applied sciences

MDPI

Article

A System for Sustainable Usage of Computing Resources Leveraging Deep Learning Predictions

Marius Cioca [1] and Ioan Cristian Schuszter [2,3,*]

[1] Faculty of Engineering, "Lucian Blaga" University of Sibiu, Emil Cioran Street, 4, 550025 Sibiu, Romania
[2] European Organization for Nuclear Research (CERN), Espl. des Particules 1, 1211 Geneva, Switzerland
[3] Department of Computer and Electrical Engineering, Universitatea din Petrosani, Strada Universitatii 20, 332009 Petrosani, Romania
* Correspondence: chrisschuszter@gmail.com or cristian.schuszter@cern.ch; Tel.: +40-726-655-822

Featured Application: The system described in this paper can be implemented at scale in order to allow computing facilities to better estimate their resource usage, thus saving them precious resources and contributing to more responsible energy consumption.

Abstract: In this paper, we present the benefit of using deep learning time-series analysis techniques in order to reduce computing resource usage, with the final goal of having greener and more sustainable data centers. Modern enterprises and agile ways-of-working have led to a complete revolution of the way that software engineers develop and deploy software, with the proliferation of container-based technology, such as Kubernetes and Docker. Modern systems tend to use up a large amount of resources, even when idle, and intelligent scaling is one of the methods that could be used to prevent waste. We have developed a system for predicting and influencing computer resource usage based on historical data of real production software systems at CERN, allowing us to scale down the number of machines or containers running a certain service during periods that have been identified as idle. The system leverages recurring neural network models in order to accurately predict the future usage of a software system given its past activity. Using the data obtained from conducting several experiments with the forecasted data, we present the potential reductions on the carbon footprint of these computing services, from the perspective of CPU usage. The results show significant improvements to the computing power usage of the service (60% to 80%) as opposed to just keeping machines running or using simple heuristics that do not look too far into the past.

Keywords: deep learning; RNN; LSTM; time-series; scaling; prediction; carbon footprint; production software

Citation: Cioca, M.; Schuszter, I.C. A System for Sustainable Usage of Computing Resources Leveraging Deep Learning Predictions. *Appl. Sci.* **2022**, *12*, 8411. https://doi.org/10.3390/app12178411

Academic Editor: Vito Conforti

Received: 13 July 2022
Accepted: 20 August 2022
Published: 23 August 2022

1. Introduction

This paper focuses on one aspect of the deep learning revolution that the field is going through, namely the application of deep learning algorithms to time-series analysis and forecasting. This field was spawned by the work mainly completed in the field of finance, where accurate predictions of stock prices can yield a large sum of money to traders [1]. However, recent advances in deep learning technologies have led to a significant increase in the accuracy and performance of these algorithms, which is why this paper investigates them and implements several in the context of efficiently managing and scaling a computing system with heavy usage.

The main novelty which this paper brings is a comparative study which evaluates the performance of various time-series model architectures. There are several previous attempts at predicting resource usage using deep learning techniques, however they focus on a single method, such as Recurrent Neural Networks (RNNs) in the case of Dugann et al. [2] or using very specific architectures, such as restricted Boltzmann machines [3].

The motivation of this paper is to compare several cutting-edge deep learning solutions for resource usage forecasting, in order to identify the most suitable one for the problem presented. The results are discussed and the potential improvements in terms of real load CPU usage are highlighted towards the end. The end-goal is the implementation of an predictor which can be used to adapt the number of resources a computing service needs in near-real time, adjusting itself to periods of high or low load. A secondary constraint of the implemented solution is to not under-allocate resources for an extended period of time, as it could cause service degradation.

The concrete contributions provided by the paper are provided below.

- The paper describes the process of gathering live metrics from distributed applications running on a Kubernetes [4] cluster, inspiring itself from a real production system deployed at the European Laboratory for Nuclear Physics (CERN). Kubernetes runs containers based on snapshots (called images). The most famous container orchestration platform that Kubernetes can run on is [5].
- A few different modern techniques of analyzing and forecasting time-series data are presented and compared with more traditional approaches.
- The performance of the different forecasting algorithms is presented, showcasing which ones are more applicable to the specific use-case presented in this paper, CPU usage forecasting.
- An experiment is conducted based on the algorithm found to be the best performer. The results from the experiment allow us to understand and quantify the potential energy-saving benefits of using this implementation. We compare the ideal scenario with the predictions provided on the validation data, with encouraging results.

2. Materials and Methods

This paper describes different modern computational intelligence techniques, mainly based on deep learning algorithms, which allow the efficient prediction of computer resource usage over time in the context of a highly-critical and used environment. The work focuses on the experimental setup that uses live production data, covering topics from data collection to the evaluation of implementing the various time-series classification algorithms.

The first part of the experimental setup consists in recording production Prometheus metrics from a running API service being called by the CERN Single Sign On service and other clients. Afterwards, several different popular forecasting algorithms that leverage deep are explored, with implementations don using the Keras deep learning framework [6]. The more in-depth analysis was performed using the Pandas [7] data analysis library and the Python programming language. Plots were provided using matplotlib, the leading solution for figures in Python.

2.1. Automatic Resource Cluster Scaling

The practice of automatically scaling clusters up and down based on various metrics is widely discussed in academic literature and actual implementations. Kubernetes is an open-source container management system developed initially as an internal project for Google [8,9]. It allows scaling services and applications to higher and higher loads, catering both to individual teams using it and for entire organizations that deploy highly-distributed systems.

Recently, many researchers have focused on the topic of automatic scaling, as it can be attractive to have an elastic system which handles peak traffic gracefully. Balla et al. identify one of the biggest disadvantages of modern Kubernetes auto-scalers, the fact that they are based on static measurements and do not adapt to their current system usage [10]. There is the built-in vertical pod autoscaler, as well as the 3rd party [11] scaler and the one provided by OpenFaas, which Balla et al. outperform via a simple linear regression algorithm, showing the potential of improving these algorithms.

Rattihalli et al. designed another system for resource-based automatic scaling of Kubernetes applications coined RUBAS [12]. They report a modest improvement in CPU

usage of around 10%. The largest downside of the system described is the lack of a truly dynamic aspect in the prediction process, as scaling decisions are performed every minute and are based on a static formula that takes into account only a limited window from the past.

Toka et al. make the point that scaling provided in Kubernetes is often reactive rather than proactive [13], highlighting the need for a more dynamic approach to scaling. They propose a solution based on machine learning algorithms, integrating an LSTM solution as well. However, the data are not collected from a real production cluster, but rather through the usage of an experimental setup. Although it seems like a good solution, the data collected appears to be relatively smooth and does not feature the noise one would generally find in real service usage.

2.2. Time-Series Analysis

The field of time-series analysis is deeply rooted in mathematical models and knowledge developed more than 50 years ago. Every process that benefits from the analysis of the data it produces over time can benefit from the wide array of techniques available in time-series analysis.

Among the first and most popular algorithms for time-series forecasting is the autoregressive integrated moving average (ARIMA), an old linear model developed around the beginning of the 20th century. However, this technique really benefited when researchers started experimenting with the combination of ARIMA and computational intelligence algorithms, such as artificial neural networks and support vector machines [1,14].

2.3. Deep Learning Models for Time-Series Analysis

Machine learning algorithms have had a wide array of applications over the past three decades, dramatically improving human ability to perform tasks that were considered hard or impossible to model using programming language code. The deep learning algorithms have led to solutions for tasks previously thought to be impossible to be solved by a computer:

- Vision and image identification through the pioneering work of Yann LeCun in automatic zip code identification off postal packages [15];
- Speech recognition through the usage of convolutional neural networks [16];
- Complex learning tasks through q-learning and reinforcement learning, coming closer to top human-level performance in tasks such as playing video games [17].

More recently, deep artificial neural networks have been widely successful in time-series analysis applied to financial data, such as the price of gold being accurately predicted by using convolutional neural networks (CNN), together with long short term memory networks (LSTM) [18]. CNNs have gained much popularity in the field of image processing, especially when it comes to recognition or classification tasks, such as face recognition [19], but recently have been proven to work on sequences of data fairly well (time-series and natural language).

When it comes to climate and carbon-footprint related work, there are several different papers that leverage deep learning for a variety of important tasks. Haq et al. [20] propose a system based on LSTMs that is capable of modeling groundwater storage change, an important topic in more arid areas of the world. Haq [21] also covers the importance and feasibility of providing powerful methods to forecast air pollution using a combination between deep neural networks and classical algorithms, such as XGBoost, random forest, or SVMs.

In terms of sequence processing, one key insight was the development of the back-propagation through time algorithm, allowing the inception and usage of recurrent neural networks (RNNs) [22]. These algorithms work best on a sequence of data, making them important to consider for the task that this paper sets out on solving. Recently, improvements have been made to the algorithm in order to improve its memory efficiency [23].

