Fine-Grained Long-Range Prediction of Resource Usage in Computer Clusters

Ivan Ukhov, Diana Marculescu, Petru Eles, and Zebo Peng

***Abstract*—In order to facilitate the development of intelligent resource managers of computer clusters, we investigate the utility of the state-of-the-art neural networks for the purpose of fine-grained long-range prediction of the resource usage in one such cluster. We consider a large data set of real-life traces and describe in detail our workflow, starting from making the data accessible for learning and finishing by predicting the resource usage of individual tasks multiple steps ahead. The experimental results indicate that such fine-grained traces as the ones considered possess a certain structure, and that this structure can be extracted by advanced machine-learning techniques and subsequently utilized for making informed predictions.**

***Index Terms*—Computer cluster, machine learning, neural network, prediction, resource management, uncertainty.**

I. Introduction and Motivation

Resource management is of great importance. It is the activity that, if adequately executed, enables one to exploit optimally the full potential of the computer system at hand. However, there is typically very little control over the production environment. In particular, the actual workload that the system will have to process at runtime is rarely known in advance. Therefore, efficient resource management is a difficult endeavor, which has to cope with inherent uncertainty.

Uncertainty can be mitigated by predicting the future and acting accordingly, which is what proactive resource managers thrive on. However, accurate and useful prediction is not an easy task either. Modern computer systems are elaborate, and their resource managers require elaborate forecasting mechanisms.

Forecasting traditionally falls within the scope of machine learning [1]. Machine learning has recently received a great amount of due attention due the renaissance in neural networks [2], which seemingly effortlessly superseded the then state-of-the-art techniques for modeling and prediction.

We posit that modern neural networks constitute a highly promising assistant for resource management. Despite the fact that resource management has already seen a number of applications of neural networks—which we shall touch upon in Sec. II—the research in this direction has been limited. In particular, only primitive architectures of neural networks have been considered, and they have been applied to relatively simple problems. This state of affairs is unfortunate provided that neural networks have been nearly revolutionary in other disciplines. Therefore, we feel strongly that more research should be conducted in order to investigate the aid that the recent advancements in the field of machine learning can give to the design of resource managers of computer systems.

In this paper, we set out to conduct one such body of research. More specifically, we study the resource usage in a large computer cluster and aim to predict this usage multiple steps ahead at the level of individual tasks executed in the cluster. To this end, we intend to use recurrent neural networks [2].

Consider the example given in Fig. 1 where the CPU usage of a task running on a machine in the cluster is depicted three times (the solid-red-to-solid-blue lines). The three cases correspond to three different time moments (the black dots) as viewed by the resource manager of the cluster. The solid red lines represent the history of the usage, which is known to the manager, while the solid blue lines represent the future usage, which is unknown to the manager. The latter is what we are to predict in this work for each task of interest and several steps ahead. In Fig. 1, our potential predictions up to four steps ahead are depicted by a set of dashed blue lines.

Such detailed (for individual tasks) and foresighted (multiple steps ahead) information about the future resource usage as the one depicted in Fig. 1 can be of great help to the resource manager. For instance, knowing the future resource usage of the tasks that are currently being executed in the cluster, the manager can more intelligently decide which of the cluster's machines the next incoming task should be delegated to.

The remainder of the paper is organized as follows. Section II provides an overview of the prior work and summarizes our contributions. In Sec. III, the problem that we address is formulated. Our pipeline for data processing is presented in Sec. IV, and our predictive model in Sec. V. The key operational aspects of our workflow are described in Sec. VI. The experimental results are reported and discussed in Sec. VII. Lastly, Sec. VIII provides a number of concluding remarks.

II. Prior Work and Our Contributions

Let us have a look at a number of studies that leverage various techniques from the field of machine learning in order to facilitate resource management in computer systems.

