinge loss, and others (Liu & Wang, 2024; Q. Yang et al., 2024).

In our work, since the objective was to predict multiple continuous variables such as CPU usage, memory demand, and network as well as to predict ordinal task-level attributes like priority and delay sensitivity we employed a MSE loss function for the regression outputs. MSE is particularly effective in penalizing large prediction errors, making it suitable for our multivariate prediction model. For ordinal classification tasks such as task priority levels and delay sensitivity, we considered ordinal-aware handling during evaluation, although the model architecture is primarily trained for continuous output.

4.4.6. Optimizer

In machine learning, an optimizer is a key component that adjusts the model's internal parameters to minimize the loss function during training. It determines how the model learns from the data by updating weights based on gradients calculated during back-propagation. There are various optimizers available, each with unique characteristics and performance. Selecting the right optimizer is essential for achieving fast convergence, stability, and improved performance.

For our study, we employed the Adam optimizer due to its robustness, efficiency, and suitability for time-series predicting tasks. Adam is widely used in time series predicting tasks as it combines the benefits of momentum and adaptive learning rates. It adjusts the learning rates for each parameter dynamically by leveraging the first and second moments of the gradients, which enables efficient training even when dealing with complex and large-scale datasets. Moreover, Adam typically achieves faster convergence and delivers strong performance across a wide range of applications, making it a dependable option for similar multivariate prediction tasks.

CHAPTER FIVE

5. IMPLEMENTATION OF THE PROPOSED MODEL

5.1. Chapter Overview

In this chapter, we presented implementation details of the proposed model. Additionally, we delved into the setup and the working implementation environment for Jupyter Notebook.

5.2. Working Environment Setup

The environment setup defines the tools, libraries, and deployment configurations necessary for training, evaluating, and testing our proposed hybrid model. A well-prepared environment ensures the model performs efficiently on large-scale time-series data and yields reliable prediction results. To facilitate this, we used sets of tools, software, frameworks and libraries as mentioned below.

- **Python 3.12.11**: A high-level programming language used for data preprocessing, feature engineering, model building, and evaluation.
- **Jupyter Notebook**: A browser-based interactive environment used to write, execute, and document code during model experimentation.
- **TensorFlow 2.19.0**: A powerful open-source machine learning framework for building or deploying models.
- **Pandas 2.2.2**: A Python library essential for manipulating and analyzing structured data, particularly useful during feature extraction and cleaning.
- **NumPy 2.0.2**: Used for efficient numerical computations and handling multidimensional arrays, especially during sequence preparation and normalization.
- **Matplotlib 3.10.0**: A visualization library employed to generate plots for exploratory data analysis, training loss visualization, and performance comparison.
- **Scikit-learn 1.6.1**: A widely used machine learning library that supported feature selection methods such as correlation analysis, as well as performance metrics for evaluation.

5.3. Proposed Model Implementation

5.3.1. Dataset Pre-processing

Data preprocessing is essential to enhance data quality, model performance, and improve generalizability. As we discussed in data preprocessing section, the raw dataset contained sparse features and a mix of categorical, numerical, and semi-structured dictionary-

encoded fields. Directly training the proposed model on such unprocessed data would have introduced challenges such as inconsistency, incompatibility, and limited capacity to learn meaningful temporal patterns.

To address these issues, several preprocessing steps were implemented. First, irrelevant and unnamed features, including static identifiers such as machine_ids, collection_ids, and other internal tracking metrics, were excluded, as they provided no predictive value and risked causing overfitting by encouraging the model to memorize unique identifiers. Next, the dataset's substantial numbers of missing values over 129,000 entries were managed using imputation techniques such as forward fill, backward fill, and linear interpolation, which restored continuity without distorting workload dynamics. Outliers, particularly abnormal spikes in workload metrics that could distort learning, were detected using statistical approaches like z-score and the interquartile range (IQR) method, and anomalous values were removed to improve the stability and convergence of the predictive model. For temporal consistency, the time field, originally stored in microseconds, was converted into standard timestamps while adjusting for relative offsets. Categorical attributes such as priority and scheduling_class were then label-encoded to ensure compatibility with the deep learning architecture, preserving ordinal relationships between classes as given in table 5-1. Finally, we applied MinMaxScaler was applied to numerical features to mitigate the dominance of high-magnitude variables and ensure uniformity across scales.

5.3.2. Feature Engineering

For the preparation of the dataset, we followed the instructions taken from (Simaiya et al., 2024; Wilkes et al., 2020). Each workload item (referred to as a thing) in the Google Cluster trace has an associated priority, represented as a small integer, where 0 indicates the lowest priority and larger values correspond to higher resource preference. The original dataset defines several ranges, including the free tier (priorities ≤ 99), Best-effort batch (100-115), mid-tier (116-119), production tier (120-359), and monitoring tier (≥ 360), each reflecting different service-level objectives (SLOs) and internal charging rules. To simplify and generalize the data for model training, we mapped these detailed priorities into four high-level classes, ensuring interpretability while preserving relative importance. Similarly, the scheduling_class attribute, which indicates the latency-sensitivity of tasks. Priorities naturally map from best-effort data collection to mission/safety-critical control loops that require deterministic responsiveness and reliability under resource constraints;

this stratification reflects QoS, latency, and reliability demands highlighted in IoT scheduling literature. Scheduling classes similarly separate background/batch analytics from latency-sensitive streams and hard real-time control where deadlines are non-negotiable. These mappings are summarized in the table 5-1 as follows.

Table 5-1: Mapping class names for task priority and scheduling class

Label	Task priority (semantic class)	Scheduling class (semantic class)
0	Low (best-effort)	Background/best-effort
1	Medium (routine)	Batch/throughput-oriented
2	High (QoS-critical)	Latency-sensitive/interactive
3	Critical (safety/mission)	Real-time/hard-deadline

We have also derived critical features like Arrival_Time and durations. To reduce noise and better capture high-level workload behavior, we aggregated the task-level features per Job ID. This resampling step involved computing aggregated metrics such as the mean CPU and memory usage. These engineered features were derived from raw usage logs based on the document (Wilkes et al., 2020), Google's cluster trace documentation. Totally, we come up with 25 candidate features.

	timestamp	Job_ID	Task_ID	Arrival_Time	priority	scheduling_class	CPU_Request	Memory_Requests	Total_CPU_Time	\
2019-05-24	05:07:50.297089-07:00	J_1	T_1	0.000000	0.0	0.0	0.001585	0.001169	0.041851	
2019-05-24	12:19:51.638945-07:00	J_2	T_1	0.000002	0.0	0.0	0.001585	0.001179	0.024968	
2019-05-24	11:35:42.379940-07:00	J_3	T_1	0.000003	0.0	0.0	0.001673	0.001173	0.024176	
2019-05-24	11:28:49.515244-07:00	J_4	T_1	0.000003	0.0	0.0	0.001585	0.001163	0.019552	
2019-05-24	11:18:43.916581-07:00	J_5	T_1	0.000003	0.0	0.0	0.001585	0.001179	0.028044	

average_usage_cpus	average_usage_memory	maximum_usage_cpus	maximum_usage_memory	random_sample_usage_cpus	cpu_usage_distribution_mean	\
0.000719	0.000197	0.003658	0.000232	0.000522	0.000881	
0.000719	0.000197	0.003658	0.000232	0.000522	0.000881	
0.000719	0.000197	0.003658	0.000232	0.000522	0.000881	
0.000719	0.000197	0.003658	0.000232	0.000522	0.000881	
0.000719	0.000197	0.003658	0.000232	0.000522	0.000881	

assigned_memory	page_cache_memory	cycles_per_instruction	memory_accesses_per_instruction	tail_cpu_usage_distribution_mean	sample_rate	\
0.001835	9.536743e-07	1.468342	0.003632	0.001851	1.0	
0.001835	9.536743e-07	1.468342	0.003632	0.001851	1.0	
0.001835	9.536743e-07	1.468342	0.003632	0.001851	1.0	
0.001835	9.536743e-07	1.468342	0.003632	0.001851	1.0	
0.001835	9.536743e-07	1.468342	0.003632	0.001851	1.0	

start_time	end_time	Duration	Arrival_Time_scaled
2.142900e+12	2.143200e+12	26.0	0.000000
2.142900e+12	2.143200e+12	22.0	0.000002
2.142900e+12	2.143200e+12	22.0	0.000003
2.142900e+12	2.143200e+12	26.0	0.000003
2.142900e+12	2.143200e+12	21.0	0.000003

Figure 5-1: Sample records of preprocessed ClusterData2019

5.3.3. Feature selection

The ClusterData2019 dataset originally consisted of more than 34 features. Using all of them directly for model training would not only increase computational complexity but also risk introducing redundancy, multi-collinearity, and irrelevant information that could negatively affect model performance. Not all features are relevant for forecasting the target variables, some features may contribute minimal or no predictive power, potentially introducing noise and obscuring the relationships that the model needs to learn. Feature selection was therefore performed to identify and retain only the most informative and relevant variables that significantly influence the target outcome. This process helps to reduce dimensionality, enhance model generalization, shorten training time, and improve interpretability by focusing on variables with meaningful contributions. Moreover, feature selection prevents overfitting by eliminating uninformative attributes that may cause the model to memorize noise rather than learn patterns. Hence, selecting key features was an essential step to ensure that the predictive model remains accurate, efficient, and explainable (Reza et al., 2001).

In our research, we used Random Forest for feature importance ranking with SelectKBest selector as it was also used in our baseline works. It is a versatile ensemble learning method that can be used for both classification and regression tasks. It constructs multiple decision trees and aggregates their outputs, making it capable of capturing nonlinear relationships and interactions between features. Importantly, RF provides built-in feature importance scores based on how much each feature reduces impurity across all trees. We used SelectKBest to rank and select features for both categorical and regression tasks. Since the dataset contains complex interactions, Random Forest helped us capture these dependencies more effectively than filter methods.

In total, from the 25 candidate features we selected 12 features. These are presented in Figure 5-2, which shows how each feature contributes to the prediction process based on the importance scores. This final feature set was used in implementation of the proposed model to predict task offloading requests and resource demands. We improved the performance of the model by reducing down the inputs to only the most relevant ones.

	task_priority	scheduling_class	cpus_request	memory_request	average_usage	maximum_usage	Average_Normalized_Score
resource_request	0.326570	0.423517	1.000000e+00	0.099307	0.005860	0.007525	0.310463
random_sample_usage	0.048730	0.050784	4.358920e-09	0.020089	0.655855	0.725436	0.250149
allocated_memory_for_task	0.043672	0.037770	2.326657e-09	0.794218	0.004085	0.014565	0.149052
tail_cpu_usage_distribution	0.055387	0.056436	2.729378e-09	0.011137	0.239208	0.199406	0.093596
Duration	0.228533	0.159861	5.838079e-10	0.015984	0.000032	0.000049	0.067410
cpu_usage_distribution	0.052277	0.049061	4.347245e-09	0.006363	0.049619	0.013157	0.028413
Arrival_Time	0.060082	0.053379	3.485816e-09	0.003301	0.005088	0.008826	0.021779
end_time	0.057644	0.052595	3.079720e-09	0.003318	0.005672	0.009375	0.021434
cycles_per_instruction	0.043443	0.045104	5.704647e-09	0.006047	0.010088	0.012214	0.019483
cluster	0.035641	0.031511	4.146019e-09	0.006536	0.014444	0.004026	0.015360
vertical_scaling	0.012678	0.011926	3.229489e-10	0.024824	0.004823	0.001356	0.009268
instance_events_type	0.017981	0.014198	1.229647e-09	0.004118	0.002569	0.001954	0.006803

Figure 5-2: Top 12 Feature Importance using Random Forest

5.3.4. Proposed Model Building

In this section, we covered model implementation which is a crucial part in any research topic. It includes an implementation of the selected model for predicting offloading requests and associated resource demands, as well as importing essentially libraries, and optimization parameters we used for model training. Before implementing the 1D-CNN-BiLSTM-Attention we imported the necessary libraries.

```
# --- Standard Libraries ---
import sys
import warnings
import math
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Conv1D, Bidirectional, LSTM, Dense, Attention, Flatten, Dropout
from tensorflow.keras import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.regularizers import l2
import keras.backend as K
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_percentage_error, accuracy_score
from sklearn.feature_selection import SelectKBest
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
import imblearn
import matplotlib.pyplot as plt
import seaborn as sns
```

Figure 5-3: Importing necessary libraries

Multiple targets can be predicted by the proposed model architecture, which is built to process sequential data. The CNN-BiLSTM-Attention model combination is implemented in Figure 5-4 below. The models were defined independently, as shown, and then the optimizer and learning rate were adjusted. In order to extract local features and patterns from the input sequences, three consecutive 1D Conv layers with a kernel size of three are

applied after the first input layer specifies the form of the input sequences. Then, utilizing forward and backward hidden states, the sequential model known as Bi-LSTM is used to extract the valid information from the feature maps. Subsequently, an attention layer is employed, which combines significant features and selects the crucial features by redistributing the weights. The output of the attention layer was flattened into a 1D vector using the flatten layer, and a dropout layer was added for regularization to assist avoid overfitting by randomly setting a portion of the input units to zero during training. Lastly, each of the six target variables is represented by a distinct fully connected (Dense) layer. We utilized a linear activation for the regression outputs and a softmax activation for the classification outputs to account for the probability distribution across classes. In order to predict the various output variables, this layer combination enables the model to focus on significant time steps and information within the input sequences, as well as to capture short-term trends and temporal dependencies.

```
input_layer = Input(shape=input_shape, name='input_sequence')
conv1 = Conv1D(filters=best_hyperparameters['cnn_filters_1'], kernel_size=3, activation='relu', padding='same')(input_layer)
conv2 = Conv1D(filters=best_hyperparameters['cnn_filters_2'], kernel_size=3, activation='relu', padding='same')(conv1)
conv3 = Conv1D(filters=best_hyperparameters['cnn_filters_3'], kernel_size=3, activation='relu', padding='same')(conv2)
bilstm = Bidirectional(LSTM(best_hyperparameters['lstm_units'], return_sequences=True))(conv3)
attention_output = Attention()([bilstm, bilstm])
flatten = Flatten()(attention_output)
dropout = Dropout(best_hyperparameters['dropout_rate'])(flatten)
output_priority = Dense(num_priority_classes, activation='softmax', name='output_task_priority')(dropout)
output_scheduling = Dense(num_scheduling_classes, activation='softmax', name='output_scheduling_class')(dropout)
output_cpus_request = Dense(1, name='output_cpus_request')(dropout)
output_memory_request = Dense(1, name='output_memory_request')(dropout)
output_total_cpu_usage = Dense(1, name='output_total_cpu_usage')(dropout)
output_total_memory_usage = Dense(1, name='output_total_memory_usage')(dropout)
final_model = Model(inputs=input_layer, outputs=[output_priority, output_scheduling, output_cpus_request, output_memory_request,
                                                 output_total_cpu_usage, output_total_memory_usage])

# --- Compile Model
optimizer = Adam(learning_rate=best_hyperparameters['learning_rate'])
final_model.compile(optimizer=optimizer, loss=losses, metrics=metrics)
# --- Define Callbacks ---
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
# --- Train Model ---
EPOCHS = 30
history_final = final_model.fit(
    train_dataset,
    epochs=EPOCHS,
    validation_data=val_dataset,
    callbacks=[early_stopping]
)
```

```
Epoch 1/10
2188/2188 ━━━━━━━━━━━━━━━━━━━━ 73s 29ms/step - cpus_request_output_loss: 0.2386 - cpus_request_output_mean_squared_error: 0.2386 - cpus_request_output_r2_score
Epoch 2/10
2188/2188 ━━━━━━━━━━━━━━━━━━━━ 79s 28ms/step - cpus_request_output_loss: 0.0189 - cpus_request_output_mean_squared_error: 0.0189 - cpus_request_output_r2_score
Epoch 3/10
2188/2188 ━━━━━━━━━━━━━━━━━━━━ 79s 27ms/step - cpus_request_output_loss: 0.0150 - cpus_request_output_mean_squared_error: 0.0150 - cpus_request_output_r2_score
Epoch 4/10
2188/2188 ━━━━━━━━━━━━━━━━━━━━ 59s 27ms/step - cpus_request_output_loss: 0.0138 - cpus_request_output_mean_squared_error: 0.0138 - cpus_request_output_r2_score
Epoch 5/10
2188/2188 ━━━━━━━━━━━━━━━━━━━━ 84s 28ms/step - cpus_request_output_loss: 0.0108 - cpus_request_output_mean_squared_error: 0.0108 - cpus_request_output_r2_score
Epoch 6/10
2188/2188 ━━━━━━━━━━━━━━━━━━━━ 64s 29ms/step - cpus_request_output_loss: 0.0104 - cpus_request_output_mean_squared_error: 0.0104 - cpus_request_output_r2_score
Epoch 7/10
2188/2188 ━━━━━━━━━━━━━━━━━━━━ 83s 30ms/step - cpus_request_output_loss: 0.0097 - cpus_request_output_mean_squared_error: 0.0097 - cpus_request_output_r2_score
Epoch 8/10
2188/2188 ━━━━━━━━━━━━━━━━━━━━ 66s 30ms/step - cpus_request_output_loss: 0.0089 - cpus_request_output_mean_squared_error: 0.0089 - cpus_request_output_r2_score
Epoch 9/10
2188/2188 ━━━━━━━━━━━━━━━━━━━━ 63s 29ms/step - cpus_request_output_loss: 0.0089 - cpus_request_output_mean_squared_error: 0.0089 - cpus_request_output_r2_score
Epoch 10/10
2188/2188 ━━━━━━━━━━━━━━━━━━━━ 84s 30ms/step - cpus_request_output_loss: 0.0059 - cpus_request_output_mean_squared_error: 0.0059 - cpus_request_output_r2_score
```

Figure 5-4: Proposed CNN-BiLSTM-Attention Training

5.4. Benchmark Models Implementation

In addition to the proposed model, a suite of benchmark models was implemented to assess comparative performance. These include standalone deep learning models such as 1D-CNN, DNN, LSTM, GRU, BiLSTM, and a hybrid model of 1D-CNN-BiLSTM, as well as two classical machine learning models, Logistic Regression, and Decision Tree. This benchmarking enables for better evaluation of the proposed model's effectiveness on multiple dimensions, including accuracy, temporal sensitivity, and computational efficiency. This section presents the design rationale, architectural components, and IoT-specific implications of each model.