Recurrent neural networks have been used for a wide range of tasks in the deep learning community, some even developing algorithms that can predict CPU usage using them [2]. However, the predictions focus on a single type of architecture and do not perform a comparison with all the other algorithms that may be useful for the task. Rao et al. [24] more recently attempted to efficiently predict CPU usage based on time-series data and discovered that LSTM networks provide better results than the classic ARIMA approach. The same result in a different field has been found by Siami et al., in their paper presenting financial forecasting using time-series models [25]. From the same field of LSTMs used for price forecasting, Sagheer and Kotb [26] present a paper successfully tweaking the architecture in order to accurately predict oil prices on the market.

The authors are not aware of other papers that perform a thorough comparative study between the different types of neural network architectures which could be used to perform this forecasting task.

Lastly, concepts such as "attention" that spawn from natural language processing and recurrent neural networks can be efficiently used in order to forecast and analyze time-series data as shown by Qin et al. in 2017 [27].

2.4. The Environmental Impact of Computing

When resource usage and the carbon footprint of services are discussed, optimistic opinions mention that the cost of highly computational workloads, such as machine learning, will end up peaking and then diminishing over time [28]. However, it is important to realise the finite nature of computing resources in a typical data center, regardless of how large they are. It should not be only the economic incentive that drives innovation in the heart of computing resource scaling, in order to achieve more with a smaller footprint.

Fridgen et al. [29], like many other researchers of the past few years, concentrate on the massive amount of energy consumed by two intensive and widespread computing processes: machine learning and cryptocurrency mining on systems, such as Bitcoin (the most widely traded and mined cryptocurrency, the precursor to the current age of blockchain-based applications [30]). Based on the findings of Fridgen et al., there is some hope though that the existence of these high-cost systems would drive further investment into renewable energy sources. In terms of papers leveraging deep learning solutions (such as LSTMs), another notable example is that of Farsi et al. [31], which describe a novel LSTM-based used in order to forecast power grid loads in the short term.

There are many other bodies of work that treat different aspects of human carbon impact, such as the incentive to split deliveries, in order to achieve less carbon emissions [32].

3. System Setup and Deep Learning Algorithms

3.1. Overview

The service architecture consists of several components that are part of a critical software system deployed in production, with tens of thousands of daily users. The main points of interest are the Kubernetes pods running the API, as their number should ideally vary over time, in order to ensure optimal resource allocation and not have idle pods lying unused. This is especially important in data centers, where optimal resource allocation ensures less power usage. Previous research, such as GreenCloud [33], focuses on heuristics meant to lower power consumption.

The data collected shows certain recurring usage patterns, where the load of the service is abnormally high, signalling the need to scale up the number of pods running the API. Figure 1 showcases the apparent hourly seasonality of the CPU usage data collected.

We notice in Figure 1 that there are several spikes occurring at roughly the same time every hour (twice), which could boost the accuracy of the algorithm. In fact, the spike in usage comes from automated processes that are synchronising data, e.g., about one's membership to the many thousands of authorization groups that a person can belong to at CERN.

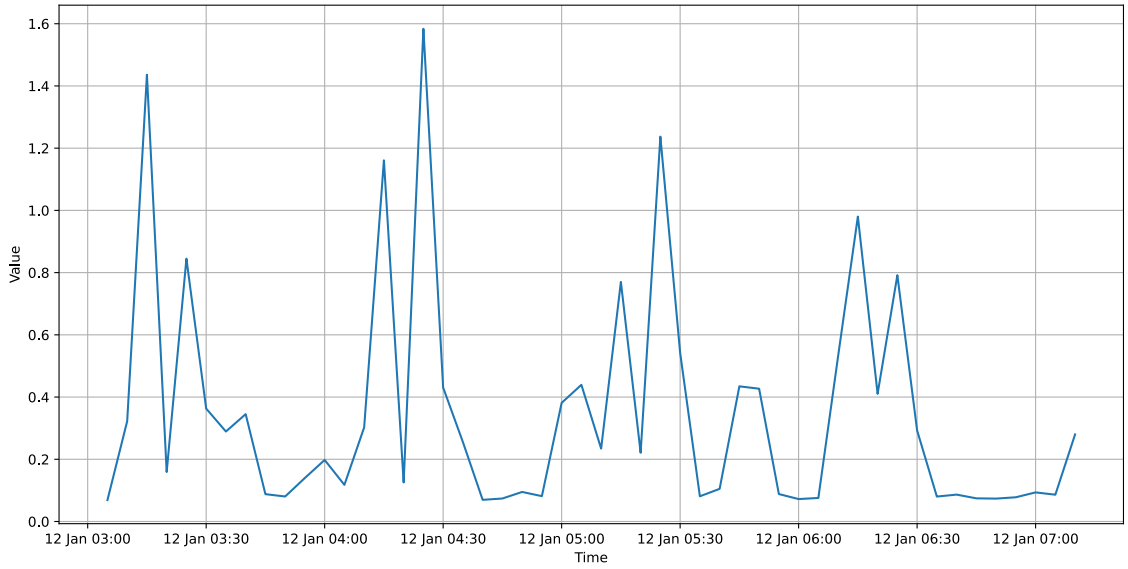

Figure 1. Sample from the collected CPU usage.

Going further into the analysis, we take into consideration the classic metrics that allow us to analyze time-series data and understand its underlying properties:

- Trend—if the data are increasing or decreasing in the long term.
- Seasonality—if certain periodic changes occur in the data and can be spotted by removing the other pieces of data.
- Residuals—the noise in the data which can be represented in this case by ad-hoc users of the API which cause some load on the systems, or other anomalies.

We analyse a 24 h piece of data (09:00 18 January 2022 to 09:00 19 January 2022) from the source dataset, using the Statsmodels [34] Python library. The day is a regular working Tuesday, meaning that we should expect to see some more usage as opposed to, e.g., a weekend day. The results can be seen in Figure 2.

Comparing the results with those from Figure 3 (data between 09:00 15 January–09:00 16 January, Saturday and Sunday), we observe that the data present much more residuals when sampled from a weekday when the API is being called by various clients in an irregular fashion. Additionally, one can observe that the weekday trend is significantly higher during the working day (between 9 and 17), while diminishing after the working day is over and showcasing only the automated spikes. The seasonality points out the 2 spikes that occur every hour from automated processed, as it could be expected from the initial analysis.

3.2. System Architecture and Data Collection

The system design, briefly explained, is a standard distributed architecture based on the Kubernetes container orchestration tool, with one load-balancer in front serving several pods that run the system's API. Additionally, the same pods contain a second container for Prometheus [35], responsible for gathering local data and serving it to the main instance.

Lastly, the data from each of the pods is gathered and aggregated into a central Prometheus instance, which allows querying the dynamically changing list of pods. In case one of the metrics disappears it means the destruction of one of the pods, which will be replaced by a newly-created one, ensuring a fault-tolerant system.

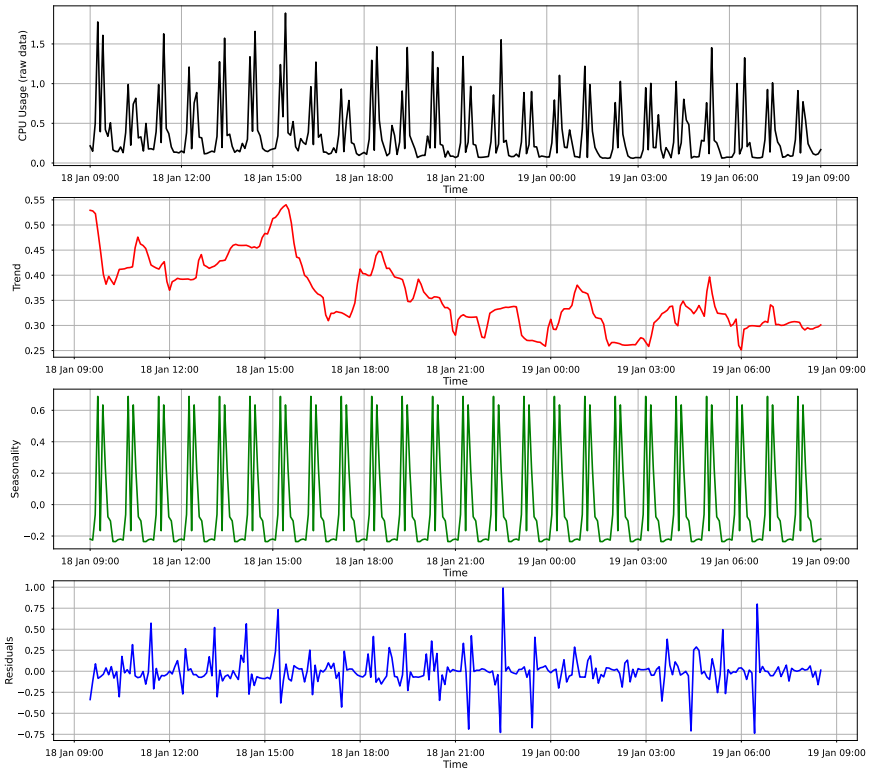

Figure 2. Decomposition (weekday).

