Christian Bauer

Long Short-Term Memory Based Adaptive Scheduling for Big Data Pipelines Across the Computing Continuum

MASTER THESIS

submitted in fulfilment of the requirements for the degree of

Diplom-Ingenieur

Programme: Master's programme Informatics

Alpen-Adria-Universität Klagenfurt

Evaluator
Postdoc-Ass. Priv.-Doz. Dr. Dragi Kimovski
Alpen-Adria-Universität Klagenfurt
Institut für Informationstechnologie

Klagenfurt, May 2023

Contents

List of Figures

List of Tables

Listings

Acknowledgement

First, let me thank Dragi Kimovski for assisting me with this Master's thesis through his advice, excellent mentoring as well as his warm character, humour and kindness. I thank Narges Mehran for the valuable insights gained during our discussions and the snacks that recharged our spirits. Also, I'd like to thank my cat Tinka for occasionally being a little fluff of cuddliness. Finally, I want to thank my girlfriend Raffaela Mori for her encouragement, understanding, and patience during the period of this work as well as for being the amazing person that inspires me, makes me laugh and makes me feel at home.

Affidavit

Christian Bauer, m. p.

Klagenfurt am Wörthersee, 22nd of May 2023

Abstract

This thesis tackles resource utilisation issues in large-scale distributed infrastructures comprising cloud, fog, and edge systems. Accurate resource utilisation estimates are essential for proper infrastructure functioning. However, current estimates tend to be overestimated by over 30%, resulting in the allocation of more resources than necessary and leading to instability and inefficiency. Additionally, scheduling tasks with communication delays is a well-known NP-hard problem, both for homogeneous and heterogeneous resources. Machine learning approaches have been proposed, but there is no consensus on the best approach. Deep learning algorithms have shown promise in providing accurate predictors. This thesis proposes using Long-Short-Term Memory to train a resource utilisation estimator on large datasets and fine-tune it for specific infrastructures to reduce over-provisioning, increase resource utilisation, stabilize the infrastructure, and optimise task scheduling. The thesis focuses on providing a proof-of-concept solution, evaluating it on a simulated infrastructure, and discussing potential improvements and limitations.

Keywords

Long-Short-Term Memory, machine learning, resource utilisation, cloud, fog, edge, task scheduling, resource utilisation estimation.

Chapter 1

Introduction

The Internet and computational devices have penetrated all fields and areas. The data produced by this interconnected network of computers increases by the day. This vast amount of data is often referred to as *Big Data.* Big Data includes healthcare data such as remote patient monitoring, treatment, sensor data of autonomous smart cities, communications, media and entertainment and other industries. Most of this produced data is unstructured and needs to be processed in order to be usable and analysed. As the computational demands of processing large amounts of data are increasing and more complex data structures are being sent on the Internet, e.g. video streams, the processing of data is requiring more powerful resources. The ever-increasing amount of data to process resulted in the need for more powerful computing resources such as the *Cloud.* The Cloud physically resides in large-scale data centres that enable applications to quickly scale depending on resource utilisation spikes. Additionally, they possess the required computing power to analyse Big Data in a performant manner.

Yet, while the Cloud is capable to process large amounts of data, it also has various disadvantages to only using a Cloud system for Big Data processing. To use the processing power or access the storage units of the Cloud system, an Internet connection is required. Given the increasing amount of data to be sent to the Cloud, this also results in requiring fast internet speed for the Cloud processing to be used in a reasonable time frame. Another disadvantage is the potential downtime of the Cloud system, either by an unstable Internet connection or a system failure at the Cloud system. This downtime can lead to total failures of infrastructures that entirely rely on the Cloud for data storage and processing. Also, the physical distance of the Cloud system to the devices producing data leads to latencies due to sending information back and forth over the Internet. Real-time applications require fast processing times, thus sending data and waiting for information provided by a Cloud system is not suitable for this application type. These disadvantages resulted in creating the *Computing Continuum.* The Computing Continuum covers Cloud, Fog and Edge systems that promise to omit the disadvantages of solely relying on a Cloud system.

Modern large-scale distributed infrastructures that consist of a vast amount of heterogeneous resources require complex workflow sequences in order to functionally operate. Additionally, specific domain knowledge is required to be able to properly map tasks residing on a big data pipeline onto heterogeneous resources. Yet, this often results in a degradation of the performance of the system since research shows that humans tend to over-provision the available resources.

> In a data center, computing resources such as CPU and memory are usually managed by a resource manager. The resource manager accepts resource requests from users and allocates resources to their applications. A commonly known problem in resource management is that users often request more resources than their applications actually use. This leads to the degradation of overall resource utilization in a data center [4].

This chapter covers the introduction of the master thesis. First, the motivation and scope of the thesis will be described in section 1.1. Followed by the explanation, of what the existing research problems are, that we want to improve upon in section 1.2. Next are the research objectives in section 1.3, which describe briefly what the gained insights and improvements upon the research problems are. And in section 1.4 the outline of the thesis briefly describes the structure and content of all subsequent chapters.

1.1 Motivation and Scope

The motivation is to improve resource utilisation in distributed infrastructures. The targeted distributed infrastructures consist of a vast amount of computing resources on cloud, fog and edge layers. Mapping and deploying tasks on these infrastructures is a challenging problem. In order to enable large-scale distributed infrastructures to function properly, it is necessary to provide estimates regarding the resource utilisation of deployable tasks to these infrastructures. A major problem when providing resource utilisation estimates is their quality of accuracy, as shown by the authors of [4], tasks provided by users overestimate the resource utilisation by more than 30% on average. This results in a higher resource allocation than necessary to execute and finish tasks in time and also results in a less stable infrastructure since tasks might not be deployed on fitting resources, since all their capacity might be allocated. Task scheduling with communication delays is a well-researched problem that is known to be strongly NP-hard problems for both homogeneous and heterogeneous resources [5]. Numerous polynomial-time heuristic algorithms have been proposed that provide non-optimal solutions, yet do so within a reasonable time frame in order to be used for scheduling. Recent developments in machine learning and large datasets that trace the behaviour of hyper-scale data centres [6] while executing various task types lay a promising groundwork to use deep learning techniques to provide a resource utilisation estimator that is trained on said immense amount of data and then fine-tuned for the specific infrastructure it is part of. Various machine learning approaches to improve resource utilisation prediction and scheduling emerged in recent years as are mentioned in section 3. Yet, there is no consensus on the best machine learning approach and while task scheduling is challenging, deep learning algorithms made promising progress in providing more accurate predictors in many fields. The wastage occurring in utilizing resources on large-scale systems (or computing clusters) is further described in sections 2.7 and 6.2. This wastage is the basis for the evaluation of potential improvements upon the currently used prediction methods.

The scope is the analysis of available literature on machine learning techniques used to predict hardware utilisation of tasks and how the authors did implement the state-of-the-art algorithms to improve the predictions. An overview of the theoretical concepts regarding the required terminology of scheduling and machine learning is done in order to provide the reader with knowledge of the subjects in this thesis. Also, an analysis of real data traces is done, that compares the actual hardware utilisation regarding the CPU and memory allocation and the deviation of the provided utilisation prediction corresponding to both hardware types. This analysis also shows the need to improve upon the currently used variant of prediction to better utilise the available hardware clusters. An implementation of a Long-Short Term Memory neural network model and its related components and architecture are further elaborated upon. Different model configurations are analysed and compared regarding their prediction accuracy.

1.2 Research Problems

The deviation of the predicted provisioning or utilisation of heterogeneous resources done by humans often leads to the degradation of the entire system. That degradation includes deployed tasks not finishing within an expected time frame or even the failure of a resource, degrading the stability of the system further.

> In practice, the planned or expected performance of production units often deviates from the actual performance. Most of these deviations are negative, which means that the actual performance is worse than the expected performance. Apparently, expectations about future performances are often too optimistic [7].

Naive predictions of the resource utilisation generated by humans are likely to be over-utilising resources, which leads to resource wastage since those resources are not optimally operating.

These deviations from the actual values are also referred to as *disturbances*. As is stated in [7], these disturbances can be divided into three categories:

1. Disturbances regarding the capacity,
2. Disturbances related to orders,
3. Disturbances related to the measurement of data.

While the paper uses the disturbances on *production scheduling*, i.e. finding a schedule for a real facility, the propositions can be translated into our domain of distributed systems that represent the machines/workers and big data pipelines filled with tasks that represent the tasks that are to be scheduled to the machines. After scheduling and deploying the tasks onto the machines, they need to be executed or finished by a worker. Capacity disturbances are caused by the machine's capacity and involve scenarios such as partial or complete failure of a hardware resource or trying to execute tasks on resources that are not capable of handling those tasks because the specific requirements are not met.

Order-related disturbances are aspects that delay the process of individual orders (tasks). This could be the unavailability of required data that is calculated by a preceding task the current task depends on which has not finished its execution.

The disturbance related to the measurement of data is that of processing times and hardware utilisation requirements that are estimated before deploying a task onto the distributed system. This type of disturbance is the aforementioned deviation of the predicted utilisation of heterogeneous resources that are investigated in this thesis. These cover the hardware utilisation requirements and the already existing estimations done by domain experts and users. While analysing the monitoring traces of popular cloud providers, it becomes apparent that hardware is not optimally used.

1.3 Research Goals and Objectives

The research objectives of this thesis are to provide a prediction method for a distributed system handling big data that can predict how much system resources are necessary and provide a rule-based analyser that monitors the system components for over-utilisation. This is achieved by analysing existing real-data traces provided by cloud providers and using the gained knowledge to build fine-tuned predictors for the given task pipelines. One objective is to improve upon the existing prediction methods, such as predictions provided by users of cloud services, which have to provide an estimation of the resource utilisation the task that is to be deployed will require. The prediction performance is measured using regression metrics in order to make informed decisions regarding prediction accuracy. In order to be comparable with user-provided estimations, first those user estimations have to be analysed to gather informations regarding their prediction accuracy.

The research goals are defined in the following points.

- An analysis of different publicly available monitoring traces. This analysis should provide insights to improve the data quality by transforming the datasets and designing better-suited machine learning models, that are fine-tuned for the datasets.

- Reviewing different methods and theories for resource utilisation prediction.

- A proof of concept machine learning component that is capable of improving resource utilisation compared to other methods, which includes evaluation of the improvement.

- Evaluating the machine learning component and existing prediction methods based on regression metrics and improving the models based on the gathered data insights to outperform said existing prediction methods.

- Providing the adapted data prediction done by the machine learning component in a reasonable time frame in order to be usable by the matching-based scheduler.

1.4 Thesis Outline

First, in chapter Background and Related Work the necessary background regarding monitoring, computation on large-scale distributed systems, a scheduling and adaptation approach and forecast prediction with machine learning is explained, followed by related work in machine learning-based forecast prediction and findings in publicly available data traces. In chapter State of the Art, published research is analysed regarding their approach for similar research questions. In chapter Model the methodology of this thesis is explained. In chapter Architecture and Implementation the architecture of the software is explained, followed by the preprocessing of data traces mentioned in Cloud Monitoring and Execution Traces and the adaptation approach used. Next, in chapter Evaluation and Results an analysis of the available data is done, followed by the metrics used for the evaluation, and finally, the different evaluation scenarios and their results are described. Finalizing the thesis with the chapter Conclusions and Future Work that contains the conclusions about the findings of the evaluation in Conclusions as well as the Future Work that mentions possible improvements upon the current state of the software.

Chapter 2

Background and Related Work

2.1 Monitoring

Monitoring is the process of continuously observing and tracking a system, process, or activity to gather data, identify trends, and detect any deviations or issues. Monitoring has the purpose of ensuring that the system is functioning as intended and also identifying any potential problems early on in order to mitigate them. Additionally, monitoring provides necessary information for effective decision-making and problem-solving. There are different monitoring types, including *system-, environmental-, process-, performance- and security monitoring.* We mainly focus on **system monitoring**. Monitoring can be performed manually or through the use of automated tools and technologies. The choice of monitoring approach will depend on the specific requirements of the system or process being monitored.

System monitoring involves monitoring the performance and availability of computer systems, applications and networks. This includes monitoring CPU, server memory, routers, switches, bandwidth, and applications as well as the performance and availability of important network devices. These systems are also used to track information regarding items such as free disk space on available hard drives, and the temperature of hardware components such as the CPU. The advantage of using system monitoring is that in case of an occurring error or failure of a system component, the monitoring system will immediately send a notification and in the best case be able to fix the problem. Additionally, collecting data when a problem occurs also enables continuous improvement and easier debugging of faulty components.

2.2 Computing Continuum

The **Computing Continuum** consists of devices that reside in cloud, fog and edge layers.

Cloud Layer is a high-performance computing layer used for big data processing and analysis. Cloud computing allows scaling without investing in physical hardware and servers and it entails the delivery of a variety of computing services over the

Internet. Physically, cloud computing resides in large data centres. These data centres are responsible for managing physical resources such as servers, routers, switches, etc.

Fog Layer is a decentralised computing layer between the source of data and a central cloud platform. Similar to edge computing, the fog layer provides processing power and services such as switches, and routers closer to the location the data is extracted from. Fog resources ensure the services to end devices in the edge layer and can compute, transfer and store the data temporarily. A connection to cloud data centres via the IP core networks enables the fog layer to interact and cooperate with the cloud for enhancing processing and storage capabilities.

Edge Layer brings processing and storage systems as close as possible to the application, device or component that generates and collects data. Edge computing helps to minimise the processing time by removing the need for data transfer to a central cloud processing system and back to the endpoint afterwards.

2.3 Scheduling and Adaptation

This section elaborates on the scheduling of computing resources, the overall types of scheduling approaches and their advantages and disadvantages and also on adaptation that is used to improve the scheduling process.

2.3.1 Scheduling

The goal of scheduling is to allocate resources effectively to tasks in order to achieve one or multiple objectives at the same time while working within processing and resource constraints. The definition of a resource varies depending on the context and may be processors, network components etc. Similarly, the definition of a task also varies and common task types to be scheduled are threads and processes. The scheduling activity is done by a process called *scheduler*. Schedulers are designed to aim to fulfil one or multiple goals while keeping the computer resources busy. Tasks that are to be scheduled that have dependencies to other tasks can be represented as a *Directed Acyclic Graph (DAG)*. The DAG scheduling problem is known to be NP-complete for homogenous and heterogeneous processing resources [8].

2.3.1.1 Static Scheduling

Static scheduling refers to scheduling tasks in advance, based on a predetermined set of rules or algorithms. The generated schedule remains fixed, hence the name static scheduling, regardless of changing system conditions or task requirements. Static scheduling has the advantage of being able to make better use of the data if its properties are sufficiently accurate. Since most static schedulers are rather simple to implement, additional advantages are their predictability and low complexity. A major disadvantage of these schedulers is that they are not flexible, meaning that a created schedule can't be simply changed by adding, removing or modifying tasks, but the scheduling algorithm has to be executed offline again after each modification. This scheduling approach is often used in systems where the tasks have a fixed duration and the workload is predictable.

2.3.1.2 Dynamic Scheduling

Dynamic schedules are determined during the runtime (real-time) and might change while executing based on the current system conditions and task requirements. The schedule is frequently updated, and tasks are assigned to resources based on their priority and availability. Additionally, the scheduler rearranges the tasks to reduce stalls of the resources while maintaining data flow and exception behaviour. The advantages of dynamic scheduling are that it can handle dependencies that are unknown at compile time, it allows code compiled for one pipeline to run efficiently on a different pipeline and it simplifies the compiler. Therefore, this approach is often used in systems where tasks have varying durations and the workload is unpredictable. Dynamic scheduling is more flexible compared to static scheduling and can respond to changes regarding the system conditions and task requirements but this also requires more complex algorithms and also might need more computational resources to do so.

2.3.1.3 Adaptation of Static Schedules

Adaptation of static schedules (now referred to as adaptive scheduling) refers to the approach where a static schedule is generated in advance but also uses an adaptive component that enables flexible adaptation based on changing conditions or unforeseen events. In adaptive scheduling, a schedule is generated based on algorithms or a set of rules that take expected and known parameters such as workload and resource availability into account. However, the schedule is also designed to be adjusted, and therefore, the schedule is adaptive, if there are changes in the system conditions or task requirements. This approach combines the advantages of static scheduling, which is predictable and easy to implement, with the flexibility of dynamic scheduling, which can respond to changing conditions. Static scheduling with adaptation can provide a more efficient and reliable schedule than pure dynamic scheduling, which can be more computationally intensive and prone to errors.

> In summary, *adaptability* can be defined as an aggregate measure of key software characteristics that support customization of software functionality after initial development [9].

A more accurate representation of task properties improves the resulting scheduling plan of static schedulers as they rely on accurate task descriptions to calculate a task order that fulfils the schedule goals.

For example, an adaptive schedule could be used in a manufacturing plant where the production line is designed to operate at a certain pace. The schedule could be created in advance, based on the expected demand and capacity of the production line. However, if there are any unexpected changes in demand or if a machine breaks down, the schedule can be adjusted to accommodate these changes while still maintaining the overall production goals.

2.4 Supervised Learning

Supervised learning is a machine learning paradigm that is applied when available data consists of labelled data points, that not only contain features but also a label that is associated with these features. Supervised learning is defined by its use of labelled datasets to train algorithms to classify data or predict outcomes accurately. As input data is fed into the model, it adjusts its weights until the model has been fitted appropriately, which occurs as part of the cross-validation process. Supervised Learning uses a training set to teach models to yield the desired output. This training dataset includes inputs and correct outputs, which allow the model to learn over time. The algorithm measures its accuracy through a loss function, adjusting until the error rate has been sufficiently minimized. A training dataset with n are of the form $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ such that x_i is defined as the *feature vector* of the i-th data point and y_i is defined as its label. A supervised learning algorithm will train to learn a function g such that $g : X \to Y$, where X is the input space and Y is the output space. The function g is often represented as a scoring function $f : X \times Y \to \mathbb{R}$ such that g is defined as $g(x) = \arg\max f(x, y)$ such that g returns the y value with the highest score. Supervised learning can be separated into Classification and Regression problems.

In supervised learning for scheduling the objective is to learn parameters of the neural network such that the trained networks can replicate the behaviour of pre-existing scheduling methods, such as mathematical optimisations, and search methods, for example *Priority Dispatching Rule (PDR)*, a heuristic rule that assigns jobs to machines based on their priorities while considering the current status of the system. Supervised learning can also be used for forecast models that predict the resource utilisation or the runtime of tasks. This is done by providing information such as the capacity of a resource, the naive allocation prediction provided by a human and other factors to the feature vector and including the target that is to be predicted to the labels. The training dataset for this prediction task is usually a time series forecasting dataset. In a time series forecasting dataset, the order of data points is ascending, meaning that the structure and order of the data points is used as additional information that is used for the prediction.

2.4.1 Classification

A classification problem is when the output value is a categorical data type, such as “cat” and “dog”. This section is added for completeness but won’t be discussed further since it serves no purpose for this master’s thesis.

2.4.2 Regression

Regression is used to predict continuous-valued outputs. That is, find a relationship between the input and output data generated by an unknown but fixed distribution represented by a function denoted as $f(x)$. A regression model does so by showing whether the observed changes in the dependent variable are also associated with changes in the independent (explanatory) variables. In figure 2.1 an example regression is shown by the orange line that is done by linear regression. As can be seen, the linear regression

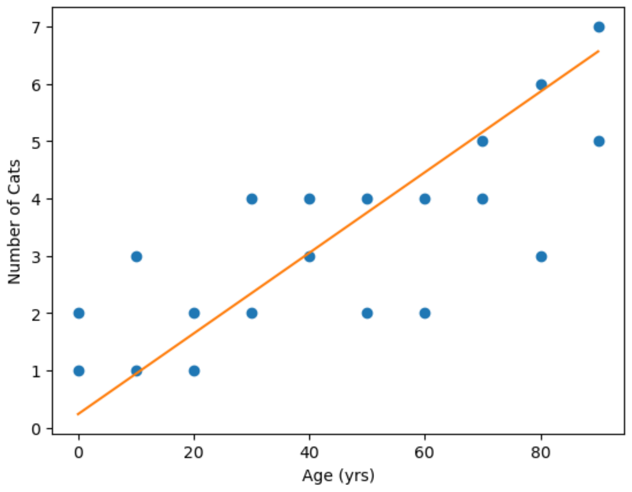


Figure 2.1 Linear Regression Example

line is estimating the values of the data points by trying to fit closest to all data points that are shown as the blue dots. This is done by using the function $f(x)$ to find a best-fit line and see how much the data is dispersed around this line. The blue dots in the figure show the actual values. In this example, they represent the number of cats owned at a certain age. *Note that this linear regression example is a fictive scenario.*

> Regression is a statistical method used in finance, investing, and other disciplines that attempt to determine the strength and character of the relationship between one dependent variable (usually denoted by Y) and a series of other variables (known as independent variables) [10].