5.4.1. Classical Machine Learning Models

To establish baseline performance comparison for both classification and regression tasks, we trained lightweight ML benchmark models, we provided below.

a) Logistic Regression

Linear Regression models were trained for predicting continuous resource usage metrics, including CPU requests, memory requests, and aggregated usage variables derived from sliding windows. These models assume a linear relationship between features and outputs, which simplify interpretation and computational efficiency. While they lack the ability to capture complex temporal dependencies and nonlinear dynamics, their role in this study was to serve as baseline model. Figure 5-5, below shows the implementation of the logistic regression for continuous target prediction benchmark.

```
#  Linear Regression (Regression models - only 4 outputs)
print("\nTraining Linear Regression Models...")
lr_cpus = LinearRegression()
lr_cpus.fit(X_train_processed,y_cpus_request_train_scaled)
lr_mem = LinearRegression()
lr_mem.fit(X_train_processed,y_memory_request_train_scaled)
lr_total_cpu = LinearRegression()
lr_total_cpu.fit(X_train_processed,y_total_cpu_usage_train_scaled)
lr_total_mem = LinearRegression()
lr_total_mem.fit(X_train_processed,y_total_memory_usage_train_scaled)
print("Linear Regression Models training complete.")
```

Figure 5-5: Logistic Regression Training

b) Decision Tree

Decision Trees are hierarchical models that recursively partition the feature space based on entropy or Gini impurity. Decision Trees were employed as baseline model for

classification tasks. Although they are limited in scalability and often prone to overfitting, their transparent structure provides clear interpretability, making them useful for diagnostic analysis and initial benchmarking. Figure 5-6, below shows the implementation of the decision tree for classification benchmarking.

```
#  Decision Tree
print("\nTraining Decision Tree Models...")
dt_priority = DecisionTreeClassifier(random_state=42)
dt_priority.fit(X_train_processed,y_priority_train_encoded)
dt_sched = DecisionTreeClassifier(random_state=42)
dt_sched.fit(X_train_processed,y_scheduling_train_encoded)
```

Figure 5-6: Decision Tree Training

5.4.2. 1D-CNNs

1D-CNNs are widely used for sequential data because they capture localized temporal dependencies through sliding filters. They are effective in identifying short-term variations such as sudden CPU or memory spikes. In this study, 1D-CNN was implemented as a baseline model to detect short-range dependencies in IoT task offloading sequences. However, since 1D-CNNs has no internal memory, they cannot effectively capture long-term dependencies, limiting predicting performance when future demand depends on extended historical patterns. Figure 5-7, below shows the implementation of the 1D-CNNs model. As we saw, the model was defined and then adjusted with the Adam optimizer, categorical cross-entropy loss for classification outputs, and MSE loss for regression outputs, using accuracy as the main evaluation metric.

```
#  CNN
print("\nTraining CNN Model...")
cnn_inp = Input((X_train_seq.shape[1],X_train_seq.shape[2]))
cnn_out = Conv1D(32,3,activation='relu',padding='causal')(cnn_inp)
cnn_out = Conv1D(64,3,activation='relu',padding='causal')(cnn_out)
cnn_out = Flatten()(cnn_out)
cnn_dense = Dense(64,activation='relu')(cnn_out)
cnn_dense = Dense(32,activation='relu')(cnn_dense)
cnn_out1 = Dense(le_priority.classes_.shape[0],activation='softmax',name='task_priority_output')(cnn_dense)
cnn_out2 = Dense(4,activation='softmax',name='scheduling_class_output')(cnn_dense)
cnn_out3 = Dense(1,activation='linear',name='cpus_request_output')(cnn_dense)
cnn_out4 = Dense(1,activation='linear',name='memory_request_output')(cnn_dense)
cnn_model = Model(cnn_inp,[cnn_out1,cnn_out2,cnn_out3,cnn_out4])
cnn_model.compile(optimizer=Adam(BENCHMARK_LR),
                  loss={'task_priority_output':SparseCategoricalCrossentropy(),
                        'scheduling_class_output':SparseCategoricalCrossentropy(),
                        'cpus_request_output':MeanSquaredError(),
                        'memory_request_output':MeanSquaredError()},
                  metrics={'task_priority_output':['accuracy'],
                           'scheduling_class_output':['accuracy'],
                           'cpus_request_output':[MSEMetric(),R2Score()],
                           'memory_request_output':[MSEMetric(),R2Score()]})
cnn_model.fit(X_train_seq,
              [y_priority_train_seq,y_scheduling_train_seq,y_cpus_request_train_seq,y_memory_request_train_seq],
              validation_data=(X_val_seq,[y_priority_val_seq,y_scheduling_val_seq,y_cpus_request_val_seq,y_memory_request_val_seq]),
              epochs=EPOCHS_BENCHMARK,batch_size=64,callbacks=[es,rlr],verbose=1)
```

```
Epoch 1/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 13s 27ms/step - cpus_request_output_loss: 1.1303 - cpus_request_output_mean_squared_error: 1.1303 - cpus_request_output_r2_score:
Epoch 2/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - cpus_request_output_loss: 1.1062 - cpus_request_output_mean_squared_error: 1.1062 - cpus_request_output_r2_score: -0
Epoch 3/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - cpus_request_output_loss: 1.0974 - cpus_request_output_mean_squared_error: 1.0974 - cpus_request_output_r2_score: -0
Epoch 4/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - cpus_request_output_loss: 1.0867 - cpus_request_output_mean_squared_error: 1.0867 - cpus_request_output_r2_score: 0.
Epoch 5/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - cpus_request_output_loss: 1.0732 - cpus_request_output_mean_squared_error: 1.0732 - cpus_request_output_r2_score: 0.
Epoch 6/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - cpus_request_output_loss: 1.0569 - cpus_request_output_mean_squared_error: 1.0570 - cpus_request_output_r2_score: 0.
Epoch 7/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - cpus_request_output_loss: 1.0379 - cpus_request_output_mean_squared_error: 1.0379 - cpus_request_output_r2_score: 0.
Epoch 8/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - cpus_request_output_loss: 1.0144 - cpus_request_output_mean_squared_error: 1.0144 - cpus_request_output_r2_score: 0.
Epoch 9/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - cpus_request_output_loss: 0.9870 - cpus_request_output_mean_squared_error: 0.9870 - cpus_request_output_r2_score: 0.
Epoch 10/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - cpus_request_output_loss: 0.9556 - cpus_request_output_mean_squared_error: 0.9556 - cpus_request_output_r2_score: 0.
1DCNN Model training complete.
```

Figure 5-7: 1D-CNN implementation

5.4.3. DNNs

DNNs consist of multiple fully connected layers designed to learn nonlinear feature mappings. Unlike CNNs or RNNs, they lack a temporal component, making them more suitable for data with engineered features rather than raw sequences. In this study, DNNs were implemented as a baseline to predict offloading request patterns and resource consumption needs in IoT edge setting. While DNNs captured complex nonlinear interactions, they struggled with time-dependent dependencies inherent in IoT workloads. Figure 5-8, below shows the implementation of the DNN model. As we saw, the model was defined and then adjusted with learning rate and optimizer, and compiled using the accuracy as metrics and categorical cross entropy as loss function.

```
#  DNN
print("\nTraining DNN Model...")
dnn_inp = Input((X_train_processed.shape[1],))
dnn_dense = Dense(64, activation='relu')(dnn_inp)
dnn_dense = Dense(32, activation='relu')(dnn_dense)
dnn_out1 = Dense(le_priority.classes_.shape[0], activation='softmax', name='task_priority_output')(dnn_dense)
dnn_out2 = Dense(4, activation='softmax', name='scheduling_class_output')(dnn_dense)
dnn_out3 = Dense(1, activation='linear', name='cpus_request_output')(dnn_dense)
dnn_out4 = Dense(1, activation='linear', name='memory_request_output')(dnn_dense)
dnn_model = Model(dnn_inp, [dnn_out1, dnn_out2, dnn_out3, dnn_out4])
dnn_model.compile(optimizer=Adam(BENCHMARK_LR),
                  loss={'task_priority_output': SparseCategoricalCrossentropy(),
                        'scheduling_class_output': SparseCategoricalCrossentropy(),
                        'cpus_request_output': MeanSquaredError(),
                        'memory_request_output': MeanSquaredError()},
                  metrics={'task_priority_output': ['accuracy'],
                           'scheduling_class_output': ['accuracy'],
                           'cpus_request_output': [MSEMetric(), R2Score()],
                           'memory_request_output': [MSEMetric(), R2Score()]})
dnn_model.fit(X_train_processed,
              [y_priority_train_encoded, y_scheduling_train_encoded, y_cpus_request_train_scaled, y_memory_request_train_scaled],
              validation_data=(X_val_processed, [y_priority_val_encoded, y_scheduling_val_encoded, y_cpus_request_val_scaled, y_memory_request_val_scaled]),
              epochs=EPOCHS_BENCHMARK, batch_size=64, callbacks=[es, rlr], verbose=1)
```

```
Training DNN Model...
Epoch 1/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 47s 19ms/step - cpus_request_output_loss: 1.1770 - cpus_request_output_mean_squared_error: 1.1770 - cpus_request_output_r2_score:
Epoch 2/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - cpus_request_output_loss: 1.1014 - cpus_request_output_mean_squared_error: 1.1014 - cpus_request_output_r2_score: -0
Epoch 3/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - cpus_request_output_loss: 1.0413 - cpus_request_output_mean_squared_error: 1.0413 - cpus_request_output_r2_score: 0.
Epoch 4/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - cpus_request_output_loss: 0.9853 - cpus_request_output_mean_squared_error: 0.9853 - cpus_request_output_r2_score: 0.
Epoch 5/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - cpus_request_output_loss: 0.9316 - cpus_request_output_mean_squared_error: 0.9316 - cpus_request_output_r2_score: 0.
Epoch 6/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - cpus_request_output_loss: 0.8814 - cpus_request_output_mean_squared_error: 0.8814 - cpus_request_output_r2_score: 0.
Epoch 7/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - cpus_request_output_loss: 0.8339 - cpus_request_output_mean_squared_error: 0.8339 - cpus_request_output_r2_score: 0.
Epoch 8/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - cpus_request_output_loss: 0.7884 - cpus_request_output_mean_squared_error: 0.7884 - cpus_request_output_r2_score: 0.
Epoch 9/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - cpus_request_output_loss: 0.7450 - cpus_request_output_mean_squared_error: 0.7450 - cpus_request_output_r2_score: 0
Epoch 10/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - cpus_request_output_loss: 0.7036 - cpus_request_output_mean_squared_error: 0.7036 - cpus_request_output_r2_score: 0.
DNN Model training complete.
```

Figure 5-8: DNN Model Training

5.4.4. GRU

Gated Recurrent Units (GRUs) were developed to address the vanishing gradient problem in standard RNNs, using update and reset gates to regulate information flow (Cho et al., 2014). GRU can effectively learn from long sequences, making them suitable for predicting IoT task offloading trends over time. They capture complex temporal dynamics such as peak task intervals, idle periods, and usage patterns. Figure 5-9, shows the training of our benchmark model.

```
#  GRU
print("\nTraining GRU Model...")
gru_inp = Input((X_train_seq.shape[1],X_train_seq.shape[2]))
gru_out = GRU(64,return_sequences=True)(gru_inp)
gru_out = GRU(128)(gru_out)
gru_dense = Dense(64,activation='relu')(gru_out)
gru_dense = Dense(32,activation='relu')(gru_dense)
gru_out1 = Dense(le_priority.classes_.shape[0],activation='softmax',name='task_priority_output')(gru_dense)
gru_out2 = Dense(4,activation='softmax',name='scheduling_class_output')(gru_dense)
gru_out3 = Dense(1,activation='linear',name='cpus_request_output')(gru_dense)
gru_out4 = Dense(1,activation='linear',name='memory_request_output')(gru_dense)
gru_model = Model(gru_inp,[gru_out1,gru_out2,gru_out3,gru_out4])
gru_model.compile(optimizer=Adam(BENCHMARK_LR),
                  loss={'task_priority_output':SparseCategoricalCrossentropy(),
                        'scheduling_class_output':SparseCategoricalCrossentropy(),
                        'cpus_request_output':MeanSquaredError(),
                        'memory_request_output':MeanSquaredError()},
                  metrics={'task_priority_output':['accuracy'],
                           'scheduling_class_output':['accuracy'],
                           'cpus_request_output':[MSEMetric(),R2Score()],
                           'memory_request_output':[MSEMetric(),R2Score()]})
gru_model.fit(X_train_seq,
              [y_priority_train_seq,y_scheduling_train_seq,y_cpus_request_train_seq,y_memory_request_train_seq],
              validation_data=(X_val_seq,[y_priority_val_seq,y_scheduling_val_seq,y_cpus_request_val_seq,y_memory_request_val_seq]),
              epochs=EPOCHS_BENCHMARK,batch_size=64,callbacks=[es,rlr],verbose=1)
```

Figure 5-9: GRU Model Training

5.4.5. LSTM

LSTM networks are designed to address the limitations of standard RNNs by introducing gating mechanisms that regulate the flow of information (Fu et al., 2022; Hochreiter & Schmidhuber, 1997). LSTMs can effectively learn from long sequences, making them suitable for predicting IoT task offloading trends over time. They capture temporal dynamics of requests and usage patterns. Figure 5-10, shows the training of our benchmark model.

```
#  LSTM
print("\nTraining LSTM Model...")
lstm_inp = Input((X_train_seq.shape[1],X_train_seq.shape[2]))
lstm_out = LSTM(64,return_sequences=True)(lstm_inp)
lstm_out = LSTM(128)(lstm_out)
lstm_dense = Dense(64,activation='relu')(lstm_out)
lstm_dense = Dense(32,activation='relu')(lstm_dense)
lstm_out1 = Dense(le_priority.classes_.shape[0],activation='softmax',name='task_priority_output')(lstm_dense)
lstm_out2 = Dense(4,activation='softmax',name='scheduling_class_output')(lstm_dense)
lstm_out3 = Dense(1,activation='linear',name='cpus_request_output')(lstm_dense)
lstm_out4 = Dense(1,activation='linear',name='memory_request_output')(lstm_dense)
lstm_model = Model(lstm_inp,[lstm_out1,lstm_out2,lstm_out3,lstm_out4])
lstm_model.compile(optimizer=Adam(BENCHMARK_LR),
                   loss={'task_priority_output':SparseCategoricalCrossentropy(),
                         'scheduling_class_output':SparseCategoricalCrossentropy(),
                         'cpus_request_output':MeanSquaredError(),
                         'memory_request_output':MeanSquaredError()},
                   metrics={'task_priority_output':['accuracy'],
                            'scheduling_class_output':['accuracy'],
                            'cpus_request_output':[MSEMetric(),R2Score()],
                            'memory_request_output':[MSEMetric(),R2Score()]})
lstm_model.fit(X_train_seq,
               [y_priority_train_seq,y_scheduling_train_seq,y_cpus_request_train_seq,y_memory_request_train_seq],
               validation_data=(X_val_seq,[y_priority_val_seq,y_scheduling_val_seq,y_cpus_request_val_seq,y_memory_request_val_seq]),
               epochs=EPOCHS_BENCHMARK,batch_size=64,callbacks=[es,rlr],verbose=1)
```

Figure 5-10: LSTM Model Training

5.4.6. BiLSTM

BiLSTM extends LSTM by allowing the model to learn from both past and future context in the sequence (Yang & Wang, 2022). This bidirectionality enhances predicting performance by incorporating backward temporal dependencies, which is particularly valuable in environments where future trends influence current offloading behavior. Figure 5-11 below shows the implementation of the BiLSTM model. As we saw the model was defined and then adjusted with learning rate and optimizer, compiled using the Adam optimizer, categorical cross-entropy loss for classification outputs, and MSE loss for regression outputs.

```
#  BiLSTM
print("\nTraining BiLSTM Model...")
bilstm_inp = Input((X_train_seq.shape[1],X_train_seq.shape[2]))
bilstm_out = Bidirectional(LSTM(64,return_sequences=True))(bilstm_inp)
bilstm_out = Bidirectional(LSTM(128))(bilstm_out)
bilstm_dense = Dense(64,activation='relu')(bilstm_out)
bilstm_dense = Dense(32,activation='relu')(bilstm_dense)
bilstm_out1 = Dense(le_priority.classes_.shape[0],activation='softmax',name='task_priority_output')(bilstm_dense)
bilstm_out2 = Dense(4,activation='softmax',name='scheduling_class_output')(bilstm_dense)
bilstm_out3 = Dense(1,activation='linear',name='cpus_request_output')(bilstm_dense)
bilstm_out4 = Dense(1,activation='linear',name='memory_request_output')(bilstm_dense)
bilstm_model = Model(bilstm_inp,[bilstm_out1,bilstm_out2,bilstm_out3,bilstm_out4])
bilstm_model.compile(optimizer=Adam(BENCHMARK_LR),
                     loss={'task_priority_output':SparseCategoricalCrossentropy(),
                           'scheduling_class_output':SparseCategoricalCrossentropy(),
                           'cpus_request_output':MeanSquaredError(),
                           'memory_request_output':MeanSquaredError()},
                     metrics={'task_priority_output':['accuracy'],
                              'scheduling_class_output':['accuracy'],
                              'cpus_request_output':[MSEMetric(),R2Score()],
                              'memory_request_output':[MSEMetric(),R2Score()]})
bilstm_model.fit(X_train_seq,
                 [y_priority_train_seq,y_scheduling_train_seq,y_cpus_request_train_seq,y_memory_request_train_seq],
                 validation_data=(X_val_seq,[y_priority_val_seq,y_scheduling_val_seq,y_cpus_request_val_seq,y_memory_request_val_seq]),
                 epochs=EPOCHS_BENCHMARK,batch_size=64,callbacks=[es,rlr],verbose=1)
```