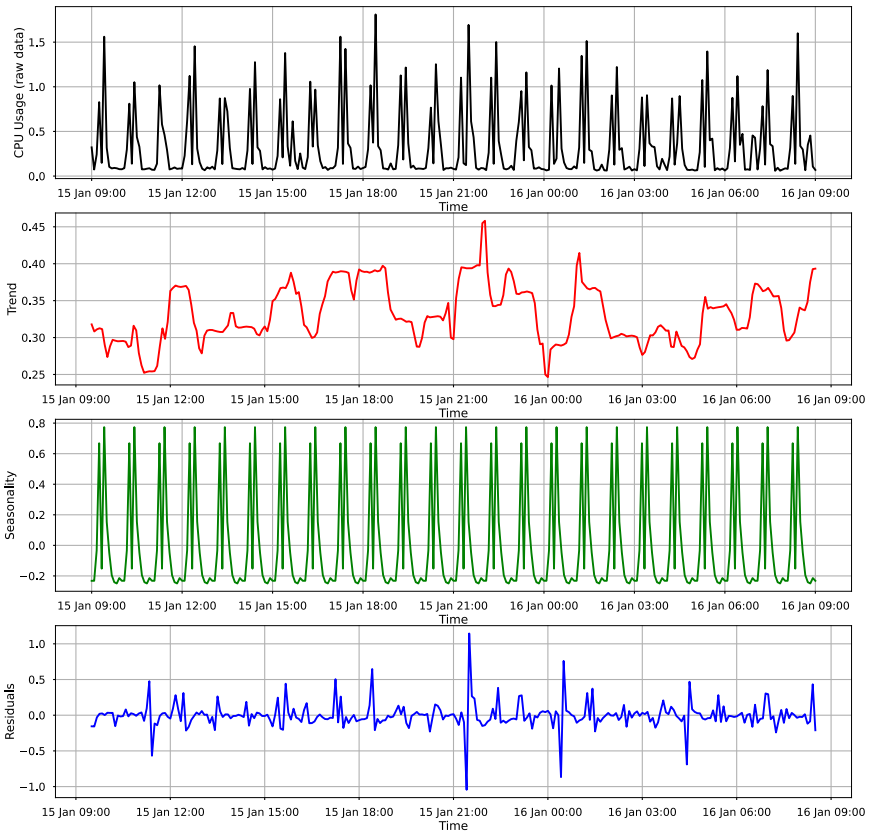

Figure 3. Decomposition (weekend).

Figure 4 shows all the monitored components in the system.

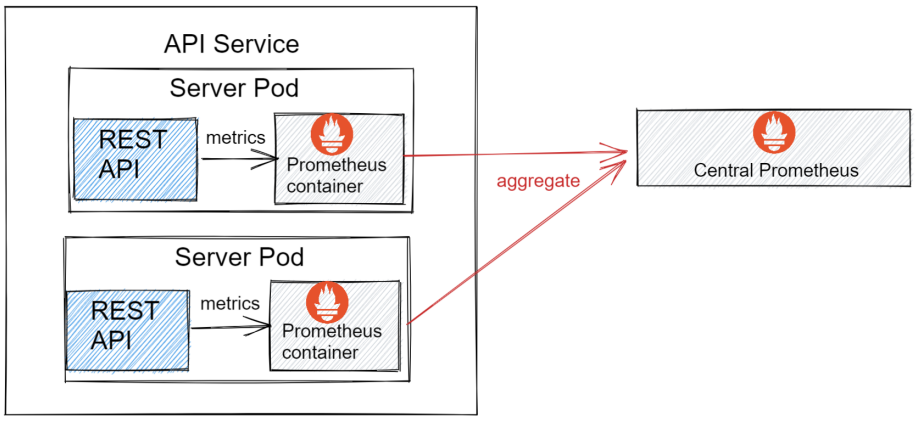

Figure 4. System architecture overview—metric collection.

The data collection process is influenced by the system's architecture. Prometheus [35] is used as the main store for application server metrics. One of the main advantages are fact that Prometheus allows clients to run fast queries that automatically aggregate data into buckets of time-series data, much faster than an equivalent SQL database would be able to. As such, in recent years it has become very popular for building large scalable monitoring systems, for example in the domain of data centers [36].

In Listing 1, the query used to gather the data from the production environment is given. The rate function takes care of returning values depending on the rate of change in the data over a window of 5 min. In principle, the query performs a rolling aggregation of the CPU usage at every 5 min, allowing the authors to capture essential information in one value. The query is performed via a REST API call that allows automated gathering of data to be fed to the online algorithm.

Listing 1. Prometheus query for collecting the data.

```
sum(rate(
process_cpu_seconds_total{
  kubernetes_namespace =~
  "api-process"}[5m]
  )
)
```

The data are returned in the Javascript Object Notation (JSON) format as a list of tuples, the first one containing the UNIX timestamp of the measurement, while the second one being the aggregate value as a floating point number. The format can be clearly seen in Listing 2.

Listing 2. Aggregated data from Prometheus.

```
{   "resultType": "matrix",
    "result": [
        { "metric": {},
          "values": [
          [1641773100, "0.08205714228572106"]
    ]}]
}
```

Once the data are collected, it is processed into a data-frame as part of a Python job using the Pandas data analytics library [7]. These data can then be used to develop the forecasting algorithms.

3.3. Classical Time Series Analysis Algorithms

In order to have a control method, we decided to compare the deep learning implementations developed to be run on our dataset with a more traditional algorithm, the Auto Regressive Integrated Moving Average (ARIMA) algorithm, namely the implementation provided by the Statsmodels library [34].

The algorithm requires 3 parameters that control the 3 different models that form up an ARIMA model:

- p—controls the auto regression (AR) part of the algorithm.
- d—controls the differencing (I) part of the algorithm, the number of non-seasonal differences.
- q—controls the moving average (MA) part of the algorithm, the number of lagged forecast errors.

We used a simple grid search to try several values for the parameters, using the algorithm that provides the best root mean square error (RMSE) metric on the validation data. We used an 80–20 train/test split on the collected dataset to test and validate the model. The value 0 for any of the three metrics represents a "deactivation" of that specific part of the model.

The values tested in the grid search are:

- p—[0, 1, 2, 4, 6, 8, 10];
- d—[0, 1, 2, 3];
- q—[0, 1, 2, 3].

Using the above search, we concluded that the best parameters for our dataset are (6, 0, 2) and we used them for comparison of the results with the other algorithms.

3.4. Deep Learning Algorithms

The paper offers a comparative study of three popular deep neural network architectures and validates their implementations in Tensorflow [37] on the collected CPU metrics. The following types of deep neural networks are used in the implementation of the algorithms:

- Convolutional neural network (CNN) model—3 layers of 1-dimensional convolutions followed by a batch normalization layer and a rectified linear unit layer to apply non-linear transformation to the data. The resulting tensors are passed through a last pooling layer that averages the results of the convolutions and is fed into a 1-neuron dense layer that outputs a predicted value.
- Recurring neural network (RNN) model—2 layers of a simple recurring neural network are used in order to train the model, the first layer feeding a sequence into the second one, then passing the information to a fully-connected dense neural network layer.
- Long short-term memory (LSTM) model—2 layers of an LSTM based neural network, using specialized recurring neural network cells that have a special memory layer feeding into each of the neurons in the sequence. The layers both return sequences and feed into another pair of dense neural network layers before outputting a prediction.

The loss function used in order to perform the stochastic gradient descent optimization for each of the models is the Huber loss function, which has a history of being adapted to time-series and regression problems [38].

Additionally, in order to find the optimal learning rate in order to avoid over or under-fitting the model, a learning rate scheduler is used for each algorithm for 100 epochs. The output of the scheduler seen on a graph shows the correct moment in which the loss on the training set starts increasing due to a too-high learning rate. Figures 5–7 show the output in terms of mean absolute error and loss for the models, at each epoch. As it gets higher, the loss starts increasing and it is essential to pick the learning rate that was found right before the loss begins decaying.

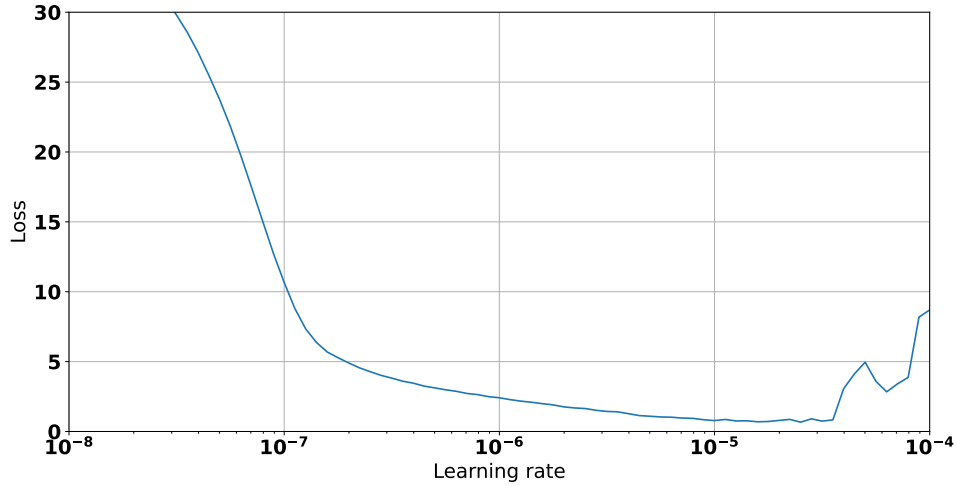

Figure 5. Learning rate schedule output—CNN.