Therefore, regression analysis is a useful tool for finding associations of variables that are observed in data. Yet, those associations do not indicate causation but merely correlation. This is because regression only captures correlations between variables that are observed in data and quantifies if the found correlation is statistically significant. In order for proper interpretations regarding the regression results, several assumptions about the data and the model itself must hold.

2.5 Recurrent Neural Network

Recurrent Neural Networks (RNNs) are a type of neural network designed to handle sequential data. Opposed to traditional feedforward neural networks, which receive a fixed-sized vector as input and produce a fixed-sized vector as output, RNNs are designed to handle sequences of variable length and produce a hidden state that summarizes information from the entire sequence. In an RNN, each unit in the network processes the input sequence one-time step at a time, maintaining a hidden state that summarizes information from all previous time steps. At each time step, the hidden state is updated based on both the current input and the hidden state from the previous time step. The

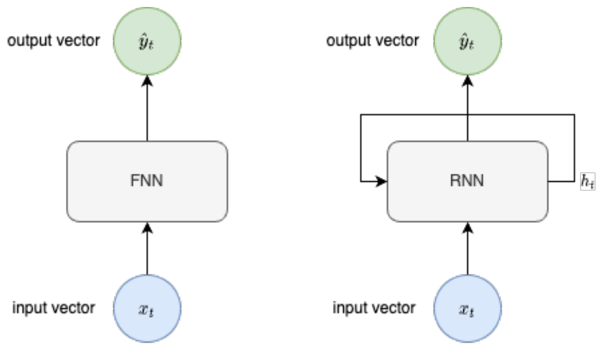


Figure 2.2 FNN vs. RNN

hidden state is then used to generate the output for that time step, and the process is repeated for each time step in the sequence. RNNs are used for a variety of tasks, including natural language processing, speech recognition, and video analysis. They can also be used to generate sequences, such as in text generation or music composition.

The ability to maintain information across a sequence of inputs makes RNNs a fitting tool for modelling sequential data.

> ... recurrent neural networks contain cycles that feed the network activations from a previous time step as inputs to the network to influence predictions at the current time step. These activations are stored in the internal states of the network which can in principle hold long-term temporal contextual information. This mechanism allows RNNs to exploit a dynamically changing contextual window over the input sequence history [11].

Yet, this also results in them being prone to Vanishing Gradients and Exploding Gradients , which make it cumbersome to train the network effectively. To address these issues, several variants of RNNs have been developed, Long-Short Term Memory networks and Gated Recurrent Units (GRUs).

2.5.1 Recurrent Cell Architecture

A traditional feed-forward neural network (FNN) is unidirectional, meaning that they have a single direction and hence cannot persist information over a time step t. Looping structures are added to a feed-forward neural network that enables the persistence of information about time-series or sequential data. This is the reason RNNs are known as "recurrent" neural networks.

As can be seen in figure 2.2, the RNN has an additional loop inside it to persist time-series information. The loop structure enables the RNN to apply a *recurrence relation* (see 2.5.2) at every time step in order to process a sequence.

The rectangle containing the RNN label is defined as a “recurrent cell”. This workflow of a single RNN cell can be seen in more detail in figure 2.3, where the previous cell state h_{t-1} and the current input x_t are used as the input of the recurrent cell, get combined and forwarded to the *tanh activation function*, which returns the output vector $\hat{y}_t$ and h_t (the recurrence relation).

2.5.2 Recurrence Relation

The recurrence relation is applied at every time step to process a sequence.

The recurrent relation seen in figures 2.2, 2.3 is denoted as h_t and is defined as $h_t = f_w(h_{t-1}, x_t)$, where f_w is a function that is parametrized by the weights, h_{t-1} is the previous state and x_t is the input vector at time step t. With the addition of h_{t-1}, the model is now also taking the previous time step into account when updating the current time step.

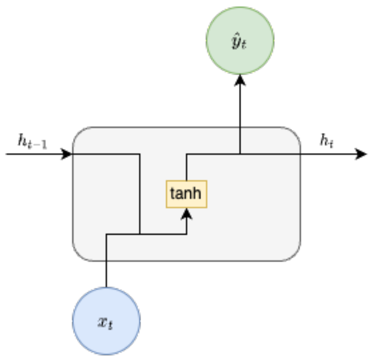


Figure 2.3 Single RNN Cell

Definition 1: Recurrence Relation

The recurrence relation in a more mathematical notation is:

$$h_t = \tanh(W_{hh}^T h_{t-1} + W_{xh}^T x_t)$$

Both the input vector x_t as well as the previous state h_{t-1} are multiplied with the two separate weight matrices W_{xh}^T and $W^T hh$ respectively, combined and fed to the tanh activation function. Finally, the tanh function returns the output vector $\hat{y}_t$ at time step t.

2.5.3 RNN Loop Unfolding

The loop unfolding will describe how RNN handles sequential data at every time step. As can be seen in figure 2.4, the model is adding the input at every time step and generates an output $\hat{y}_i$ for every time step. The weight matrix W_{hh} that is used for every time step for updating the previous state is the same for every time step. The weight matrix W_{xh} is applied to every x_i and is also the same for every time step i.

— x_i denotes the input value at time step i.

— $\hat{y}_i$ denotes the output value at time step i.

— W_{hh} denotes the weight matrix to update the previous state.

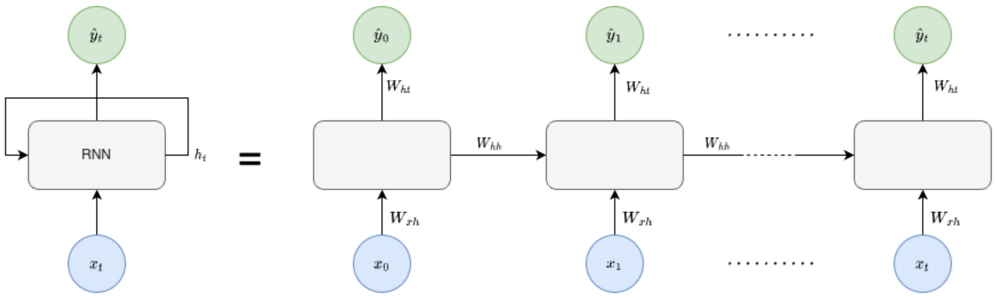


Figure 2.4 RNN Loop Unfolding

— W_{xh} denotes the weight matrix that is applied to the input value at every time step.

The output vectors $\hat{y}_0, \hat{y}_1, \hat{y}_2, \dots, \hat{y}_t$ can be used to calculate the separate losses $L_0, L_1, L_2, \dots, L_t$ at each time step t.

2.5.4 Loss Calculation and Weight Updates for RNN

The training of an unfolded RNN is done through multiple time steps, as can be seen in figure 2.4. The overall loss is defined as $L = L_0, L_1, \dots, L_t$ and is calculated from the outputs $\hat{y}_0, \hat{y}_1, \dots, \hat{y}_t$ for each time step t in the *forward-propagation* process. Then the

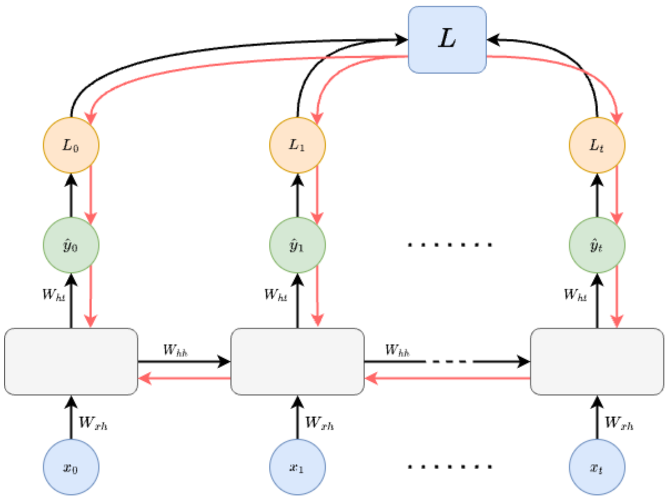


Figure 2.5 Loss Calculation and Weight Update in RNNs

total loss L is used to propagate backwards and calculate the *back-propagation* in order to update the weights of the model. This is shown in figure 2.5, where the black arrows display the forward-propagation step that is accumulated as the total loss L, and then the back-propagation shown as the red arrows update the weights by using the partial derivative.

> The central problem that back-propagation solves is the evaluation of the influence of a parameter on a function whose computation involves several elementary *steps*. The solution to this problem is given by the chain rule, but back-propagation exploits the particular form of the functions used at each step (or layer) to provide an elegant and local procedure [12].

Backpropagation is the practice of fine-tuning the weights of a neural network based on the error rate (i.e. loss) obtained in the previous epoch. Proper tuning of the weights ensures lower error rates, making the model reliable by increasing its generalization.

2.5.5 Common Problems and Shortcomings of RNNs

Regularly experienced problems of RNNs are the exploding or vanishing gradient problems.

> The motivation behind why they happen is that it is hard to catch long-haul conditions as a result of a multiplicative angle that can be dramatically diminishing/expanding regarding the number of layers [13].

While unfolding, the error gradient is calculated as the sum of all gradient errors across time steps. Therefore, the loss calculation in unfolded RNNs is also known as *backpropagation through time (BPTT)*. Over time while calculating the error gradients the domination of the multiplicative term increases due to the chain rule application. And thus the gradients either explode or vanish. These problems occur when the sequence is too long and this may result in the model training with either null weights (i.e. the model won't learn while training) or exploding weights.

Exploding Gradients problem occurs when many of the values (i.e. weight matrices, or gradients themselves) involved in the repeated gradient computations are greater than 1. If this is the case, then gradients become extremely large and optimising them becomes computationally intensive. To solve the Exploding Gradients problem, a process called *Gradient Clipping* is applied, which scales the gradient values to smaller values less than 1.

Vanishing Gradients problem occurs when many of the values (i.e. weight matrices or gradients themselves) that are involved in the repeated gradient computations are too small or less than 1. Opposed to the Exploding Gradients problem, the gradients become smaller with each repeated computation of the gradients. This results in the problem of *long term dependency*, where smaller sequences can be remembered and the weights updated accordingly. But for longer sequences, the model will unlikely be able to yield a good prediction performance.

There are multiple solutions to the Vanishing Gradient problem, such as changing the activation function from tanh to *Rectified Linear Unit (ReLU)*. Also initialising the weights of the model can solve this problem, though tailored heuristics (such as the *Xavier Weight Initialisation*) should be used to result in omitting the Vanishing Gradients problem and also increase the efficiency of the training. Another solution is to change the architecture of the neural network and add more complex recurrent units that are mentioned in section 2.6.

Slow and Complex Training Since RNNs are recurrent one of their fundamental problems is that they require a lot of time for training when compared to FNNs. Additionally, RNNs need to calibrate the previous outputs as well as current inputs into a state change function, which in turn makes RNNs harder to implement and customize and more complex to train.

Difficult to Process Longer Sequences As already mentioned earlier, training RNNs on too-long sequences is difficult without taking any measures to improve the prediction performance. This is especially true while using the *tanh* activation function.

2.6 Long-Short Term Memory

Long-Short Term Memory (LSTM) [1] [14] are a type of Recurrent Neural Network architecture but are designed to improve upon the issues of regular RNN models as are mentioned in Vanishing Gradients and Exploding Gradients . The main architectural improvement for LSTMs over traditional RNNs is that they have introduced LSTM Memory Cell Architecture to produce paths, which let gradients flow for a long duration, and thus gradients will not vanish. Compared to RNNs, LSTMs are better suited for learning long-term dependencies in sequential or time-series data that have temporal dependence such as natural language processing, speech recognition, music generation and financial prediction.

> Hence standard RNNs fail to learn in the presence of time lags greater than 5 – 10 discrete time steps between relevant input events and target signals. The vanishing error problem casts doubt on whether standard RNNs can indeed exhibit significant practical advantages over time window-based feedforward networks. A recent model, "Long Short-Term Memory" (LSTM), is not affected by this problem. LSTM can learn to bridge minimal time lags in excess of 1000 discrete time steps by enforcing constant error flow through "constant error carousels" (CECs) within special units, called cells [15].

LSTMs are a type of deep learning algorithm that can be trained end-to-end, making them highly flexible and suitable for complex problems where the underlying relationships between input and output are not well understood.

2.6.1 LSTM Memory Cell Architecture

The LSTM architecture is comprised of a series of memory cells that store information and gate mechanisms that are used to control the flow of information into and out of the cells. Compared to RNN cells, they are more complex and require more computations in general. The introduction of *self-looping* to produce paths is the main architectural improvement for LSTMs over RNNs. This additional component diminishes the problem of the vanishing gradients problem and enables gradients to flow for a long duration when extended with gate units as is described in the upcoming sections. LSTM memory

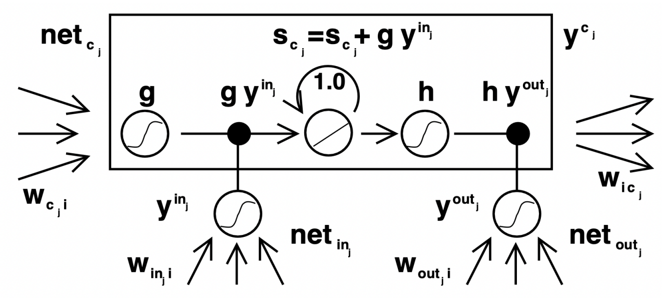


Figure 2.6 Architecture of a memory cell and its gate units [1]

cells can store information for an extended period compared to regular RNNs and its *cell state* functions as the memory of the network. The architecture of a memory cell and its gates can be seen in figure 2.6 that was taken directly from the paper [1].

2.6.1.1 LSTM Gate Structure

The key building block in LSTM cells is a gate structure. Each memory cell has gate components that control the flow of information into and out of the cell. Information is either added or removed through these gates. The usage of gates to control the information flow allows the LSTM model to selectively remember or forget information, making it well-suited for tasks that require long-term memory. The gate types are as follows:

2.6.1.2 Input Gate

This gate determines the flow of new information and how much from the current time step should be added to the cell state. The input gate is implemented as a multiplicative gate that protects other units such as the cell state from perturbation by irrelevant input information.

2.6.1.3 Forget Gate

This gate determines how much of the previously obtained information should be forgotten in order for the network to maintain a long-term memory of relevant information and discard irrelevant information.

Definition 2: Forget Gate

The forget gate is denoted by

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right)$$

where t is the time step and i is the memory cell, $x^{(t)}$ is the current input vector, h^t is the current hidden state containing the outputs of all the LSTM cells, b^f is the bias, U^f the input weights and W^f the recurrent weights for the forget gates.

2.6.1.4 Output Gate

The output gate controls what information is being used for the cell state, and is sent to the network as input for the next time step. The output of the output gate is referred to as the *hidden state*, denoted as $h_i^{(t)}$ and is sent to the next network layer or also used for predictions.

2.6.1.5 Sigmoid Layer and Pointwise-Multiplication

Optionally, information can also be passed through those gates via a Sigmoid layer and point-wise multiplication as can be seen in figure 2.7. The Sigmoid layer is used to map

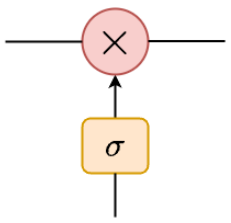


Figure 2.7 Sigmoid Layer and Pointwise Multiplication

the input to an output in a range of 0 and 1, which is used to determine how much of the information is captured while passing through the gate, and how much of the information will be retained while the pass-through. If the Sigmoid output is 0, no information is kept and if the Sigmoid output is 1, all information about the input is kept.

2.6.1.6 Back-Propagation Through Time

The LSTM architecture implements so-called *backpropagation through time (BPTT)*, a variant of back-propagation that allows the gradients to flow backwards through the entire sequence of inputs as can be seen in figure 2.5.

2.6.2 LSTM Learning Process

1. Forget Irrelevant History The process of forgetting irrelevant history is done by the forget gates (see LSTM Gate Structure). This process is beneficial since not all information in a sequence is important.

2. Perform Computations and Store Relevant Information The importance of sequential information is decided by the LSTM model, which either keeps information that is relevant to keep or will be discarded by the forget gates in order to be able to store new information.

Definition 3: Forget Gate

This process is mathematically represented as:

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}\right)$$

The computation of new information is done via the input gates. The internal cell state is updated by the previous hidden state $h^{(t-1)}$, the current input $x^{(t)}$ and the bias $b^{(t)}$ that is passed to the Sigmoid activation function that weights the value by transforming it between 0 and 1 based on how relevant the information is.

3. Self-Loop to Update Internal State Once enough data was gathered by the model, the self-loop weight f_i^t is used to update the internal state of the model.

Definition 4: Internal State

$$s_i^{(t)} = f_i^t s_i^{t-1} + g_i^t \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right)$$

First, point-wise multiplication that was calculated of the previous cell state denoted as $s_i^{(t-1)}$ is applied. Next, the output of input gate g_i^t with the computation of the biases, input and recurrent weights are calculated and then added to the self-loop weight.

4. Output Gate In the last step, the output is forwarded by the output gate. This output is also called *hidden state*, which is sent to the next network component and also used for predictions. This hidden state is denoted by h_i^t and has the mathematical representation:

Definition 5: Output Gate

$$h_i^{(t)} = \tanh\left(s_i^{(t)}\right) \sigma \left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)} \right)$$

The new cell state that was calculated in 3. Self-Loop to Update Internal State is passed to the tanh activation function. Then is multiplied by the Sigmoid output of the neural network operation that was performed on the input of the current time step and the previous outputs.

2.6.3 Stacked LSTM

The original LSTM model is comprised of a single hidden LSTM layer and its output is consumed by a standard feedforward output layer. A stacked LSTM is an extension of this model, and it can be defined as an LSTM model comprised of multiple LSTM layers. An LSTM layer above another layer provides a sequence output rather than a single value output to the LSTM layer below. Each layer contains multiple memory cells and those layers are stacked onto each other, giving the name to this variant. The usage of stacked LSTM hidden layers is responsible for making the model deeper. The success of neural networks is generally attributed to the increased depth of neural networks in recent years, resulting to be applicable for a wide range of challenging prediction and optimisation problems.

> [The success of deep neural networks] is commonly attributed to the hierarchy that is introduced due to the several layers. Each layer processes some part of the task we wish to solve, and passes it on to the next. In this sense, the DNN can be seen as a processing pipeline, in which each layer solves a part of the task before passing it on to the next, until finally the last layer provides the output [16].

The additional hidden layers are understood to recombine the learned representation from prior layers and create new representations at a high level of abstraction. Thus, since LSTMs operate on sequential data, the addition of layers adds levels of abstraction of input observations over time.

> While it is not theoretically clear what is the additional power gained by the deeper architecture, it was observed empirically that deep RNNs work better than shallower ones on some tasks [17].

While the above quote states that it is not theoretically proven as to why a deeper RNN architecture is beneficial for predictions, stacked LSTMs have become a stable technique for challenging sequence prediction problems. The stacked LSTM architecture is similar to adding additional hidden layers to the Neural Network to make it deeper. Those additional hidden layers are understood to recombine the learned representation of prior layers and use them to create new representations at a higher level of abstraction than the previous layer.

2.7 Cloud Monitoring and Execution Traces

Cloud monitoring involves the tracking of Quality of Service (QoS) parameters of a heterogeneous environment (ie. VM, storage, network and appliances), as well as the physical resources those environments reside on. Additionally, the applications that are running in those environments and the data that is stored and hosted on them. Data that is gathered by monitoring tools are often referred to as *monitoring traces*. Large companies such as Google and Alibaba make some of their gathered monitoring traces public in order to enable researchers to analyse datasets of large-scale cloud systems. These monitoring traces can be used for *workload characterizations*, that is simulating a representation of various production workloads in order to use them for studies regarding scheduling or resource management strategy.