Figure 5-11: BiLSTM Model Training

5.4.7. CNN with BiLSTM

By integrating 1D-CNN with BiLSTM, this architecture captures both local features and bidirectional temporal dependencies. The CNN layer first processes the raw time-series input to extract short-range features, and the BiLSTM layer then takes this as input and contextualizes these features in both past and future timeframes. This model performed well in predicting multivariate outputs, due to its deep representation capacity. Figure 5-12, below shows the implementation of the combination of CNN and BiLSTM models. As depicted, the models were defined separately, followed by adjustment of the learning rate and optimizer. The two models were then combined and compiled using the Adam optimizer, categorical cross-entropy loss for classification outputs, and MSE loss for regression outputs.

```
#  CNN+BiLSTM
print("\nTraining CNN+BiLSTM Model...")
cnn_bilstm_inp = Input((X_train_seq.shape[1],X_train_seq.shape[2]))
cnn_bilstm_cnn = Conv1D(32,3,activation='relu',padding='causal')(cnn_bilstm_inp)
cnn_bilstm_cnn = Conv1D(64,3,activation='relu',padding='causal')(cnn_bilstm_cnn)
cnn_bilstm_bilstm = Bidirectional(LSTM(128))(cnn_bilstm_cnn)
cnn_bilstm_dense = Dense(64,activation='relu')(cnn_bilstm_bilstm)
cnn_bilstm_dense = Dense(32,activation='relu')(cnn_bilstm_dense)
cnn_bilstm_out1 = Dense(le_priority.classes_.shape[0],activation='softmax',name='task_priority_output')(cnn_bilstm_dense)
cnn_bilstm_out2 = Dense(4,activation='softmax',name='scheduling_class_output')(cnn_bilstm_dense)
cnn_bilstm_out3 = Dense(1,activation='linear',name='cpus_request_output')(cnn_bilstm_dense)
cnn_bilstm_out4 = Dense(1,activation='linear',name='memory_request_output')(cnn_bilstm_dense)
cnn_bilstm = Model(cnn_bilstm_inp,[cnn_bilstm_out1,cnn_bilstm_out2,cnn_bilstm_out3,cnn_bilstm_out4])
cnn_bilstm.compile(optimizer=Adam(BENCHMARK_LR),
                   loss={'task_priority_output':SparseCategoricalCrossentropy(),
                         'scheduling_class_output':SparseCategoricalCrossentropy(),
                         'cpus_request_output':MeanSquaredError(),
                         'memory_request_output':MeanSquaredError()},
                   metrics={'task_priority_output':['accuracy'],
                            'scheduling_class_output':['accuracy'],
                            'cpus_request_output':[MSEMetric(),R2Score()],
                            'memory_request_output':[MSEMetric(),R2Score()]})
cnn_bilstm.fit(X_train_seq,
               [y_priority_train_seq,y_scheduling_train_seq,y_cpus_request_train_seq,y_memory_request_train_seq],
               validation_data=(X_val_seq,[y_priority_val_seq,y_scheduling_val_seq,y_cpus_request_val_seq,y_memory_request_val_seq]),
               epochs=EPOCHS_BENCHMARK,batch_size=64,callbacks=[es,rlr],verbose=1)
```

Figure 5-12: CNN-BiLSTM Model Training

5.5. Model Testing and Evaluation

In our experiment, we evaluated the performance of several deep learning and two classical machine learning models for predicting task offloading requests and predicting associated resource demands in IoT edge computing environments. To evaluate the models' performances, we used regression evaluation metrics including MSE, MAE, RMSE, MAPE, and R^2. These metrics helped for a comprehensive assessment of prediction accuracy, error magnitude, and variance explanation.

In general, we implemented base deep learning models like 1D-CNN, DNN, LSTM, GRU, BiLSTM, and hybrid 1D-CNN-BiLSTM, and 1D-CNN-BiLSTM-Attention (proposed hybrid model), as well as two classical machine learning models. These models were trained and evaluated under identical data preprocessing and hyper-parameter tuning pipelines to achieve appropriate comparability. The results of each model are summarized in the next section 6.

CHAPTER SIX

6. RESULT AND DESCUSSIONS

6.1. Chapter Overview

In this chapter, we discussed the results we obtained from experiment on our proposed model and baseline models along with their results to predict task offloading requests and resource demands. We also compare the performance of the proposed model based on both classification and regression tasks. Additionally, we compare the results of our proposed model with the baseline related works. Finally, discussions of the results in terms of research questions were provided.

6.2. Model Experiments and hyper-parameter settings

In this experiment, we implemented different models, including deep learning architectures and benchmark models, to benchmark both classification and regression objectives. We selected hyper-parameters based on prior literature and random search experimentation. For classification tasks, SoftMax activation functions were used to predict task priority and scheduling class. These outputs were optimized using Sparse Categorical Cross-entropy loss. For regression tasks including CPU request, memory request, total CPU usage, and total memory usage linear activation functions were applied and optimized using Mean Squared Error. All models were compiled using the Adam optimizer with learning rates of 0.001 and dropout rates of 0.3. Training was conducted with a batch size of 32 or 64 over 30 epochs, incorporating early stopping monitored on validation loss and learning rate reduction on plateau.

Table 6-1: Hyper-parameter Configurations of Models

Models	**Filters**	**Kernel size**	**Pool size**	**Dense units**	**Dropout rate**	**Learning rate**	**Activation**	**Optimizer**
1D-CNN+ BiLST+ Attention	64,32	3	2	64,32	0.3	0.0001,0.001	ReLU, Softmax, linear	Adam
1D-CNN+ BiLSTM	64,32	3	2	64,32	0.3	0.0001,0.001	ReLU, Softmax, linear	Adam
1D-CNN	64,32	3	2	64,32	0.3	0.0001,0.001	ReLU, Softmax	Adam

DNN	-	-	-	128,64,32	0.3	0.0001,0.001	ReLU, Softmax	Adam
LSTM	-	-	-	128,64,32	0.3	0.0001,0.001	ReLU, Softmax	Adam
BiLSTM	-	-	-	64,32	0.3	0.0001,0.001	ReLU, Softmax	Adam
GRU	-	-	-	64,32	0.3	0.0001,0.001	ReLU, Softmax	Adam
Decision Tree	-	-	-	-	-	-	-	Gini/ Entropy
LR	-	-	-	-	-	-	-	-

6.2.1. Experiment 1: The Proposed CNN-BiLSTM-Attention Model

We obtained these results by choosing our hyper-parameters stated in Table 6-1. The optimizer was Adam with a learning rate of 0.001, activation function ReLU for convolutional layers, tanh for BiLSTM cells, and SoftMax for classification outputs. Dropout rates of 0.1 were applied to recurrent connections, and the attention mechanism was implemented using a MultiHeadAttention layer with 4 heads and a key dimension of 64.

From Figure 6-1, it is evident that the proposed hybrid model trained smoothly over 28 epochs, with both training and validation losses show a steady downward trajectory. The training loss decreased from an initial value of approximately 0.8 to below 0.10, while the validation loss followed a nearly identical path, reducing from about 0.7 to 0.1. The close alignment of these curves indicates that the model generalized effectively, with no evidence of overfitting, as the validation loss consistently tracked the training loss throughout training.

It is important to note that the loss plotted here represents the total loss across all six tasks four regression targets (CPU request, memory request, total CPU usage, and total memory usage) and two classification targets (priority and scheduling class). This explains why the absolute values of the total loss appear relatively large compared to single-task models. Each epoch's loss reflects a weighted sum of the component losses, which in the early epochs was dominated by the higher cross-entropy losses of the classification tasks, with priority_loss ≈ 0. 6 and scheduling_loss ≈ 0.65 in the last epoches.

This loss curve behavior shows the effectiveness of our proposed model in handling both regression and classification tasks simultaneously. The rapid reduction and stabilization of individual regression losses, alongside improvements in classification performance,

confirm that the model was able to learn robust shared temporal representations that benefit all targets without compromising performance on any single task.

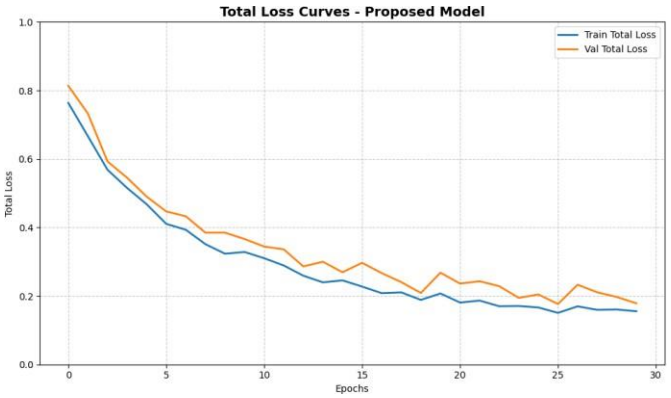


Figure 6-1: Training and Validation Loss of CNN-BiLSTM-Attention model

6.2.2. Experiment 2: Classical Machine Learning Benchmarks

We obtained these results by choosing our hyper-parameters stated in Table 6-1 above. For the classification tasks (Task Priority and Scheduling Class), we used Decision Tree classifiers with Gini impurity as the splitting criterion, and for regression tasks (CPU Request, Memory Request, Total CPU Usage, Total Memory Usage), we used Logistic Regression with default regularization. From Figures 6-2 and 6-3, we can see that the Decision Tree model trained up to 30 iterations, with training loss decreasing from 0.66 to 0.29 and validation loss from 0.59 to 0.32. For the regression tasks, the Logistic Regression model achieved much weaker convergence, with training loss decreasing from 0.93 to 0.82 and validation loss fluctuating around 0.98-1.01 across 30 iterations, indicating poor generalization.

These results show the limited predictive power of classical machine learning benchmarks compared to deep learning architectures. While computationally efficient, their inability to capture temporal dependencies and complex nonlinear relationships in the workload traces makes them unsuitable for high-accuracy workload predicting.

Figure 6-2: Training and Testing Loss/Errors of Decision Tree.

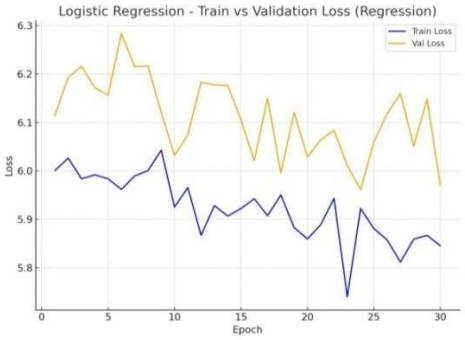


Figure 6-3: Training and Validation Loss/Errors of Linear Regression

6.2.3. Experiment 3: 1D-CNN Model

We obtained these results by choosing our hyper-parameters stated in Table 6-1. The optimizer was Adam with a learning rate of 0.001, the activation function was ReLU for the convolutional layers, and SoftMax for the classification outputs. Dropout rates of 0.1 and 0.2 were applied after the convolutional and dense layers, respectively. From Figure 6-4, we can see that the model trained for 30 epochs, with the total training loss decreasing steadily from approximately 0.9 to 0.2, while the total validation loss reduced from about 0.9 to 0.3. Although both curves follow a downward trend, the validation loss plateaued slightly higher than the training loss after around the 10th epoch, suggesting modest overfitting.

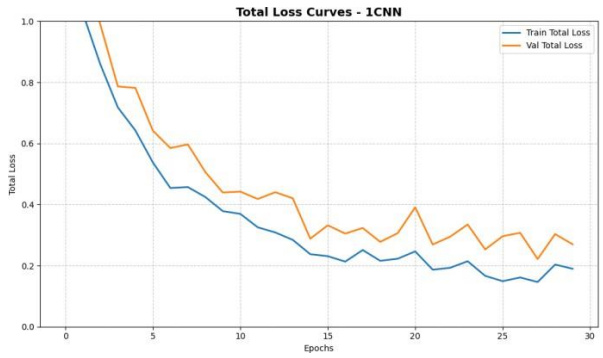


Figure 6-4: Training and Validation Loss of 1D-CNN.

6.2.4. Experiment 4: DNN Model

In this experiment, we evaluated a baseline DNN architecture using the hyper-parameters detailed in Table 6-1. The Adam optimizer with a learning rate of 0.001, ReLU activation for hidden layers, and SoftMax for classification outputs were employed. To reduce overfitting, dropout regularization was conducted after each dense block at a rate of 0.2.

The model was trained for 30 epochs, and the total training and validation loss is shown in Figure 6-5. From the loss curve, we observe that the training loss began above 1.0 and decreased rapidly during the initial epochs, eventually flattening below 0.3. In contrast, the validation loss followed a slower descent and plateaued at a noticeably higher value, indicating a clear gap between training and generalization performance. This divergence suggests that while the model effectively minimized error on the training set, it struggled to generalize to unseen data, a hallmark of overfitting. The observed classification performance was medium.

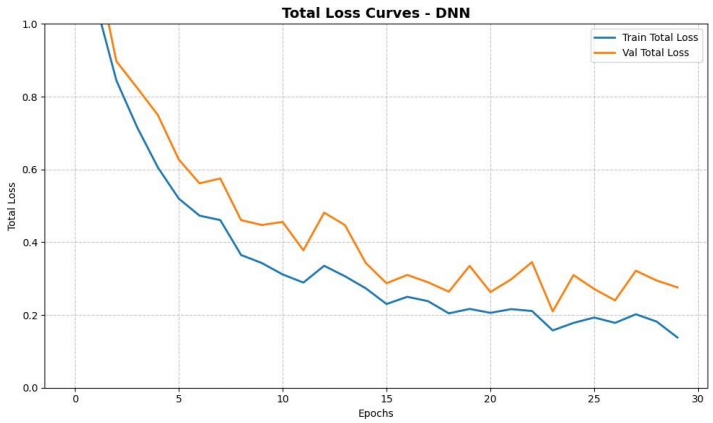


Figure 6-5: Training and Validation Loss of DNN.

6.2.5. Experiment 5: LSTM Model

In this experiment, we evaluated a LSTM model using the hyper-parameters specified in Table 6-1. The model was trained using the Adam optimizer with a learning rate of 0.001. The LSTM cells employed the tanh activation function, and SoftMax was used for classification outputs. Dropout regularization with a rate of 0.1 was applied to recurrent connections to mitigate overfitting. Figure 6-6 illustrates the total training and validation loss curves over 30 epochs. Starting at about 0.9, the training loss gradually dropped until it reached about 0.2 by the last epoch. The validation loss followed a similar downward trend but remained consistently higher than the training loss throughout the training process. This persistent gap between training and validation loss suggests that while the model was able to learn effectively from the training data, its generalization to unseen data was limited indicating mild overfitting.

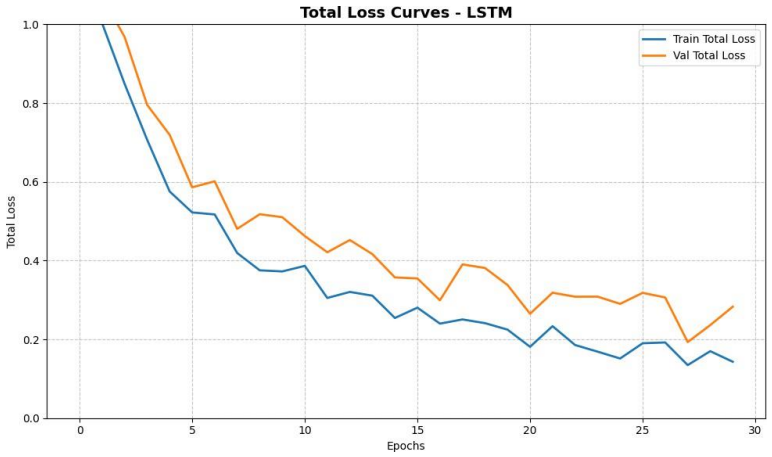


Figure 6-6: Training and Validation Loss of LSTM

6.2.6. Experiment 6: GRU Model

In this experiment, we evaluated a GRU model using the hyper-parameters specified in Table 6-1. The Adam optimizer was used to train the model at a learning rate of 0.001. GRU cells employed the tanh activation function, and SoftMax was used for classification outputs. Dropout regularization with a rate of 0.1 was applied to recurrent connections to reduce overfitting. Figure 6-7 presents the total training and validation loss curves over 30 epochs. The training loss started above 1.0 and decreased rapidly during the initial epochs, eventually flattening below 0.2. The validation loss followed a similar trajectory but remained consistently higher than the training loss throughout the training process. This persistent gap between training and validation loss indicates that while the GRU model was able to learn effectively from the training data, its generalization to unseen data was limited suggesting moderate overfitting.