In [3], temperature forecasting is based on an autoregressive moving-average model [1], enabling the development of an efficient thermal management strategy for multiprocessor sys-

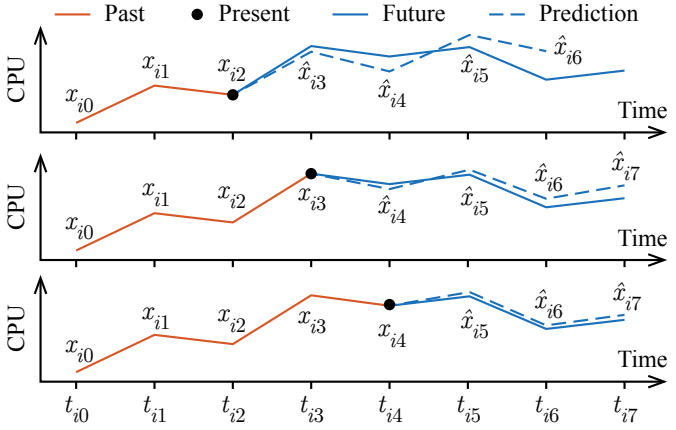

Figure 1. Example of predicting the CPU usage of a particular task up to four steps ahead at three different time moments.

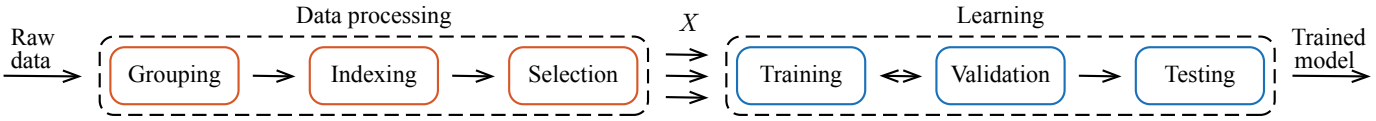

Figure 2. Overview of our workflow. The data-processing and learning stages are described in Sec. IV and Sec. VI, respectively.

tems. The work in [4] enhances runtime thermal management by providing an on-chip temperature predictor based on feed-forward neural networks [1]. The analysis and mitigation of the impact of process variation undertaken in [5] are facilitated by a linear regression model [1] trained on leakage-power measurements with the goal of predicting peak temperatures.

Closer to our work, the work in [6] is concerned with cloud data centers. The authors propose a framework for predicting the number of virtual-machine requests together with the required amount of CPU and memory. The framework makes use of k-means clustering [1] for identifying different types of requests, and it then uses Wiener filters in order to estimate the workload with respect to each identified type.

Similar to [6], the work in [7] is focused on forecasting virtual-machine requests in cloud data centers and relies on k-means clustering as the first step. Unlike [6], the main workhorse in the case of [7] is extreme learning machines, which are feed-forward neural networks mentioned earlier.

An ensemble model [1] is presented in [8] targeted at predicting the CPU usage in cloud environments. It relies on multiple models including an exponential smoothing, auto regressive, weighted nearest neighbors, and most similar pattern model. The final predictions are obtained by combing the predictions from these models by means of a scoring algorithm.

It can be seen in the above that, in general, machine learning has been extensively utilized for aiding the design of resource managers of computer systems. However, as mentioned in Sec. I, the most recent advancements in machine learning have not been sufficiently explored in this context yet. In particular, the utility of neural networks have been studied only marginally: feed-forward networks, as in [4], [7], are arguably the simplest and least powerful members of their rich family.

In addition, note that the predictions delivered by [6], [7], [8] are coarse, aggregative. The corresponding techniques treat virtual-machine requests or computational resources as a fluid and predict the amount of this fluid that will arrive or be needed at the next time step. It means that the aforementioned techniques are not capable of characterizing individual tasks. Such fine-grained information, however, can be of help for the resource manager of the computer cluster under consideration.

To conclude, only primitive architectures of neural networks have been considered in the literature on resource management, and only aggregative prediction has been addressed so far. Thus, there is a need for further exploration and development.