The traces gathered by Alibaba are collected and can be accessed in a GitHub repository[1] and are part of the *Alibaba Cluster Trace Program*. This program aims to help researchers and students to study the workload of different data centres. The repository is split into five sections that were collected for a specific purpose and have different time frames ranging from twelve hours to two months. Two sections traced the workload of machines that handled regular tasks that don't require specialised hardware. The *cluster-trace-gpu-v2020* contains a dataset collected over two months of approximately 1800 machines that have 6500 GPUs in total that describe AI/ML workloads that are provided by the *Alibaba Platform for Artificial Intelligence*. This dataset is mainly used for the Evaluation Scenarios. Additionally, some datasets focus on microservices and microarchitectures. The first contains call dependencies, response times, and call rates and the second was not available at the time of conducting the evaluation of the master's thesis. Cheng et al. [18] analysed the cluster traces of Google and Alibaba and found that warehouses have high unnecessary costs because the overall resource utilisation was between $20 - 40\%$. They performed a case study to characterize co-located long-running and batch job workloads on Alibaba clusters and observed patterns of overbooking, over-provisioning and poorly predicted resource usage and straggler issues (an unexpected phenomenon with a slow task processing speed performance encountered in distributed deep learning applications). Fengcun et al. [19] analysed the improvement of using deep reinforcement learning to schedule jobs in a cloud data centre compared to hand-crafted heuristics. They used machine learning to automatically obtain a fitness calculation method, that was able to minimize the makespan directly from experience. They have shown that their deep reinforcement learning algorithm outperforms the heuristic-based job scheduling algorithms that were used in the Alibaba traces. Weng et al. [20] did a study on the workload characterization, the found temporal patterns, and the GPU machine utilisation, where they suggested the usage of GPU sharing practices to fully utilise GPUs and the predictable duration for recurring tasks. They found that knowing the duration of ML task instances is key to improving the scheduling decisions but this would require specific framework support, which is not always possible. When investigating the temporal patterns, they concluded that GPUs are idle most of the time because of the contention of resources such as CPU or memory. The reason for this is

[1]Alibaba GitHub Repository: https://github.com/alibaba/clusterdata

that instances can use *spare resources* in the host machines, which can lead to utilising more resources than requested.

Chapter 3

State of the Art

Resource prediction based on machine learning involves the usage of forecasting machine learning algorithms that are capable of predicting future resource usage or availability based on historical data. This can be applied to a variety of resources, such as energy, and hardware utilisation. For example, in energy resource prediction, energy usage patterns may be used to train machine learning models that are capable of predicting energy consumption [21] [22].

There is a large variety of machine learning algorithms available that can be used for resource prediction, including linear regression [23], decision trees [24], random forests [25] and neural networks [26]. Neural networks that are especially promising for forecast prediction are Recurrent Neural Network and Long-Short Term Memory. Improving resource utilisation is especially promising for large-scale computational systems such as data centres and cloud-based infrastructures. The server load in a cloud computing environment is predicted in a hybrid Convolutional Neural Network - Long-Short Term Memory architecture in the paper [27]. The choice of algorithm highly depends on the specific problem and the type of data that is available. Overall, resource prediction based on machine learning can provide valuable insights and support informed decision-making by allowing organizations to better plan and allocate resources.

Tuli et al. [28] proposed a novel prediction and mitigation method using an Encoder long-short-term memory (LSTM) model for large-scale cloud computing infrastructure. This method aims at reducing the application response time while maintaining the service level agreement between the application owner and resource provider.

Thonglek et al. [4] designed a neural network model based on LSTM as a type of Recurrent Neural Network (RNN) to predict resource allocation based on historical data. This model has two LSTM layers, each of which learns the relationship between i) allocation and usage, and ii) CPU and memory. It aims to improve resource utilization in data centres by predicting the required resource for each data pipeline. They analyse the CPU and memory allocation and utilisation separately and train the machine learning model to generate more accurate forecast predictions based on the provided sequential data and the allocated resource information.

Oren et al. [29] represents resource utilisation as a combinatorial optimization problem and uses a set of Monte Carlo Decision Processes (MDP) [30], Deep Q-learning and Graph Neural Networks as heuristics to improve the resource utilisation.

Ngo et al. [31] considered multiple anomaly detection deep neural network (DNN) models with varying complexity. Afterwards, the authors explored selecting one of the models to perform autonomous detection at most IoT, Edge, or Cloud layers. For the evaluations, the authors considered the devices such as NVIDIA Jetson-TX2 and NVIDIA Devbox with four GPU TitanX, respectively, as the Edge and Cloud server machines.

Chen et al. [32, 33] proposed a learning-based method that generates resource allocation decisions to minimize latency and power consumption. Intelligent resource allocation framework (iRAF) resource allocation action is predicted and obtained through self-supervised learning, where the training data is generated from the searching process of the Monte Carlo tree search (MCTS) algorithm.

Tan and Hu [34] used deep reinforcement learning to formulate the resource allocation optimization problem, where the parameters of caching, computing, and communication are optimized jointly.

Chapter 4

Model

4.1 Application Model

The application model structure is shown in figure 4.1. The data pipeline is filled with task requests provided by users of the infrastructure. Section 4.2 describes the data pipeline model and the components that are sent to the Deployment API. It is then sent to a server that consumes the data pipeline through a *Deployment API*. Afterwards, the data pipeline is forwarded to the *Preprocessing* component that prepares the data for further computations as described in section 5.4. It is then sent to the adaptation component (see 4.6). The tasks with their adapted resource utilisation are then sent to the Scheduling component. The tasks are scheduled and mapped onto available resources. The mapped tasks are provided to the *Orchestration* component which then deploys the tasks onto the resources defined in the schedule.

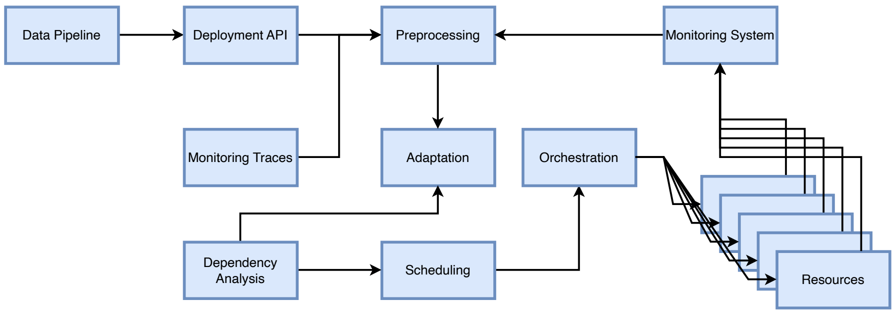


Figure 4.1 Application Model

The state of the resources is tracked by the monitoring system (see 4.5) and the states of all resources are collected with the *Monitoring* component. In case of a rule violation is detected on a resource by the Monitoring component, the tasks deployed on the resource are evaluated for possible adaptation.

4.2 Data Pipeline Model

We model Big Data pipeline workflow received through the DEF-PIPE tool as $W = M, E, Q, M_{SOURCE}, M_{SINK}$, where M is the set of Big Data pipeline tasks, E denotes the Big Data pipelines, Q is the data queue between the tasks, M_{SOURCE} is the data producers (sources) and M_{SINK} is the data consumers (sinks).

4.3 Resource Model

We model the devices provided by the R-MARKET tool as a set of resources distributed over the computing continuum. We define the set of resources as $R = r_1, r_2, \ldots, r_n$, where $n = |R|$ denotes the total number of resources. Thereafter, CPU_j^{util}shows a "non-idle" CPU of a resource r_j. The $CPU_j^{util}$is defined as CPU_j^{util}= CPU_j^{busy}+ CPU_j^{wait}. $CPU_j^{busy}$and CPU_j^{wait}, respectively, show the amount of CPU that is "busy" with processing and "waiting" for other processes to be finished. In addition, we define the utilised memory as $MEM_j^{util} = \frac{MEM_j^{busy}}{MEM_j^{total}}$.

4.4 Network Model

We model the network channels between the computing resources as the round-trip latency LAT and network bandwidth BW between the resources r_i and r_j. Afterwards, we denote the number of data bytes/packets sent and received to/from resources as: $DATA_{sent} = (NET_{bytesSent}, NET_{packetsSent})$ denotes the number of packets (in bytes) sent by the system to other resources. $DATA_{recv} = (NET_{bytesRecv}, NET_{packetsRecv})$ denotes the number of packets (in bytes) received by the system from other resources.

4.5 Monitoring Model

The monitoring states $S_t$that is sent at time step $t$from the resources to the Monitoring and Analysis component consisting of all monitoring metrics collected from the computing continuum. Then, we define a capacity of a resource as MONrt, denoting the monitored data of resource r at time step t. In this case, St is defined as $S_t = MON_{r1}^t, MON_{r2}^t, \ldots, MON_{rn}^t$ as the combination of monitored data of a resource r_i at a time step t.

4.6 Resource Prediction Model

The monitoring system on the computing continuum records the performance metrics, such as CPU, GPU, memory utilisation and network usage, along with the runtime execution of tasks. We train an LSTM-based machine learning model on the monitored and historical data. Then, the output is obtained from the LSTM prediction model. In the case that the difference between the prediction and monitored data is less than a threshold value (e.g., 0.1), we obtain the predicted under-utilized resources for task scheduling. Otherwise, we tune the training model parameters or add more historical data to improve the prediction accuracy.

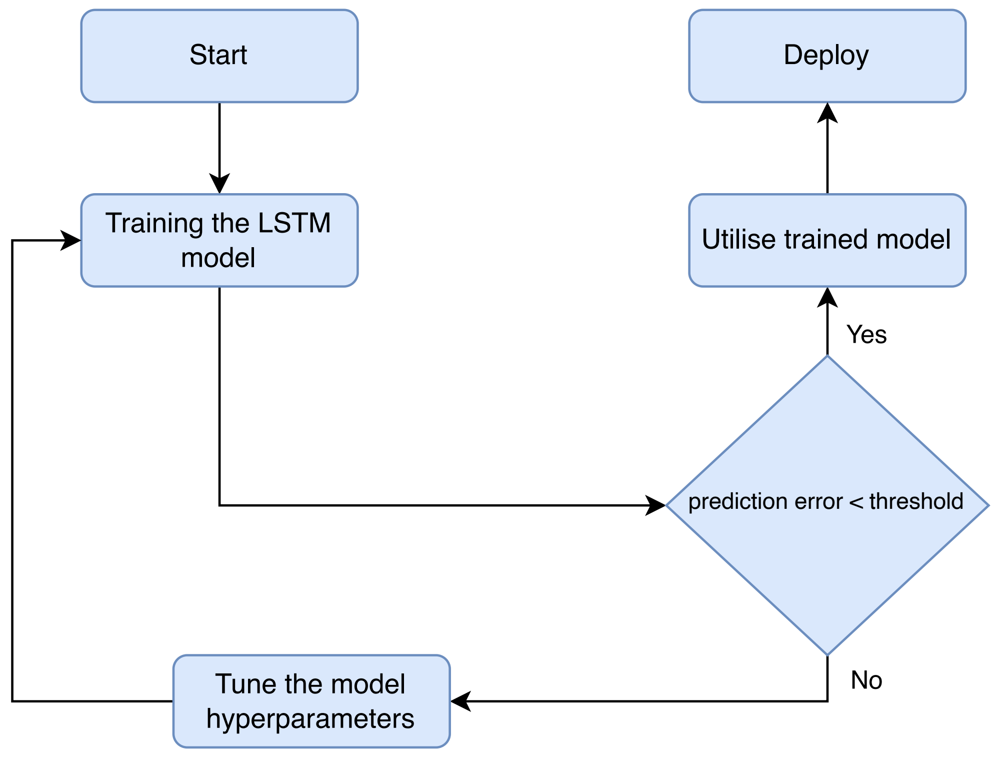


Figure 4.2 Flow chart of resource prediction

Chapter 5

Architecture and Implementation

5.1 Scheduling and Adaptation Approach

5.1.1 Scheduling and Adaptation Functional Architecture

The architecture of the integrated SAA scheduling and adaptation tool consists of five components, displayed in Figure 5.1.

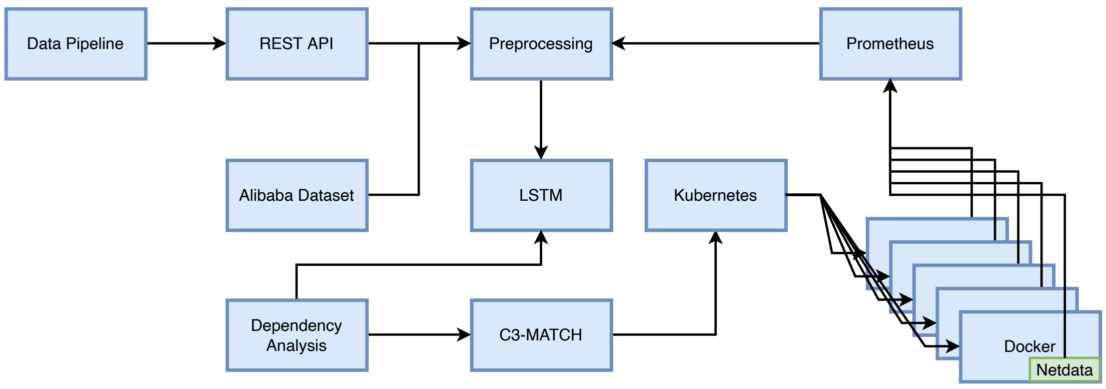


Figure 5.1 SAA Architecture

Data Pipeline consists of tasks provided by users that are to be deployed on the Computing Continuum. These tasks are forwarded to a REST API on our endpoint.

REST API enables the integration of the SAA scheduling and adaptation tool with deployment and orchestration systems.

Alibaba Dataset is used to train the prediction model on real data traces consisting of tasks provided as a pipeline to a GPU cluster. The dataset is further described in chapter Evaluation and Results.

Preprocessing is used to transform the data coming from either the data pipeline or the Alibaba dataset into a form that is possible to be fed to the machine learning models for training and inference. The methods used for preprocessing are described in section 5.4.

LSTM deep neural networks are used to predict the resource utilisation for the tasks incoming from the data pipeline. The predicted resources are CPU and memory and both are predicted for each task. Further description of the LSTM variants used is found in section 5.4.

Dependency analysis orders the candidate tasks scheduled on each resource in a preference list based on the aggregate pipeline communication time to each task.

Scheduling with C^3-MATCH maps the pipeline tasks to the resources using a matching theory algorithm, applied to the task and resource preference lists in response to infrastructure drifts.

Netdata is deployed on all computing devices that are used to execute the tasks and receive them from the C^3-MATCH scheduler. Netdata is a monitoring tool that tracks the status of a resource it it deployed on. The capabilities and usage is described in more details in section 5.2.3.

Prometheus fetches monitoring data from resources that have Netdata deployed to identify SLO violations or anomalies during execution. Additionally, it is used to aggregate and store monitoring traces of executed tasks to later use them for fine-tuning the LSTM model. Further details are found in section 5.2.4.

Docker is used to containerise tools such as Netdata and Prometheus and deploy them on a Kubernetes cluster. Also the tasks in the data pipeline are containerised and deployed on the resources of the Computing Continuum [35]. Additional details are found in section 5.2.1.

5.1.2 Scheduling

The scheduler is *capacity-aware algorithm* designed for asynchronous data processing workflows called C^3-MATCH .

> We model the asynchronous communication between microservices using data element queues, expressing temporary storage of communication data elements [36].

C^3-MATCH defines a scheduling approach that is similar to a matching game, where both the tasks in the pipeline and the available resources participate in this game. On the task side, each task ranks the available resources based on its resource preferences.

On the resource side, each resource also ranks the deployable tasks based on its task preferences.

5.1.3 Adaptation

The adaptation process consists of multiple components that communicate with each other via API. The usual execution order is represented by the order in the components mentioned in this section.

Monitoring and Analysis Component is responsible for monitoring the resources and tasks and storing traces of the behaviour of the system on the disk. This is necessary to be able to analyse the correlations of the tasks and resources and train the LSTM model to generate accurate predictions for future task deployments. Each available resource will run a Netdata container in the background that monitors its current state of it. This component runs a Prometheus instance that periodically scrapes all resources with a Netdata endpoint and checks each of them for rule violations. Prometheus includes a rule-based approach that triggers events in case a defined rule is broken by a resource. In this context, rules denote the expected limits of a resource such as a CPU utilisation that must not exceed a specific percentage value. Otherwise, an event is triggered that is sent to the Adaptation component (see 5.1.3) that will adapt the affected resource to function in expected limits again.

Adaptation Component uses a trained LSTM model to adapt the hardware requirements of the current tasks that reside in the task pipeline. Tasks are provided with initial hardware utilisation limits that are not to be exceeded. As already stated in 2.7, the majority of tasks utilise more hardware resources than they require. The Adapt Component analyses the tasks, their properties and their ordering and adapts the hardware utilisation based on the estimations done by the LSTM model.

5.1.4 The SAA Approach

The SAA adaptation approach is defined as a loop, that cyclically updates every component contained in the loop. The adaptation loop is shown in figure 5.2 and consists of the Big Data pipeline and the *resources* containing all registered and monitored to the *Monitoring and Analysis* component. Both the Big Data pipeline and the resources send the gathered monitoring information at a time step P_t or S_t respectively to the *Monitoring and Analysis* component, where t denotes the current time step.

The Monitoring and Analysis component then uses the monitoring data P_t and S_t and calculates an action A_{t+1} as an update for the resources, where $t+1$ denotes the next time step. This received action A_{t+1} is then used by the Resource components to reconfigure the resources if necessary. The adaptation uses monitoring of tasks and resources to retrieve the monitoring data P_t, S_t. The monitoring component is necessary to obtain information about failures or performance fluctuations along with under-, and over-utilisation of resources. The monitoring data, therefore, provide feedback if a resource can handle the additional load, and thus, more tasks will potentially be mapped

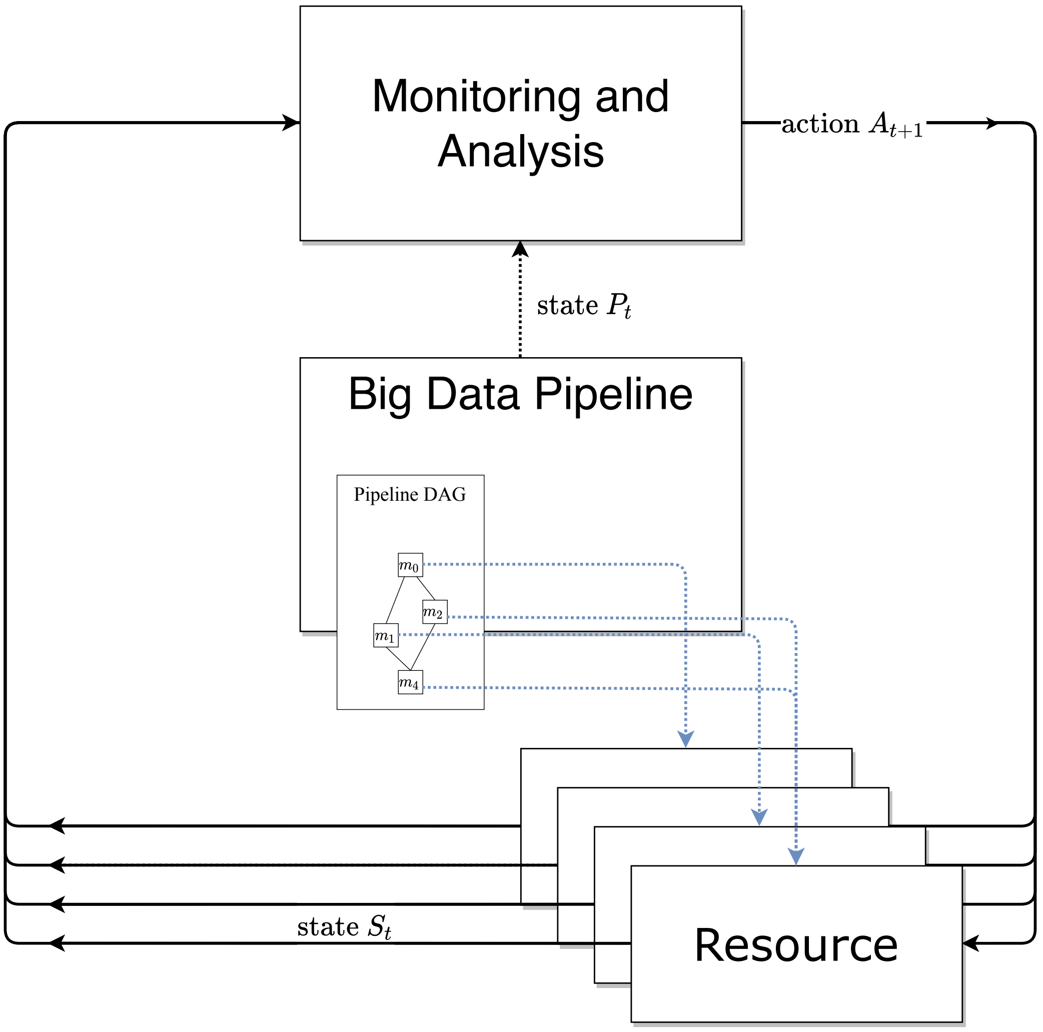


Figure 5.2 Adaptation Loop

to it for execution. In case the resource is not capable of handling the current load, less demanding tasks will be mapped to that resource in future scheduling cycles and it will be reconfigured accordingly.