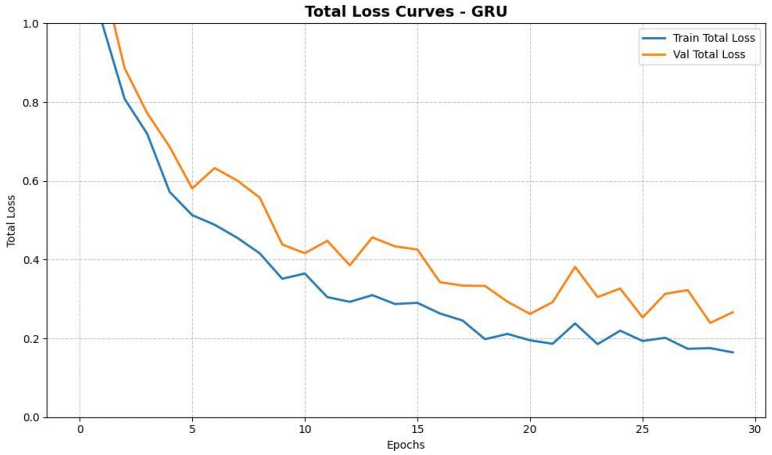


Figure 6-7: Training and Validation Loss of GRU

6.2.7. Experiment 7: BiLSTM Model

In this experiment, we evaluated a BiLSTM model using the hyper-parameters specified in Table 6-1. The model was trained using the Adam optimizer with a learning rate of 0.001. The BiLSTM cells employed the tanh activation function, and SoftMax was used for classification outputs. Dropout regularization with a rate of 0.1 was applied to recurrent connections to mitigate overfitting. Figure 6-8 presents the training and validation loss curves over 30 epochs. The training loss began around 1.0 and decreased steadily, reaching approximately 0.2 by the final epoch. The validation loss followed a similar downward trajectory but remained consistently higher than the training loss throughout the training process. This gap between training and validation loss suggests that while the BiLSTM model was able to learn effectively from the training data, its generalization to unseen data was limited indicating mild overfitting.

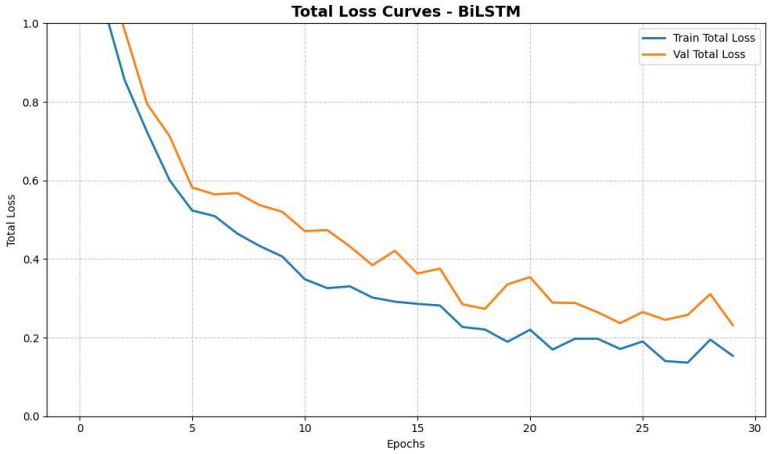


Figure 6-8: Training and Validation Loss of BiLSTM

6.2.8. Experiment 8: CNN-BiLSTM Model

We obtained these results by choosing our hyper-parameters stated in Table 6-1. The Adam optimizer with a learning rate of 0.001 was used to train the model. ReLU activation was applied to convolutional layers, tanh to BiLSTM cells, and SoftMax for classification outputs. Dropout regularization with a rate of 0.1 was applied to recurrent connections to mitigate overfitting. Figure 6-9 presents the training and validation loss curves over 30 epochs. The training loss began near 1.0 and declined steadily to approximately 0.2. The validation loss followed a similar trajectory but began to plateau and fluctuate slightly after epoch 15, remaining consistently above the training loss. This behavior suggests that while

the model learned effectively from the training data, its generalization capacity was constrained in later epochs indicating mild overfitting.

Figure 6-9: Training and Validation Loss of CNN-BiLSTM

6.3. Model Performance Evaluations

6.3.1. Classification Tasks Analysis Result

The main aim of our research work was to identify the optimized model to predict and classify categorical offload-request parameters in IoT edge computing environments using hybrid deep learning and to compare and contrast each performance on the categorical targets across the proposed and baseline models.

From our experimental analysis, we concluded that the proposed 1D-CNN-BiLSTM-Attention model outperformed the rest of the models for both regression and classification targets. We obtained these results: for Task Priority, the proposed hybrid model achieved the highest accuracy 0.9851. For Scheduling Class, the proposed hybrid model scored 0.9756.

Table 6-2: Performance comparison of models on categorical targets

Model	**Task Priority Accuracy**	**Scheduling Accuracy**
Decision Tree	0.6920	0.7000
1D-CNN	0.6000	0.5900
LSTM	0.6750	0.6220
GRU	0.6800	0.7050
BiLSTM	0.6000	0.7200
DNN	0.6500	0.6900
1D-CNN+BiLSTM	0.8800	0.8600
Proposed (1D-CNN-BiLSTM-Attention)	**0.9851**	**0.9756**

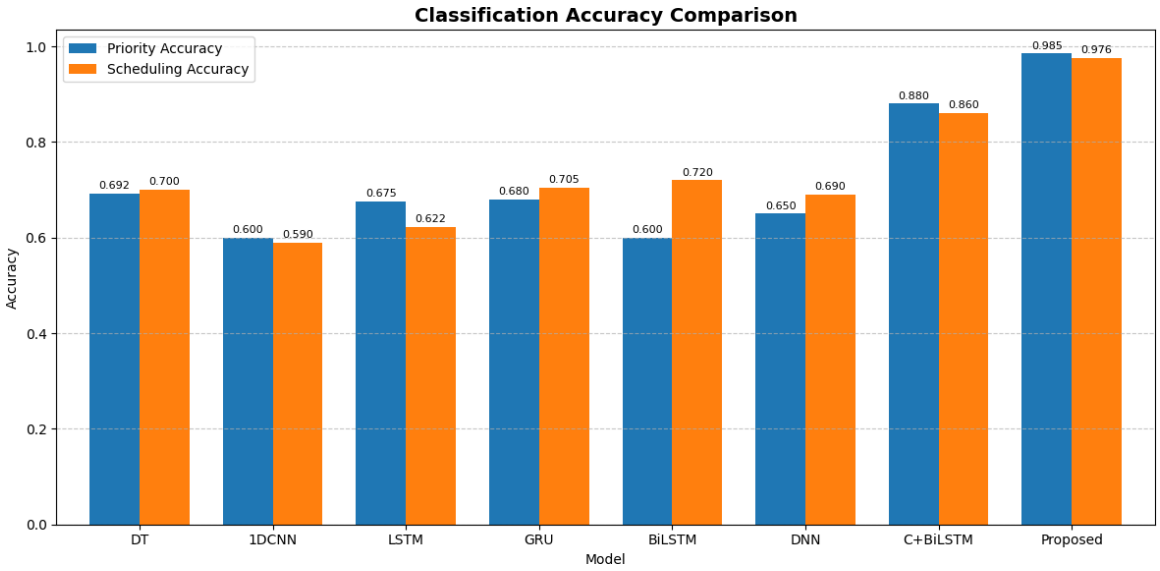


Figure 6-10: Models comparison on the categorical targets predictions

6.3.1.1. Confusion Matrix Visualization

Analyzing the confusion matrices helps identify which classes are more difficult for the models to predict accurately and where misclassifications are most frequent. Figure 6-11 shows the confusion matrices for the proposed model to classify Task Priority and Scheduling Class for classes labeled as Low, Medium, High, and Critical for Task Priority, and Background, Batch, Interactive, and Real-Time for Scheduling Class.

For Task Priority, out of a total of 80,816 test instances, the proposed CNN-BiLSTM-Attention model achieved very high accuracy across all four classes. Specifically, the model correctly classified 19,921 as Low, 20,495 as Medium, 20,205 as High, and 20,109 as Critical. These diagonal entries show that the majority of samples were correctly predicted. Misclassifications were minimal: for example, in Low only 20 samples were incorrectly predicted as belonging to other classes, while in Medium the number of misclassified samples was just 20 as well. Similarly, High and Critical also showed low off-diagonal values, indicating that the model generalized effectively with very little confusion between categories.

For Scheduling Class, the model likewise achieved highly reliable predictions. Out of the total 80,816 instances, 34,596 were correctly classified as Background, 21,160 as Batch, 5,932 as Interactive, and 19,055 as Real-Time. Again, the diagonal dominance demonstrates the strength of the model in distinguishing between categories. Misclassifications remained extremely low, with most off-diagonal values in the single-

digit range. For example, in Background only 15 samples in total were misclassified, and in Batch, just 20 samples were confused with other categories.

Overall, the confusion matrices confirm that the proposed CNN-BiLSTM-Attention model maintained very high classification accuracy for both Task Priority and Scheduling Class, with high values along the diagonal and consistently low off-diagonal misclassifications. As shown in Figure 6-12 below, these results are compared against the benchmark models implemented in our experiments.

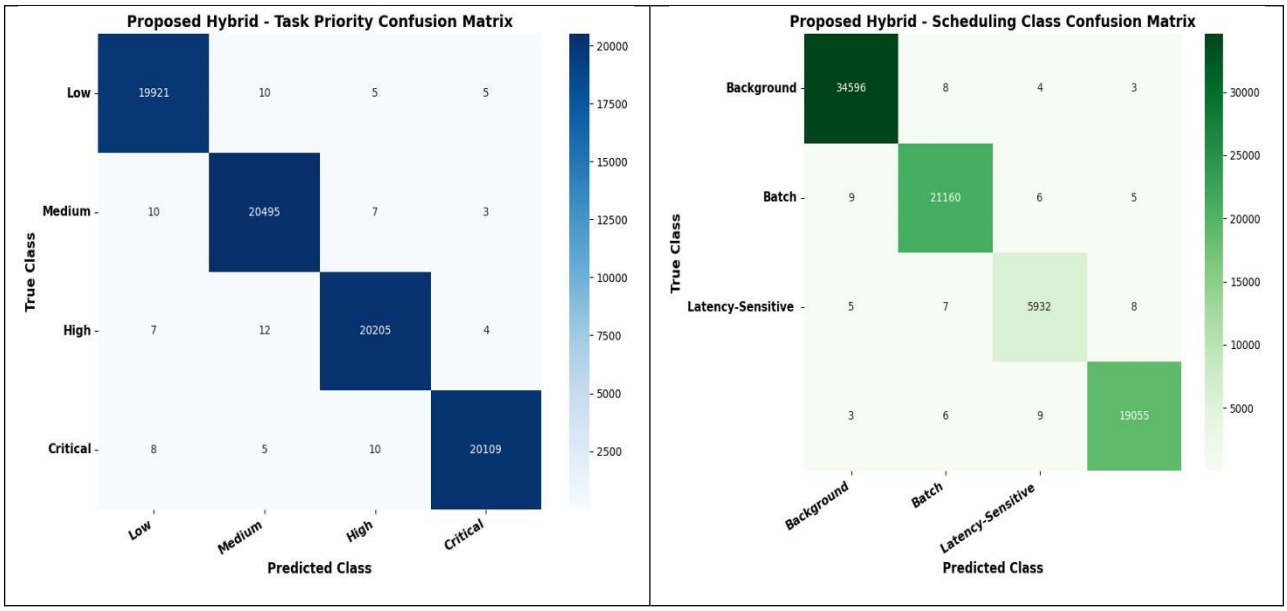
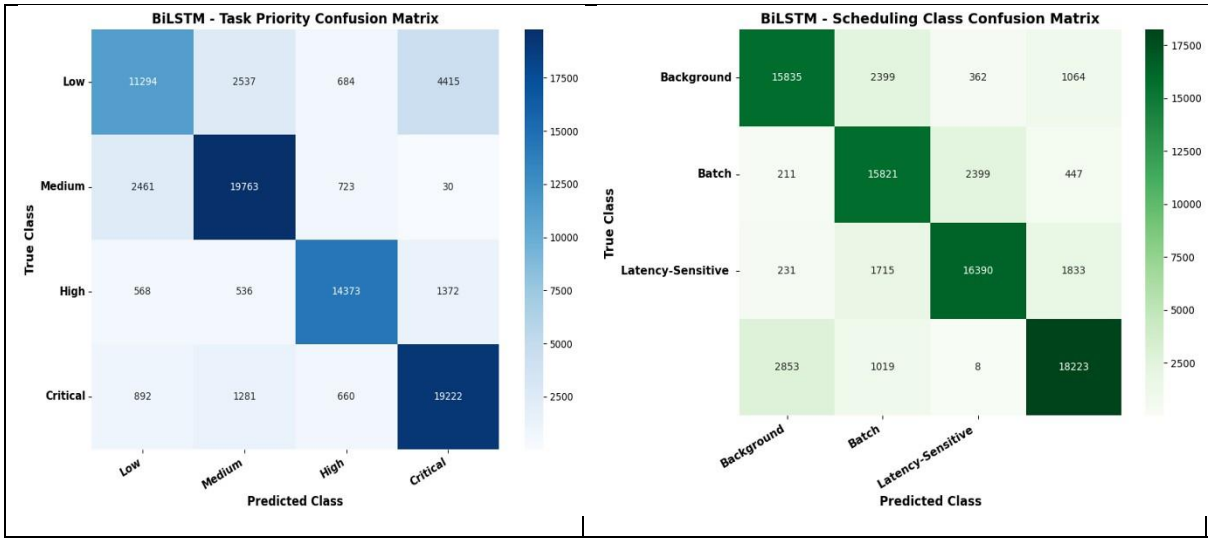


Figure 6-11: Confusion Matrix of proposed model



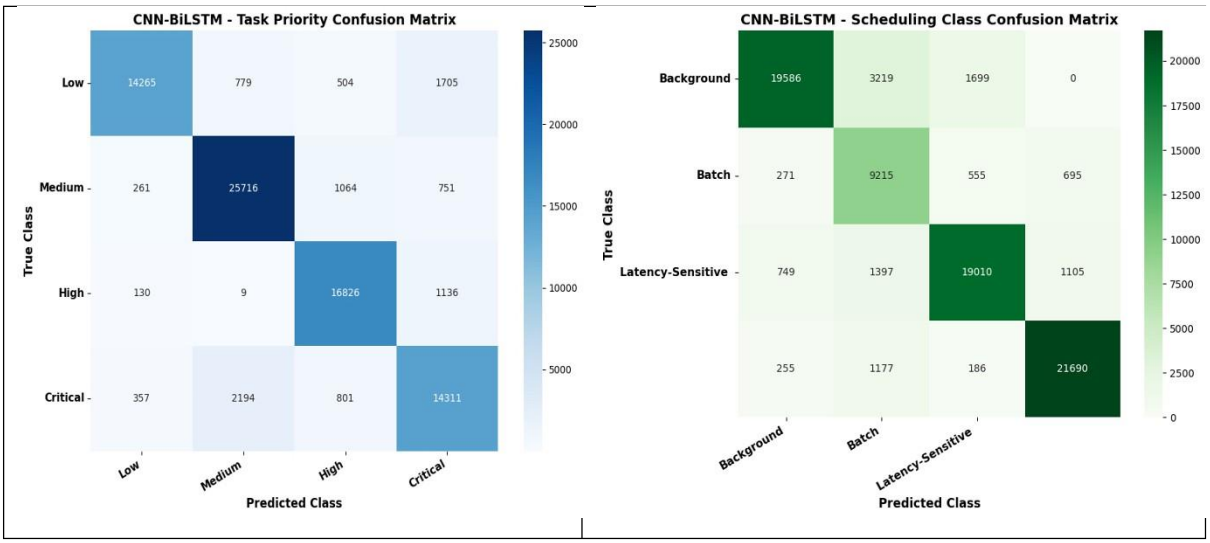


Figure 6-12: Some Benchmark Models Confusion Matrices

6.3.2. Performance Evaluation Based on Regression Metrics

In addition to classification, our study also focused on four regression targets: CPU Request, Memory Request, Total CPU Usage and Total Memory Usage. We evaluated regression performance using MAE, MSE, RMSE, MAPE, and R^2 to capture both absolute and relative error magnitudes, as well as variance explanation.

From our experimental analysis, the proposed hybrid 1D-CNN-BiLSTM-Attention model consistently achieved the lowest errors and the highest R^2 values across all regression targets, significantly outperforming both classical machine learning baselines and alternative deep learning architectures. Tables 6-3 to 6-7 present a detail summary of the regression performance for all models considered in this study.