Our work makes the following major contributions:

- We develop a data-processing pipeline for working with large data sets that makes the data readily accessible for machine learning, and we apply it to a data set of real resource-usage traces collected in a large computer cluster.
- We present a model for making fine-grained long-range prediction of the cluster's resource usage that relies on the state-of-the-art recurrent neural networks.
- We open-source the whole infrastructure that we have developed for data processing and modeling [9].

To the best of our knowledge, we are the first to investigate the utility of recurrent neural networks for predicting the resource usage in a computer cluster. In addition, we are the first to address this prediction at the level of individual tasks executed in the cluster and multiple steps ahead, providing the resource manager with more detailed and foresighted information and allowing for more elaborate management of the cluster.

III. Problem and Solution

Consider a cluster of machines that is serving a stream of tasks which are distributed across the cluster by a resource manager. Each task consumes certain resources during its execution; examples include the CPU and memory usage over time. The cluster has an adequate monitoring facility deployed so that the resource usage of the tasks is at one's disposal.

The resource-usage trace of task i is defined as a sequence of equally spaced d-dimensional measurements in time, which we shall represent as the following tensor of size $l_i \times d$:

$$x_i = (x_{i0}, \ldots, x_{ik}, \ldots, x_{i,l_i-1}) \tag{1}$$

where x_{ik} is the measurement taken at time t_{ik}, and l_i denotes the length of the sequence. Such a sequence is called fine-grained data as it contains multiple measurements over the execution of the task as opposed to having only one aggregative measurement such as the average or maximum value.

Consider now task i and suppose that the current time is t_{ik}; an illustration for $k \in \{2, 3, 4\}$ is given in Fig. 1. This means that $x_{i0}, \ldots, x_{ik}$ are known. Given these previous values, our goal is to estimate its next h values, which we shall denote by $\hat{x}_{i,k+1}, \ldots, \hat{x}_{i,k+h}$; in Fig. 1, $h = 4$. Such an estimation is called a long-range prediction as it provides multiple future values as opposed to only one. This operation is to be performed for each active task of interest at any time moment of interest.

In order to attain the objective, we reside to learning from historical data. Specifically, it is assumed first that there is a data set of resource-usage traces available, and that these past resource-usage traces are representative of the future ones:

$$X = \{x_i : i = 0, \ldots, n-1\} \tag{2}$$

where n is the total number of traces, and x_i is as in (1). We apply machine learning to the data in order to construct an adequate model offline, and this model is then used in order to make predictions at runtime. We specifically aim at investigating the utility of the state-of-the-art in machine learning; to this end, we use recurrent neural networks [2].

It should be understood that, in order for learning to be possible, the resource-usage traces that we consider have to have a certain structure that could be extracted and used for intelligent prediction. An important question is whether real-life traces of this kind, at all, exhibit such a structure. Investigating this question is part of our objective in this work.

Let us now adumbrate our workflow, which is illustrated in Fig. 2. Given raw resource-usage traces, we first (pre)process them in order to make these data suitable for the subsequent computations. This part is explained in the next section, Sec. IV, and can be seen on the left of Fig. 2. The processed traces are then used in order to obtain an adequately trained predictive model. The modeling part is covered in Sec. V while the learning one in Sec. VI; the latter can also be seen on the right of Fig. 2. The above operations are to be undertaken offline while the obtained model is supposed to be used by the resource manager of the computer system at hand at runtime.

IV. Data Processing

Before we describe our data-processing pipeline, let us first give a brief introduction to the considered data set: the Google cluster-usage traces [10]. The traces were collected over 29 days in May 2011 and encompass more than 12 thousand machines serving requests from more than 900 users.

In the data set's parlance, a user request is a job; a job comprises one or several tasks; and a task is a Linux program to be run on a single machine. Each job is ascribed a unique ID, and each task is given an ID that is unique in the scope of the corresponding job. Apart from other tables, the data set contains the so-called resource-usage table. The table records the resource usage of the executed tasks with the granularity of five minutes. Each record corresponds to a specific task and a specific five-minute interval, and it provides such measurements as the average and maximum values of the CPU, memory, and disk usage. There are more than 1.2 billion records, which correspond to more than 24 million tasks or, equivalently, resource-usage traces and to more than 670 thousand jobs.