A monitored resource information consists of the CPU, memory and storage utilization, in addition to network bandwidth usage. The Big Data pipeline and resources send their monitoring data P_t, and S_t respectively to the Monitoring and Analysis component. Afterwards, given those monitoring data, the analyser will determine the next action A_{t+1} of the resources. The monitoring feedback will be periodically retrieved from all registered resources and provided to the analyser. In case of a resource side failure, the monitoring mechanism will be alerted of the occurred anomaly. The analyser uses this monitoring data to decide if any actions $a \in A$ on the current scheduling plan have to be applied. The following actions A are considered in the adaptation approach:

Resource load within expected parameters (S_1)

When tasks on resources are well mapped and no action has to be taken for these resources, this is denoted in the action set A_{t+1} as the state S_1. This state is achieved when a resource is running within expected parameters, the estimated time of completing a task is not exceeded or other anomalies occur.

Resource load under expected parameters (S_2)

When the monitoring component detects under-utilization of a resource, tasks of the pipeline are mapped to it to be efficiently utilized. This state is categorized as S_2 in the analyser and the action set will be modified to improve the utilization of resources grouped into S_2.

Resource load over expected parameters (S_3)

If a resource is over-utilized, there are multiple options. If it is detected that horizontal scaling solves the over-utilization and the resource instance is capable of being scaled up, it will be reconfigured to be within expected utilization levels. Furthermore, if an ill-defined task is detected and mapped to a resource that isn't capable to fulfil the computation within a reasonable time frame, this task will be migrated to another resource. This is categorized as S_3 in the analyser, and the action set will be modified so that resources grouped in S_3 will be less utilized with the upcoming updates of the action set [35].

5.2 Third-Party Tools

In this section, the third-party tools that were used in this master's thesis are briefly described and where they were used. These tools range from deployment components and machine learning frameworks used to implement and train the machine learning models.

5.2.1 Docker

Docker [37] is an open-source platform for automating the deployment of applications inside containers. A container is a standalone executable package that includes everything needed to run a piece of software, including the code, runtime, libraries, environment variables, and system tools. Abstracting applications as containers provide a consistent and reproducible environment, which makes it easier to move applications between development, testing and production environments. Containers are often used to package an entire application with all its dependencies into a single container image that can easily be moved and run on any device with a Docker runtime. The docker toolbox provides tools for building, shipping and running containers, including a runtime environment for containers called Docker Engine, Docker Hub (a repository for storing and sharing images) and the Docker CLI to be able to interact with Docker via a command-line interface. In the master's thesis, Docker is used for containerising the monitoring tool mentioned in section 5.2.2 as well as the LSTM model described in section 5.5. This is done to easily deploy them inside the distributed system in a scalable manner.

5.2.2 Kubernetes

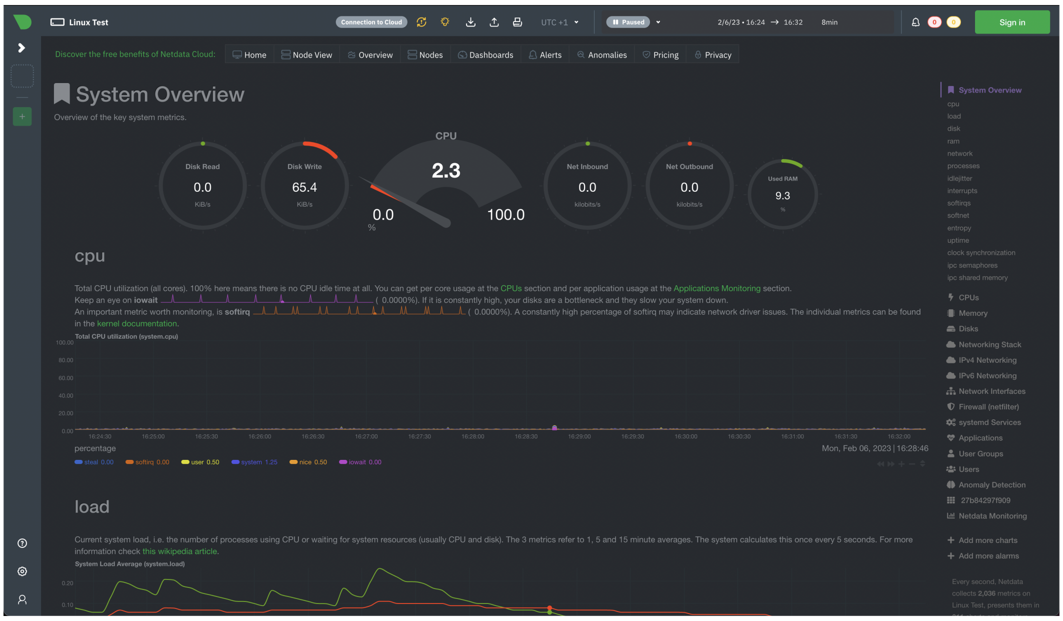
Kubernetes [38] is an open-source platform for automating the deployment, scaling, and management of containerised applications. It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF). Kubernetes provides a way to manage and organize containers (such as Docker containers) at scale, making it easier to deploy, update, and maintain applications. It does this by abstracting the underlying infrastructure and providing a unified API for managing containers. Kubernetes helps to automate many of the manual tasks involved in deploying and managing containers, such as scaling up or down the number of replicas of an application, rolling out updates, and managing network and storage resources. It also provides features for scaling and self-healing, allowing your applications to be more robust and resilient. Kubernetes has become a popular platform for deploying cloud-native applications and is widely used by organizations of all sizes across various industries.

5.2.3 Netdata

Netdata [39] is an open-source tool that collects real-time metrics, including CPU usage, disk activity, bandwidth usage and furthermore. The reason we chose Netdata for monitoring is that it is a lightweight tool mostly written in C, Python and Javascript and it requires minimal resources, which is necessary when monitoring edge devices. One of its major features is that it runs without interrupting any of the applications running on the same device. This is achieved by only using idle CPU cycles while running. Netdata provides an in-browser dashboard to analyse each metric in real time with help of visual representation. As an example, a screenshot was taken of a cloud resource in our system as can be seen in 5.3. Netdata has a vast amount of support for other tools to gather data. One of the supported tools is Prometheus, which is used to scrape the monitored data from all resources that have Netdata installed.

5.2.4 Prometheus

Prometheus [40] is an open-source application used for monitoring and alerting of events and is designed to run across various platforms in a scalable and easily deployable manner. Same as Netdata it records in real-time, and stores the gathered metrics in a time series database by using an *HTTP pull model*. It also allows real-time alerting via a rule-defining configuration and also has a flexible query language called *PromQL*, that enables the retrieval and processing of the gathered data. Prometheus has great integration with other tools such as Netdata. In the project, it is used inside a monitoring master node that is continuously scraping all resources that have Netdata installed for monitoring data. In case a predefined rule is broken at a monitored resource, such as CPU over-allocation, Prometheus triggers an event that notifies the Adaptation Component (see section 5.1) to be able to take measurements.



Total CPU utilization (all cores). 100% here means there is no CPU idle time at all. You can get per core usage at the CPUs section and per application usage at the Applications Monitoring section.
Keep an eye on **iowait** (0.0000%). If it is constantly high, your disks are a bottleneck and they slow your system down.
An important metric worth monitoring, is **softirq** (0.0000%). A constantly high percentage of softirq may indicate network driver issues. The individual metrics can be found in the kernel documentation.

Total CPU utilization (system.cpu)

percentage — Mon, Feb 06, 2023 | 16:28:46

steal 0.00 softirq 0.00 user 0.50 system 1.25 nice 0.50 iowait 0.00

Current system load, i.e. the number of processes using CPU or waiting for system resources (usually CPU and disk). The 3 metrics refer to 1, 5 and 15 minute averages. The system calculates this once every 5 seconds. For more information check this wikipedia article.

System Load Average (system.load)

Figure 5.3 Netdata Dashboard Example

5.2.5 Pandas

Pandas [41] is an open-source library created for Python and its main purpose is to provide tools for data analysis and manipulation as well as data structures for large-scale numerical tables and time series data. Pandas allow importing data from various file formats such as *comma-separated values (CSVs)*, which is the file format that Alibaba chose to store the monitoring traces. These CSV files are imported into *Pandas DataFrames*, which is the main data structure used for data manipulation with Pandas. DataFrames are used to merge the different datasets to create the training and test datasets for the LSTM model, and additionally, most of the data analysis is also done with the help of the functions that are built into the DataFrame class.

5.2.6 PyTorch

PyTorch [42] is an open-source machine learning library for Python, developed by Facebook's AI Research lab (FAIR) but is now part of the *Linux Foundation umbrella*. It provides tensor computation (similar to NumPy) with strong support for deep neural networks built on a tape-based autograd system. PyTorch is designed to provide flexibility and ease of use, with a focus on deep learning and neural networks. It can be used for various tasks such as natural language processing, computer vision, and speech recognition. PyTorch also supports deployment on various platforms, including CPUs, GPUs, and TPUs, and has an active community of users and contributors that continue to improve and expand its capabilities. PyTorch is used to build the LSTM model (see section 5.5) and the loss function (see section 5.7) to train the models and use the trained variants for inference of resource utilisation.

5.3 Data Preprocessing

Data preprocessing (also referred to as feature engineering) is the process of designing artificial features, that is, to model features that are used as the inputs for a machine learning (ML) model either during training or inference to make predictions. In short, it is the development of new data features from raw data. It is a very important step in machine learning since the later prediction performance depends on the quality of the data. The accuracy of an ML model heavily relies on a well-composed and precise set of features to make reliable predictions. Regardless of the data or ML architecture, if a feature set is not well-suited, it does have a direct impact on the ML model.

> Feature engineering is a machine learning technique that leverages data to create new variables that aren't in the training set [43].

Feature engineering involves the extraction and transformation of variables of a raw data source. This process is necessary to be able to use the data for the training of an ML model and successfully use its inference to make predictions. The steps required in feature engineering include Feature Extraction and Feature Cleansing followed by feature creation and analysis (see section 5.3.3). It can produce new features to enhance the model accuracy and simplify and speed up data transformation.

5.3.1 Feature Extraction

Feature extraction is the process of collecting and assembling all the data that is required for the ML training into a feature set. Data often resides in multiple data sources, therefore collecting training data is a challenging process since those different data sources need to be connected for it to be usable. Furthermore, data sources often have vastly different formats and types, which also needs to be taken into account. Feature extraction also faces the challenge of not distorting the original relationships or significant information of the raw data and also compressing the amount of data into manageable quantities for the ML models to process and train on. The Alibaba cluster traces consist of a multitude of datasets, each with its respective focus. Because of the distributed nature of the data, first, the common key features were required to be analysed to be able to later join the data based on these common key features to an actual training and test dataset.

5.3.2 Feature Cleansing

Feature cleansing is the process of cleaning a dataset. A major part is to add missing values or to remove data points that are not beneficial for the training of an ML model. In the Alibaba dataset, all tasks were removed that did not successfully finish their execution. The reason why the cluster of failed tasks was excluded from the training was that the ML model should be trained on successfully finished tasks since it was not stated what the reason for the failure of the tasks was and adding failed tasks could, in this case, lead to unwanted behaviour when predicting incoming tasks. And therefore, it was deemed a better approach to not include those tasks. This process is shown in listing

5.1, where only tasks will be kept that had their feature column set to `'terminated'`, which states that they did successfully terminate their execution.

```
# df is a preprocessed dataframe provided by Alibaba
df.query("status=='Terminated'", inplace=True)
df.drop(columns=['status'], inplace=True)
```

Listing 5.1 Dropping Failed Tasks from DataFrame

After dropping all failed tasks, the feature column `status` is dropped from the dataset since it now does not provide any information. Next, other feature columns are dropped as well that are either not necessary for the training or inference of the ML model or are duplicated columns that were added by the Pandas framework after joining the DataFrames. Dropped columns contain fields such as *read, write, instance name, instance ID, worker-pool name, group, user and status_t*, where status_t is simply a duplicated column that was appended while doing a join operation. Feature cleansing also involves the task of ensuring that the correct data types are used for each dataset. In the case of the mentioned dataset, it was necessary to denote the feature field `start_date` as a time-stamp as well as the index for the dataset, to ensure a time-series-based ordering of the data. Also, fields such as the *number of instances* were declared as floating point numbers. To ensure that this will not result in unwanted behaviour, this field data type was changed to an unsigned integer.

5.3.3 Feature Creation and Analysis

Feature creation involves the process of data labelling. Data labelling identifies data such as images, and text files and adds meaningful and informative labels to it by creating new variables which will be the most helpful to the ML model. This is done to provide the data labels as context to the ML model so it can learn from it. Data labelling is required for various use cases in Supervised Learning since ML models of that category require labels to train on. Features that were created for the datasets that later are fed into the LSTM model are one-hot encoded feature columns. These were either generated from the task type, which is used to differentiate between the various tasks and is used to enable the model to comprehend categorical data for training and inference. Similarly, the *number of instances* is one-hot encoded to provide additional information about a task by providing categorical data as encoded feature columns. The one-hot encoding method and how it is applied is described in more detail in section One-hot Encoding.

5.3.4 Feature Analysis

Feature analysis often uses visualisation like histograms, scatter, box and line plots to analyse if the created feature data set is correct. Feature analysis is also used to discover patterns, spot anomalies, check assumptions or test a hypothesis. The analysis of features and results is done in chapter Evaluation and Results.

5.3.5 Feature Selection

Feature selection is the process of reducing the number of input variables of a feature set when developing a predictive model. Feature selection is simply the process of selecting and excluding given features without changing them as is done with *dimensionality reduction*. A reduction of the number of input variables is desirable to both reduce the computational cost of training a model and in some cases, it also increases the prediction performance of the model.

5.4 Data Preprocessing Techniques

In this section, the data preprocessing techniques that are used on the datasets used in this thesis are further elaborated.

5.4.1 One-hot Encoding

One-hot encoding [44] is used to represent categorical data of a finite set by an index in that set, where only one element has its index set to 1, and all other values are assigned the value 0. This technique is useful when categorical data should be fed to a machine learning model since many model architectures can only handle numerical values, making it necessary to transform strings or other data types to numbers.

> One hot encoding is a process of converting categorical data variables so they can be provided to machine learning algorithms to improve predictions. *One hot encoding is a crucial part of feature engineering for machine learning* [44].

The one-hot encoding technique is used for the categorical data of *task type*, where each type of task is represented as an index in the task-type vector. Similarly, one-hot encoding is used to classify a different number of instances (or Kubernetes Replicas) into clusters, that are then mapped to an index of the encoded feature vector.

To generate one-hot encoded feature vector columns in the application, we used the Python framework `Pandas`. Pandas is a popular framework that was designed to handle big datasets that are represented with their internal `Series` and `DataFrame` data types, where `Series` is a feature column and a DataFrame is a collection of Series or a feature vector. The process of generating one-hot encoded columns for a DataFrame is shown in the listing 5.2. To generate one-hot encoded columns for a DataFrame, the column or row with the categorical data has to be isolated. In line 2, the `instances` Series is used to generate a new Series that contains the index position for the one-hot encoded feature vector. Next, in line 3 the `get_dummies()` function of Pandas is used to generate the one-hot encoded feature vector of a DataFrame data type that is stored in a variable called `dummies`. In lines 4-8, the columns of the `dummies` DataFrame are renamed to be concise with the defined instance clusters. The `inplace=True` flag is necessary since Pandas usually does not modify DataFrames directly but an operation commonly returns a copy of the modified DataFrame, and this flag omits the copy process and renames the columns directly in the DataFrame.

```python
# instances is a pd.Series that contains the instance number of a task
instance_indices = instances.apply(lambda x: get_index_pos(x <= ranges))
dummies = pd.get_dummies(instance_indices)
dummies.rename(
    columns={
        col: f'inst_clust_{col}' for col in dummies.columns
        },
    inplace=True)
# training_df is a pd.DataFrame that contains the training dataset
inst_df = training_df.join(dummies)
```

Listing 5.2 One-Hot Encoding Pandas Example

Finally, the one-hot encoded feature vector `dummies` is joined with the training DataFrame called `training_df` and the result is stored in the variable `inst_df`. An example one-hot encoding of the first few values of the training DataFrame is shown in table 5.1, where $task_0$ is categorised to be part of instance cluster 0 and the following two tasks $task_1, task_2$ are part of instance cluster 3.

Table 5.1 Example of One-Hot Encoding the Instances

index	inst_clust_0	inst_clust_1	inst_clust_2	inst_clust_3
0	1	0	0	0
1	0	0	0	1
2	0	0	0	1

Using one-hot encoding enables the LSTM model to additionally learn from categorical features such as the type of task (see evaluation scenario Adding Task Knowledge) and also to learn and infer from the number of instances (or Kubernetes replicas). These further details about each task enable the ML model to better and more accurately react to the given sequence of tasks.

5.4.2 z-Score Normalisation

z-Score Normalisation (also referred to as *standardisation*) is a process of scaling values of a dataset by first calculating its standard deviation and accounting for it. In case the standard deviation of features is different, then the range of those features will likewise differ. As a result, the outliers in the characteristics are reduced. In order to calculate a distribution of the data with a mean of 0 and a variance of 1, all the data points are first subtracted by their mean and the result of that is then divided by the variance of the distribution.

Definition 6: Mean

The mean μ is calculated with the formula:

$$\mu = \frac{1}{N}\sum_{i=1}^{N}(x_i)$$

where N denotes the number of elements in the dataset and x_i the i-th element in that dataset.

Definition 7: Standard Deviation

The standard deviation is calculated with:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

Definition 8: z-Score Normalisation

The standardisation is calculated by the formula

$$Z = \frac{x - \mu}{\sigma}$$

The z-score normalisation of data points is used to preprocess the feature dataset that is fed to the LSTM model while training or inferring tasks. Standardising the feature dataset is preferred to normalisation (see section 5.4.3) since the impact of outliers is reduced compared to the normalisation scaling method.

5.4.3 Normalisation

In **normalisation**, all values in a data set are scaled in a specified range between 0 and 1 via normalisation and represented as floating-point numbers. The normalisation of data does not influence the distribution of the features, but it does increase the effects of outliers due to the lower standard deviation after the modification. Because of this, it is necessary to deal with outliers before applying normalisation to a data set.

Definition 9: Normalisation

The normalisation scaling formula and its parameters are defined as follows:
The minimum value of a dataset is denoted as x_{min} and the maximum value as x_{max}. The value x denotes a data point that will be scaled between 0 and 1 and its new value is denoted as x_{new}.

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

The normalisation of data points is used for the label dataset that contains the labels that the ML model should predict (i.e. the hardware utilisation of resources). The normalisation is used for the label dataset because its values of it are transformed to a standard scale without distorting the difference between the values. This behaviour is useful when predicting the actual values in the label dataset.

Definition 10: De-Normalisation

The de-normalisation is used to rescale the estimation done by the LSTM model into the range of the actual values. The formula for calculating the inverse of the normalisation is:

$$x = x_{new} * (x_{max} - x_{min}) + x_{min}$$

The old value x is calculated by simply rearranging the normalisation formula and de-normalising the normalised value x_{new}.