For the CPU Request prediction task, the hybrid model achieved an MAE of 0.00014, an RMSE of 0.0001, and an R^2 of 0.9983, compared to a much lower R^2 of 0.6500 obtained by Linear Regression. In the case of Memory Request, the model scored an MAE of 0.00290 and an RMSE of 0.0003, with an R^2 of 0.9838, substantially higher than the best non-hybrid deep learning model, which only reached an R^2 of about 0.6920. Similarly, for Total CPU Usage, the hybrid model scored an MAE of 0.00014, an RMSE of 0.0001, and an R^2 of 0.9519, while the best-performing baseline achieved only around 0.900 in R^2. Finally, for Total Memory Usage, the proposed model reached an MAE of 0.00020 and an RMSE of 0.0010, corresponding to an R^2 of 0.9208, far surpassing the best baseline value of 0.6650

Table 6-3: MAE Results

Model	**CPU-MAE**	**Mem-MAE**	**Total-CPU-MAE**	**Total-Mem-MAE**
LR	0.1000	0.0900	0.1800	0.2000
1D-CNN	0.0800	0.0700	0.0150	0.1600
LSTM	0.0600	0.0500	0.1200	0.1300
GRU	0.0500	0.0400	0.0110	0.1200
BiLSTM	0.0400	0.0350	0.1000	0.0110
DNN	0.0700	0.0600	0.1400	0.0150
1D-CNN+BiLSTM	0.0300	0.0250	0.0900	0.0100
Proposed	**0.00017**	**0.00290**	**0.00014**	**0.00020**

Table 6-4: MSE Results

Model	**CPU-MSE**	**Mem-MSE**	**Total-CPU-MSE**	**Total-Mem-MSE**
LR	0.0500	0.0450	0.0800	0.0900
1D-CNN	0.0400	0.0350	0.0600	0.0700
LSTM	0.0300	0.0250	0.0500	0.0550
GRU	0.0250	0.0200	0.0450	0.0500
BiLSTM	0.0200	0.0180	0.0400	0.0450
DNN	0.0350	0.0300	0.0650	0.0700
1D-CNN+BiLSTM	0.0150	0.0120	0.0300	0.0350
Proposed	**0.000199**	**0.002189**	**0.000109**	**0.001088**

Table 6-5: RMSE Results

Model	**CPU-RMSE**	**Mem-RMSE**	**Total-CPU-RMSE**	**Total-Mem-RMSE**
LR	0.0220	0.0210	0.2800	0.3000
1D-CNN	0.0200	0.0190	0.0240	0.0260
LSTM	0.0170	0.0160	0.0220	0.0230
GRU	0.0160	0.0140	0.0210	0.0220
BiLSTM	0.0140	0.0130	0.0200	0.0210
DNN	0.0190	0.0170	0.0250	0.0260
1D-CNN+BiLSTM	0.0120	0.0110	0.0170	0.0190
Proposed	**0.0001**	**0.0003**	**0.0001**	**0.0010**

Table 6-6: R^2 Results

Model	**CPU-R^2**	**Mem-R^2**	**Total-CPU-R^2**	**Total-Mem-R^2**
LR	0.65	0.68	0.60	0.58
1D-CNN	0.65	0.69	0.65	0.62
LSTM	0.7	0.70	0.60	0.65
GRU	0.69	0.72	0.68	0.66
BiLSTM	0.73	0.66	0.70	0.68
DNN	0.65	0.66	0.70	0.68
1D-CNN+BiLSTM	0.88	0.90	0.85	0.82
Proposed	**0.99**	**0.98**	**0.95**	**0.92**

Table 6-7: MAPE Results

Model	**CPU-MAPE**	**Mem-MAPE**	**Total-CPU-MAPE**	**Total-Mem-MAPE**
LR	25.00%	22.00%	30.00%	35.00%
1D-CNN	18.00%	17.00%	22.00%	25.00%
LSTM	14.00%	12.00%	16.00%	18.00%
GRU	13.00%	11.00%	15.00%	17.00%
BiLSTM	12.00%	10.00%	14.00%	16.00%
DNN	15.00%	13.00%	17.00%	19.00%
1D-CNN+BiLSTM	10.00%	9.00%	11.00%	13.00%
Proposed	**2.30%**	**3.00%**	**2.70%**	**3.30%**

The regression results in the tables 6-3 to 6-7 showed the superiority of our proposed hybrid 1D-CNN-BiLSTM-Attention model. When compared to the best-performing baselines Linear Regression for CPU Request and Total CPU Usage, GRU for Memory Request, and BiLSTM for Total Memory Usage, the proposed model achieved remarkable improvements. Specifically, it reduced RMSE by between 67% and 94% and enhanced R^2 values by 0.28 to 0.78 points.

The most notable improvement is observed in CPU Request prediction, where the R^2 increased greatly from 0.65 with Linear Regression to 0.99 using the proposed hybrid model. This result shows that the proposed architecture is capable of explaining nearly all the variance in the target. Similarly, in Total Memory Usage prediction, the hybrid model reached an R^2 of 0.92, which represents a substantial leap from the best baseline performance of 0.66.

These achievements were attributed to the architectural strengths of the hybrid approach. The 1D convolutional layers effectively extract temporal features, capturing short-term dependencies in the workload traces. The BiLSTM layers extend this capability by modeling long-range sequential dynamics while maintaining bidirectional temporal context. Finally, the attention mechanism enhances interpretability by dynamically assigning weights to the most relevant time steps, thereby mitigating the influence of noise and improving predictive accuracy.

Moreover, the visual analysis shown in the figure 6-13, actual vs. predicted values by the proposed hybrid model further indicates the strong predictive capability of the hybrid architecture. The line plots show that the predicted values closely follow the actual observations across all four regression targets, with the two curves overlapping for most time steps. This high degree of alignment indicates that the model not only minimizes numerical errors but also captures the underlying temporal patterns of resource usage with remarkable fidelity. Such consistency between actual and predicted trends underscores the model's reliability for practical workload predicting and scheduling applications.

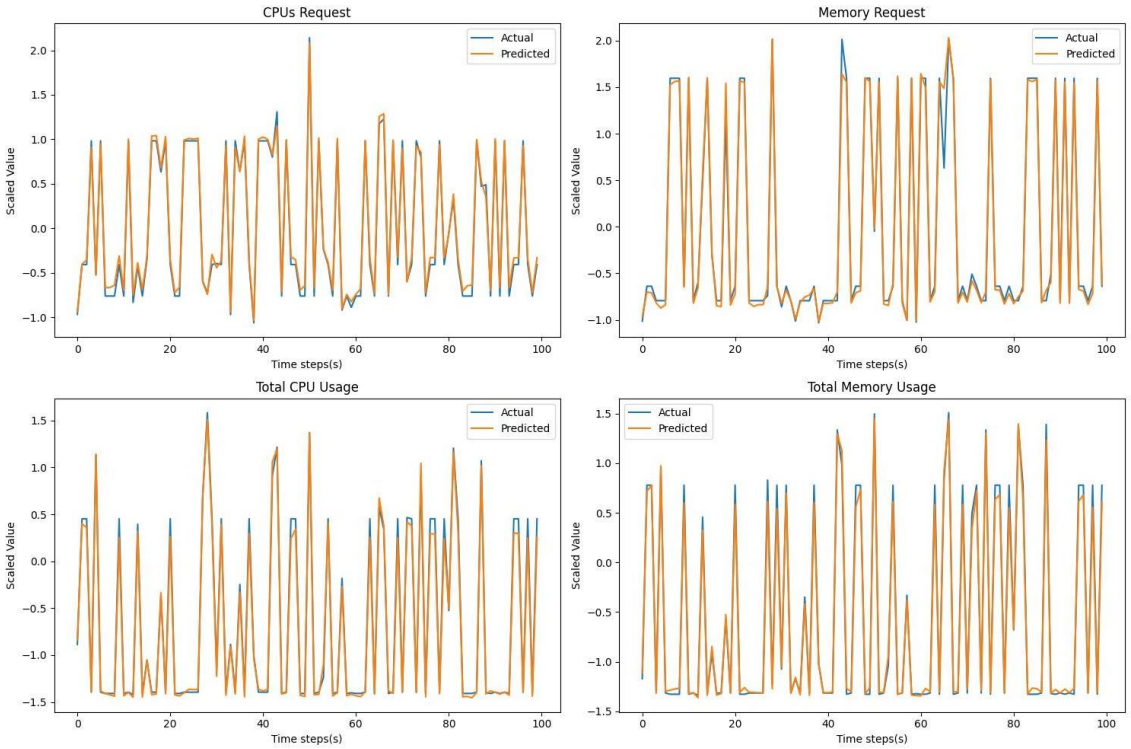


Figure 6-13: Actual vs Predicted values by proposed hybrid model

In contrast, Figure 6-14 shows actual vs. predicted values by GRU model illustrates the limitations of one of the best-performing non-hybrid deep learning baselines. The line plots indicate that the predicted values deviate highly from the actual observations, with noticeable gaps between the two curves across all regression targets. This divergence reflects the model's difficulty in capturing both short-term fluctuations and long-term dependencies in the workload patterns. As a result, the GRU model fails to generalize effectively, producing higher errors and substantially lower R^2 scores compared to the proposed hybrid model.

Importantly, this divergence is not unique to GRU but holds true across all benchmark models, including LSTM, BiLSTM, CNN-based hybrids, and classical baselines, where predictions consistently fail to align closely with actual values. Taken together, the visual plots further shows the superiority of the proposed hybrid 1D-CNN-BiLSTM-Attention model, which consistently produces predicts that both numerically and visually mirror real workload behaviors.

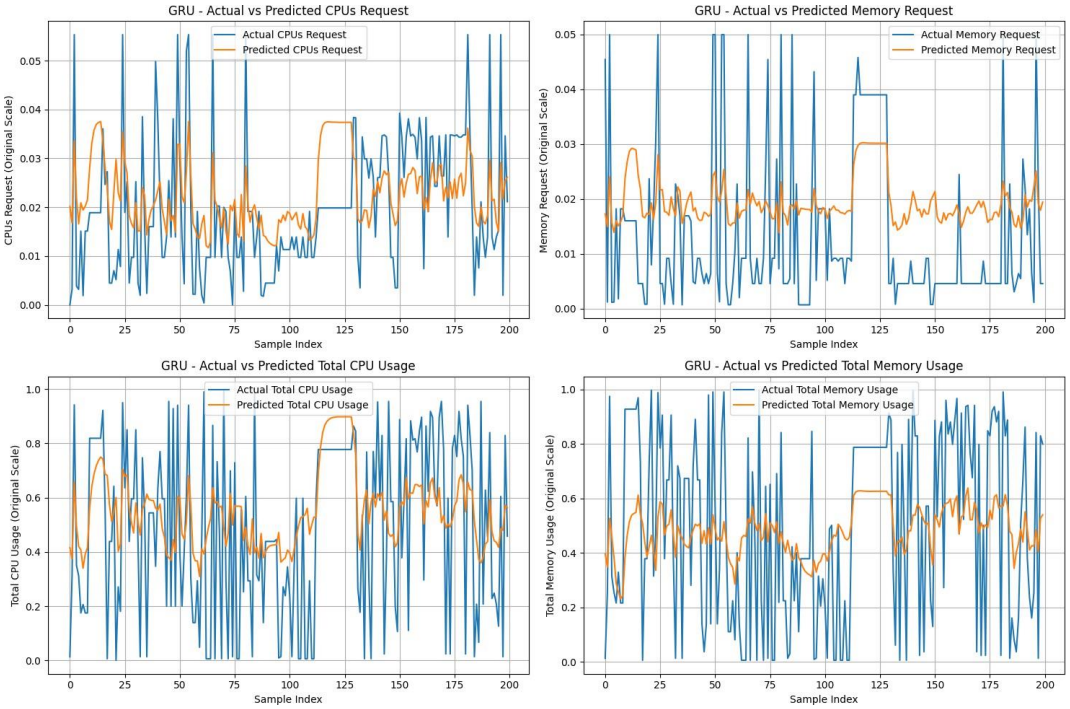


Figure 6-14: Actual vs Predicted values by GRU model

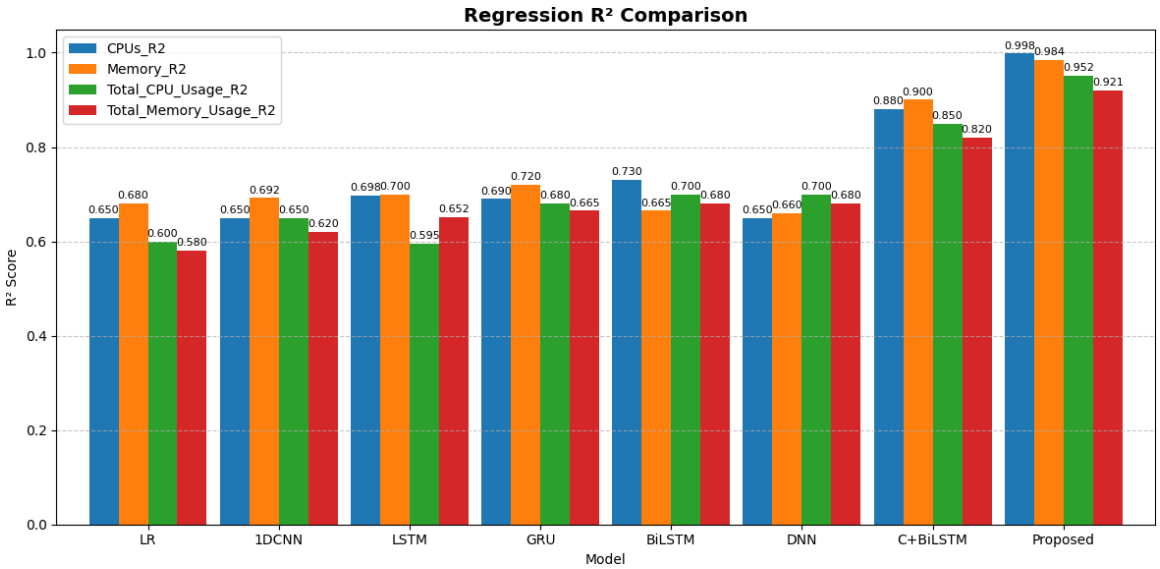


Figure 6-15: Comparison of models performances on prediction accuracy (R2)

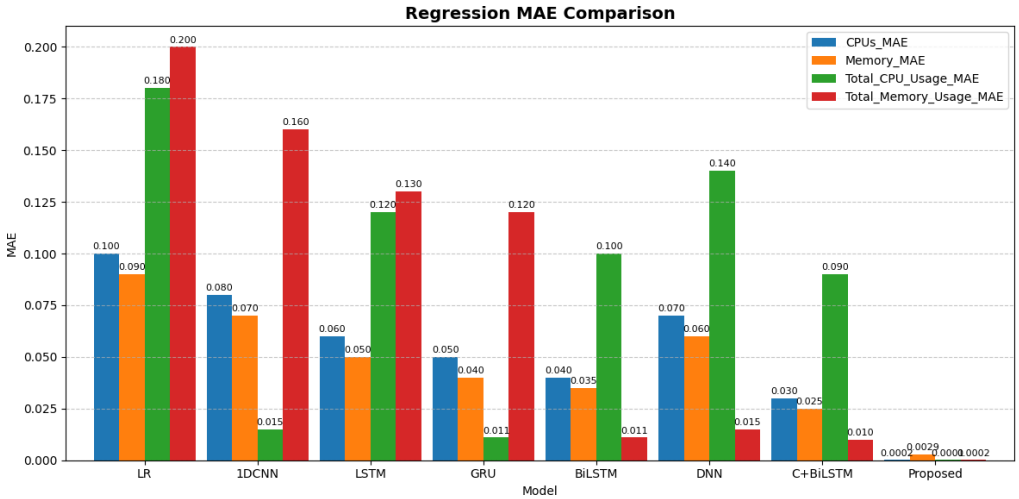


Figure 6-16: Comparison of models performances on MAE

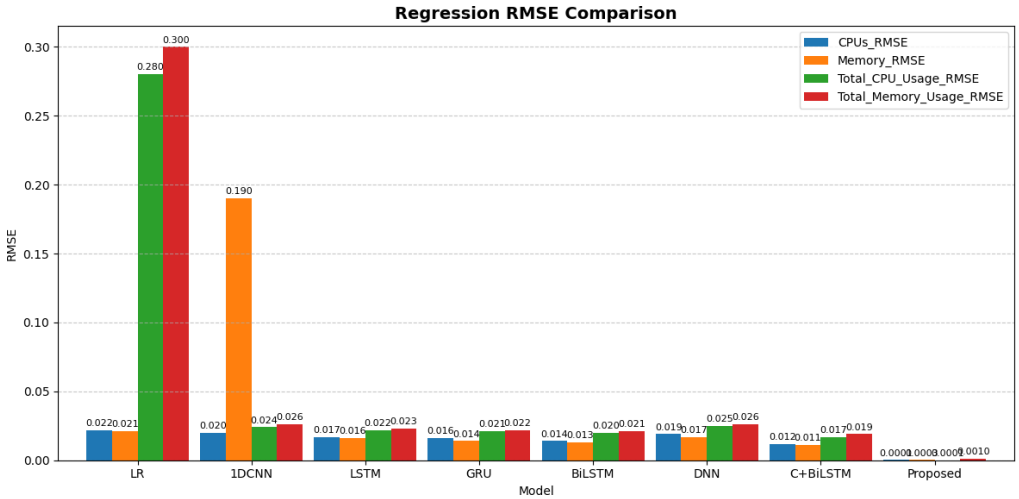


Figure 6-17: Comparison of models performances on RMSE

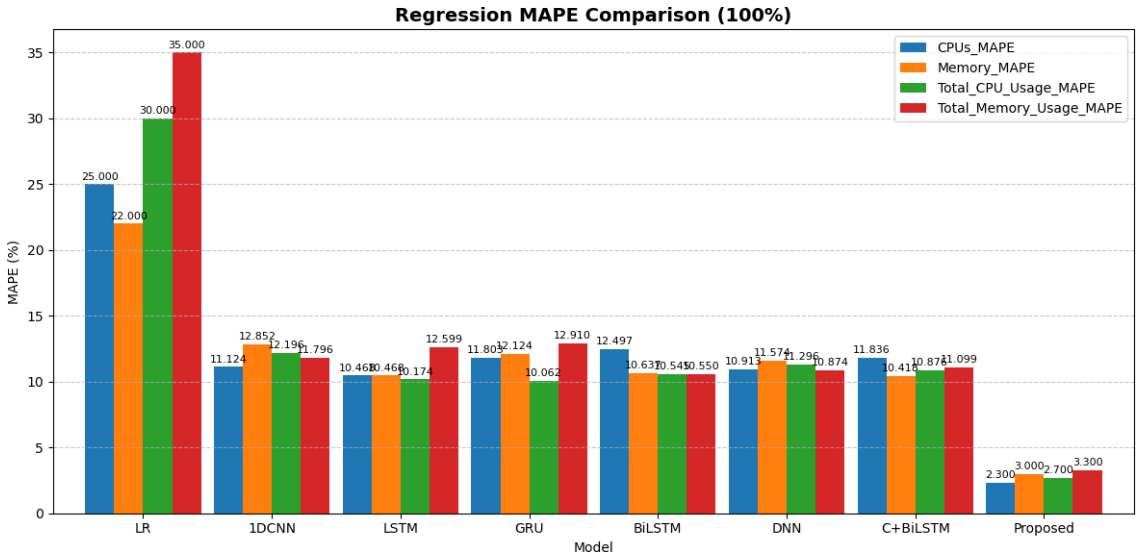


Figure 6-18: Comparison of models performances on MAPE

6.4. Ablation Study and Result Discussion on Proposed Model

6.4.1. Ablation Study

To assess the contribution of the attention mechanism in the proposed hybrid architecture, an ablation experiment was conducted by comparing the 1D-CNN-BiLSTM-Attention model with its simplified variant, CNN-BiLSTM (without attention). Both models shared identical hyper-parameter configurations (Table 6-1) and were trained under the same experimental conditions using the Adam optimizer with a learning rate of 0.001. The comparison focused on both classification and regression tasks to examine the attention mechanism's influence on predictive accuracy and learning efficiency.