The resource-usage table is provided in the form of 500 archives. Each archive contains a single file with measurements over a certain time window. Such a format is inconvenient and inefficient to work with, which is what we address in this section. Consider now the three leftmost boxes in Fig. 2.

A. Grouping

At the first stage, the data from the 500 archives are distributed into separate databases so that each such database contains all the data points that belong to a particular job, resulting in as many databases as there are jobs. In order to reduce the space requirements, only the used columns of the table are preserved. In our experiments, these columns are the start time stamp, job ID, task ID, and average CPU usage.

B. Indexing

At the second stage, an index of the traces is created in order to be able to efficiently navigate the catalog of the databases created at the previous stage. Each record in the index contains metadata about a single task, the most important of which are the task ID and the path to the corresponding database. We also include the length of the trace in question into the index in order to be able to efficiently filter the traces by their lengths.

C. Selection

At the last stage of our pipeline, a subset of the resource-usage traces is selected using the index according to the needs of a particular training session (to be discussed in Sec. VII) and then stored on disk. Concretely, the traces are fetched from the databases and stored in the native format of the machine-learning library utilized; we shall refer to a file in such a format as a binary file. Instead of writing all the selected traces into one binary file, we distribute them across several files. Such a catalog of binary files is created for each of the three commonly considered parts [1] of the data at hand: one is for training, one for validation, and one for testing; see also Fig. 2. We shall refer to these parts as X_1, X_2, and X_3, respectively.

Lastly, it is common practice to standardize data before feeding them into a machine-learning algorithm [1]. In our case, it is done along with creating the aforementioned three catalogs, which requires a separate pass over the training set.

In conclusion, the benefit of the above pipeline is in the streamlined access to the data during one or multiple training sessions. The binary files can be read efficiently as many times as needed, and they can be straightforwardly regenerated whenever the selection criteria change; note that the artifacts of the procedures in Sec. IV-A and Sec. IV-B stay the same. The presence of multiple binary files allows also for shuffling the training data at the beginning of each training epoch.

Now we are ready to make use of the processed data.

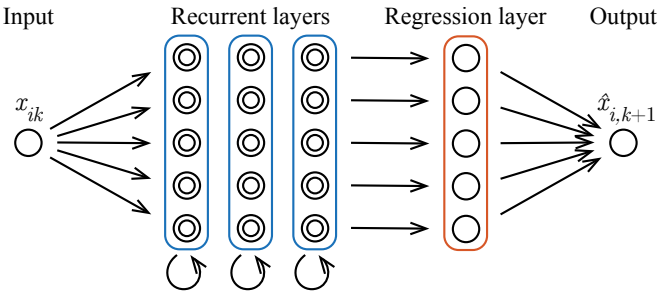

Figure 3. Schematic representation of our predictive model.

V. Modeling

As mentioned in Sec. I and Sec. III, a part of our goal is to assess the applicability of the latest advancements in neural networks [2] to modeling and prediction of fine-grained resource-usage data. The architectures of neural networks are very diverse: one network can be nothing like another. Since the data that we study are inherently sequential, it is natural to found our model on the basis of recurrent neural networks [2], which are specifically designed for such cases.

A schematic representation of our model can be seen in Fig. 3; many of the actual connections between the model's parts are simplified or not shown at all in order to make the figure legible. In the following subsections, we shall describe each part in detail. A number of important operational aspects of the model will be covered in the next section, Sec. VI.

A. Input and Output

The input to the model is a single d-dimensional data point, which can be seen on the left-hand side of Fig. 3. Similarly, the output is a single d-dimensional data point, which is depicted on the right-hand side of Fig. 3. The input x_{ik} is the value of the resource usage of an individual task at step k, and the output $\hat{x}_{i,k+1}$ is a one-step-ahead prediction of the usage.