The de-normalisation is used to rescale the predicted values forwarded by the LSTM model to the actual data range. This is required since the LSTM model is trained to predict normalised values since this commonly results in more accurate predictions.

5.5 LSTM Architecture

In this section, the architecture of the LSTM model is elaborated. This involves the description of the different layers that are used as well as some operations that are required to improve the prediction of the model. The LSTM model was implemented with PyTorch, an open-source machine-learning library for Python.

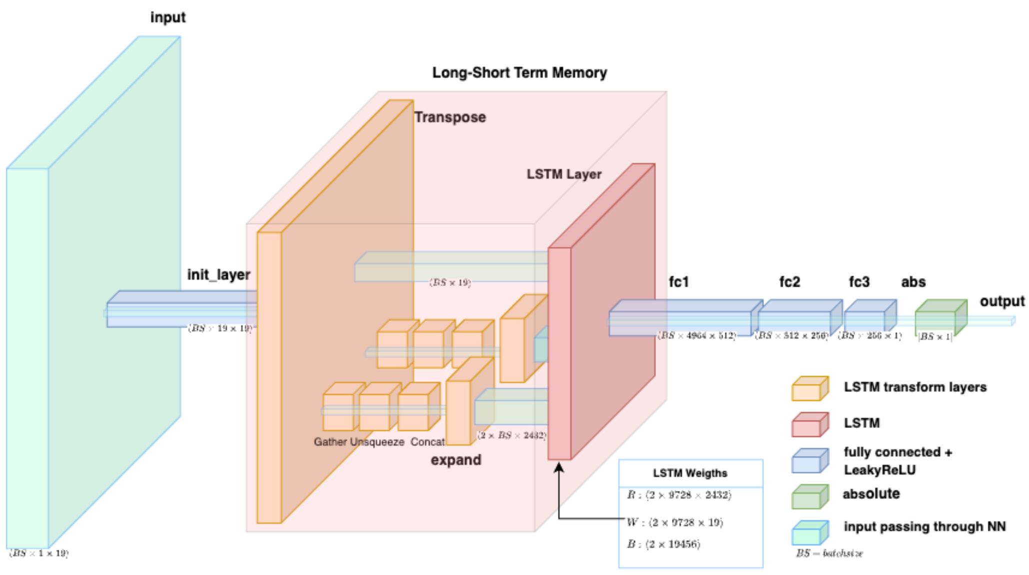

input
Long-Short Term Memory
Transpose
LSTM Layer
init_layer
$(BS \times 19)$
$(BS \times 19 \times 19)$
fc1
fc2
fc3
abs
output
$(BS \times 4964 \times 512)$
$(BS \times 512 \times 256)$
$(BS \times 256 \times 1)$
$(BS \times 1)$
Gather Unsqueeze Concat
$(2 \times BS \times 2432)$
expand
LSTM Weigths
$R : (2 \times 9728 \times 2432)$
$W : (2 \times 9728 \times 19)$
$B : (2 \times 19456)$
$(BS \times 1 \times 19)$
LSTM transform layers
LSTM
fully connected + LeakyReLU
absolute
input passing through NN
$BS - batchsize$

Figure 5.4 LSTM Model Architecture

The LSTM model implementation is called `UtilizationLSTM` and is of type `Module`. This is a default parent class for neural networks that are implemented in PyTorch. The class constructor `__init__()` has four parameters. First, is the instance reference `self` that is used to access functions and variables of an instance. The available instance variables are *input size, hidden size, number of layers* and *device* that holds the reference to the hardware that will compute the training of the neural network. PyTorch enables the training of neural networks on graphical computing units (GPU) which accelerates the training process. For being able to train on a GPU, the NVIDIA API CUDA (Compute Unified Device Architecture) needs to be installed on the device, in order to send the LSTM model on the GPU and train it on incoming data batches. Next, is the input size that refers to the number of feature set columns that were selected by the Feature Selection process. The hidden size value in the parameters also refers to the hidden layer size of the LSTM model that is used to variate the size of the model and its capabilities to remember long-term patterns in time series data. Finally, the number of layers refers to how many Stacked LSTM layers should be applied. For the Evaluation Scenarios, a default of two stacked layers is used. When using two stacked LSTM layers, the size of the hidden layers of the model is doubled. This also includes the model output, where both hidden state outputs are returned by the `forward` function. The reason why an estimation that is then fed back to the model is returned is to be able to analyse the hidden states of the intermediate model layers.

```python
class UtilizationLSTM(torch.nn.Module):

    def __init__(
        self,
        input_size: int,
        hidden_size: int,
        num_layers: int = 1,
        ) -> None:

        super(UtilizationLSTM, self).__init__()
        self.input_size: int = input_size
        self.hidden_size: int = hidden_size
        self.num_layers = num_layers
        self.device = device

        # long-short term memory layer to predict cpu usage
        self.cpu_lstm = torch.nn.LSTM(
            input_size=input_size
            hidden_size=hidden_size,
            num_layers=self.num_layers,
            batch_first=True,
        ).to(self.device)

        # long-short term memory layer to predict memory usage
        self.mem_lstm = torch.nn.LSTM(
            input_size=input_size
            hidden_size=hidden_size,
            num_layers=self.num_layers,
            batch_first=True,
        ).to(self.device)

        self.cpu_lstm_seq = self.init_sequential_layer_batchnorm(
            hidden_size)
        self.mem_lstm_seq = self.init_sequential_layer_batchnorm(
            hidden_size)
```

Listing 5.3 LSTM Constructor

Next, the two LSTM models are instantiated in lines 17 and 25 with the parameters from above. One for the prediction of CPU and one for the prediction of memory utilisation. The `batch_first` parameter provided to both models defines how the model shall return its estimation, in this implementation, the batch will be returned first, followed by the hidden state and then the internal state.

Then, a sequential layer for both LSTM models is generated by the function shown in 5.4. This sequential layer consists of three components. Each component is built with the following components:

Definition 11: Rectified Linear Unit (ReLU)

Rectified Linear Unit (ReLU) is a popular function for transforming the summed weighted input from a layer into the activation of a node or output for that input. The ReLU function can be mathematically described as

$$g(z) = \max\{0, z\}.$$

The advantages of ReLU are its computational simplicity, the representational sparsity (being able to output true zero values opposed to tanh and *sigmoid*), the linear behaviour and the training of deep neural networks via backpropagation was heavily improved when using ReLU compared to more complex methods. The function is linear for values greater than zero which results in ReLU having many of the advantages of a linear activation function when using backpropagation. Yet, ReLU is a nonlinear function since negative values are always output as zero.

Definition 12: Batch Normalisation

Batch normalisation (batch norm) is a generalisation method used to increase the stability and reduce the training time of a neural network layer by re-centring and re-scaling the inputs of the layer. Batch normalisation standardises the inputs to a layer for each mini-batch, and the resulting increase in stability reduces the number of required training epochs to train deep neural networks.

> Batch normalization provides an elegant way of *reparametrizing* almost any deep network. The reparametrization significantly reduces the problem of coordinating updates across many layers [45].

Assumptions that a subsequent layer makes during the weight update that regards the spread and distribution of inputs, will not change by a lot of the activations of the prior layer are standardised.

Normalising the input batch also has a regularising effect, which leads to a reduction of generalisation errors, similar to the usage of activation regularisation. Batch normalisation was proposed to mitigate internal covariate shift. This internal covariate shift occurs because inputs that are forwarded to the neural network and also from each layer have a corresponding distribution that affects the training process because of the randomness of the parameter initialisation and input data. Because of the resulting internal covariate shift, the succeeding network layers need to readjust their weights in order to update their state from the data that was forwarded from the proceeding layer with a bias in terms of the distribution of activated nodes. This requirement to readjust to new distributions is even more severe in deep neural networks that have many layers and the increased number of layers amplifies this effect. Batch normalisation reduces unwanted shifts propagated by layers, speeds up the training, and also produces more reliable and general models.

Definition 13: Linear (Fully-Connected) Layer

Linear layers, also known as fully-connected layers connect every input neuron to every output neuron and are commonly used in neural networks. The structure of a neural network consisting of fully-connected layers can be seen in the figure below, where each node is connected to all nodes of the previous and the succeeding layer. A linear layer is defined by three parameters, the number of inputs, the number of outputs and the batch size. Operations such as forward propagation, activation gradient computation, and weight gradient computations are directly expressed as matrix multiplications.

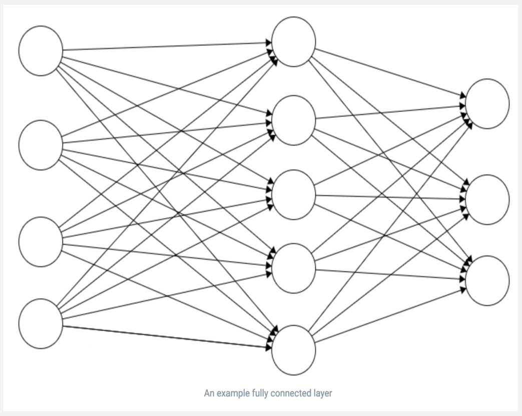


Figure 5.5 Fully-Connected Layers [2]

These LSTM and sequential layers are the core of the machine learning architecture inside the `UtilizationLSTM` class and are responsible for the estimation of the CPU and memory utilisation and are used by forwarding inputs through each layer until the estimation is calculated. The received input is sent to the `forward` function of the class that expects a PyTorch tensor as the input type.

```python
def init_sequential_layer_batchnorm(self, hidden_size: int) -> torch.nn.
    Sequential:
    return torch.nn.Sequential(
        torch.nn.ReLU(),
        torch.nn.BatchNorm1d(hidden_size),
        torch.nn.Linear(hidden_size, 512),

        torch.nn.ReLU(),
        torch.nn.BatchNorm1d(512),
        torch.nn.Linear(512, 128),

        torch.nn.ReLU(),
        torch.nn.BatchNorm1d(128),
        torch.nn.Linear(128, 1),
    ).to(device)
```

Listing 5.4 LSTM Initialise Sequential Layers with BatchNorm

As can be seen in figure 5.4 the input vector consisting of a pipeline task batch is forwarded to the *initial lstm layers*. This step is implemented in the LSTM `forward` function (see 5.5) in lines 8 and 9. The `get_hidden_internal_state` (see 5.6) function seen on the right side of lines 8 and 9 is used to create the initial internal and hidden state based on the time series batch provided as the `input` for the model. The input vector is a PyTorch tensor that is then sent to a function that generates the initial hidden and internal state based on the shape of the input vector. The general LSTM prediction model is split into two smaller LSTM components that each predict the utilisation of one resource unit. A resource unit is either the CPU usage or the allocated memory. At creation these LSTM components use the length of the input as the expected input size and the quality of the prediction performance heavily depends on the chosen hidden layer size as well as the number of Stacked LSTM layers. The final LSTM output of either resource unit is then sent to traditional sequential layers that use batch normalisation as the generalisation strategy.

Afterwards, the output of those sequential layers is concatenated column-wise as a label vector of the form $[(cpu_1, mem_1), \ldots, (cpu_{bs}, mem_{bs})]$ where bs denotes the batch size or the number of data points in the batch. At line 22 the calculated output is then sliced in order to only include the same amount of elements as the elements of the input vector. This has to be done since using multiple LSTM layers also increases the number of output elements by the simple formula $elements_{input} \times \#layers$. For the estimation of the resource utilisation, only the last estimation section of the output is required since this is the section that contains the prediction of the last LSTM layer. In line 25 the absolute values of the output are calculated. This is done to ensure that negative values are not included in the prediction since the hardware utilisation can't be less than zero.

```
1  class UtilizationLSTM(torch.nn.Module):
2
3      def __init__():
4          ...
5
6      def forward(self, input: torch.Tensor) -> torch.Tensor:
7          # Propagate input through LSTM
8          _, (cpu_ht, _) = self.cpu_lstm(input, self.
               get_hidden_internal_state(input))
9          _, (mem_ht, _) = self.mem_lstm(input, self.
               get_hidden_internal_state(input))
10
11         # Reshaping the data for the Dense layer
12         cpu_ht = cpu_ht.view(-1, self.hidden_size)
13         mem_ht = mem_ht.view(-1, self.hidden_size)
14
15         cpu_out: torch.Tensor = self.cpu_lstm_seq(cpu_ht)
16         mem_out: torch.Tensor = self.mem_lstm_seq(mem_ht)
17
18         # Concat the two tensors column-wise
19         output = torch.cat([cpu_out, mem_out], dim=1)
20
21         # Only use the last stacked lstm layer as output
22         output = output[(self.num_layers - 1) * input.size(0):]
23
```

```
        # don't allow negative values
        output = torch.abs(output)

        return output
```

Listing 5.5 LSTM Forward Function

```
def get_hidden_internal_state(self, input: torch.Tensor) -> Tuple[torch.
    Tensor, torch.Tensor]:
    hidden_state = torch.zeros(self.num_layers, input.size(0), self.
        hidden_size).requires_grad_().to(device)
    internal_state = torch.zeros(self.num_layers, input.size(0), self.
        hidden_size).requires_grad_().to(device)

    return (hidden_state, internal_state)
```

Listing 5.6 LSTM Initialise Internal and Hidden State

5.6 DataFrame Scaler

The `DataFrameScaler` is a custom class that handles the scaling of datasets that are in the form of a *Pandas DataFrame* data type. The reason for implementing a custom class for this thesis instead of the popular approach of the `StandardScaler` and `MinMaxScaler` classes provided by *scikit-learn* is that these third-party classes destroy valuable information about the dataset while scaling it to either be standardised (see z-Score Normalisation) or normalised (see Normalisation). Information that is not kept when scaling the DataFrames is data point timestamps, which are converted to ascending integer values. Similarly, the column description is also not kept after the scaling, so it is necessary to dynamically create a map of an integer to the column description. Because of behaviour, a custom Python class was implemented that scales the DataFrame to a standardised or normalised DataFrame or if required to be fed to the LSTM model converted to a PyTorch Tensor. This class ensures that the timestamps of each data point are kept and fed to the LSTM model and also keeps the column descriptions. Additionally, it is possible to define columns that should be ignored while scaling the DataFrame. This is useful for columns that were one-hot encoded (see One-hot Encoding, Adding Task Knowledge, Adding Instance Knowledge) before the scaling process. The *DataFrameScaler* class contains both the logic for standardising and normalising a DataFrame.

```
class DataFrameScaler():
    df_data_types: dict
    std_dev_df: DataFrame
    norm_dev_df: DataFrame
    scaled_df_columns: list[str]

    def __init__(
        self,
 9          df: DataFrame = None,   # the Pandas DataFrame to be scaled
10          filter_columns: list[str] = None   # DataFrame columns to ignore
11          ) -> None:
12  
13          self.filter_columns = filter_columns
14          self.df_data_types = None
15  
16          if df is not None:
17              self.fit(df)
```

Listing 5.7 DataFrameScaler Constructor and Instance Variables

As shown in lines 2-5 listing 5.7 the class contains different instance variables to hold the necessary information about a DataFrame, such as the data types of each column. The variables in lines 3-5 hold the standard deviation, normalisation DataFrames and the list of the columns that are to be scaled, respectively. The DataFrame values for the standard deviation and normalisation are calculated in the `fit()` method (see listing 5.8) that resides inside the class. These DataFrames are necessary to later be able to scale other data points that share the same distribution as the DataFrame that was used to fit the scaler. This is used for estimations forwarded by the LSTM model that are within the range of normalised values. In order to rescale the estimation into the actual range, the `norm_dev_df` is used to transform the data by using the formula for de-normalisation (see Normalisation).

```
1  def fit(self, df: DataFrame) -> None:
2      self.df_columns = df.columns
3      # preserve datatypes of dataframe
4      self.df_data_types = df.dtypes.to_dict()
5      self.scaled_df_columns = self.get_scaled_columns(
6          df, self.filter_columns)
7      self.std_dev_df = self._get_std_mean_df(df)
8      self.norm_dev_df = self._get_norm_min_max_df(df)
```

Listing 5.8 DataFrameScaler Fit Method

The `fit()` method of listing 5.8 is used to preserve the columns of the given DataFrame (line 3), the data types in each of its columns (line 4), the columns that need to be scaled, and a standard deviation and normalisation DataFrame based on the data points in the DataFrame given as a parameter. The execution `fit()` function is mandatory to be able to scale data points with the `DataFrameScaler` class even if it is not enforced to provide a DataFrame when instantiating the scaler instance. After fitting the DataFrame with the `fit()` method, data points can be (de-)standardised or (de-)normalised based on the variables calculated in this function. The calculations for the scaling are done with the internal operations provided by Pandas to ensure a performant execution independent of the number of data points.

5.7 Penalty Mean Squared Error Loss Function

The *Penalty Mean Squared Error (PMSE) loss function* is a custom loss function based on the commonly used *Mean Squared Error (MSE) loss function* [46] (see 5.7).

The Mean Squared Error metric measures how close a Regression line is to a set of data points. It is a risk function that corresponds to the expected value y_i of the squared error loss (see 2.1). A larger MSE indicates that the data points are widely dispersed around its mean, whereas a smaller MSE indicates the opposite.

Definition 14: Mean Squared Error

The *Mean Squared Error (MSE)* of an estimator (i.e. trained LSTM model) measures the average of the squares of the errors, which is the average squared difference between the actual value y and the estimated values $\hat{y}$.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

The squaring is critical to reducing the complexity with negative signs. If used as a loss function (criterion), the machine learning algorithm will update its weights in order to reduce the calculated loss value produced by the MSE loss function.

Penalty in this loss function refers to increasing the loss of predicted values $\hat{y}_i$ that are lower than the actual value y_i.

Definition 15: Calculation of the Penalised Prediction Value $\hat{y'}_i$

The penalty is defined as a positive real number ranging from zero to infinity and its mathematical representation is $Penalty = [0, \infty]$. The penalty value denoted as *Penalty* is subtracted from a predicted value $\hat{y}_i$ if it is smaller than the actual value y_i. A subtraction is required to further increase the distance between the actual value and the (under-allocated) predicted value. The result of this calculation is the updated penalised prediction value $\hat{y'}_i$. This conditional mathematical representation is shown in the following:

$$\hat{y'}_i = \begin{cases} \hat{y}_i - Penalty, & \text{if } \hat{y}_i < y_i \\ \hat{y}_i, & \text{otherwise} \end{cases}$$

Therefore, the more the prediction values are below the actual values, the more the penalty value is applied, and thus the error of a predicted data batch increases.

The custom PMSE was implemented because the more information was included in the feature set while training, the more likely the LSTM model is to predict values that are lower than the actual value (see Evaluation Scenarios). Since an estimation lower than the actual value could lead to problems such as a task not finishing in the required time frame, or an *out of memory* exception, that not only interrupts the system to terminate the task but could also lead to system failure. The reason is that a penalty is added to the predicted value in case of being lower than the actual value. With this measure in place, the LSTM model is more likely to predict values that are higher than the actual value. While this might introduce resource wastage, it is still preferable to introduce some resource wastage than provide an unstable system configuration.

Definition 16: Penalty Mean Squared Error

The *Penalty Mean Squared Error (PMSE)* of an estimator (i.e. trained LSTM model) measures the average of the squares of errors similar to *mean squared error* but also applies a penalty to an estimated value $\hat{y}_i$ that is lower than the actual value y_i (see 5.7). The mathematical representation of PMSE is:

$$PMSE = \frac{1}{N} \sum_{i=1}^{N} \left(y_i - \hat{y'}_i \right)^2$$

As can be seen, it is similar to the representation of MSE (see 5.7), but uses a modified estimated value $\hat{y'}_i$ that was penalised before the calculation.