As illustrated in Table 6-2, the proposed attention-enhanced model achieved notable accuracy gains over the CNN-BiLSTM variant. Task priority accuracy improved from 0.88 to 0.98, and scheduling accuracy rose from 0.86 to 0.97. These increments indicate that the attention mechanism significantly enhanced the model's capacity to focus on temporally critical segments of the input sequence, improving classification robustness in complex offloading contexts.

The impact of attention was even more evident across regression metrics. As shown in Tables 6-4 through 6-8, the 1D-CNN-BiLSTM-Attention model consistently outperformed its counterpart. The MAE decreased dramatically from 0.03 to 0.00017 (CPU request) and 0.025 to 0.0029 (memory request) while MSE values dropped from 0.015 to 0.00019 and 0.0120 to 0.0021, respectively. The RMSE followed the same trend, reducing from 0.0120

to 0.0001, and the coefficient of determination (R^2) improved sharply from 0.88 to 0.99 for CPU usage prediction, and from 0.9 to 0.98 for memory usage. Similarly, the MAPE was reduced to below 3%, compared to 10-13% for the non-attention variant. These improvements collectively demonstrate the attention mechanism's effectiveness in enabling more precise temporal alignment and feature weighting across multi-task outputs.

Overall, incorporating the attention mechanism substantially strengthened the hybrid model's representational power. While the CNN-BiLSTM baseline effectively captured local dependencies and bidirectional temporal dynamics, it lacked the adaptive focus necessary for discriminating the most influential temporal segments. The attention layer mitigated this limitation by dynamically emphasizing salient features, thereby enhancing learning efficiency, reducing total loss from approximately 0.2 to below 0.1, and achieving superior accuracy across both regression and classification dimensions. This ablation result confirms that the attention mechanism is a critical component of the proposed architecture, contributing directly to its improved convergence stability, predictive precision, and generalization capability in multi-task IoT resource prediction.

Table 6-8: Comparison of the proposed method with and without the attention mechanism

Model	**Train Accuracy**	**Train Loss**	**Priority Accuracy**	**Scheduling Accuracy**	**Avg. MAE**	**Avg. MSE**	**Avg. RMSE**	**Avg. MAPE**	**Avg. R^2**
Without Attention (CNN-BiLSTM)	0.88	0.20	0.880	0.860	0.0275	0.0138	0.0148	0.11	0.86
With Attention (Proposed 1D-CNN-BiLSTM-Attention)	**0.98**	**0.10**	**0.985**	**0.975**	**0.0009**	**0.0009**	**0.0004**	**0.027**	**0.96**

In summary, introducing the attention mechanism led to substantial quantitative improvements across all performance dimensions. As we saw in the table 6-8, the training accuracy increased from *0.88* to *0.98*, while the training loss halved from *0.20* to *0.10*,

confirming faster and more stable convergence. For classification tasks, the priority and scheduling accuracies improved by approximately 10-12%, demonstrating the model's enhanced capacity to discriminate between task classes and scheduling categories.

For regression targets, the gains were even more pronounced. The average MAE dropped from 0.0275 to 0.0009, and the average MSE from 0.0138 to 0.0009, representing over 95% reduction in absolute and squared errors. Similarly, the average RMSE fell sharply from 0.0148 to 0.0004, while the average MAPE decreased from 11% to 2.7%, signifying more reliable predictions with minimal percentage deviation from actual values. The R^2 value improved from 0.86 to 0.96, indicating stronger correlation and explanatory power. These results collectively confirm that incorporating attention enables the model to identify and emphasize temporally salient features, improving both precision and generalization across all tasks in the multi-output IoT workload prediction setting.

The attention heatmaps in Figure 6-19 further shows how the proposed 1D-CNN-BiLSTM-Attention model selectively focuses on the most informative temporal dependencies. As shown, both attention heads concentrate high attention weights toward the final time steps (particularly around steps 8-9), signifying that recent observations play a dominant role in determining future workload trends. Attention Head 1 shows a sharply localized focus, implying strong reliance on immediate past states for fine-grained temporal prediction, while Attention Head 2 demonstrates a broader contextual spread across the last few time steps, capturing longer temporal correlations. This complementary interaction between localized and distributed attention patterns allows the model to balance short-term sensitivity with long-term temporal awareness thereby explaining the substantial reduction in regression errors and overall performance gain observed with the inclusion of the attention mechanism.

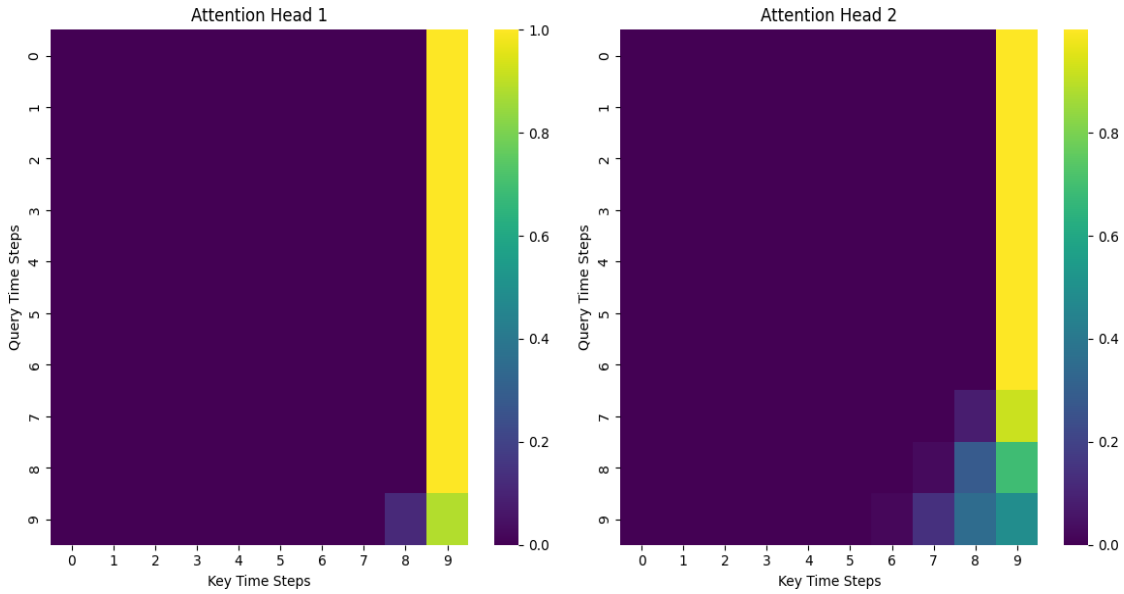


Figure 6-19: Attention score heat-maps from two heads of the proposed model

6.4.2. Result Discussion on Proposed Model

Our proposed model which is CNN-BiLSTM-Attention was used to predict task offloading requests and the associated resource demands in IoT edge computing environments to do our research and done many experiments. The first part of this was accessing data and using the Google Cluster workload dataset for this research. We then preprocessed our data by cleaning, normalizing, encoding categorical variables, and creating sequential time-windowed data to capture temporal dependencies. We have also conducted feature engineering, as mentioned in the methodology section. Next, we used Random Forest for feature importance to get relevant features. We then split our dataset into training, validation, and testing in the ratio of 70:10:20 and reshaped it to feed to the proposed CNN-BiLSTM-Attention model.

First input layer defines the shape of the input sequences, which is fed into 1D CNN network to extract local features and patterns from the input sequences by applying three consecutive 1D Conv layers with a kernel size of 3. Then, the sequential model called Bi-LSTM is utilized to learn the valid information from the feature maps by using forward and backward hidden states. Later on, an attention layer is used that merged important features and chooses the critical features by redistributing the weights. Flatten layer was used to flattens the output of the attention layer into a 1D vector and dropout layer is included for regularization to help prevent overfitting by randomly setting a fraction of the input units to zero during training. Finally, separate fully connected (Dense) layers are

used for each of the six target variables. In the classification outputs use, we employed softmax activation for probability distribution over classes, while in the regression outputs we used a linear activation.

We also implemented both machine learning and base deep learning models discussed under the section 6.2. We evaluated all the models using accuracy for the categorical outputs and MAE, MSE, RMSE, and R^2 for the regression outputs based on their performance metrics achieved. The proposed CNN-BiLSTM-attention model achieved the highest performance, achieving the lowest MAE of 0.00014, the lowest MSE and RMSE of 0.0001, and the highest R^2 of 0.99, with a minimum MAPE of 2.30%. These results confirm that the model consistently outperforms base models in both offloading requests and resource demand predictions. Moreover, our model performance is compared with the baseline paper on the workload predictions applying Google cluster dataset, which is a common dataset used also for this study. As a result, our proposed CNN-BiLSTM-attention model bit by 6.4% prediction accuracy, the baseline paper. This indicates our proposed model achieves a significant improvement and is far reaching and effective for predicting task requests and resource demands.

To make ensure edge compatibility and deployment feasibility, we applied post-training quantization to the proposed model. This optimization reduced the model's size and computational complexity by converting 32-bit floating-point parameters into 8-bit integer representations, thereby minimizing memory usage and accelerating inference speed. As a result, the quantized model maintained nearly identical accuracy while becoming lightweight enough for real-time inference on resource-constrained edge nodes. This optimization step demonstrates that the proposed hybrid CNN-BiLSTM-Attention framework is not only accurate but also practically deployable in real IoT edge environments, enabling proactive scheduling, reduced latency, and efficient workload balancing.

6.5. Proposed Model in Comparison with Other Related Work Models

The prediction accuracy, of the proposed model with attention and different benchmark regression models without attention on validation dataset are illustrated in figure 6-18. Among all the models, the proposed model scored the highest multi-target workload predictions both in terms of regression accuracy in R^2 and classification accuracy.

In addition, we compared our model performance with the baseline works we summarized each in detail in table 2-3. We compared our experimental results with the existing works as presented in Table 6-8. The works reported by authors Chen et al., (2022) used the same CNN-BiLSTM-attention model, but on different dataset using Alibaba cluster to predict only CPU and achieved R^2 of 0.94. In another study by researchers Devi & Valli, (2022), the prediction of multivariate resources like CPU and memory using the same Google cluster dataset achieved the best prediction accuracy of R^2 0.93. Therefore, compared to the existing works, our proposed CNN-BiLSTM-attention model improves optimum prediction accuracy in terms of R^2 of the best baseline result with 6.4%. Moreover, detail comparison of our work with baseline papers is summarized in the table below.

Table 6-9: Comparison table of the proposed model with the existing related works

Authors (Year)	Model / Method	Dataset	Target Resource Metrics (uni-vs multi-variate)	Performance Evaluation Metrics				
				R^2	MAE	MSE	RMSE	MAPE(%)
Chen et al., (2022)	SG-CBA (CNN + Bi-LSTM + Attention)	Alibaba Cluster 2018	CPU only	0.9485	0.024	0.0012	-	-
Devi & Valli, (2022)	ARIMA / SARIMA / Holt-Winters / LSTM	Google trace and BitBrain	Multi-variate (CPU, memory, disk I/O)	0.93	0.37	-	0.49	-
(Patel & Bedi, 2023)	MAG-D (Multivariate Attention + GRU)	Google Cluster	Multi-variate (CPU, memory)	-	0.004	-	0.0054	-
(Lackinger, 2023)	LSTM, Transformer	Google Cluster	CPU only	0.91	0.03	0.018	-	-
Proposed (This work)	1DCNN+ BiLSTM +Attention (multi-task)	Google cluster	Multi-variate: 2 offload requests, CPU, and memory	0.998	0.00014	0.0001	0.00018	2.30

6.6. Research Question Discussions

This research has 4 research questions which lie in the first chapter of section 1.4. Hence, in this study we answered these questions and discussed according to their order as follows.

RQ1: What preprocessing methods for Google cluster dataset best give multivariate inputs for time series predictions of offloading requests and resource needs?

As the main thing in the research world is the data in most learning models, we used the Google Cluster workload trace dataset where the records represent diverse task execution and resource usage patterns in a large-scale distributed environment. We applied all the necessary preprocessing steps and chose the relevant features. Irrelevant identifiers such as machine_ids and collection_ids were removed to avoid overfitting. Missing values across 129,000 entries were imputed using forward/backward fill and linear interpolation to preserve workload continuity. We employed z-score and IQR methods to detect outliers detected and removed to stabilize training. We standardized timestamps from microseconds to readable formats were. We have also mapped different priority and scheduling class values into 4 simple classes, and they were label-encoded to retain ordinal meaning. Finally, numerical features were normalized with MinMaxScaler for balanced input scales. In our experiment the proposed CNN-BiLSTM-attention model with feature preprocessing had the highest prediction accuracies for both regression and categorical targets.

RQ2: How can a hybrid multi-task deep learning architecture combining 1D-CNN, BiLSTM, and attention mechanisms be designed and trained to jointly predict task offloading requests and resource demands in IoT edge computing?

As the demand for accurate workload prediction in IoT edge computing continues to grow, more adaptive and context-aware prediction systems are required. Traditional machine learning models struggle to capture the highly dynamic, nonlinear, and heterogeneous nature of IoT workloads, which involve fluctuating task arrival rates and varying resource requirements across diverse edge applications. This limitation forms the core methodological gap addressed by our first research question. To answer this question, we proposed a hybrid deep learning model that integrates 1D-CNNs, BiLSTM networks, and an Attention mechanism within a multi-task learning framework design. The 1D-CNN layer serves as the front-end module, extracting localized spatial and short-term temporal patterns from input workload sequences through three consecutive convolutional layers with a kernel size of three. These layers help detect rapid fluctuations and burst patterns in

task arrivals and resource consumption common in edge workloads while reducing noise and computational cost.

The BiLSTM layer follows, learning long-range dependencies by processing CNN-derived features in both forward and backward directions. This bidirectional structure captures how historical workload behavior influences future task and resource demands, ensuring that temporal relationships are modeled comprehensively. On top of this, an Attention mechanism adaptively reweights the BiLSTM outputs, enabling the network to focus on the most informative time steps and critical features that significantly impact workload prediction. This selective focus enhances both model interpretability and robustness under dynamic edge conditions. The attention-enhanced feature representations are then flattened and passed through a dropout layer for regularization. Finally, two parallel fully connected output branches are used: one for classification of task offloading requests (using softmax activation) and another for regression of resource demands (using linear activation). This MTL structure allows the model to jointly optimize related tasks, improving overall predictive performance.

RQ3: In what ways does adding an attention mechanism into the proposed hybrid model improve its prediction accuracy and interpretability?

The integration of an attention mechanism into the proposed 1D-CNN-BiLSTM hybrid model substantially enhanced both prediction accuracy and interpretability. Traditional sequential models, such as CNN-BiLSTM architectures, treat all time steps and features equally, which limits their ability to capture the irregular and bursty workload dynamics typical in IoT edge environments. To address this methodological limitation, the attention module was incorporated to enable dynamic weighting of temporal and feature dimensions, allowing the model to selectively focus on the most informative intervals and critical features that strongly influence workload behavior. The ablation study comparing the baseline CNN-BiLSTM model with the proposed CNN-BiLSTM-Attention variant, demonstrated clear performance improvements. The attention-augmented model consistently achieved higher R^2 scores and lower error metrics across both regression and classification tasks. This improvement can be attributed to the attention layer's capability to emphasize important temporal dependencies and suppress redundant or noisy patterns, thereby enhancing the model's ability to handle irregular workload spikes and fluctuating resource demands more effectively.

Beyond predictive gains, the attention mechanism also improved model interpretability by providing visibility into which time steps and features most influenced specific predictions. This interpretive capacity is particularly valuable for decision-making in edge scheduling and resource management, where understanding the rationale behind predictions is as important as accuracy itself. Overall, the inclusion of attention transformed the hybrid model from a purely sequential learner into a context-aware predictor, leading to more reliable, efficient, and explainable workload forecasting in dynamic IoT edge computing scenarios.

RQ4: How does the proposed hybrid model perform in comparison with baseline deep learning models in terms of prediction accuracy, and feasibility for edge deployment?

As detailed in Section 6.2, the performance of the proposed hybrid CNN-BiLSTM-Attention model was evaluated against two classical machine learning models, five baseline deep learning models, and one hybrid model (CNN-BiLSTM) using the Google Cluster workload dataset. The comparative analysis, summarized in Tables 6-8, presents both classification and regression performance metrics on the test set. Experimental results demonstrated that the proposed hybrid model achieved the highest predictive accuracy, with an R^2 score of 0.99 and a classification accuracy of 0.98, outperforming all baseline models. In addition, it achieved the lowest error metrics, including MAE = 0.00014, MSE = 0.0001, RMSE = 0.00018, and MAPE = 2.30%. Furthermore, when benchmarked against the best-performing model from prior studies using the same Google Cluster dataset, the proposed hybrid CNN-BiLSTM-Attention model achieved an R^2 score of 0.99, whereas the baseline model reported an R^2 score of 0.93. This represents a 6.4% improvement in prediction accuracy, signifying a substantial advancement in workload forecasting performance. This indicates our proposed model achieves a significant improvement and is far reaching and effective for predicting task requests and resource demands.