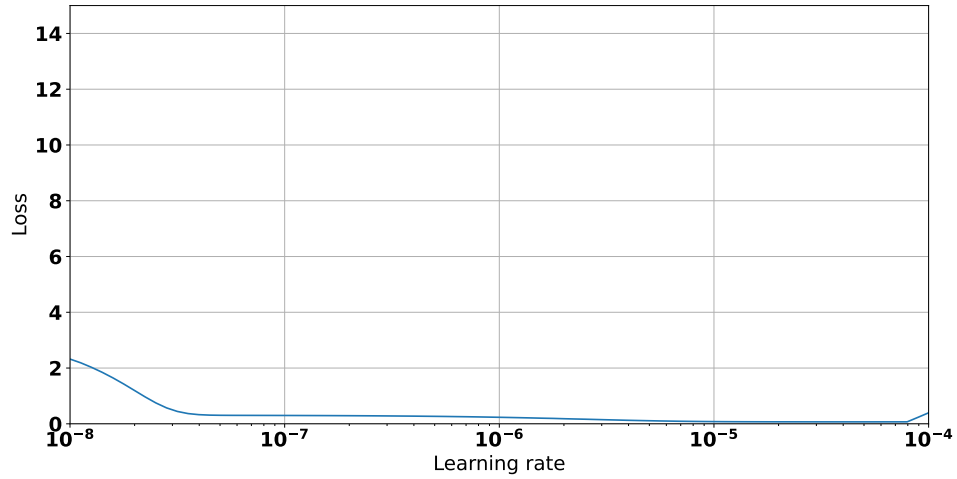

Figure 6. Learning rate schedule output—LSTM.

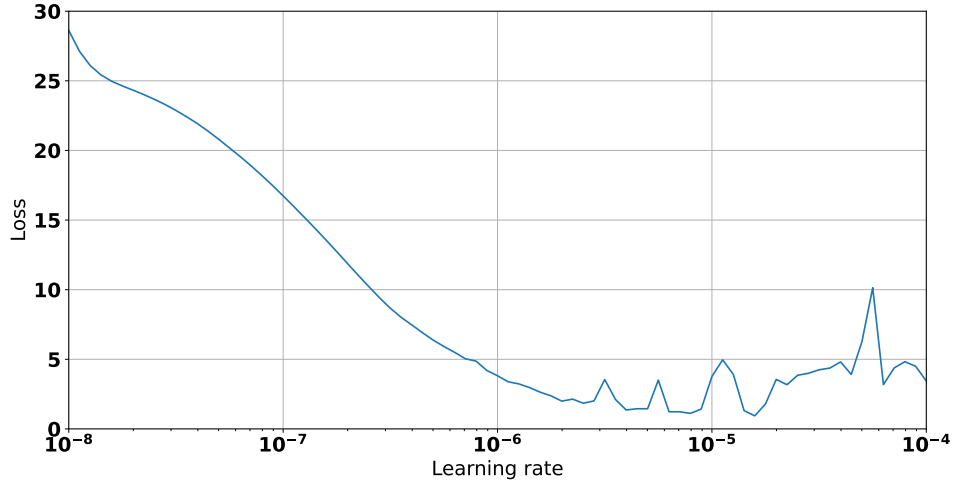

Figure 7. Learning rate schedule output—RNN.

The rest of the parameters, such as those used to establish the batch sizes and the window of data taken into consideration, were selected based on practicality. A batch of 32 windows in the data are taken for each training epoch, while the window size is equal to 20. We took 20 as it represents 100 min per input to the algorithm, 1 h and 20 min, a realistic value to be used when doing online prediction in the future.

Table 1 holds a breakdown of the number of parameters available for training each of the models, as well as the optimal learning rate that was found.

The optimal learning rate can be obtained using the built-in LearningRateScheduler class provided in the Keras implementation of Tensorflow 2. Using stochastic gradient descent with an initial low learning rate 10^{-8} and a momentum of 0.9, Figures 5–7 show the results of training for an epoch with each step of the parameter. The higher the learning rate, the quicker the algorithm converges. However, fast convergence leads to potentially missing the global minimum and "bouncing around" the target, eventually increasing the loss. The values presented in Table 1 are the ones just before the loss starts increasing, leading to the conclusion that they are the parameters for which conversion occurs the fastest.

The number of parameters comes not by choice but from the design of each of the networks. The trainable parameters of the models are further described, layer by layer, in Tables 2–4.

Table 1. Number of parameters and optimal learning rate for each architecture.

Architecture	No. Parameters	Optimal LR
CNN	25,793	1.6×10^{-5}
LSTM + DNN	17,573	1×10^{-5}
RNN	4961	7×10^{-6}

Table 2. Parameter breakdown per model—RNN.

Layer	Parameters
lambda (Lambda)	0
simple_rnn (SimpleRNN)	1680
simple_rnn_1 (SimpleRNN)	3240
dense (Dense)	41
lambda_1 (Lambda)	0

Table 3. Parameter breakdown per model—CNN.

Layer	Parameters
lambda (Lambda)	0
conv1d (Conv1D)	256
batch_normalization	256
re_lu (ReLU)	0
conv1d_1 (Conv1D)	12,352
batch_normalization_1	256
re_lu_1 (ReLU)	0
conv1d_2 (Conv1D)	12,352
batch_normalization_2	256
re_lu_2 (ReLU)	0
global_average_pooling1d	0
dense (Dense)	65
lambda_1 (Lambda)	0

Table 4. Parameter breakdown per model—LSTM.

Layer	Parameters
lambda (Lambda)	0
bidirectional	33,792
bidirectional_1	41,216
dense (Dense)	65
lambda_1 (Lambda)	0

4. Results

The results from the various model implementations show promising results, enabling the development of a real production system that alerts system administrators about impending load increases. This adds the potential to integrate the system with Kubernetes in an automatic manner, making it easy to scale resources depending on their forecasted requirements.

The dataset obtained from the CPU metrics covers the data collected from the 5 January 2022 to the 25 January 2022, a total of 20 days in which usage patterns of the service are likely to be visible.

Taking the results and applying them to the validation dataset, Figures 8–11 graphically show the output of the predictions for all three architectures developed in the writing of this paper. The window seen in each of the figures represents the last 50 samples of the validation dataset (5 min intervals each), potentially being the most unseen piece of the data for the algorithms, offering insight into how the system would perform in real life.

Table 5 discusses the details of each of the algorithm implementations, giving some insight as to how each of them performs. One very important metric for evaluating the model performance on the validation set is the mean absolute error (MAE), which computes how close the predictions are to the real values that were achieved.

Table 5. Training times and losses for each architecture.

Architecture	Training Time/Epoch	Val. MAE	Val. Loss
CNN	8 ms	0.8446	0.1265
LSTM + DNN	9 ms	0.2175	0.0475
RNN	1 s 29 ms	0.1653	0.0843
ARIMA	N/A	0.2384	N/A

In Table 5, the value "N/A" stands for "not applicable", as the training process does not require multiple epochs, as in the case of deep learning algorithms.

Although most of the models seem to perform fairly well, there is a clear pattern of over-fitting and under-fitting visible between the different implementations of the algorithms. In the case of the CNN, the graph and the MAE points to the fact that the network does not fully capture the non-linearity of the data, leading to predictions that are higher than the actual values they should represent. The peaks in the data are represented but the valleys are not "deep" enough. Despite being a generally viable solution for forecasting data that presents seasonality and other repetitive metrics, the results (including those from Figure 8) show that the deep learning algorithms are better capable of predicting the irregularities in the dataset. The MAE for the ARIMA model has been shown to be better than the CNN solution, with the downside of taking a long time to test and for the grid search algorithm to finish successfully. The fact that the training time increases significantly with the addition of data is another drawback of using the ARIMA model.