B. Recurrent Layers

The core of the model is formed by a number of recurrent layers, which are also called cells. They are shown as a group

of blue boxes in Fig. 3. The network can be made as many layers deep as needed. Each cell is composed of a number of units, which are depicted by double circles in Fig. 3. We let c and u be the number of cells and units per cell, respectively.

A unit is the smallest processing elements. The key characteristics of a unit are that it has internal memory, and that it has access to its previous output, which makes it recurrent. There are different types of units; each one defines how a unit lets data flow through it and updates its memory. One notable type is called LSTM [11], which stands for *long short-term memory*. It has been designed to overcome the problems of traditional recurrent networks—such as vanishing gradient when training—and it is now one of the most widely used types. The recurrent layers of our model are LSTM cells.

In addition, during training, each cell is enhanced by a dropout mechanism [12], which gives control over the regularization of the model and is to prevent potential overfitting [1]. We let p be the probability of dropping an output of a cell.

C. Regression Layer

The output of the last cell is typically a large tensor, which is proportional in size to the number of units in the cell. Each entry of such a tensor can be considered as a feature that the network has extracted and activated in accordance with the trace that is currently being fed into the model. The task now is to combine these features in order to produce a single prediction. To this end, we mount a regression layer on top of the last cell, which is depicted by a red box in Fig. 3. Unlike the recurrent layers in Sec. V-B, which feature highly nonlinear transformations, this layer performs an affine transformation.

To summarize, we have described a predictive model that is composed of a number of LSTM cells and a linear regression layer. Due to its internal memory, the model is capable of efficiently taking into account the entire past of the resource-usage trace under consideration when making predictions. Let us now discuss how the model is supposed to be used.

VI. Learning

The output of the data-processing pipeline in Sec. IV is the data set X, which is split into three parts: X_1 is for the training stage, X_2 for the validation stage, and X_3 for the testing stage. We now elaborate on the operations that take place during these three stages, which are also displayed in Fig. 2.

A. Training

The model in Sec. V has a large number of parameters that have to be learned during training; they are primarily various weights and biases inside the layers. For this purpose, X_1 is utilized. The training is undertaken via backpropagation through time using stochastic gradient descent [2] whose objective is to minimize a certain loss function, which we shall specify shortly. There are two aspects to be discussed first.

The first concerns the way a single resource-usage trace is fed into the model. To begin with, the internal memory is nullified before feeding a trace. Next, note that a trace, as in (1), has multiple data points ($l_i > 1$), and that two traces are likely to have different lengths ($l_i \neq l_j$) since the execution times of two tasks are likely to differ. With this in mind, all the points of a trace are fed in one pass using a technique called dynamic unrolling. An illustration for $l_i = 4$ is given in Fig. 4,

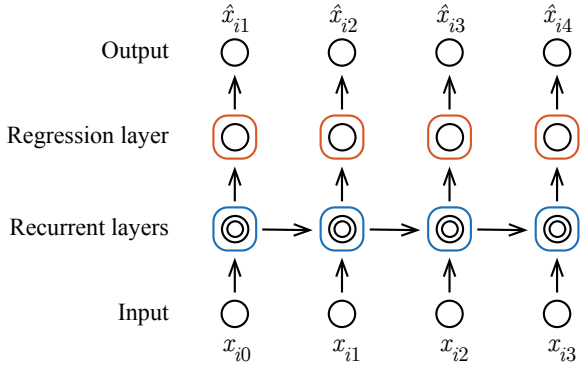

Figure 4. Example of dynamic unrolling during the model's usage.

in which the representation in Fig. 3 has been simplified even further and rotated 90° counterclockwise. It can be seen that the model has been replicated as many times as there are data points in the trace. However, it is still the same model, and all the replicas share the same parameters and internal memory. It can also be seen in Fig. 4 how information flows from one time step to the next, which is what makes the model recurrent.