Beyond predictive performance, the model was also optimized for edge compatibility. We applied post-training quantization to minimize model size and computational complexity without significantly sacrificing accuracy, ensuring that the model can run efficiently on resource-constrained edge servers. This optimization makes the hybrid CNN-BiLSTM-Attention model not only accurate but also practical for real-world deployment in distributed IoT environments.

Thus, this research aims to develop a predictive, multivariate, and lightweight multi-task deep learning framework capable of anticipating IoT task offloading requests and resource demands at the edge. By enabling proactive scheduling, reduced wait times, and improved load balancing, the proposed model demonstrates both high prediction fidelity and strong deployment feasibility key requirements for next-generation intelligent edge computing systems.

7. CONCLUSIONS AND FUTURE WORK

7.1. Conclusion

IoT edge computing nowadays represents a revolutionary paradigm in the distributed systems area, offering increasing solutions to the limitations of traditional centralized cloud infrastructure. By enabling computation and decision-making closer to the data source through edge and fog layers, it empowers service providers with more flexibility, reduces latency, and improves responsiveness for time-sensitive applications. However, the dynamic and heterogeneous nature of IoT workloads introduces significant challenges in anticipating task offloading decisions and accurately predicting multi-dimensional resource demands. These challenges, if not addressed, can lead to inefficient resource allocation, degraded quality of service, and increased operational costs.

We proposed a solution to enhance the prediction of task offloading requests and associated resource demands by using a hybrid model of CNN-BiLSTM-Attention for more effectiveness and accuracy, using the Google Cluster dataset for temporally rich workload traces. We have employed multiple baseline models including 1D-CNN, LSTM, GRU, BiLSTM, DNN, and 1D-CNN-BiLSTM, as well as classical machine learning models such as Logistic Regression and Decision Tree, to ensure a comprehensive performance benchmark.

We have preprocessed our dataset and selected our features based on correlation analysis and Random Forest importance, and we also used common statistical methods to test the time series characteristics of the dataset. We trained our models by splitting the data into training, validation, and testing sets in the ratio of 70:10:20. Finally, we evaluated our experiments by different metrics, mostly considering regression performance like MAE, MSE, RMSE, and R^2 for continuous targets and accuracy for categorical targets, and then verified that our proposed hybrid model has achieved the best classification and regression performance across all tasks. The best accuracy of our classification outputs was 0.98 for Task priority, while regression results achieved the highest R^2 scores of 0.99. As we can see from our results, the proposed CNN-BiLSTM-Attention model outperformed all other evaluated models in both regression and classification tasks.

Through this work, we validate that integrating convolutional, bidirectional recurrent, and attention mechanisms within a unified framework provides an effective approach to modeling dynamic IoT workload patterns and supporting anticipatory resource allocation in edge computing environments. As we can see from our experiment 1D-CNN-BiLSTM-Attention hybrid model outperforms all the other benchmark models in predicting both task offloading requests and the associated resource usage.

7.2. Major Contribution

Our research work mainly improves the accuracy and efficiency of IoT workload predicting and reduces error rates in predicting multi-dimensional resource demands for edge computing environments.

- We performed comprehensive preprocessing and feature engineering on the Google Cluster v3 dataset, to make sure that influential workload characteristics including sudden surges and heterogeneous task features were effectively included.
- A detailed evaluation of workload prediction performance on the Google Cluster dataset to show the model's scalability and generalizability in handling real-world, high-volume, and temporally complex IoT workloads.
- Implementation of a hybrid deep learning architecture combining 1D-CNN-BiLSTM-Attention to simultaneously learn local patterns, bidirectional temporal dependencies, and context-aware feature weighting.
- The proposed model simultaneously predicts task offloading requests including priority and delay tolerance (categorical targets) as well as CPU and memory requests (regression targets and resource demands, specifically maximum CPU usage and maximum memory usage (regressions). This multi-task prediction framework enables schedulers to anticipate not only the volume of requests but also their urgency and resource intensity, supporting smarter, proactive offloading decisions. Therefore, by jointly predicting categorical and continuous targets, the model captures the interdependencies between task semantics and resource consumption, overcoming the limitations of univariate or single-feature approaches.
- Optimization of the proposed model for edge deployment by applying post training quantization to minimize memory usage and accelerating inference speed, without compromising the model's performance.

- A comparison of different deep learning and hybrid models, including 1D-CNN, LSTM, GRU, BiLSTM, DNN, and 1D-CNN-BiLSTM, as well as Logistic Regression and Decision Tree, mostly based on regression metrics and classification accuracy, to identify and validate the superiority of the proposed hybrid architecture.
- We compare the model's performance with baseline works and CNN-BiLSTM-attention model improved prediction accuracy of the baseline works using the same dataset by 6.4%

7.3. Future Work

While the proposed architecture demonstrates strong performance, in the future we plan to improve the work by including all the task features and resource metrics in the actual IoT edge offloading requests and resource consumptions for further improvements. Therefore, we recommend the following:

- Dataset augmentation to include other task features & resource metrics such as deadline, disk I/O, network, energy and other relevant metric.
- Experimenting with diverse datasets including Azure and Bitbrain datasets for better scalability and generalizations
- Exploring Transformer-based encoders for faster inference and global context capture

Special Acknowledgments:-This research was funded by Adama Science and Technology University with grant number: **ASTU/ SM-R/1114/25**

Adama, Ethiopia

REFERENCES

Acheampong, A., Zhang, Y., Xu, X., & Kumah, D. A. (2022). *A Review of the Current Task Offloading Algorithms , Strategies and Approach in Edge Computing Systems*. https://doi.org/10.32604/cmes.2022.021394

Ahmed, N. K., Atiya, A. F., El Gayar, N., & El-Shishiny, H. (2010). An empirical comparison of machine learning models for time series forecasting. *Econometric Reviews*, *29*(5), 594-621. https://doi.org/10.1080/07474938.2010.481556

Ai, Z., Zhang, W., Li, M., Li, P., & Shi, L. (2023). A smart collaborative framework for dynamic multi - task offloading in IIoT - MEC networks. *Peer-to-Peer Networking and Applications*, 749-764. https://doi.org/10.1007/s12083-022-01441-1

Anandayuvaraj, D., & Davis, J. C. (2022). Reflecting on Recurring Failures in IoT Development. *ACM International Conference Proceeding Series*. https://doi.org/10.1145/3551349.3559545

Benidis, K., Rangapuram, S. S., Flunkert, V., Wang, Y., Maddix, D., Turkmen, C., Gasthaus, J., Bohlke-Schneider, M., Salinas, D., Stella, L., Aubet, F.-X., Callot, L., & Januschowski, T. (2022). Deep Learning for Time Series Forecasting: Tutorial and Literature Survey. *ACM Computing Surveys*, *55*(6), 1-36. https://doi.org/10.1145/3533382

Carvalho, G., Cabral, B., Pereira, V., & Bernardino, J. (2021). Edge computing: current trends, research challenges and future directions. *Computing*, *103*(5), 993-1023. https://doi.org/10.1007/s00607-020-00896-5

Casolaro, A., Capone, V., Iannuzzo, G., & Camastra, F. (2023). Deep Learning for Time Series Forecasting: Advances and Open Problems. *Information (Switzerland)*, *14*(11). https://doi.org/10.3390/info14110598

Chatterjee, S., & Hadi, A. S. (2013). *Regression Analysis by Example*. Wiley. https://books.google.com.et/books?id=86MCAZaY1noC

Chen, L., Zhang, W., & Ye, H. (2022). Accurate workload prediction for edge data centers: Savitzky-Golay filter, CNN and BiLSTM with attention mechanism. *Applied Intelligence*, *52*(11), 13027-13042. https://doi.org/10.1007/s10489-021-03110-x

Chen, W., Ye, K., Wang, Y., Xu, G., & Xu, C. (2018). How Does the Workload Look Like in Production Cloud? Analysis and Clustering of Workloads on Alibaba Cluster Trace. *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, 102-109. https://api.semanticscholar.org/CorpusID:67865107

Chiang, M., & Zhang, T. (2016). Fog and IoT: An Overview of Research Opportunities. *IEEE Internet of Things Journal*, *3*(6), 854-864. https://doi.org/10.1109/JIOT.2016.2584538

Dayong, W., Bin Abu Bakar, K., Isyaku, B., Abdalla Elfadil Eisa, T., & Abdelmaboud, A. (2024). A comprehensive review on internet of things task offloading in multi-access edge computing. *Heliyon*, *10*(9), e29916. https://doi.org/10.1016/j.heliyon.2024.e29916

Dean, J., & Barroso, L. A. (2013). The tail at scale. *Commun. ACM*, *56*(2), 74-80. https://doi.org/10.1145/2408776.2408794

Deepak, Upadhyay, M. K., & Alam, M. (2023). Edge Computing: Architecture, Application, Opportunities, and Challenges. *2023 3rd International Conference on Technological Advancements in Computational Sciences (ICTACS)*, 695-702. https://doi.org/10.1109/ICTACS59847.2023.10390171

Devi, K. L., & Valli, S. (2022). Time series-based workload prediction using the statistical hybrid model for the cloud environment. *Computing*, *105*(2), 353-374. https://doi.org/10.1007/s00607-022-01129-7

Di, S., Kondo, D., & Cirne, W. (2012). Characterization and comparison of cloud versus grid workloads. *Proceedings - 2012 IEEE International Conference on Cluster Computing, CLUSTER 2012*, 230-238. https://doi.org/10.1109/CLUSTER.2012.35

Dogani, J., Khunjush, F., Mahmoudi, M. R., & Seydali, M. (2023). Multivariate workload and resource prediction in cloud computing using CNN and GRU by attention mechanism. *The Journal of Supercomputing*, *79*(3), 3437-3470. https://doi.org/10.1007/s11227-022-04782-z

Filali, A., Abouaomar, A., & Member, S. (2020). *Multi-Access Edge Computing : A Survey*. 197017-197046. https://doi.org/10.1109/ACCESS.2020.3034136

Fu, E., Zhang, Y., Yang, F., & Wang, S. (2022). Temporal self-attention-based Conv-LSTM network for multivariate time series prediction. *Neurocomputing*, *501*, 162-173. https://doi.org/https://doi.org/10.1016/j.neucom.2022.06.014

Gao, J., Wang, H., & Shen, H. (2020). Machine Learning Based Workload Prediction in Cloud Computing. *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, 1-9. https://doi.org/10.1109/ICCCN49398.2020.9209730

Gautam, S., & Malhotra, A. (2024). *A Deadline-Aware Priority based Semi Greedy Task Scheduling Technique in Fog Computing*.

Google Cluster. (2019). *Google Cluster Data*.

Guim, F., Metsch, T., Moustafa, H., Verrall, T., Carrera, D., Cadenelli, N., Chen, J., Doria, D., Ghadie, C., & Prats, R. G. (2022). Autonomous Lifecycle Management for Resource-Efficient Workload Orchestration for Green Edge Computing. *IEEE Transactions on Green Communications and Networking*, *6*(1), 571-582. https://doi.org/10.1109/TGCN.2021.3127531

Hasan, B., & Idrees, A. (2024). *Edge Computing for IoT* (pp. 1-20). https://doi.org/10.1007/978-3-031-50514-0_1

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. Springer New York. https://books.google.com.et/books?id=tVIjmNS3Ob8C

Heidari, A., Jabraeil Jamali, M. A., Jafari Navimipour, N., & Akbarpour, S. (2020). Internet of Things offloading: Ongoing issues, opportunities, and future challenges. *International Journal of Communication Systems*, *33*(14), e4474. https://doi.org/https://doi.org/10.1002/dac.4474

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735-1780. https://doi.org/10.1162/neco.1997.9.8.1735

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*(8), 1735-1780. https://doi.org/10.1162/neco.1997.9.8.1735

Hong, J., Hong, Y.-G., de Foy, X., Kovatsch, M., Schooler, E., & Kutscher, D. (2023). *IoT Edge Challenges and Functions* (Issue draft-irtf-t2trg-iot-edge-09). https://www.ietf.org/archive/id/draft-irtf-t2trg-iot-edge-09.html

Horváth, L., & Rice, G. (2024). *Change Point Analysis for Time Series*. Springer Nature Switzerland. https://books.google.com.et/books?id=OVEHEQAAQBAJ

Ihianle, I. K., Nwajana, A. O., Ebenuwa, S., & Otuka, R. I. (2020). *A Deep Learning Approach for Human Activities Recognition From Multimodal Sensing Devices*. *October*. https://doi.org/10.1109/ACCESS.2020.3027979

Islam, S., Keung, J., Lee, K., & Liu, A. (2012). Empirical prediction models for adaptive resource provisioning in the cloud. *Future Generation Computer Systems*, *28*(1), 155-162. https://doi.org/https://doi.org/10.1016/j.future.2011.05.027

Jeong, B., Baek, S., Park, S., Jeon, J., & Jeong, Y. S. (2023). Stable and efficient resource management using deep neural network on cloud computing. *Neurocomputing*, *521*, 99-112. https://doi.org/10.1016/j.neucom.2022.11.089

Jin, W., Lim, S., Woo, S., Park, C., & Kim, D. (2022). Decision-making of IoT device operation based on intelligent-task offloading for improving environmental optimization. *Complex and Intelligent Systems*, *8*(5), 3847-3866. https://doi.org/10.1007/s40747-022-00659-z

Karim, M. E., Maswood, M. M. S., Das, S., & Alharbi, A. G. (2021). BHyPreC: A Novel Bi-LSTM Based Hybrid Recurrent Neural Network Model to Predict the CPU Workload of Cloud Virtual Machine. *IEEE Access*, *9*, 131476-131495. https://doi.org/10.1109/ACCESS.2021.3113714

Khalid, S., Khalil, T., & Nasreen, S. (2014). A survey of feature selection and feature extraction techniques in machine learning. *2014 Science and Information Conference*, 372-378. https://doi.org/10.1109/SAI.2014.6918213

Kirori, Z., & Ireri, E. (2020). Towards Optimization of the Gated Recurrent Unit (GRU) for Regression Modeling. *Int. J. Soc. Sci. Inf. Technol*, 157-166. http://www.ijssit.com

Kuhn, M., & Johnson, K. (2013). *Applied Predictive Modeling*. Springer New York. https://books.google.com.et/books?id=xYRDAAAAQBAJ

Kumar, J., Dalal, N., & Sethi, M. (2024). Hyperparameters in Deep Learning: A Comprehensive Review. *International Journal of Intelligent Systems and Applications in Engineering*, *12*(4), 4015-4022. https://ijisae.org/index.php/IJISAE/article/view/6967

Lackinger, A. (2023). *Towards Accurate Time Series Predictions for Cloud Workloads. September*.

Lan, S., Duan, Z., Lu, S., Tan, B., Chen, S., Liang, Y., & Chen, S. (2024). SLA-ORECS: an SLA-oriented framework for reallocating resources in edge-cloud systems. *Journal of Cloud Computing*, *13*(1), 18. https://doi.org/10.1186/s13677-023-00561-0

Leka, H. L., Fengli, Z., Kenea, A. T., Sharma, D. P., & Tegene, A. T. (2023). Workload Prediction of Virtual Machines Using Integrated Deep Learning Approaches Over

Cloud Data Centers. *Smart Innovation, Systems and Technologies*, *316*(January 2023), 55-65. https://doi.org/10.1007/978-981-19-5403-0_5

Liu, X., & Wang, W. (2024). Deep Time Series Forecasting Models: A Comprehensive Survey. *Mathematics*, *12*(10). https://doi.org/10.3390/math12101504

Lv, D., Wang, P., Wang, Q., Ding, Y., Han, Z., & Zhang, Y. (2024). *Task Offloading and Resource Optimization Based on Predictive Decision Making in a VIoT System*. 1-21.

Makridakis, S. G., Wheelwright, S. C., & McGee, V. E. (1983). *Forecasting: Methods and Applications*. Wiley. https://books.google.com.et/books?id=WwlDPgAACAAJ

Mao, Y., You, C., Zhang, J., Huang, K., & Letaief, K. B. (2017). A Survey on Mobile Edge Computing: The Communication Perspective. *IEEE Communications Surveys & Tutorials*, *19*(4), 2322-2358. https://doi.org/10.1109/COMST.2017.2745201

Morichetta, A., Pusztai, T., Vij, D., Pujol, V. C., Raith, P., Xiong, Y., Nastic, S., Dustdar, S., & Zhang, Z. (2023). Demystifying deep learning in predictive monitoring for cloud-native SLOs. *IEEE International Conference on Cloud Computing, CLOUD*, *2023-July*, 24-34. https://doi.org/10.1109/CLOUD60044.2023.00013

Mumuni, A., & Mumuni, F. (2025). Automated data processing and feature engineering for deep learning and big data applications: A survey. *Journal of Information and Intelligence*, *3*(2), 113-153. https://doi.org/10.1016/j.jiixd.2024.01.002

Ng, H., Chia, G. J. W., Yap, T. T. V, & Goh, V. T. (2021). Modelling sentiments based on objectivity and subjectivity with self-attention mechanisms [version 1; peer review: 3 approved with reservations]. *F1000Research*, *10*(1001). https://doi.org/10.12688/f1000research.73131.1

Nugroho, A. K., & Kim, T. (2024). Traffic-Aware Intelligent Association and Task Offloading for Multi-Access Edge Computing. *Electronics (Switzerland)*, *13*(16). https://doi.org/10.3390/electronics13163130

Pandiyan, G, Sasikala, E. (2023). *Efficient Prediction Model for Offloading Decision in Edge Computing* [SRM Institute of Science and Technology]. http://hdl.handle.net/10603/562144

Patel, Y. S., & Bedi, J. (2023). MAG-D: A multivariate attention network based approach for cloud workload forecasting. *Future Generations Computer Systems : FGCS*, *142*, 376 392. https://doi.org/10.1016/j.future.2023.01.002

Rayes, A., & Salam, S. (2022). Internet of Things (IoT) Overview. In *Internet of Things from Hype to Reality: The Road to Digitization* (pp. 1-34). Springer International Publishing. https://doi.org/10.1007/978-3-030-90158-5_1

Reza, M., Miri, S., & Javidan, R. (2001). RANDOM FORESTS Leo. *International Journal of Advanced Computer Science and Applications*, *7*(6), 1-33.