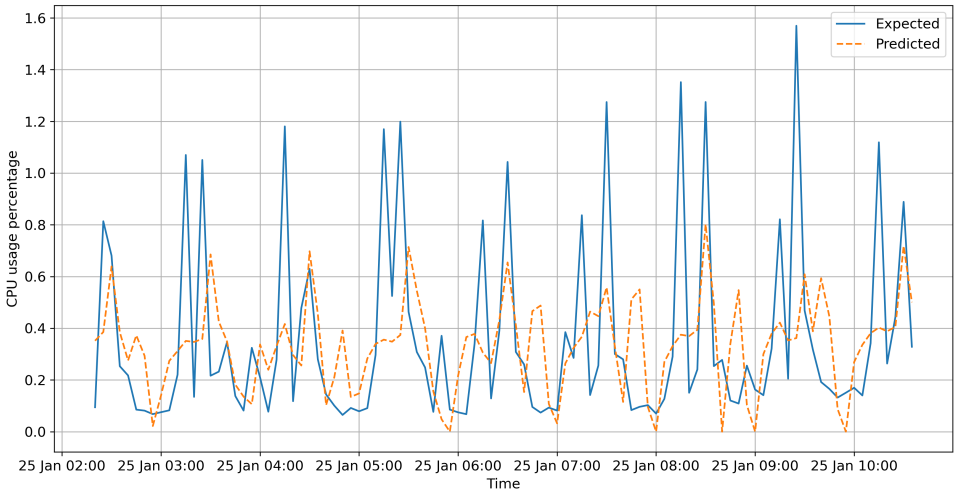

Figure 8. Sample validation predictions—ARIMA.

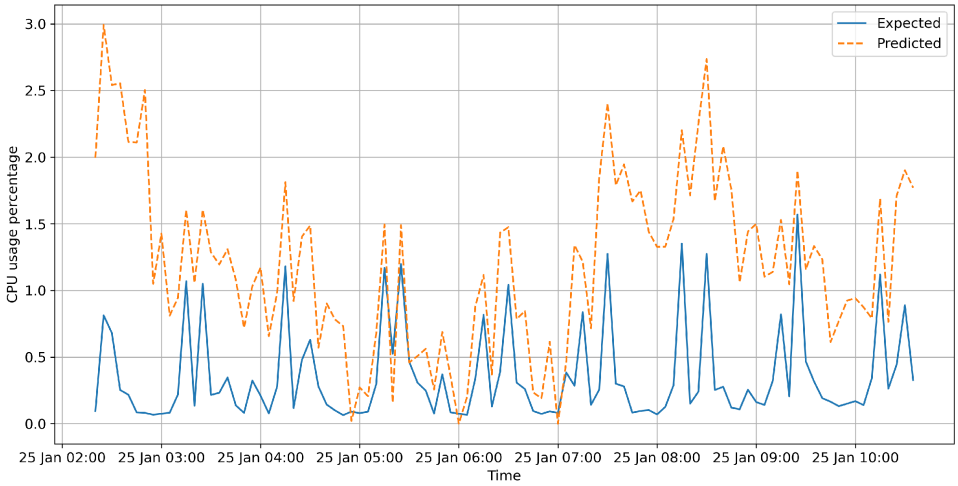

Figure 9. Sample validation predictions—CNN.

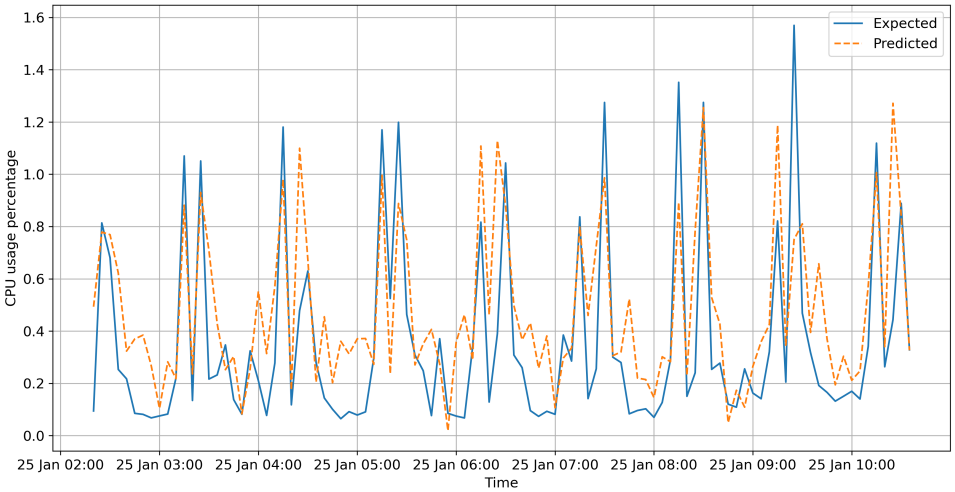

Figure 10. Sample validation predictions—RNN.

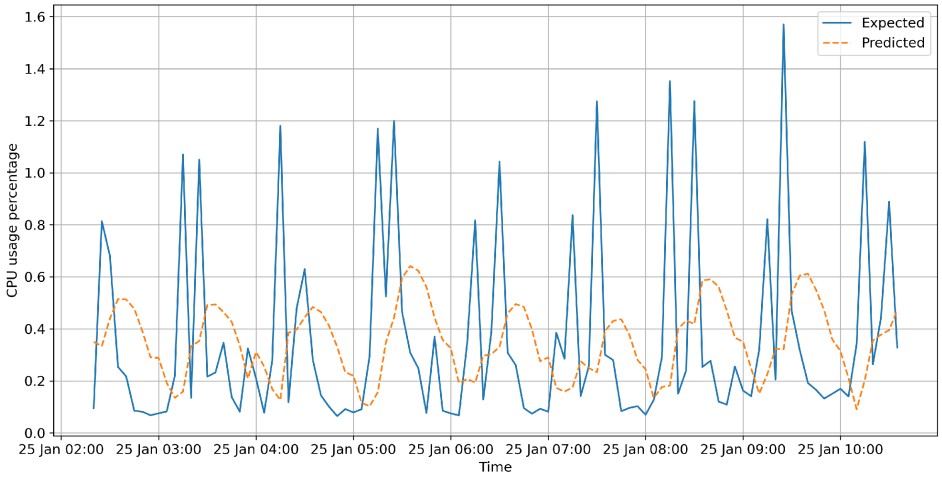

Figure 11. Sample validation predictions—LSTM.

In the case of the LSTM implementation, the algorithm converged quite fast to a low MAE and loss, actually achieving lower loss on the dataset than in the case of the recurring neural network implementation. Additionally, looking at the graph in Figure 12 shows us that the mean absolute error does not go below a specific value, indicating the fact that the algorithm is not specialized enough. In terms of frequency, the data were collected at 5-min intervals for the duration of 20 days, with the last 2 days being used as validation data for the algorithm implementations validation steps.

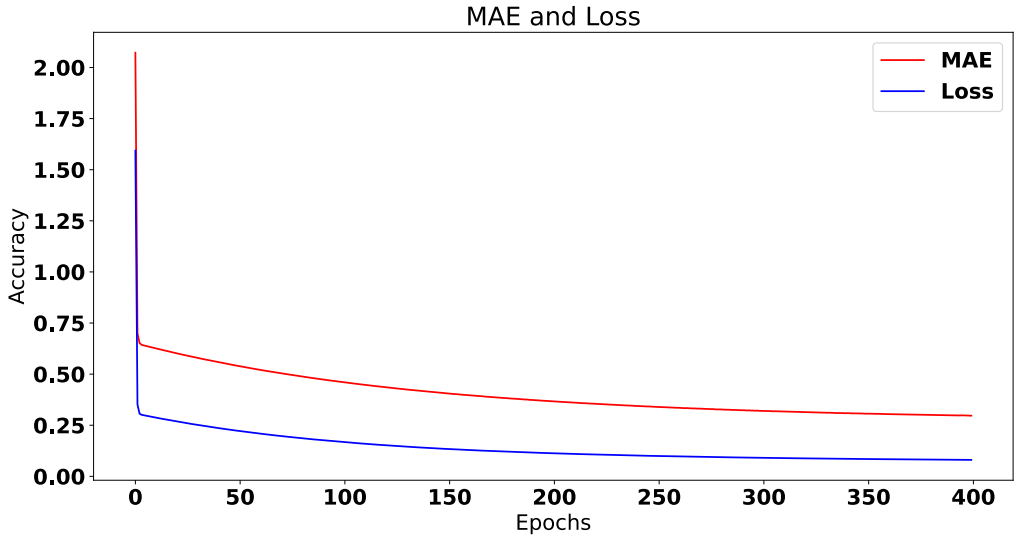

Figure 12. MAE and loss—LSTM training.

The RNN has fewer parameters and offers the best results, a trade-off between the other 2 under and over fitting algorithms. Essentially, it would seem that the LSTM model tends to be too complex for the nature of the task provided and ends up over-fitting to the time-series data. The loss quickly drops after 50 epochs and does not improve afterwards, as one can see in Figure 13. The CNN solution converges very fast and has a slightly higher error than the other implementations, as observer in Figure 14.

Resource Usage Impact

This section discusses the significance of the more detailed results we collected in view of understanding how the algorithm's application in a production system would influence power consumption. We devised an experiment for measuring the fitness of the

forecasts provided by the best algorithm implementation out of the ones that were studied, the RNN solution.