Now, it is not efficient to feed only one trace at a time due to the inevitable overhead imposed by the computations involved. Therefore, these computations should be performed in batches whenever possible. Since $l_i \neq l_j$ in general, it is not possible to stack multiple arbitrary traces into a single tensor directly. In order to circumvent this problem, we reside to bucketing. Specifically, each read trace is put into one of many queues depending on its length. When a queue, accumulating traces of length from some l' to l'', has collected the desired number of traces—denoted by b and referred to as the batch size—it pads traces shorter than l'' with zeros and emits a tensor of size $b \times l'' \times d$ to be further consumed by the model.

The loss function that we minimize during training is the mean squared error (MSE) [1] of one-step predictions over the whole batch. The correct prediction for the very last time step, which goes beyond the time window of the traces in question, is assumed to be zero. For instance, in Fig. 4, $\hat{x}_{i4}$ has no x_{i4} to be compared with; x_{i4} is assumed to be zero.

B. Validation

As it is the case with arguably all nontrivial machine-learning models, the one presented in Sec. V has a number of hyperparameters. They include the number of cells c, number of units per cell u, and probability of dropout p, which are introduced in Sec. V-B. Unlike ordinary parameters, which are to be optimized during training (see Sec. VI-A), the hyperparameters are to be set prior to training and kept unchanged thereafter. The impact of the hyperparameters is profound; therefore, they should be carefully selected.

The validation set X_2 is used to assess the performance of the model trained (using X_1 as usual) under different configurations of the hyperparameters of the model. As before, the error metric utilized is the MSE, and it is beneficial to perform these computations in batches. The trained model that has the best performance on X_2 is then chosen as the one to be used.

Despite all the techniques employed to speed up training, it is still a time-consuming operation. As a result, brute-force search in the space of hyperparameters for the best configuration is impractical; a certain intelligent strategy should be followed.

In our workflow, we use the Hyperband algorithm [13]. Instead of adaptively choosing new configurations to evaluate—which is the case with many algorithms of this kind—it adaptively allocates resources to configurations chosen at random, which has been shown to be a very efficient strategy. In particular, the algorithm allows one for saving a lot of compute power, which otherwise would be burnt in vain evaluating overtly inadequate configurations of hyperparameters. In this context, *resources* refers a user-defined measure of how extensively a configuration is exercised. For instance, it can be the amount of wall-clock time spent or the number of training steps taken; in our experiments in Sec. VII, we use the latter.

C. Testing

After a trained model has been selected during the validation stage, it has to be assessed anew [1]: one cannot aver that the error with respect to X_2 is a good estimate of the generalization error because the selection was biased (we deliberately chose the configuration that had the best performance on X_2).

In order to have an unbiased evaluation, the testing set X_3 is utilized. As it is with training and validation, the MSE is considered as a quality metric, and the bucketing mechanism is used (see Sec. VI-A). However, unlike training and validation, the error is calculated in a more elaborate way as follows.

Recall first that our objective is making long-range predictions of resource usage (see Sec. III). Note also that, in Sec. VI-A and Sec. VI-B, we are only concerned with what happens one time step ahead. The reason is that we would like to have as high throughput as possible since the training and validation operations are to be performed many times. Testing, on the other hand, is done only once, and it is during testing we make and assess multiple-step-ahead predictions.

In order to compute long-rage predictions, we use refeeding: at time step k, the predicted value $\hat{x}_{i,k+1}$ is fed into the model as if it was the actual resource usage $x_{i,k+1}$ at step $k+1$, which is not yet known at step k. It might be helpful to consider the example given in Fig. 1. The process continues until all the desired h future values are estimated. It is natural to expect that the more accurate the one-step-ahead prediction is, the more accurate the multiple-step-ahead one will be.