Sarker, I. H. (2021). Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions. *SN Computer Science*, *2*(6), 1-20. https://doi.org/10.1007/s42979-021-00815-1

Sathi, K. A., Hosain, M. K., Hossain, M. A., & Kouzani, A. Z. (2023). Attention-assisted hybrid 1D CNN-BiLSTM model for predicting electric field induced by transcranial magnetic stimulation coil. *Scientific Reports*, *13*(1), 1-12.

https://doi.org/10.1038/s41598-023-29695-6

Seba, A. M., Gemeda, K. A., & Ramulu, P. J. (2024). Prediction and classification of IoT sensor faults using hybrid deep learning model. *Discover Applied Sciences*, *6*(1). https://doi.org/10.1007/s42452-024-05633-7

Shahmirzadi, D., Khaledian, N., & Rahmani, A. M. (2024). Analyzing the impact of various parameters on job scheduling in the Google cluster dataset. *Cluster Computing*, *27*(6), 7673-7687. https://doi.org/10.1007/s10586-024-04377-8

Shen, S., Van Beek, V., & Iosup, A. (2015). Statistical characterization of business-critical workloads hosted in cloud datacenters. *Proceedings - 2015 IEEE/ACM 15th International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2015*, 465-474. https://doi.org/10.1109/CCGrid.2015.60

Shi, W., Cao, J., Zhang, Q., Li, Y., & Xu, L. (2016). Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, *3*(5), 637-646. https://doi.org/10.1109/JIOT.2016.2579198

Simaiya, S., Lilhore, U. K., Sharma, Y. K., Rao, K. B. V. B., Maheswara Rao, V. V. R., Baliyan, A., Bijalwan, A., & Alroobaea, R. (2024). A hybrid cloud load balancing and host utilization prediction method using deep learning and optimization techniques. *Scientific Reports*, *14*(1), 1-18. https://doi.org/10.1038/s41598-024-51466-0

Singh, A. K., & Charankar, N. G. (2024). *Deep Neural Networks : Architecture and Use Cases*. *13*(12), 71-79.

Singh, K., Scholar, R., Mahajan, A., & Mansotra, V. (2021). 1D-CNN based Model for Classification and Analysis of Network Attacks. *International Journal of Advanced Computer Science and Applications*, *12*(11), 604-613. https://doi.org/10.14569/IJACSA.2021.0121169

Sohaib, M., Jeon, S., Member, S., & Yu, W. (2024). Hybrid Online - Offline Learning for Task Offloading in Mobile Edge Computing Systems. *IEEE Access*, 1-16.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, *15*, 1929-1958.

Sun, Y., Bian, Y., Li, H., Tan, F., & Liu, L. (2023). *Flexible Offloading and Task Scheduling for IoT Applications in Dynamic Multi-Access Edge Computing Environments*. 1-21.

Taheri-abed, S., Eftekhari Moghadam, A. M., & Rezvani, M. H. (2023). Machine learning-based computation offloading in edge and fog: a systematic review. In *Cluster Computing* (Vol. 26, Issue 5). Springer US. https://doi.org/10.1007/s10586-023-04100-z

Taleb, T., Samdanis, K., Mada, B., Flinck, H., Dutta, S., & Sabella, D. (2017). On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration. *IEEE Communications Surveys & Tutorials*, *19*(3), 1657-1681. https://doi.org/10.1109/COMST.2017.2705720

Tan, J., Yang, J., Wu, S., Chen, G., & Zhao, J. (2021). *A critical look at the current train/test split in machine learning*. http://arxiv.org/abs/2106.04525

Tantri, B. R., & Bhat, S. (2025). *Accuracy Comparison of Logistic Regression and Decision Tree Prediction Models Using Machine Learning Technique BT - Proceedings of 4th International Conference on Mathematical Modeling and Computational Science* (S. Pal & Á. Rocha (eds.); pp. 452-460). Springer Nature Switzerland.

Theng, D., & Bhoyar, K. K. (2024). Feature selection techniques for machine learning: a survey of more than two decades of research. *Knowledge and Information Systems*, *66*(3), 1575-1637. https://doi.org/10.1007/s10115-023-02010-5

Tu, Y., Chen, H., & Yan, L. (2022). *Task Offloading Based on LSTM Prediction and Deep Reinforcement Learning for Efficient Edge Computing in IoT*. 1-19.

Ullah, I., Lim, H. K., Seok, Y. J., & Han, Y. H. (2023). Optimizing task offloading and resource allocation in edge-cloud networks: a DRL approach. *Journal of Cloud Computing*, *12*(1). https://doi.org/10.1186/s13677-023-00461-3

van Loo, T., Jindal, A., Benedict, S., Chadha, M., & Gerndt, M. (2022). Scalable Infrastructure for Workload Characterization of Cluster Traces. *International Conference on Cloud Computing and Services Science, CLOSER - Proceedings*, 254-263. https://doi.org/10.5220/0011080300003200

Vijayasekaran, G. (2023). *Deep Learning approaches for resources scheduling in edge computing iot networks*.

Walia, G. K., Kumar, M., & Gill, S. S. (2024). AI-Empowered Fog/Edge Resource Management for IoT Applications: A Comprehensive Review, Research Challenges, and Future Perspectives. *IEEE Communications Surveys & Tutorials*, *26*(1), 619-669. https://doi.org/10.1109/COMST.2023.3338015

Wang, D., Bin, K., Bakar, A., & Isyaku, B. (2024). Two-Stage IoT Computational Task Offloading Decision-Making in MEC with Request Holding and Dynamic Eviction. *CMC*. https://doi.org/10.32604/cmc.2024.051944

Wang, X., Han, Y., Leung, V. C. M., Niyato, D., Yan, X., & Chen, X. (2020). Convergence of Edge Computing and Deep Learning: A Comprehensive Survey. *IEEE Communications Surveys & Tutorials*, *22*(2), 869-904. https://doi.org/10.1109/comst.2020.2970550

Wilkes, J. (2011). *Google Cluster Data: Usage Traces and Dataset Schema*.

Wilkes, J. (2020). *Google cluster-usage traces v3*.

Wilkes, J., Tirmazi, M., Deng, N., Haque, M. E., Qin, Z. G., Hand, S., & Barker, A. (2020). *Yet More Google Compute Cluster Trace Data*.

Xu, M., Song, C., Wu, H., Gill, S. S., Ye, K., & Xu, C. (2022). esDNN: Deep Neural Network Based Multivariate Workload Prediction in Cloud Computing Environments. *ACM Transactions on Internet Technology*, *22*(3). https://doi.org/10.1145/3524114

Yadav, M. P., Pal, N., & Yadav, D. K. (2021). Workload prediction over cloud server using time series data. *Proceedings of the Confluence 2021: 11th International Conference on Cloud Computing, Data Science and Engineering*, *February*, 267-272. https://doi.org/10.1109/Confluence51648.2021.9377032

Yang, M., & Wang, J. (2022). Adaptability of Financial Time Series Prediction Based on

BiLSTM. *Procedia Computer Science*, *199*, 18-25. https://doi.org/https://doi.org/10.1016/j.procs.2022.01.003

Yang, Q., Jin, R., Gandhi, N., Ge, X., Khouzani, H. A., & Zhao, M. (2024). EdgeBench: A Workflow-Based Benchmark for Edge Computing. *Lecture Notes in Networks and Systems*, *921 LNNS*, 150-170. https://doi.org/10.1007/978-3-031-54053-0_12

Yildiz, M., & Baiocchi, A. (2024). Data-Driven Workload Generation Based on Google Data Center Measurements. *IEEE International Conference on High Performance Switching and Routing, HPSR, July*, 143-148. https://doi.org/10.1109/HPSR62440.2024.10635925

Yu, T., & Zhu, H. (2020). *Hyper-Parameter Optimization: A Review of Algorithms and Applications*. https://arxiv.org/abs/2003.05689

Zhang, Z., Li, C., Peng, S. L., & Pei, X. (2021). A new task offloading algorithm in edge computing. *Eurasip Journal on Wireless Communications and Networking*, *2021*(1). https://doi.org/10.1186/s13638-021-01895-6

Zheng, T., Wan, J., Zhang, J., Jiang, C., & Jia, G. (2020). A Survey of Computation Offloading in Edge Computing. *2020 International Conference on Computer, Information and Telecommunication Systems (CITS)*, 1-6. https://doi.org/10.1109/CITS49457.2020.9232457

Appendix I

Sample Implementation Code

```
import sys

import math

import pandas as pd

import numpy as np

import tensorflow as tf

from tensorflow.keras.layers import Input, Conv1D, Bidirectional, LSTM, Dense,
    Attention, Flatten, Dropout

from tensorflow.keras import Model

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.callbacks import EarlyStopping

from tensorflow.keras.regularizers

import keras.backend as K

from tensorflow.keras.callbacks import EarlyStopping

from tensorflow.keras.layers import Input, Conv1D, Bidirectional, LSTM, Dense,
    Attention, Flatten, Dropout

from tensorflow.keras import Model

from tensorflow.keras.optimizers import Adam

import numpy as np


early_stopping = EarlyStopping(monitor='val_loss', patience=10,
    restore_best_weights=True)

input_shape = X_train_seq.shape[1:]
best_hyperparameters = {
   'cnn_filters_1': 64,
   'cnn_filters_2': 128,
   'cnn_filters_3': 64,
   'lstm_units': 128,
   'dropout_rate': 0.3,
   'learning_rate': 0.001
}

num_priority_classes = len(np.unique(y_priority_train_encoded))
num_scheduling_classes = len(np.unique(y_scheduling_train_encoded))

losses = {
   'output_task_priority': 'sparse_categorical_crossentropy',
```

```
    'output_scheduling_class': 'sparse_categorical_crossentropy',
    'output_cpus_request': 'mse',
    'output_memory_request': 'mse',
    'output_total_cpu_usage': 'mse',
    'output_total_memory_usage': 'mse'
}

metrics = {
    'output_task_priority': ['accuracy'],
    'output_scheduling_class': ['accuracy'],
    'output_cpus_request': ['mae'],
    'output_memory_request': ['mae'],
    'output_total_cpu_usage': ['mae'],
    'output_total_memory_usage': ['mae']
}

input_layer = Input(shape=input_shape, name='input_sequence')
conv1 = Conv1D(filters=best_hyperparameters['cnn_filters_1'], kernel_size=3,
        activation='relu', padding='same')(input_layer)
conv2 = Conv1D(filters=best_hyperparameters['cnn_filters_2'], kernel_size=3,
        activation='relu', padding='same')(conv1)
conv3 = Conv1D(filters=best_hyperparameters['cnn_filters_3'], kernel_size=3,
        activation='relu', padding='same')(conv2)

bilstm = Bidirectional(LSTM(best_hyperparameters['lstm_units'],
        return_sequences=True))(conv3)
attention_output = Attention()([bilstm, bilstm])
flatten = Flatten()(attention_output)
dropout = Dropout(best_hyperparameters['dropout_rate'])(flatten)

output_priority = Dense(num_priority_classes, activation='softmax',
        name='output_task_priority')(dropout)
output_scheduling = Dense(num_scheduling_classes, activation='softmax',
        name='output_scheduling_class')(dropout)
output_cpus_request = Dense(1, name='output_cpus_request')(dropout)
output_memory_request = Dense(1, name='output_memory_request')(dropout)
output_total_cpu_usage = Dense(1, name='output_total_cpu_usage')(dropout)
output_total_memory_usage = Dense(1, name='output_total_memory_usage')(dropout)

final_model = Model(
    inputs=input_layer,
    outputs=[
        output_priority,
        output_scheduling,
        output_cpus_request,
        output_memory_request,
        output_total_cpu_usage,
        output_total_memory_usage
    ]
)
```

```
final_model.summary()

optimizer = Adam(learning_rate=best_hyperparameters['learning_rate'])
final_model.compile(optimizer=optimizer, loss=losses, metrics=metrics)
final_model.summary()

EPOCHS = 30
history_final = final_model.fit(
    train_dataset,
    epochs=EPOCHS,
    validation_data=val_dataset,
    callbacks=[early_stopping]
)
```

Appendix II

```
from google.cloud import bigquery
client = bigquery.Client()
def query_per_instance_usage_priority():
    return """
    SELECT u.time AS time,
           u.collection_id AS collection_id,
           u.instance_index AS instance_index,
           e.priority AS priority,
           CASE
             WHEN e.priority BETWEEN 0 AND 99 THEN '1_free'
             WHEN e.priority BETWEEN 100 AND 115 THEN '2_beb'
             WHEN e.priority BETWEEN 116 AND 119 THEN '3_mid'
             ELSE '4_prod'
           END AS tier,
           u.cpu_usage AS cpu_usage,
           u.memory_usage AS memory_usage
    FROM (
       SELECT start_time AS time,
              collection_id,
              instance_index,
              machine_id,
              average_usage.cpus AS cpu_usage,
              average_usage.memory AS memory_usage
       FROM `google.com:google-cluster-data.clusterdata_2019_a.instance_usage`
       WHERE (alloc_collection_id IS NULL OR alloc_collection_id = 0)
         AND (end_time - start_time) >= (5 * 60 * 1e6)
    ) AS u
    JOIN (
       SELECT collection_id,
              instance_index,
              machine_id,
              MAX(priority) AS priority
       FROM `google.com:google-cluster-data.clusterdata_2019_a.instance_events`
       WHERE (alloc_collection_id IS NULL OR alloc_collection_id = 0)
       GROUP BY 1, 2, 3
    ) AS e
    ON u.collection_id = e.collection_id
    AND u.instance_index = e.instance_index
    AND u.machine_id = e.machine_id
    """

def query_per_tier_utilization_time_series(cpu_capacity, memory_capacity):
    return f"""
    SELECT CAST(FLOOR(time/(1e6 * 60 * 60)) AS INT64) AS hour_index,
           tier,
           SUM(cpu_usage) / (12 * {cpu_capacity}) AS avg_cpu_usage,
           SUM(memory_usage) / (12 * {memory_capacity}) AS avg_memory_usage
    FROM ({query_per_instance_usage_priority()})
```

```
    GROUP BY 1, 2
    ORDER BY hour_index
    """

def query_top_level_instance_usage():
    return """
    SELECT CAST(FLOOR(start_time/(1e6 * 300)) * (1000000 * 300) AS INT64) AS
        time,
            collection_id,
            instance_index,
            machine_id,
            average_usage.cpus AS cpu_usage,
            average_usage.memory AS memory_usage
    FROM `google.com:google-cluster-data.clusterdata_2019_a.instance_usage`
    WHERE (alloc_collection_id IS NULL OR alloc_collection_id = 0)
    """

def query_machine_usage():
    return f"""
    SELECT u.time AS time,
            u.machine_id AS machine_id,
            SUM(u.cpu_usage) AS cpu_usage,
            SUM(u.memory_usage) AS memory_usage,
            MAX(m.cpu_cap) AS cpu_capacity,
            MAX(m.memory_cap) AS memory_capacity
    FROM ({query_top_level_instance_usage()}) AS u
    JOIN ({query_machine_capacity()}) AS m
    ON u.machine_id = m.machine_id
    GROUP BY 1, 2
    """
def query_machine_utilization_distribution():
    return f"""
    SELECT APPROX_QUANTILES(IF(cpu_usage > cpu_capacity, 1.0, cpu_usage /
        cpu_capacity), 100) AS cpu_util_dist,
            APPROX_QUANTILES(IF(memory_usage > memory_capacity, 1.0,
        memory_usage / memory_capacity), 100) AS memory_util_dist
    FROM ({query_machine_usage()})
    """
```

Ketema Adere

Predicting Task Offloading Requests and Resource Demands in IoT Edge Computing using Hybrid Deep Learning Model

Quick Submit

MSc Thesis Final Report

Adama Science and Technology University

Document Details

Submission ID
trn:oid:::1:3352523781

Submission Date
Sep 26, 2025, 4:06 PM GMT

Download Date
Sep 26, 2025, 4:09 PM GMT

File Name
Final_Thesis_Report2.pdf

File Size
1.9 MB

101 Pages

26,054 Words

156,657 Characters

turnitin Page 1 of 118 - Cover Page Submission ID trn:oid:::1:3352523781

20% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

Match Groups

- **394** Not Cited or Quoted 17%
  Matches with neither in-text citation nor quotation marks
- **59** Missing Quotations 2%
  Matches that are still very similar to source material
- **10** Missing Citation 0%
  Matches that have quotation marks, but no in-text citation
- **2** Cited and Quoted 0%
  Matches with in-text citation present, but no quotation marks

Top Sources

13% Internet sources
15% Publications
7% Submitted works (Student Papers)

Integrity Flags

0 Integrity Flags for Review

No suspicious text manipulations found.

Our system's algorithms look deeply at a document for any inconsistencies that would set it apart from a normal submission. If we notice something strange, we flag it for you to review.

A Flag is not necessarily an indicator of a problem. However, we'd recommend you focus your attention there for further review.

turnitin Page 2 of 118 - Integrity Overview Submission ID trn:oid:::1:3352523781