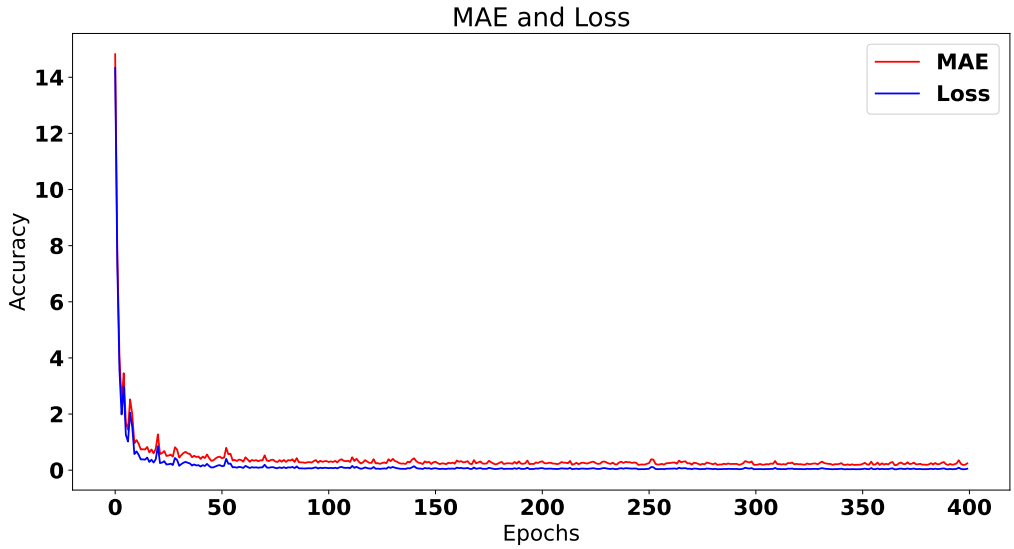

Figure 13. MAE and loss—RNN training.

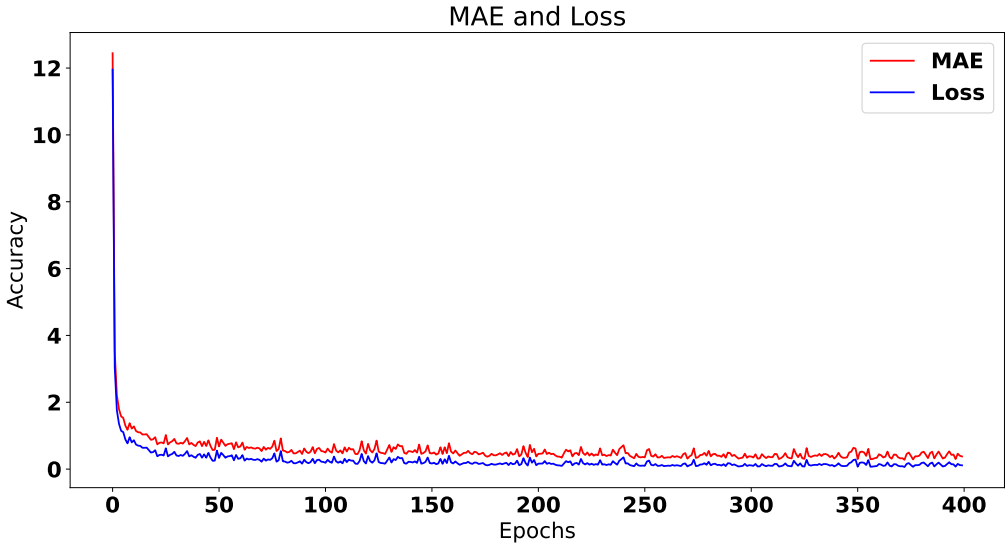

Figure 14. MAE and loss—CNN training.

We name ratio the amount of 5-minute time-intervals that the algorithm should consider as being under heavy load. The experiment was performed on the validation data, roughly 2 days, from the 24–25 January 2022.

The "ratio" value represents the instant when a prediction considers scaling up the system (for example, from 1 to 2 containers). We simplified the model by assuming the system uses 0 units of power when the predicted value is below the threshold and 1 when the prediction is over the ratio value. By computing some statistics (which can be consulted in Table 6), we can compare the number of instances when the system should ideally scale up or down to the real predictions.

We look at the real instances in which the system went over the threshold of computing power needed to scale up (the ratio) and measure how many times the prediction was correct, allowing us to understand the amount of true positives in the system. When the predicted or real value for the CPU usage is not above the ratio, we consider that the system is scaled down, using 0 units of power for that interval of time.

We took both the training and the validation data in the table, in order to see that the performance of the system is not significantly worse on the validation data.

Table 6. Table of measurements for instances where real data and forecasts go above the ratio used in the experiment.

	Ratio	Real Over	Forecast Over	Real All % Over	All % Over	Val. Real Over	Val. Over	Val. Real %	Val. %
0	0.1	3216	4072	72.8	92.2	330	390	79.5	94.0
1	0.2	2173	3397	49.2	76.9	229	331	55.2	79.8
2	0.3	1517	2349	34.4	53.2	147	232	35.4	55.9
3	0.4	1049	1580	23.8	35.8	97	156	23.4	37.6
4	0.5	891	1171	20.2	26.5	83	113	20.0	27.2
5	0.6	808	975	18.3	22.1	77	92	18.6	22.2
6	0.7	731	843	16.6	19.1	69	83	16.6	20.0
7	0.8	654	706	14.8	16.0	67	66	16.1	15.9
8	0.9	537	548	12.2	12.4	53	51	12.8	12.3

The legend for Table 6 is the following:

- Ratio—the CPU usage over which the system should scale up.
- Real Over—the number of real instances in which the CPU usage surpassed the value (training + test data).
- Forecast Over—the number of forecasts where the CPU usage is above the ratio value.
- Real All % Over—the percentage of real predictions where the CPU usage values were over the ratio.
- All % Over—the percentage of forecasts where the CPU usage values were over the ratio.
- Val. Real Over—the number of real instances where the values where over the ratio in the validation data.
- Val. Over—the number of forecast instances where the values where over the ratio in the validation data.
- Val. Real %—the percentage of real measurements where the CPU usage values were over the ratio in the validation data.
- Val.%—the percentage of forecasts where the CPU usage values were over the ratio in the validation data.

We can observe that the real CPU usage is lower than the predicted one when taking the threshold lower than 0.4 (both for the training and validation data). For example, at 0.2 the forecast is over that value 79.8% of the time while the real values were around 55.2%. This shows a trend for the algorithm to overestimate the amount of CPU resources needed at a certain point in time, which is not necessarily something undesirable. However, in the context of lowering energy consumption, the best iteration of the algorithm for us is the one that provides forecasts at all the instances when they also occur (5 min in the future).

However, when we look at the values when the ratio is larger than 0.4, it can be observed that the predictions and the real CPU usage converge. This means that on most instances where the CPU usage was over the ratio, the forecasting algorithm predicted similar values. As these are 5 min time-frames, this is desirable since the scaling does not have to occur instantly and can smoothly take place. This is to be expected of a heuristic in general and should be used only as an informative item. In practice, scaling around 40–50% seems to give results that are promising enough and would work in a real-life implementation.

Given the data above, we decided to analyze the overall gains in terms of energy consumption. To simplify the computation process, we consider the scaled-down mode of operation as using 0 units of energy. When the system is under a load larger than the ratio value, that time is counted as using 1 unit of energy. The measurements were performed only on the validation dataset.

The 3 cases present on the graphs are as follows:

- Orange—the worst-case scenario (identical for every instance), where the system is constantly in high CPU-usage mode.
- Green—the best-case scenario, matching the real CPU resource usage. The ideal scaling system would already know if the CPU usage will be above the threshold in the next 5 min, scaling up before the load occurs.
- Blue—the scenario where prediction values larger than the ratio are considered "scale up" commands.
- Red—a simple heuristic that mimics what a traditional resource scheduler would do. It takes the previous 2 measurements and compares it to the current timestamp. If the difference between the mean of the previous 2 measurements and the current reading is larger than the "ratio", a scale up is issued. Alternatively, if the difference between the current value and the previous means is smaller by ratio %, a scale down is issued.

Figure 15 shows the values obtained for each of the different cases.