The `PenaltyMSELoss` class inherits from the PyTorch Module package in order to be able to use it as the loss function for the error calculation of the predicted and actual values. The `PenaltyMSELoss` class constructor (see listing 5.9 line 4) has two parameters, the `self` keyword that is used to access instance parameters, and the `penalty` parameter, that is expected to be of type `float`. Inside the class constructor, an `MSELoss` instance (line 6) will be created in order to calculate the final loss values based on the mean square error calculation.

```
1  class PenaltyMSELoss(torch.nn.Module):
2      penalty: float
3
4      def __init__(self, penalty: float = 0.05):
5          super().__init__()
6          self.mse = torch.nn.MSELoss()
7          self.penalty = penalty
8
9      def forward(self, pred: torch.Tensor, actual: torch.Tensor) -> float:
10         under_alloc_indices = pred < actual
11         pred[under_alloc_indices] = pred[under_alloc_indices] - self.
               penalty
12
13         return self.mse(pred, actual)
```

Listing 5.9 Penalty Mean Squared Error Python Class

The `forward` function (see 5.9 line 9) is used to calculate the loss between the estimation (denoted as `pred`) and actual values (denoted as `actual`). Both the estimation and actual values are represented as PyTorch tensors, a multi-dimensional matrix that contains the elements of a single data type, in this case floating point numbers. The function also expects its calculated output to be of type `float`, which is a common data type for loss functions. In line 10 the indices of all predictions smaller than the actual values that would result in an under-allocation of resources are stored in a `boolean` array. This array is then used in line 11 to update all values that match the indices stored in the array by subtracting the penalty value. After updating all predicted values both tensors are then forwarded to the mean squared error function in line 13, which does the final calculation of the loss. The MSE function then returns the value to the calling function `forward` and this returns the loss value as type `float`.

Chapter 6

Evaluation and Results

6.1 Over-, Under-Allocation of Data

Over-, and under-allocation are terms used to describe if a predicted value for a task is either higher (over-allocated) or lower (under-allocated) than the actual value the task required. Over-, and under-allocated values that are predicted by an LSTM model or the user predictions are referred to as over-estimated or under-estimated values further on. Predicted values are elements of prediction vectors and get compared to the actual values that are elements of a vector containing the actual values. These predicted/actual values correspond to a real task that was deployed on a GPU cluster.

Definition 17: Calculation of Over/Under-Estimated Value

The prediction vector is defined as $\hat{P} = \{p_1, p_2, \ldots, p_n\}$, where $p_i, 1 \leq i \leq n$ is a predicted value of a task. The actual vector is defined as $\hat{A} = \{a_1, a_2, \ldots, a_n\}$, where a_i is the actual value of a task, i.e., the value that was traced while executing the task.

$$p_i = \begin{cases} \text{Over-Allocated}, & \text{if } p_i > a_i \\ \text{Under-Allocated}, & \text{otherwise} \end{cases}$$

Each predicted value is then stored in either a vector that contains over, or underestimated values to determine the distribution for each LSTM variant.

6.2 Trace Data

This section contains the analysis of the Alibaba datasets used in section 6.3. This data analysis builds on top of data analysis done in [19]. Alibaba provides numerous monitoring traces clustered in different data clusters collected in a GitHub repository [47]. For the evaluation of the prediction performance of the LSTM model configurations, the GPU cluster dataset `cluster-trace-gpu-v2020` was chosen. Analysis and characterisation of the GPU cluster data were done in [20]. The data structure of the monitoring traces is shown in figure 6.1, where a job consists of multiple tasks that get deployed on one or more instances depending on the defined number of instances.

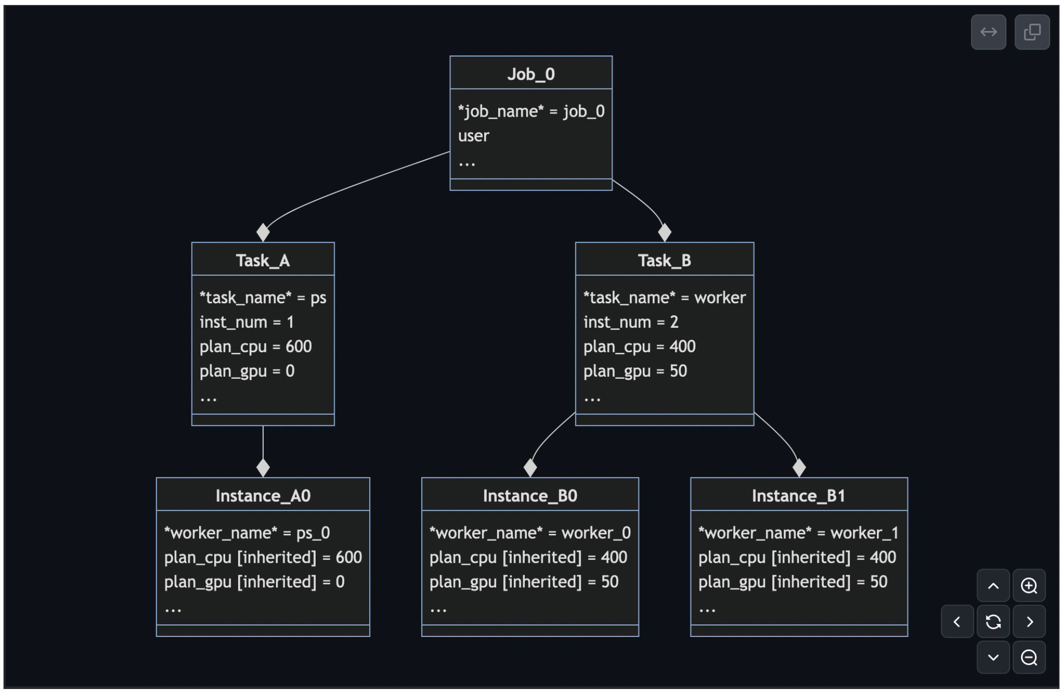


Figure 6.1 GPU Trace Data Structure [3]

6.2.1 CPU Data Analysis

In table 6.1 the actual CPU utilisation of all tasks is shown. The dataset contains 5000 elements and the CPU utilisation is represented in percentage, thus a CPU utilisation of 200 states that a task utilised two full CPU cores on average. The mean of the

Table 6.1 Actual CPU Utilisation

	CPU Utilisation (in %)
mean	516.073
std	881.832
min	1.023
25%	103.632
50%	208.749
75%	528.076
max	7790.371

CPU utilisation in this dataset is 516.073, so on average, a task required 516.073 CPU utilisation, or 5.16 CPU cores for its execution. The standard deviation is approximately 8.8 CPU cores per task, therefore the *average* distance from the mean value is 8.8 CPU core utilisation. In at least one case, CPU cores were utilised with only 1% (see *min* in table), as can be seen in the minimum value of one in the table. The first quartile shows that a full CPU core is used, and the second quartile that 2.08 CPU cores are utilised. Next, the third quartile shows a CPU utilisation of 5.28 CPU cores and finally, the maximum CPU utilisation used is approximately 78 CPU cores. This high CPU utilisation as the maximum value is due to a task being deployed as many instances on multiple devices and the CPU utilisation being summed up for the task.

6.2.2 Memory Data Analysis

In table 6.2 the actual memory utilisation of all tasks is shown. This is the same dataset as is used for the CPU data analysis in section 6.2.1 and contains the corresponding memory utilisation of the tasks. Memory utilisation is represented in gigabytes (GB). Therefore, a memory utilisation of 10 means that a task required 10 GB at maximum for finishing its execution.

Table 6.2 Actual Memory Utilisation

	Memory Utilisation (in GB)
mean	17.203
std	74.761
min	0.003
25%	2.160
50%	7.699
75%	15.976
max	1992.484

The mean memory utilisation in this dataset is approximately 17.2 GB for an average task. The standard deviation is 74.76, thus the distribution of memory utilisation compared to the mean is rather high. The minimum is shown as being about 3 megabytes for a task in the dataset. Up to the first quartile, the memory utilisation reaches 2 GB, for the second quartile 7.7 GB, and in the third quartile a value of approximately 16 GB. The maximum memory utilisation is approximately 1992 gigabytes, and same as for the CPU utilisation, a task might consist of multiple instances, and their memory utilisation is summed up similarly.

6.3 Evaluation Setup

In this section, the evaluation setup will be described in detail. This involves what tools and technologies were used and how they were combined in order to evaluate our test cases. Additionally, the Forecasting Metrics used are briefly described since they are crucial for the forecast prediction performance comparisons.

6.3.1 Hardware

The training of the Long-Short Term Memory model was done on a GPU server provided by ITEC, University of Klagenfurt. The server specifications are as follows:

Processor	Intel Xeon Gold 5218 CPU @ 2.30GHz (64 cores)
GPU	2 x NVIDIA Quadro RTX 8000 GPU (48 GB RAM)
Main Memory	754 GB
Operating System	Ubuntu Linux 18.04 LTS

6.3.2 LSTM Hyperparameter

The hyperparameters of all LSTM variants are stored in `YAML` configuration files. A configuration file used for the Instance LSTM (see 6.4.4) is shown in listing 6.1.

> A hyperparameter is a machine learning parameter whose value is chosen before a learning algorithm is trained. Hyperparameters should not be confused with parameters. In machine learning, the label parameter is used to identify variables whose values are learned during training [48].

The differences for the LSTM variants are found in the values `include_tasks` and `include_instance`. If both values are set to `False`, the LSTM model only uses the capacity of the mapped computing resource and user predictions as seen in the evaluation scenario 6.4.2. For the Task LSTM (see 6.4.3), the value `include_tasks` is set to `True`. For both the Instance LSTM and the Penalty LSTM (see 6.4.5) both the values `include_tasks` and `include_instance` are set to `True` as seen in the mentioned listing. The `loss` value only changes for the Penalty LSTM that uses the penalty loss function (see 5.7) instead of the *mean squared error function*.

All other hyperparameters are identical for all LSTM variants in the evaluation scenarios (see 6.4) to be able to compare the performance impact of feeding additional feature columns to the LSTM models.

```yaml
---
dataset:
  small_df: True
  include_tasks: True
  include_instance: True
  batch_size: 2000

model:
  name: "Utilization LSTM Model with Instances"
  save_model: True
  num_epochs: 100
  learning_rate: 0.0002
  hidden_size: 4000
  num_layers: 2

  scheduler:
    patience: 50
    factor: 0.5
    min_lr: 0.00001
    eps: 0.000000001

  loss: "MSE"

evaluation_path:
```

```
save_to_file: True
loss_progression: "evaluation/all/loss_progression_with_instances.csv"
training_prediction_path: "evaluation/all/
    util_lstm_train_with_instances.csv"
test_prediction_path: "evaluation/all/util_lstm_test_with_instances.csv
    "
```

Listing 6.1 LSTM Model Config File Example

6.3.3 Weights & Biases

Weights & Biases (W&B)[1] is a software company that provides an AI platform for machine learning and deep learning. Their platform provides tools for tracking experiments, visualizing models, and collaborating with team members. The platform provides a centralized repository for all of the information related to a machine learning project, including code, data, models, and results. W&B offers a range of features to help users better understand their models, including interactive visualizations of model architecture, weight distributions, and training metrics. The platform also provides a suite of tools for tracking experiments, which makes it easy to compare different models and understand the impact of changes to the code or data.

6.3.4 Forecasting Metrics

Forecasting metrics [49] are measures used to evaluate the accuracy of forecasting models. These metrics are used to compare different models, assess the quality of the forecasts, and identify areas for improvement.

The choice of forecasting metric will depend on the specific goals and requirements of the forecasting task. Some metrics may be more appropriate for certain types of data or models, while others may be better suited for comparing different models. Overall, the use of appropriate forecasting metrics is critical for evaluating the accuracy of forecasting models and for identifying areas for improvement.

6.3.4.1 Root Mean Square Error (RMSE)

Root Mean Square Error (RMSE) [50] is a metric used for evaluating the accuracy of forecasting models. It measures the average magnitude of the error between the predicted values and the actual values and provides a useful way to compare the magnitude of the errors in different models. The formula for RMSE is:

Definition 18: Root Mean Square Error

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(Predicted_i - Actual_i)^2}{N}}$$

[1]https://wandb.ai

where n denotes the number of data points and *Actual* and *Predicted* are the actual and predicted values, respectively. RMSE provides a measure of the magnitude of the errors in the predictions, with lower values indicating a more accurate model. RMSE is particularly useful when the goal is to minimize the magnitude of the errors in the predictions. One important thing to note about RMSE is that it is sensitive to the scale of the data. This means that it is more appropriate to use RMSE when the scale of the actual and predicted values is similar. Because of the scale sensitivity of RMSE, the number of actual and predicted values is equivalent for all our evaluation scenarios in order to be able to use RMSE for a comparison between different approaches. Overall, RMSE is a widely used and useful metric for evaluating the accuracy of forecasting models and can provide valuable information for understanding the magnitude of the errors in the predictions.

6.3.4.2 Mean Absolute Percentage Error (MAPE)

Mean Absolute Percentage Error (MAPE) [51] is a commonly used metric for evaluating the accuracy of forecasting models. It measures the average percentage difference between the predicted values and the actual values. The formula for MAPE is:

Definition 19: Mean Absolute Percentage Error

$$MAPE = \frac{1}{n} \times \sum \left| \frac{Actual - Predicted}{Actual} \right| \times 100$$

where n denotes the number of data points and *Actual* and *Predicted* are the actual and predicted values, respectively. MAPE provides a percentage error, which makes it easy to interpret and compare the accuracy of forecasting models.

However, there are some limitations to using MAPE, such as the fact that it can become undefined when the actual value is zero (in our case if there is no resource utilisation at some time step i), and it can be sensitive to outliers. Despite these limitations, MAPE is a widely used metric for evaluating forecasting models, especially if it is important to understand the relative magnitude of the errors in the predictions. Overall, MAPE provides a useful way to measure the accuracy of forecasting models and can help to identify areas for improvement in the model or the data. One drawback of using MAPE is its asymmetric nature, which results in it being biased towards higher values. This drawback is the reason the variant Symmetric Mean Absolute Percentage Error (sMAPE) is also used for comparing the predictions.

6.3.4.3 Symmetric Mean Absolute Percentage Error (sMAPE)

Symmetric Mean Absolute Percentage Error (sMAPE) [52] is a metric used for evaluating the accuracy of forecasting models. It measures the average percentage difference between the predicted values and the actual values and is symmetrical in that it treats positive and negative errors equally. In our evaluation of the prediction performance, this is especially useful since both over and under-utilisation are present in all prediction variants. The formula for sMAPE is:

Definition 20: Symmetric Mean Absolute Percentage Error

$$sMAPE = \frac{1}{n} \times \sum \left| \frac{Actual - Predicted}{(Actual + Predicted) \div 2} \right| \times 100$$

where n denotes the number of data points and *Actual* and *Predicted* are the actual and predicted values, respectively. sMAPE provides a percentage error similar to Mean Absolute Percentage Error (MAPE), which makes it easy to interpret and compare the accuracy of forecasting models. Unlike MAPE, sMAPE is symmetrical, since it doesn't become undefined when the actual value is zero, and another difference is that it is less sensitive to outliers. Overall, sMAPE is a useful metric for evaluating the accuracy of forecasting models and can provide valuable information for understanding the relative magnitude of the errors in the predictions. While sMAPE addresses the asymmetric nature and drawback of MAPE, it simultaneously creates another asymmetric value. The cause of this is the denominator while overestimating and underestimating. If the predictor underestimates, the sMAPE metric will be higher compared to overestimation, because the variable *Predicted* raises the difference in both cases.

Therefore, both the MAPE and the sMAPE metrics are used in the Evaluation and Results chapter to compare the different prediction variants.

6.3.4.4 Resource Wastage

The resource wastage for a single task is defined as the difference between the predicted and the actual value. Since wastage refers to utilising more resources than necessary, only predicted values higher than the actual values are considered for this calculation.

6.4 Evaluation Scenarios

In evaluation scenarios 1-4, the forecast performance of different machine learning training scenarios is evaluated and compared with each other. Each of them builds on top of the previous evaluation scenario to compare the improvement.

6.4.1 User-Defined Hardware Utilisation

This evaluation scenario analysis the user-defined hardware utilisation. In this variant, users are required to estimate the hardware utilisation of their tasks when providing those tasks to the GPU clusters. This user-defined prediction is the comparison base for improvement in the evaluation scenarios that will follow.

6.4.1.1 CPU Prediction Analysis

In this section, the user-defined prediction regarding CPU utilisation is analysed and compared with the actual CPU utilisation (see table 6.3). The CPU values are in percentage similar to section 6.2. As can be seen in the *mean* row, the user-predicted CPU utilisation is approximately 116 % or 1 CPU core over-allocated. Also, the first quartile in the user-predicted column already allocates four CPU cores while the required amount is close to one CPU core, which results in a CPU resource wastage of three CPU cores. In the second and third quartiles, the user-predicted allocation was 600 % in both cases, yet in the second quartile, the required amount is about 2.88 times less than allocated. In the third quartile, about half a CPU core was allocated but idle. As stated in [4], users over-estimate the actual requirements of their tasks in most cases. Additionally, while the general trend is to over-estimate the required amount of CPU cores, the estimation of the maximum value in the actual CPU dataset with the utilisation of 7790.37% or 78 CPU cores could not be predicted successfully, where the maximum in the user-predicted dataset only reached a value of 6400%, or 64 CPU cores. In table 6.4

Table 6.3 User Predicted CPU - Actual CPU Comparison

	User Predicted CPU	Actual CPU
mean	632.81	516.07
std	496.24	881.83
min	5.00	1.02
25%	400.00	103.63
50%	600.00	208.75
75%	600.00	528.08
max	6400.00	7790.37

the different metrics used for the comparison of the actual and predicted data are shown. This table is in the following evaluation scenarios used to compare the performance of the LSTM model to user-predicted values. The metrics Root Mean Square Error (RMSE), Mean Absolute Percentage Error (MAPE) and Symmetric Mean Absolute Percentage Error (sMAPE) are used for the performance comparison. Also included in the table are the allocation distribution columns for *Over-Allocation (OA)* and *Under-Allocation (UA)* calculated by comparing each predicted value with its corresponding actual value and if the predicted value is higher, it is added to the over-allocated section and if the value is smaller then it is added to the under-allocated section. As seen in the *OA* and *UA* columns, the CPU over-allocation is at almost 73%.

Table 6.4 Metrics for User Predicted CPU Allocation

	RMSE	MAPE	sMAPE	OA	UA
User CPU	812.497	355.049	89.466	72.8	27.2

6.4.1.2 Memory Prediction Analysis

In this section, the user-defined prediction regarding memory utilisation is analysed and compared with the actual memory utilisation (see table 6.5). The memory values are shown in gigabytes (GB), similar to section 6.2.2. The *mean* in the user-predicted column shows that the required memory was overestimated by a factor of 1.56 or 9.69

GB. The standard deviation (std) of the *mean* was lower in the user-predicted dataset by approximately 60 GB. This difference can be seen as users are more likely to predict similar values for their tasks, as can be seen in the dataset with very similar user-predicted values, which result in a lower standard deviation. At the minimum amount, users predicted to require at least 2 GB of memory, while the actual minimum was 0 MB. While it is necessary to provide an actual value to the GPU cluster, the user-provided prediction still heavily overestimated the actual utilisation. In the quartiles, the trend to over-estimate the memory requirements continues whereas, in the first quartile, the factor of over-estimation is 6.78 or 12.5 GB. The amount of memory allocation is identical in both the second and third quartiles with 29.3 GB. The factor of over-estimation for the second quartile is 3.8 or 21.6 GB and in the third quartile, the factor is 1.81 or 13.3 GB. Similar to the CPU utilisation, the memory utilisation predicted by the user is less than half as much as is required for the maximum value. In the dataset of actual memory utilisation, the maximum requires approximately 1992 GB, yet the user predicted a memory utilisation of 146.48 GB, which is 13.6 times less than was required.

Table 6.5 User Predicted Memory - Actual Memory Comparison

	User Predicted Memory	Actual Memory
mean	26.89	17.20
std	15.26	74.76
min	2.00	0.00
25%	14.65	2.16
50%	29.30	7.70
75%	29.30	15.98
max	146.48	1992.48

The metrics of the table 6.6 for calculating the deviation from the actual values are the same as for the CPU utilisation prediction in table 6.4. Similar to the CPU prediction, the over-allocation is predominant in the user-predicted memory allocation with 80.8%. This table is used to compare the memory predictions of the different LSTM model configurations.

Table 6.6 Metric for User Predicted Memory Allocation

	RMSE	MAPE	sMAPE	OA	UA
User Memory	73.853	3672.256	97.613	80.8	19.2

6.4.2 Simple LSTM Model

The simple LSTM model denotes a Long-Short Term Memory model that was trained with a feature set that only contains the allocated CPU and memory a user provided for the submitted task as well as the capacity of the worker pool resource the task should be deployed to. Additionally, since LSTMs are designed to handle sequential data, the order of the incoming tasks is also a piece of important information for the model, and thus cannot be randomized as is common for other machine learning algorithms.