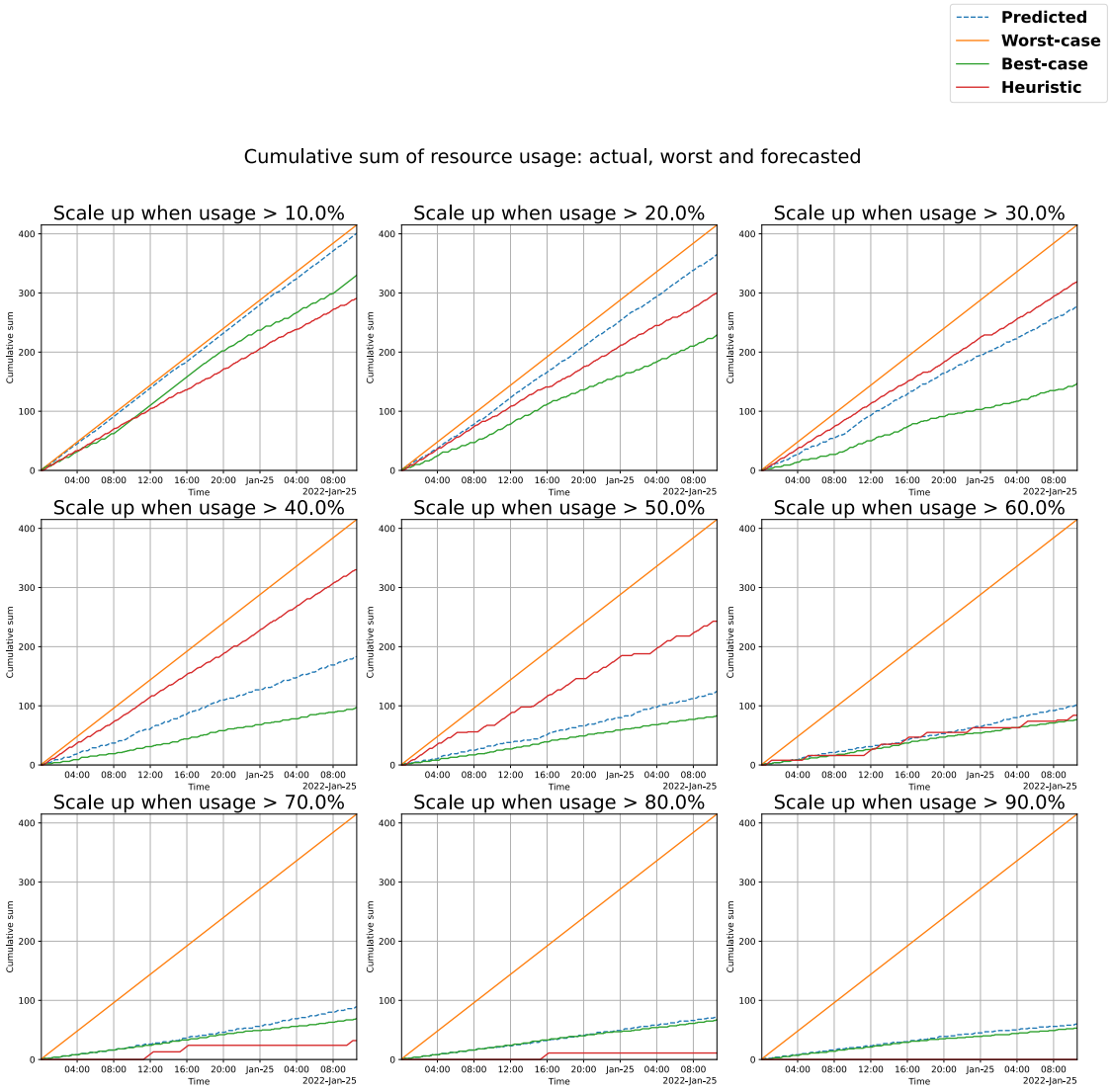

Figure 15. Cumulative sum of resource usage.

We can now observe the values in Table 7 and the relative performance of the forecasting algorithm compared to the best and the worst cases. As observed on the cumulative sum graphs, the forecasting performance becomes better once the ratio value increases, as there are fewer instances where the predictor has the chance to miss. The key observation to make is that the ratio experiment was an exploratory matter devised to see at which point one should consider scaling up, as the system could potentially stay scaled down even at 30% CPU usage.

Table 7. Cumulative sums at various ratio levels.

	Ratio	Worst	Forecast	Ideal	% From Worst	% From Ideal
0	0.1	415	390	330	94.0	118.2
1	0.2	415	331	229	79.8	144.5
2	0.3	415	232	147	55.9	157.8
3	0.4	415	156	97	37.6	160.8
4	0.5	415	113	83	27.2	136.1
5	0.6	415	92	77	22.2	119.5
6	0.7	415	83	69	20.0	120.3
7	0.8	415	66	67	15.9	98.5
8	0.9	415	51	53	12.3	96.2

Taking a higher threshold eliminates the intervals where the CPU usage is lower and the system does not need to stay scaled up. However, in a real-life scenario, one would need to balance the practicality of the solution with the risk of keeping the system under heavy load and degrading the performance of the system. Looking at the predictions, a value around 40% looks reasonable enough and not overly sensitive.

On average, across all experiments, we obtained an energy consumption 60% less than the worst-case scenario and 28% more than the ideal scenario. If we take only the cases with a ratio larger than 0.4, the numbers become 80.5% better than the worst-case scenario and 14% worse than the ideal scenario.

As expected, the cumulative sum of the heuristic metric lowers a lot when reaching higher values of the "ratio", showing that the heuristic does not perform any up-scaling unless there is a long period in which the system is potentially overworking. We consider the heuristic to be quite efficient for a metric that is so simple, but it would most likely under-perform the predictive model in a real-world scenario.

To argue a bit more about the significance of the heuristic result, the numbers point to it having a cumulative sum of resource usage that is much lower than even the ideal case when we have "ratio" values that are higher than 0.6. How is this possible?

The answer is that the ideal model can be perceived as the bottom threshold of minimal resource usage which is necessary to ensure the smooth operation of the system. One could scale down more often or scale up only when the system is really strained, as in the case of the heuristic. However, this scenario means that we are not allocating enough resources and in a real-world scenario it would likely degrade the service. It is for this reason that the heuristic provides a poorer overall performance of the system than the predictive case.

Using the above results, we can confidently say that a best case scenario of applying the scaling algorithm to an operational data center could lead to significant improvements in term of computing resource usage and power consumption optimization. In reality, many aspects are involved in data center power usage, so simple scaling down of machines is not enough as powering off hardware ends up being at the compromise of Quality of Service in the eventuality of a power cut or similar event [39].

5. Conclusions

In this paper, a real-world example of deep learning algorithms applied to time-series prediction was introduced. Three different architectures were evaluated on the collected

data and their performance was discussed in detail, as well as highlighting the benefits this scaling could bring. Including this system as part of a Kubernetes automatic scaler could yield significant improvements in terms of resource usage, around 60% in a real-world scenario. The online (constant) training of the algorithm on new data could help keep the scaler up-to-date, even in the situation when the usage patterns change significantly.

The presented results show that several different deep learning techniques can be successfully applied to the complex task of CPU resources usage. Additionally, the novelty of the study was represented by the comparison provided between the three different architectures. Similar papers in the field choose to focus on one specific implementation [2,3,24], without discussing the implications of trying out different architectures and comparing them in order to obtain the best results for the problem at hand. The most efficient architecture out of the three tested was identified, achieving the goal of the paper to efficiently predict resource usage.

Additionally, the data extracted from the real validation data and compared with the predictions has shown the potential for the algorithm to forecast high-load periods with only 20% less accuracy than an ideal "oracle" which can see the future CPU usage of the service. Compared to the worst-case scenario, improvements of up to 80% could be seen in terms of raw computing power usage over a certain window of time, as shown in Figure 15. These promising results could have some true real-world impact, especially when applied at the scale of an entire data center.

Comparing this paper to other similar systems developed and introduced, the advantage of this solution is mainly posed by the usage of a real production dataset coupled with a thorough analysis and discussion on the deep learning algorithm's effectiveness in practice. One disadvantage that would be addressed at a future date is the lack of comparison of this solution with other publicly available datasets.

In terms of future work, one possible path of development lies at the intersection of the fields of Computer Science and Systems Engineering. An automated scaler could be implemented using the Kubernetes Go API and added to any system that would require it. The algorithm requires a very small window of data (around 1 h of 5 min intervals) points in order to offer a good prediction for the future.

The forecasting algorithm that we designed, for example when compared to RUBAS [12], allows us to improve the performance of the algorithm over time, given that it collects data from the same system. As we previously mentioned, many of the other proposed methods in the literature rely on static formulas and limited windows of data. Although it is true that the algorithm requires a certain amount of data to offer a prediction, that data can be used as training data for future predictions, improving its performance. Additionally, the recurrent aspect of the implemented networks allow past events (such as a spike in the CPU usage) to influence future predictions in a truly dynamic way, offering more robust results regardless of the data that is input.

Due to the information provided above, this makes it easy to implement solutions that scale dynamically based on the prediction output. This topic shall be the subject of discussion in a future paper.

Author Contributions: Conceptualization, M.C. and I.C.S.; Data curation, I.C.S.; Funding acquisition, M.C.; Investigation, I.C.S.; Methodology, M.C. and I.C.S.; Project administration, M.C.; Software, I.C.S.; Supervision, M.C.; Visualization, I.C.S.; Writing—original draft, I.C.S.; Writing—review and editing, M.C. and I.C.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Lucian Blaga University of Sibiu and Hasso Plattner Foundation research grants LBUS-IRG-2021-07.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The dataset collected for the purpose of this study is available publicly at the following URL https://github.com/saibot94/cpu-dataset-prometheus (accessed on 22 August 2022). Additional to the JSON file containing the measurements, a Python script is provided that parses the data into a Pandas dataframe and allows it to be used for any classification task. There is a short introduction on the dataset in the README file. The data does not contain any specific information subject to data protection agreements.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

CNN	Convolutional neural network
LSTM	Long-short term memory
LR	Learning rate
RNN	Recurrent neural network
API	Application programmer interface
CPU	Central processing unit
CERN	The European Organization for Particle Physics

References

1. Pai, P.F.; Lin, C.S. A hybrid ARIMA and support vector machines model in stock price forecasting. *Omega* **2005**, *33*, 497–505. [CrossRef]
2. Duggan, M.; Mason, K.; Duggan, J.; Howley, E.; Barrett, E. Predicting host CPU utilization in cloud computing using recurrent neural networks. In Proceedings of the 2017 12th International Conference for Internet Technology and Secured Transactions (ICITST), Cambridge, UK, 11–14 December 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 67–72.
3. Qiu, F.; Zhang, B.; Guo, J. A deep learning approach for VM workload prediction in the cloud. In Proceedings of the 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), Shanghai, China, 30 May–1 June 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 319–324.
4. Kubernetes Documentation. Available online: https://kubernetes.io/docs/home/ (accessed on 6 July 2022).
5. Docker Container Platform. Available online: https://docs.docker.com/get-started/overview/ (accessed on 6 July 2022).
6. Gulli, A.; Pal, S. *Deep Learning with Keras*; Packt Publishing Ltd.: Birmingham, UK, 2017.
7. Pandas Data Analysis Library. Available online: https://pandas.pydata.org/ (accessed on 6 July 2022).
8. Luksa, M. *Kubernetes in Action*; Simon and Schuster: New York, NY, USA, 2017.
9. Verma, A.; Pedrosa, L.; Korupolu, M.; Oppenheimer, D.; Tune, E.; Wilkes, J. Large-scale cluster management at Google with Borg. In Proceedings of the Tenth European Conference on Computer Systems, Bordeaux, France, 21–24 April 2015; pp. 1–17.
10. Balla, D.; Simon, C.; Maliosz, M. Adaptive scaling of Kubernetes pods. In Proceedings of the NOMS 2020—2020 IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, 20–24 April 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1–5.
11. Knative Scaler. Available online: https://github.com/knative/docs (accessed on 6 July 2022).
12. Rattihalli, G.; Govindaraju, M.; Lu, H.; Tiwari, D. Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes. In Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), Milan, Italy, 8–13 July 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 33–40.
13. Toka, L.; Dobreff, G.; Fodor, B.; Sonkoly, B. Machine learning-based scaling management for kubernetes edge clusters. *IEEE Trans. Netw. Serv. Manag.* **2021**, *18*, 958–972. [CrossRef]
14. Zhang, G.P. Time series forecasting using a hybrid ARIMA and neural network model. *Neurocomputing* **2003**, *50*, 159–175. [CrossRef]
15. LeCun, Y.; Boser, B.; Denker, J.S.; Henderson, D.; Howard, R.E.; Hubbard, W.; Jackel, L.D. Backpropagation applied to handwritten zip code recognition. *Neural Comput.* **1989**, *1*, 541–551. [CrossRef]
16. Abdel-Hamid, O.; Mohamed, A.R.; Jiang, H.; Deng, L.; Penn, G.; Yu, D. Convolutional neural networks for speech recognition. *IEEE/ACM Trans. Audio Speech Lang. Process.* **2014**, *22*, 1533–1545. [CrossRef]
17. Vinyals, O.; Babuschkin, I.; Czarnecki, W.M.; Mathieu, M.; Dudzik, A.; Chung, J.; Choi, D.H.; Powell, R.; Ewalds, T.; Georgiev, P.; et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* **2019**, *575*, 350–354. [CrossRef]
18. Livieris, I.E.; Pintelas, E.; Pintelas, P. A CNN–LSTM model for gold price time-series forecasting. *Neural Comput. Appl.* **2020**, *32*, 17351–17360. [CrossRef]
19. Lawrence, S.; Giles, C.L.; Tsoi, A.C.; Back, A.D. Face recognition: A convolutional neural-network approach. *IEEE Trans. Neural Netw.* **1997**, *8*, 98–113. [CrossRef]

20. Haq, M.A.; Jilani, A.K.; Prabu, P. Deep Learning Based Modeling of Groundwater Storage Change. *CMC-Comput. Mater. Contin.* **2021**, *70*, 4599–4617.
21. Haq, M.A. Smotednn: A novel model for air pollution forecasting and aqi classification. *Comput. Mater. Contin.* **2022**, *71*, 1.
22. Werbos, P.J. Backpropagation through time: What it does and how to do it. *Proc. IEEE* **1990**, *78*, 1550–1560. [CrossRef]
23. Gruslys, A.; Munos, R.; Danihelka, I.; Lanctot, M.; Graves, A. Memory-efficient backpropagation through time. *Adv. Neural Inf. Process. Syst.* **2016**, *29*, 1–9.
24. Rao, S.N.; Shobha, G.; Prabhu, S.; Deepamala, N. Time Series Forecasting methods suitable for prediction of CPU usage. In Proceedings of the 2019 4th International Conference on Computational Systems and Information Technology for Sustainable Solution (CSITSS), Bengaluru, India, 20–21 December 2019; IEEE: Piscataway, NJ, USA, 2019; Volume 4, pp. 1–5.
25. Siami-Namini, S.; Namin, A.S. Forecasting economics and financial time series: ARIMA vs. LSTM. *arXiv* **2018**, arXiv:1803.06386.
26. Sagheer, A.; Kotb, M. Time series forecasting of petroleum production using deep LSTM recurrent networks. *Neurocomputing* **2019**, *323*, 203–213. [CrossRef]
27. Qin, Y.; Song, D.; Chen, H.; Cheng, W.; Jiang, G.; Cottrell, G. A dual-stage attention-based recurrent neural network for time series prediction. *arXiv* **2017**, arXiv:1704.02971.
28. Patterson, D.; Gonzalez, J.; Hölzle, U.; Le, Q.; Liang, C.; Munguia, L.M.; Rothchild, D.; So, D.R.; Texier, M.; Dean, J. The carbon footprint of machine learning training will plateau, then shrink. *Computer* **2022**, *55*, 18–28. [CrossRef]
29. Fridgen, G.; Körner, M.F.; Walters, S.; Weibelzahl, M. Not all doom and gloom: How energy-intensive and temporally flexible data center applications may actually promote renewable energy sources. *Bus. Inf. Syst. Eng.* **2021**, *63*, 243–256. [CrossRef]
30. Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Bus. Rev.* **2008**, *4*, 21260.
31. Farsi, B.; Amayri, M.; Bouguila, N.; Eicker, U. On short-term load forecasting using machine learning techniques and a novel parallel deep LSTM-CNN approach. *IEEE Access* **2021**, *9*, 31191–31212. [CrossRef]
32. Wu, D.; Wu, C. Research on the Time-Dependent Split Delivery Green Vehicle Routing Problem for Fresh Agricultural Products with Multiple Time Windows. *Agriculture* **2022**, *12*, 793. [CrossRef]
33. Liu, L.; Wang, H.; Liu, X.; Jin, X.; He, W.B.; Wang, Q.B.; Chen, Y. GreenCloud: A new architecture for green data center. In Proceedings of the 6th International Conference Industry Session on Autonomic Computing and Communications Industry Session, Barcelona, Spain, 15 June 2009; pp. 29–38.
34. Statsmodels Data Analysis Library. Available online: https://www.statsmodels.org/stable/index.html (accessed on 7 August 2022).
35. Turnbull, J. Monitoring with Prometheus. Turnbull Press. 2018. Available online: https://www.prometheusbook.com/MonitoringWithPrometheus_sample.pdf (accessed on 19 August 2022).
36. Chen, L.; Xian, M.; Liu, J. Monitoring System of OpenStack Cloud Platform Based on Prometheus. In Proceedings of the 2020 International Conference on Computer Vision, Image and Deep Learning (CVIDL), Chongqing, China, 10–12 July 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 206–209.
37. Martin, A.; Ashish, A.; Paul, B.; Eugene, B.; Chen, Z.; Citro, C.; Corrado, G.S.; Davis, A.; Dean, J.; Devin, M. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. 2015. Available online: https://www.tensorflow.org/ (accessed on 6 July 2022).
38. Balasundaram, S.; Prasad, S.C. Robust twin support vector regression based on Huber loss function. *Neural Comput. Appl.* **2020**, *32*, 11285–11309. [CrossRef]
39. Pries, R.; Jarschel, M.; Schlosser, D.; Klopf, M.; Tran-Gia, P. Power consumption analysis of data center architectures. In Proceedings of the International Conference on Green Communications and Networking, Colmar, France, 5–7 October 2011; Springer: Berlin/Heidelberg, Germany, 2011; pp. 114–124.