6.4.2.1 CPU Prediction Analysis

In this section, the CPU prediction of the simple LSTM model and the user-provided prediction are compared in tables 6.7 and 6.8. The *mean* of the predictions the LSTM model provided is lower by approximately one and the user prediction is higher by one CPU core than the actual *mean* of 5.16 CPU cores. In regards to the *standard deviation*, the LSTM prediction with 705.771 is closer to the actual value of 881.832 than the value of the user prediction with 496.245. This indicates that the LSTM model is able to better predict changes in task utilisation than users that provide the tasks. The *minimum* of the LSTM model with 3.03 is also closer to the actual value with 1.023 than the user-predicted minimum value of 5. Similarly, the first and second quartile in the LSTM prediction dataset is closer to the actual data values, yet the LSTM model is likely to predict values lower than the actual value. As already signalled by a lower *mean* value, the second and third quartiles of the LSTM predictions both are lower than the similar quartiles of the actual dataset. In the third quartile, the user predictions of the CPU utilisation outperform the LSTM model, since these predictions only overestimate the CPU utilisation by 72% of a CPU core, while the LSTM model underestimates the utilisation by approximately 2.5 CPU cores. This trend continues in the predicted maximum value. The LSTM model value is even lower than the value provided by the user, and therefore even smaller than the actual value. A reason for the underestimation of values might be that the actual dataset consists of a large amount of *low* utilisation values, as indicated by the second quartile. Since this LSTM model version is only provided with the user prediction of the tasks, the GPU worker pool capacity, and the actual CPU utilisation (after training in the loss function), it is likely that based on the limited information, the model learned to predict lower CPU utilisation values, since most of the actual data points are lower than the *mean*. In table 6.8 the predictions of

Table 6.7 Comparison of Simple LSTM and User Predicted CPU

	Actual CPU	Simple LSTM CPU	User Predicted CPU
mean	516.073	392.630	632.809
std	881.832	705.771	496.245
min	1.023	3.030	5.000
25%	103.632	97.586	400.000
50%	208.749	118.014	600.000
75%	528.076	281.472	600.000
max	7790.372	5793.996	6400.000

the LSTM model and the users are compared with commonly used regression metrics. A lower value denotes a better result by being closer to the actual values. The LSTM model did perform better in all three used regression metrics. Also, as indicated by a lower *mean* value, the LSTM model is more likely to predict values lower than the actual value, which results in under-utilisation.

Table 6.8 Regression Metrics - Simple LSTM and User Predicted CPU

	RMSE	MAPE	sMAPE	OA	UA
Simple LSTM CPU	797.289	206.062	85.626	41.94	58.06
User Predicted CPU	812.497	355.049	89.466	72.80	27.20

6.4.2.2 Memory Prediction Analysis

In this section, the memory prediction of the simple LSTM model and the user-provided prediction are compared in tables 6.9 and 6.10. As can be seen in table 6.9, the LSTM model did perform worse in almost all categories except the *standard deviation, minimum* and *maximum*. Similarly to the CPU prediction, the lack of information is a likely reason why the prediction performance regarding memory utilisation did not achieve good results. The second and third quartiles are identical both in the user predictions and the LSTM predictions and could signal that the LSTM model relies heavily on the user predictions but is not able to improve upon them with the limited information available. The comparatively worse performance of the LSTM model is also reflected in

Table 6.9 Comparison of Simple LSTM and User Predicted Memory

	Actual Memory	Simple LSTM Memory	User Predicted Memory
mean	17.203	29.904	26.895
std	74.761	39.634	15.259
min	0.003	1.951	2.000
25%	2.160	22.178	14.648
50%	7.699	24.620	29.297
75%	15.976	24.620	29.297
max	1992.484	550.056	146.484

table 6.10, where the LSTM predictions have higher loss values than the user predictions for all regression metrics. The over-, and underestimation of the actual values are similar for the LSTM and the user-based predictions, which is another indicator that the LSTM model heavily relied on the information provided by the user predictions.

Table 6.10 Regression Metrics - Simple LSTM and User Predicted Memory

	RMSE	MAPE	sMAPE	OA	UA
Simple LSTM	77.902	7711.391	109.870	77.8	22.2
User Predicted	73.853	3672.256	97.613	80.8	19.2

6.4.3 Adding Task Knowledge

In this evaluation scenario, the feature set used to train the LSTM model contains additionally to the feature set arguments of the Simple LSTM Model also *task knowledge*. This task knowledge refers to the type a task is classified as. The task knowledge was first extracted from the data set and transformed with the One-hot Encoding method. The one-hot encoding method is useful in general if categorical data needs to be provided to machine learning models. The tasks are mapped to a finite set of a one-hot encoded feature vector and their corresponding index is represented as the value 1. The additional information about the task type enables the LSTM model to better discern the actual utilisation that is required for each task. The task knowledge also provides information regarding the order of tasks, since patterns of reoccurring tasks also help the model to generate more accurate predictions in theory. This LSTM model is referred to as **Task LSTM** model in the following analysis sections.

6.4.3.1 CPU Prediction Analysis

In this section, the CPU prediction of the Task LSTM model and the user-provided prediction are compared in tables 6.11 and 6.12. As can be seen in table 6.11, the *mean* of the LSTM model that additionally receives information about the task types is closer to the actual *mean* and is the closest of the currently presented prediction methods. The *standard deviation* of the Task LSTM is smaller than the value of the Simple LSTM but closer than the value of the user prediction. The predicted *minimum* value with 2.395 is closer to the actual value of approximately 1 than both the Simple LSTM and user prediction. Also, the three quartiles of the Task LSTM variant combined are the closest to the actual quartiles, which indicates that the model managed to learn the distribution of the dataset the best of the currently presented predictions. The *maximum* value of the Task LSTM prediction is the farthest from the actual value, which could be a result of the lower *standard deviation* of this variant, and also that providing the task knowledge is not sufficient to predict utilisation spikes in the GPU cluster.

Table 6.11 Comparison of LSTM Variants and User Predicted CPU

	Actual CPU	Task LSTM CPU	Simple LSTM CPU	User CPU
mean	516.073	454.205	392.630	632.809
std	881.832	579.213	705.771	496.245
min	1.023	2.395	3.030	5.000
25%	103.632	129.884	97.586	400.000
50%	208.749	249.392	118.014	600.000
75%	528.076	662.490	281.472	600.000
max	7790.371	5634.635	5793.996	6400.000

In table 6.12, the two introduced LSTM variants are compared as well as the user-predicted values for CPU. The Task LSTM did perform better than the Simple LSTM and user predictions regarding the RMSE metric, yet only performed better than the user-predicted values regarding the MAPE metric and worse by approximately 6.8 than the Simple LSTM, which is similar to an error increase of 7%. An explanation for the worse performance is the distribution of over-, and underestimated values in the datasets. As discussed in section 6.3.4.2, the MAPE metric is biased towards higher values, therefore, overestimated values result in a higher overall loss. The Task LSTM did perform better than both the Simple LSTM and the user predictions regarding the sMAPE metric. Also the distribution of over-, and underestimated values is close to equal for the Task LSTM predictions.

Table 6.12 Regression Metrics - Task LSTM and User Predicted CPU

	RMSE	MAPE	sMAPE	OA	UA
Task LSTM	688.089	212.796	83.017	48.14	51.86
Simple LSTM	797.289	206.062	85.626	41.94	58.06
User Predicted	812.497	355.049	89.466	72.80	27.20

Therefore, providing categorical data can positively influence the prediction performance of the machine learning model. This is also seen in the predictions of the Task LSTM model. The additional knowledge about what type of tasks are present in the pipeline enabled the ML model to create more accurate predictions of the CPU utilisation for each task.

6.4.3.2 Memory Prediction Analysis

In this section, the memory prediction of the Task LSTM model and the user-provided prediction are compared in tables 6.13 and 6.14.

The *mean* memory utilisation prediction of both the LSTM variants performed slightly worse than the prediction provided by users. All three prediction variants were approximately $9 - 12$ GB overestimated on average. For the *standard deviation* the Task LSTM prediction did achieve the closest result, which indicates that it is capable of adapting its prediction more actively than the other two prediction variants. Similarly, the Task LSTM model did achieve to predict a *minimum* value close to the actual *minimum* as opposed to the other two variants. In the first and second quartiles, the Task LSTM also performed more accurately than both the Simple LSTM and the user predictions but did perform worse than the Simple LSTM in the third quartile. The actual *maximum* could be predicted the best with the Task LSTM, yet the estimation is still approximately 3 times too small.

As seen in table 6.14 for the RMSE metric the Task LSTM did perform the best of the three variants. Yet for both the *MAPE* and *sMAPE* metrics, it did perform worse. An explanation for the high *MAPE* value of the Task LSTM is the high *standard deviation* in combination with the too high *mean* value. The Task LSTM, while being more proactive in predicting utilisation spikes also is more likely to predict higher utilisation. Predicting utilisation spikes when none do occur does result in such high loss values of the *MAPE* metric.

Table 6.13 Comparison of LSTM Variants and User Predicted Memory

	Actual MEM	Task LSTM MEM	Simple LSTM MEM	User MEM
mean	17.203	29.134	29.904	26.895
std	74.761	63.342	39.634	15.259
min	0.003	0.156	1.951	2.000
25%	2.160	4.537	22.178	14.648
50%	7.699	14.679	24.620	29.297
75%	15.976	27.924	24.620	29.297
max	1992.484	698.983	550.056	146.484

Table 6.14 Regression Metrics - Task LSTM and User Predicted Memory

	RMSE	MAPE	sMAPE	OA	UA
Task LSTM	61.897	32287.182	119.715	56.66	43.34
Simple LSTM	77.902	7711.391	109.870	77.8	22.2
User Predicted	73.853	3672.256	97.613	80.80	19.20

6.4.4 Adding Instance Knowledge

In this evaluation scenario, the feature set used to train the LSTM model contains additionally to the feature set arguments of the Adding Task Knowledge also information about the number of instances required to be deployed. Further on, the model trained with the additional instance knowledge is referred to as **Instance LSTM**. While the knowledge about the number of instances is similarity one-hot encoded as is the task knowledge (see 6.4.3), adding one feature vector column for every instance number would have resulted in exploding the feature data set with 800 additional columns since the minimum instance number is 1 and the maximum is 800. For this reason, the feature vector for the one-hot encoded instances was first clustered into 10 buckets in 50 iterations by k-means clustering [53]. In the following prediction analysis sections 6.4.4.1, 6.4.4.2, the Instance LSTM model is compared to the Task LSTM model of evaluation scenario 6.4.3 and the user predictions (see 6.4.1.1).

6.4.4.1 CPU Prediction Analysis

In this section the CPU prediction of the Instance LSTM is compared to the prediction variants mentioned in section 6.4.4. For the comparison, the tables 6.15 and 6.16 are used. As can be seen in the first table, the *mean* value of the Instance LSTM with 571.784 is the closest to the actual *mean* value of 516.073. Also the *standard deviation* of the Instance LSTM model is the closest to the actual value. This signals that the Instance model learned to react to CPU spikes in the dataset. Yet, when comparing the *minimum* values, the first, second and third quartiles, the Instance LSTM has a worse performance than the Task LSTM and overestimated the actual values in the third quartile even more than the user-predicted variant. When predicting the *maximum*, the

Instance LSTM model did perform better than any LSTM model presented before, and also better than the user-predicted variant. The provided data in table 6.15 indicates that the Instance LSTM model is capable of predicting the *mean* value of the data, but also tends to overestimate the actual utilisation in order to account for CPU spikes.

Table 6.15 Comparison of Instance LSTM and User Predicted CPU

	Actual CPU	Instance LSTM CPU	Task LSTM CPU	User CPU
mean	516.073	571.784	454.205	632.809
std	881.832	690.524	579.213	496.245
min	1.023	4.240	2.395	5.000
25%	103.632	215.433	129.884	400.000
50%	208.749	434.128	249.392	600.000
75%	528.076	821.406	662.490	600.000
max	7790.371	6954.110	5634.635	6400.000

As seen in table 6.16, the Instance LSTM has the lowest *RMSE* value. The *MAPE* and *sMAPE* both had a higher loss for the Instance LSTM than the Task LSTM variant but performed better than the user-predicted variant. Compared to the Task LSTM model it is more likely to overestimate the actual value. The likely reason is that the Instance LSTM provided additional information about the number of instances of a task that should be deployed. This additional information lets the model predict a CPU utilisation higher than the actual value since the tasks are denoted to be deployed on multiple resources.

Table 6.16 Regression Metrics - Instance LSTM and User Predicted CPU

	RMSE	MAPE	sMAPE	OA	UA
Instance LSTM	614.645	294.449	87.975	57.04	42.96
Task LSTM	688.089	212.796	83.017	48.14	51.86
User Predicted	812.497	355.049	89.466	72.80	27.20

While the Instance LSTM model prediction performance does look promising in terms of *mean* and *standard deviation* (see 6.15), it did not do perform well in other mentioned metrics. This LSTM variant is successful in terms of predicting CPU spikes more accurately than any other presented prediction, yet this also results in predicting CPU utilisation spikes in tasks that do not require such a vast amount of CPU resources, even if deployed on multiple resources.

6.4.4.2 Memory Prediction Analysis

In this section the memory prediction of the Instance LSTM is compared to the prediction variants mentioned in section 6.4.4. As seen in table 6.17, the Instance LSTM model did overestimate on average by a factor of 2.5 compared to the actual *mean*. The *standard deviation* is closer to the actual value than the user-predicted variant, yet the estimation performed worse than the one of the Task LSTM model. The *minimum* value could be predicted the best with the Instance LSTM model, similarly the *maximum* value. This indicates that this LSTM variant is sensitive to utilisation spikes. Yet, the performance in all three quartiles is worse than both the prediction of the Task LSTM and the user predictions, which is not surprising given the high *mean* value.

Table 6.17 Comparison of Instance LSTM and User Predicted Memory

	Actual MEM	Instance LSTM MEM	Task LSTM MEM	User MEM
mean	17.203	43.098	29.134	26.895
std	74.761	56.496	63.342	15.259
min	0.003	0.055	0.156	2.000
25%	2.160	22.294	4.537	14.648
50%	7.699	27.067	14.679	29.297
75%	15.976	62.938	27.924	29.297
max	1992.484	1208.922	698.983	146.484

As seen in table 6.18, the Instance LSTM was performing the best regarding *RMSE* loss and also performed better than the Task LSTM regarding *MAPE* but performed worse than the Task LSTM regarding *sMAPE*. Approximately 70% of the predicted values were overestimated by the Instance LSTM instance, which is more than all other variants currently presented.

Table 6.18 Regression Metrics - Instance LSTM and User Predicted Memory

	RMSE	MAPE	sMAPE	OA	UA
Instance LSTM	60.415	16855.899	131.774	69.9	30.1
Task LSTM	61.897	32287.182	119.715	56.66	43.34
User Predicted	73.853	3672.256	97.613	80.8	19.2

While the Instance LSTM did have good prediction performance regarding CPU utilisation, the prediction of memory utilisation is viable for detecting memory utilisation spikes but did otherwise overestimate the majority of values. The reason for the *MAPE* loss value being smaller than the value of the Task LSTM is, that the memory utilisation spikes were not predicted by the Task LSTM, hence the bad prediction performance.

6.4.5 Training with Penalty Mean Squared Error Function

As discussed in the previous evaluation scenarios, additional information regarding a task (see section 6.4.3) will result in closer predictions to the actual value but also the LSTM model is more likely to predict values that are smaller than the actual value. Since underestimating the actual utilisation of both CPU and memory resources can lead to erroneous behaviour while executing tasks, it is preferable to overestimate. Therefore, a custom loss function was designed and implemented (see Penalty Mean Squared Error Loss Function) to counter the tendency of the LSTM model with an increasing number of feature columns to predict values lower than the actual value. This LSTM variant that is using said custom loss function is from now on referred to as *Penalty LSTM*.

6.4.5.1 CPU Prediction Analysis

In this section, the CPU prediction of the Penalty LSTM is compared to the prediction variants mentioned in section 6.4.4.1. The predicted *mean* value of the Penalty LSTM variant is performing worse by a value of approximately 20 than the Instance LSTM model. Also, the distance of the *standard deviation* value is larger for the Penalty LSTM model than the Instance LSTM but performs better than the user predictions. In the first and second quartiles, the Penalty LSTM did perform worse than the Instance LSTM, but slightly better for the third quartile. The predicted *maximum* value is the worst of the compared prediction variants.

Table 6.19 Comparison of Penalty, Instance LSTM and User Predicted CPU

	Actual CPU	Penalty LSTM CPU	Instance LSTM CPU	User CPU
mean	516.073	595.122	571.784	632.809
std	881.832	550.389	690.524	496.245
min	1.023	4.481	4.240	5.000
25%	103.632	248.302	215.433	400.000
50%	208.749	571.664	434.128	600.000
75%	528.076	808.049	821.406	600.000
max	7790.371	5776.425	6954.110	6400.000

As seen in table 6.20, the Penalty LSTM did perform slightly worse than the Instance LSTM regarding the *RMSE* loss value but performed better for both the *MAPE* and *sMAPE* metrics. Also, the overallocation of the Penalty LSTM is 70%, so an increase of 13% when using the PMSE loss function compared to the Instance LSTM variant.

Table 6.20 Regression Metrics - Penalty LSTM and User Predicted CPU

	RMSE	MAPE	sMAPE	OA	UA
Penalty LSTM	637.124	275.275	82.988	70.16	29.84
Instance LSTM	614.645	294.449	87.975	57.04	42.96
User Predicted	812.497	355.049	89.466	72.80	27.20

Given the metrics and the preferably overestimated values of table 6.20, this LSTM variant is preferable to the other prediction variants for CPU utilisation prediction.

6.4.5.2 Memory Prediction Analysis

In this section, the memory prediction of the Penalty LSTM is compared to the prediction variants mentioned in section 6.4.4.2. The Penalty LSTM variant performed the worst for predicting memory utilisation in all categories except the *maximum* value as seen in table 6.21. This is not surprising given the PMSE loss function did further shift the already overestimating tendency of the memory section into penalising the remaining underestimated values.

Table 6.21 Comparison of Penalty, Instance LSTM and User Predicted Memory

	Actual MEM	Penalty LSTM MEM	Instance LSTM MEM	User MEM
mean	17.203	65.128	43.098	26.895
std	74.761	64.252	56.496	15.259
min	0.003	1.639	0.055	2.000
25%	2.160	19.961	22.294	14.648
50%	7.699	60.018	27.067	29.297
75%	15.976	85.906	62.938	29.297
max	1992.484	1251.511	1208.922	146.484

Similarly to the results in table 6.21, the results of the metrics for the Penalty LSTM in table 6.22 show that this LSTM variant did perform the worst for predicting memory utilisation of tasks.

Table 6.22 Regression Metrics - Penalty LSTM and User Predicted Memory

	RMSE	MAPE	sMAPE	OA	UA
Penalty LSTM	77.541	27603.612	133.192	87.38	12.62
Instance LSTM	60.415	16855.899	131.774	69.9	30.1
User Predicted	73.853	3672.256	97.613	80.80	19.20

While the actual CPU utilisation did gain some performance gains using the PMSE loss function, it simultaneously decreased the prediction performance for the actual memory utilisation.

6.4.6 Training Time

The evaluation scenario regarding the training time measures the time that is required for the LSTM model to train for one epoch. Each LSTM variant shown in table 6.23 was trained with 100 epochs and the number of data elements was 5000 for each of them. The data points only differed in the dimensions, i.e., the number of feature columns.

Table 6.23 Training Time Comparison of LSTM Variants

Seconds/Epoch	Simple LSTM	Task LSTM	Instance LSTM	Penalty LSTM
mean	2.3487	2.3853	2.3042	2.4001
std	0.0729	0.0949	0.0621	0.0922
min	2.2285	2.1869	2.1697	2.2086
max	2.5651	2.6898	2.5993	2.8930

As seen by the *mean* value, each of the LSTM variants had a similar training time average ranging from approximately $2.3-2.4$ seconds. Given that each model was trained with 100 epochs, this results in an average training time difference of 100 seconds between the fastest (Instance LSTM) and the slowest (Penalty LSTM) variant.

6.4.7 Inference Time

The evaluation scenario regarding the inference time measures the time that is required for the LSTM model to forward a prediction. The time measurement is done on different batch sizes. This is important to be able to forward the resource utilisation prediction to the scheduler in an acceptable time frame. A low inference time by the LSTM model is mandatory in order for its predictions to be usable by the scheduler of the Scheduling and Adaptation approach. Too high inference times can lead to task pipeline congestion if the scheduler has to wait for tasks to arrive from the LSTM model. The required inference time depends on three major factors, the batch size, i.e., the number of tasks provided at once to the LSTM model to calculate a prediction of their hardware utilisation.

Table 6.24 Inference Times of LSTM Variants

	Simple LSTM	Task LSTM	Instance LSTM	Penalty LSTM
mean	0.065724	0.066120	0.067292	0.067271
std	0.034361	0.033345	0.033334	0.034054
min	0.009691	0.009126	0.016284	0.010067
max	0.114571	0.114148	0.120081	0.114867

The next factor is the complexity of the LSTM model, where a more complex neural network architecture also requires more internal calculations while forwarding the input data (the tasks of the pipeline). The complexity of the LSTM model can be derived from the number of layers and LSTM cells as well as used hyper-parameters, most importantly the *hidden size* of the LSTM layers, which has the most impact both on the internal complexity and the amount of space the LSTM model requires. The *hidden size* in all LSTM variants is the same, and this is the reason the inference time seen in figure 6.2 for all models is close to each other.

Next, the hardware the LSTM model resides on has a major influence on the time that is required to infer a prediction from input data. Deep learning nowadays heavily relies on the usage of modern GPUs that use so-called *tensor cores* specifically build for machine learning tasks to accelerate both the training and the inference times.

As can be seen in figure 6.2 and table 6.24 the inference times of all LSTM variants are similar regarding all batch sizes except for the Instance LSTM variant (see 6.4.4). The Instance LSTM is the worst-performing variant in both the *minimum* and *maximum* values with the *minimum* being 1.78 worse than the best-performing (Task LSTM) variant. This is shown in the x-axis for the smallest batch size of 100 in figure 6.2 by the noticeably higher line for the Instance LSTM. Similarly, the Instance LSTM has worse inference time performance for higher batch sizes as seen in the rightmost area of the mentioned figure by the line spike corresponding to the model. An explanation of

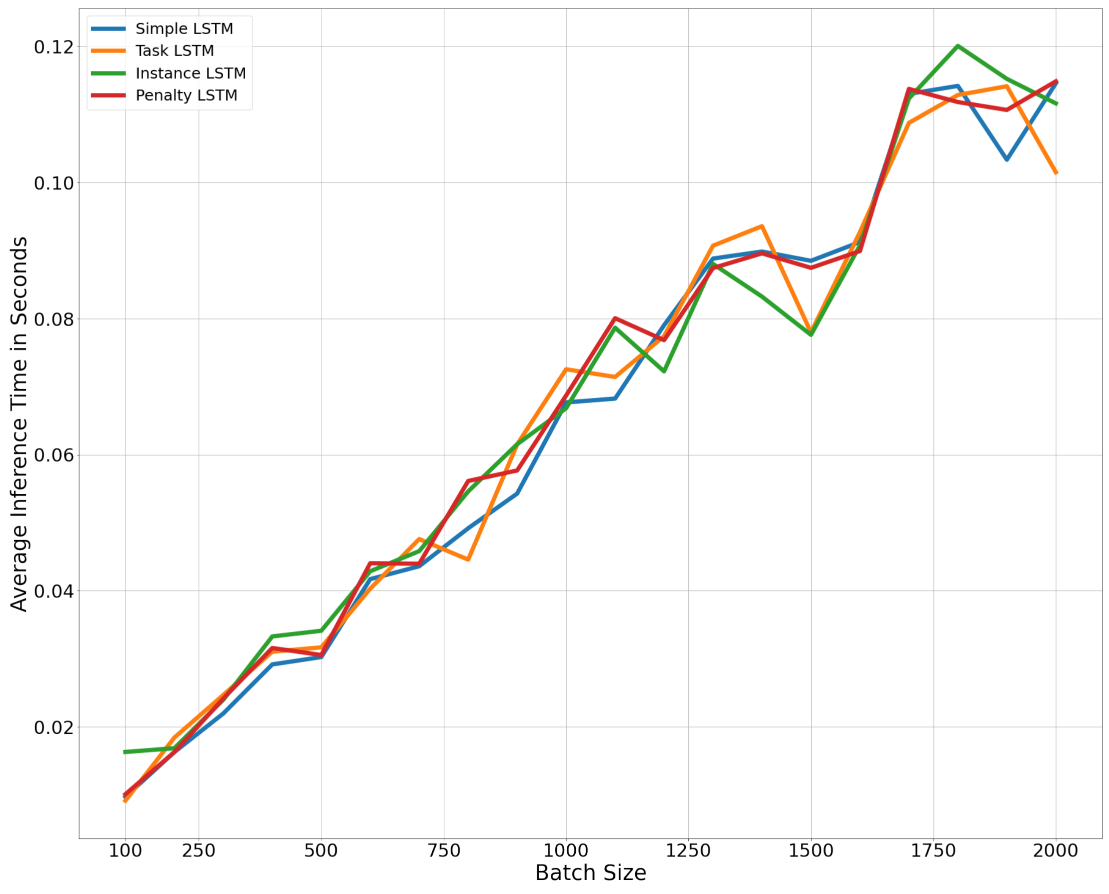


Figure 6.2 Average Inference Time per Batch Size

why the Instance LSTM might be the worst-performing model might be the additional complexity of the feature dataset that is forwarded to the model, yet the Penalty LSTM is fed the same feature dataset and only differs in the loss function that is used to train the model.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this master's thesis, I provided an approach for solving resource utilisation in large-scale computing clusters. This was done by introducing a new machine learning approach based on Long-Short-Term Memory (LSTM) neural networks that I fully developed. The evaluation scenarios that were used to analyse the performance of the LSTM models were tested with real data provided by Alibaba. The source of this dataset are monitoring traces of a GPU cluster of approximately 1800 computing devices. The required utilisation for each task was estimated for both CPU and memory requirements.

The acquired inference results were compared with a dataset that contained the actual resource utilisation of CPU and memory for each task and each LSTM model variant. Additionally, the LSTM results were also compared with the user-predicted values for each task which were provided to the Alibaba GPU cluster for the task to be deployed with the estimated allocation values.

Regarding the evaluation scenarios, the Task LSTM (see 6.4.3) did perform the best for predicting the CPU utilisation for the general use case. Yet, it could not predict CPU utilisation spikes, and the model variants Instance LSTM and Penalty LSTM did perform better when predicting those spikes. The Instance LSTM also had better prediction accuracy regarding memory utilisation compared to the Task LSTM. Both variants did perform better than the user predictions regarding the RMSE metric, yet performed worse for *MAPE* and slightly worse for *sMAPE*.

The custom loss function *Penalty Mean Squared Error (PMSE)* did perform better in predicting utilisation spikes, yet it also is likely to over-estimate the resource utilisation, which results in a worse performance than the user predictions for memory utilisation also had the better prediction performance for CPU utilisation regarding the metrics *MAPE* and *sMAPE* than the Instance LSTM variant. The improvements are promising and further evaluations regarding this loss function or a new variant (see section **??**) should be done in future work.

For the Root Mean Square Error (RMSE) metric, the LSTM variants performed $2\% - 27\%$ better than the user predictions for CPU utilisation, and for memory utilisation the LSTM variants Task and Instance LSTM performed $17\% - 20\%$ better and the other variants slightly worse with 5% than the user predictions. For the Mean Absolute Percentage Error (MAPE), the LSTM variants performed $19\% - 53\%$ better for CPU utilisation but worse for the memory utilisation by $70\% - 160\%$ compared to the user predictions. For the Symmetric Mean Absolute Percentage Error (sMAPE), the LSTM variants did perform $1.6 - 7.5\%$ better for CPU utilisation but slightly worse for memory utilisation by $12\% - 31\%$ than the user predictions. Therefore, there is potential for improvement regarding memory utilisation.

Overall, the LSTM models did perform worse when predicting memory utilisation the more information is fed as a feature set to them and the more complex they have become. Additionally, not one LSTM variant did outperform the other LSTM model variants in all regression metrics. Each had a specific strength, which will require additional research to generalise the training in a manner to result in an LSTM model that is suitable for most task types. The prediction improvements for CPU utilisation did improve regarding every regression metric, yet the predictions for memory utilisation were still outperformed in many scenarios by the user predictions. The improvements in CPU utilisation predictions indicate that similar improvements are possible for memory utilisation, even if they have yet not been achieved.

7.2 Future Work

This section contains possible future work on how the contents of this thesis would be extended in the future. These extensions cover topics such as energy consumption and CO_2 emissions, implementation details, comparisons with other datasets, comparisons with other machine learning architectures, improving the PMSE loss function and using additional machine learning components to improve the prediction performance of the LSTM variants covered in this thesis.

In the current state of the implementation, the energy consumption is not taken into account. Energy consumption and the resulting CO_2 emissions of computing devices have become important topics in recent years.

While the components of the Adaptation Loop described in section 5.1.3 work independently and to some degree can be used with one another, the full integration of the Adaptation Loop is yet not fully implemented nor tested. In the following sections, missing integration steps are explained, as to why they need to be implemented in order for the system to be functional.

While the LSTM model that used the PMSE loss function did gain performance gains over other variants in some scenarios, it also was likely to overestimate the actual resource utilisation for both CPU and memory usage. A well-suited penalty value could improve prediction accuracy.

The results of the LSTM variants are promising for the chosen GPU cluster traces. Yet, these cluster traces only contain various machine-learning tasks. Alibaba provides access to other monitoring traces, each with its specific focus, such as micro-services. Similarly, Google provides monitoring traces of their clusters, thus one possible evaluation scenario for future work is the comparison of a GPU cluster dataset by Google to see how well the LSTM variants perform on these cluster traces.

While researching possible resource utilisation machine learning approaches, other model architectures were also considered. One very promising machine learning architecture is called *Graph Neural Networks (GNN)*. GNNs are capable of doing inference on an arbitrary number of tasks (similar to LSTMs), yet one major advantage is their characteristic of not requiring the tasks to be ordered.

One reason the Alibaba dataset of the GPU cluster was chosen is task independence, i.e. no task has dependencies on other tasks in the dataset. This characteristic of independence made the process of predicting resource utilisation simpler to implement. Since many tasks that are being executed on distributed infrastructures do have dependencies on other tasks and are often deployed onto different resources, it is necessary to take these dependencies into account when predicting the utilisation and deploying the tasks to resources.

Bibliography

[1] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, pp. 1735–1780, Nov. 1997.

[2] "Fully connected layer." https://www.fastaireference.com/tabular-data/fully-connected-layer.

[3] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, and J. He, "Clusterdata/cluster-trace-gpu-v2020 at master · alibaba/clusterdata." https://github.com/alibaba/clusterdata/tree/master/cluster-trace-gpu-v2020.

[4] K. Thonglek, K. Ichikawa, K. Takahashi, H. Iida, and C. Nakasan, "Improving Resource Utilization in Data Centers using an LSTM-based Prediction Model," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 1–8, Sept. 2019.

[5] M. Orr and O. Sinnen, "Optimal task scheduling for partially heterogeneous systems," *Parallel Computing*, vol. 107, p. 102815, Oct. 2021. https://www.sciencedirect.com/science/article/pii/S0167819121000636.

[6] K. Van Ooteghem, "What is a hyperscale data center?." https://www.parallels.com/blogs/ras/hyperscale-data-center/, Jan. 2023.

[7] P. P. Stoop and V. C. Wiers, "The complexity of scheduling in practice," *International Journal of Operations & Production Management*, vol. 16, pp. 37–53, Jan. 1996. https://doi.org/10.1108/01443579610130682.

[8] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, pp. 260–274, Mar. 2002.

[9] B. S. Doerr, T. Venturella, R. Jha, C. D. Gill, and D. C. Schmidt, "Adaptive scheduling for real-time, embedded information systems," in *Gateway to the New Millennium. 18th Digital Avionics Systems Conference. Proceedings (Cat. No. 99CH37033)*, vol. 1, pp. 2–D, IEEE, 1999.

[10] B. Beers, "What is Regression? Definition, Calculation, and Example." https://www.investopedia.com/terms/r/regression.asp.

[11] H. Sak, A. Senior, and F. Beaufays, "Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition." http://arxiv.org/abs/1402.1128, Feb. 2014.

[12] Y. LeCun, D. Touresky, G. Hinton, and T. Sejnowski, "A theoretical framework for back-propagation," in *Proceedings of the 1988 Connectionist Models Summer School*, vol. 1, pp. 21–28, 1988.

[13] D. Parikh, "Disadvantages of RNN." https://iq.opengenus.org/disadvantages-of-rnn/, Jan. 2021.

[14] A. Graves and A. Graves, "Long short-term memory," *Supervised sequence labelling with recurrent neural networks*, pp. 37–45, 2012.

[15] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to Forget: Continual Prediction with LSTM," *Neural Computation*, vol. 12, pp. 2451–2471, Oct. 2000. https://doi.org/10.1162/089976600300015015.

[16] M. Hermans and B. Schrauwen, "Training and Analysing Deep Recurrent Neural Networks," in *Advances in Neural Information Processing Systems*, vol. 26, Curran Associates, Inc., 2013. https://papers.nips.cc/paper/2013/hash/1ff8a7b5dc7a7d1f0ed65aaa29c04b1e-Abstract.html.

[17] Y. Goldberg, "A primer on neural network models for natural language processing," *Journal of Artificial Intelligence Research*, vol. 57, pp. 345–420, 2016.

[18] Y. Cheng, Z. Chai, and A. Anwar, "Characterizing Co-located Datacenter Workloads: An Alibaba Case Study." http://arxiv.org/abs/1808.02919, Aug. 2018.

[19] L. Fengcun and H. Bo, "DeepJS: Job Scheduling Based on Deep Reinforcement Learning in Cloud Data Center." https://github.com/FengcunLi/CloudSimPy/blob/c103672f51d6617707501f05548a7df6090cdca5/playground/paper/F0049-4.19.pdf, Feb. 2023.

[20] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, "MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 945–960, USENIX Association, 2022.

[21] M. K. M. Shapi, N. A. Ramli, and L. J. Awalin, "Energy consumption prediction by using machine learning for smart building: Case study in Malaysia," *Developments in the Built Environment*, vol. 5, p. 100037, Mar. 2021. https://www.sciencedirect.com/science/article/pii/S266616592030034X.

[22] E. Rich and J. Gao, "DeepMind AI reduces energy used for cooling Google data centers by 40%." https://blog.google/outreach-initiatives/environment/deepmind-ai-reduces-energy-used-for/, July 2016.

[23] S. Weisberg, *Applied Linear Regression*, vol. 528. John Wiley & Sons, 2005.

[24] S. B. Kotsiantis, “Decision trees: A recent overview,” *Artificial Intelligence Review*, vol. 39, pp. 261–283, 2013.

[25] L. Breiman, “Random forests,” *Machine learning*, vol. 45, pp. 5–32, 2001.

[26] J. A. Anderson, *An Introduction to Neural Networks.* MIT press, 1995.

[27] E. Patel and D. S. Kushwaha, “A hybrid CNN-LSTM model for predicting server load in cloud computing,” *The Journal of Supercomputing*, vol. 78, no. 8, pp. 1–30, 2022.

[28] S. Tuli, S. S. Gill, P. Garraghan, R. Buyya, G. Casale, and N. Jennings, “Start: Straggler prediction and mitigation for cloud computing environments using encoder lstm networks,” *IEEE Transactions on Services Computing*, 2021.

[29] J. Oren, C. Ross, M. Lefarov, F. Richter, A. Taitler, Z. Feldman, D. Di Castro, and C. Daniel, “SOLO: Search online, learn offline for combinatorial optimization problems,” in *Proceedings of the International Symposium on Combinatorial Search*, vol. 12, pp. 97–105, 2021.

[30] F. James, “Monte Carlo theory and practice,” *Reports on progress in Physics*, vol. 43, no. 9, p. 1145, 1980.

[31] M. V. Ngo, T. Luo, H. Chaouchi, and T. Q. Quek, “Contextual-bandit anomaly detection for IoT data in distributed hierarchical edge computing,” in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1227–1230, IEEE, 2020.

[32] J. Chen, S. Chen, Q. Wang, B. Cao, G. Feng, and J. Hu, “iRAF: A deep reinforcement learning approach for collaborative mobile edge computing IoT networks,” *IEEE Internet of Things Journal*, vol. 6, no. 4, pp. 7011–7024, 2019.

[33] J. Chen, S. Chen, S. Luo, Q. Wang, B. Cao, and X. Li, “An intelligent task offloading algorithm (iTOA) for UAV edge computing network,” *Digital Communications and Networks*, vol. 6, no. 4, pp. 433–443, 2020.

[34] R. Q. Hu, “Mobility-aware edge caching and computing in vehicle networks: A deep reinforcement learning,” *IEEE Transactions on Vehicular Technology*, vol. 67, no. 11, pp. 10190–10203, 2018.

[35] D. Kimovski, C. Bauer, N. Mehran, and R. Prodan, “Big Data Pipeline Scheduling and Adaptation on the Computing Continuum,” in *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 1153–1158, June 2022.

[36] N. Mehran, Z. N. Samani, D. Kimovski, and R. Prodan, "Matching-based Scheduling of Asynchronous Data Processing Workflows on the Computing Continuum," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 58–70, IEEE, 2022.

[37] Docker, "Docker Documentation Overview." https://docs.docker.com/get-started/, Mar. 2023.

[38] The-Linux-Foundation and Kubernetes, "Kubernetes Documentation / Getting started." https://kubernetes.io/docs/setup/.

[39] Netdata, "Getting started — Learn Netdata." https://learn.netdata.cloud/docs/getting-started/, Mar. 2023.

[40] Prometheus, "Overview — Prometheus." https://prometheus.io/docs/introduction/overview/.

[41] Pandas, "Pandas documentation — pandas 1.5.3 documentation." https://pandas.pydata.org/docs/.

[42] The-Linux-Foundation, "PyTorch." https://www.pytorch.org.

[43] H. Patel, "What is Feature Engineering — Importance, Tools and Techniques for Machine Learning." https://towardsdatascience.com/what-is-feature-engineering-importance-tools-and-techniques-for-machine-learning-208
Sept. 2021.

[44] A. Fawcett, "Data Science in 5 Minutes: What is One Hot Encoding?." https://www.educative.io/blog/one-hot-encoding.

[45] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT press, 2016.

[46] O. Köksoy, "Multiresponse robust design: Mean square error (MSE) criterion," *Applied Mathematics and Computation*, vol. 175, no. 2, pp. 1716–1729, 2006.

[47] Q. Weng and H. Ding, "Alibaba Cluster Trace Program." Alibaba, Apr. 2023. https://github.com/alibaba/clusterdata.

[48] M. Rouse, "Hyperparameter." https://www.techopedia.com/definition/34625/hyperparameter-ml-hyperparameter, Aug. 2022.

[49] A. Botchkarev, "Performance metrics (error measures) in machine learning regression, forecasting and prognostics: Properties and typology," *arXiv preprint arXiv:1809.03006*, 2018.

[50] T. Chai and R. R. Draxler, "Root mean square error (RMSE) or mean absolute error (MAE)?–Arguments against avoiding RMSE in the literature," *Geoscientific model development*, vol. 7, no. 3, pp. 1247–1250, 2014.

[51] A. De Myttenaere, B. Golden, B. Le Grand, and F. Rossi, “Mean absolute percentage error for regression models,” *Neurocomputing*, vol. 192, pp. 38–48, 2016.

[52] V. Kreinovich, H. T. Nguyen, and R. Ouncharoen, “How to estimate forecasting quality: A system-motivated derivation of symmetric mean absolute percentage error (SMAPE) and other similar characteristics,” 2014.

[53] J. A. Hartigan and M. A. Wong, “Algorithm AS 136: A k-means clustering algorithm,” *Journal of the royal statistical society. series c (applied statistics)*, vol. 28, no. 1, pp. 100–108, 1979.