There is more to it. Consider how a trained model will be used in practice. Potentially at each step k, one might want to predict the next h values of the resource usage of task i, that is, $\hat{x}_{i,k+1}, \ldots, \hat{x}_{i,k+h}$. Therefore, in order to test the model properly, we have to sweep over all the time steps of the trace in question while making h predictions at each step. An important aspect to note here is that the state of the model's memory should be saved before computing $\hat{x}_{i,k+1}, \ldots, \hat{x}_{i,k+h}$ at step k and restored before feeding $x_{i,k+1}$ in order to advance to step $k+1$. The memory becomes contaminated when one feeds predictions instead of observations into the model.

At this point, the main aspects of our workflow, which is illustrated in Fig. 2, have been discussed. The output of the workflow is a predictive model that has been trained on X_1, validated on X_2, and tested on X_3.

VII. Experimental Results

The infrastructure developed for the experiments presented below is open source and available online at [9]. The implementation is based on TensorFlow [14]. The experiments are conducted on a GNU/Linux machine equipped with 8 CPUs Intel Core i7-3770 3.4 GHz, 16 GB of RAM, and an HDD of 500 GB. The machine has no modern GPUs; therefore, the reported results have an immense room for improvement.

A. Data Processing

Recall that the considered data set is the Google cluster-usage traces [10] described in Sec. IV. In the experiments, we focus on one particular resource ($d = 1$), which is the CPU usage of the tasks executed in the cluster. The resource-usage table contains two apposite columns: the average and maximal CPU usage over five-minute intervals; we extract the former.

The grouping and indexing steps of the data-processing pipeline described in Sec. IV-A and Sec. IV-B, respectively, and depicted in Fig. 2 take approximately 57 hours to finish (no parallelism). Since they have to be done only once, their computational cost can be safely considered negligible.

Regarding the selection stage delineated in Sec. IV-C, we filter those resource-usage traces that contain 5–50 data points; consequently, $l_i \in [5, 50]$ in (1). Such traces constitute around 74% of the total number of traces (around 18 out of 24 million). We experiment with a random subset of two million traces, which is around 11% of the 5–50 resource-usage traces; hence, $n = 2 \times 10^6$ in (2). The data sets X_1, X_2, and X_3 constitute 70%, 10%, and 20% of X, respectively. Fetching and storing on disk these many traces take approximately four hours. Recall that this operation has to be repeated only when the selection criteria change, which happens rarely in practice.

B. Learning

The training stage (see Sec. VI-A and recall Fig. 2) is configured as follows. Ten buckets or, equivalently, queues are used according to the following rule: $l < 6 < 7 < 8 < 9 < 10 < 15 < 20 < 30 < 40 \leq 50$. The batch size b is set to 64. The optimization algorithm that is employed for minimizing the loss function is Adam [15], which is an adaptive technique. The algorithm is applied with its default settings.

Regarding the validation stage, the considered hyperparameters are the number of cells c (the blue boxes in Fig. 3), number of units per cell u (the double circles in Fig. 3), and probability of dropout p as discussed in Sec. VI-B. More concretely, we let $c \in \{1, 2, 3, 4, 5\}$, $u \in \{100, 200, 400, 800, 1600\}$, and $p \in \{0, 0.25, 0.5\}$, which yields 75 different combinations in total. The candidates are explored by means of the Hyperband algorithm introduced in Sec. VI-B with its default settings. The maximum budget for one configuration is set to four training epochs, which correspond to $4 \times 0.7 \times 2 \times 10^6 = 5.6 \times 10^6$ resource-usage traces or $5.6 \times 10^6 \div 64 = 87\,500$ training steps.

The above exploration, which encompasses both the training and validation stages, takes approximately 15 days to finish. During this process, we run up to four training sessions in parallel, which typically keeps all the eight CPUs busy. It should be noted that, since the training, validation, and testing data sets have been cached on disk as a result of our data-processing pipeline described in Sec. IV, individual training sessions do not have any overhead in this regard.

The results of the validation stage are given in Table I, which shows the mean squared error (MSE) of the top 10 configurations of the hyperparameters as measured using X_2

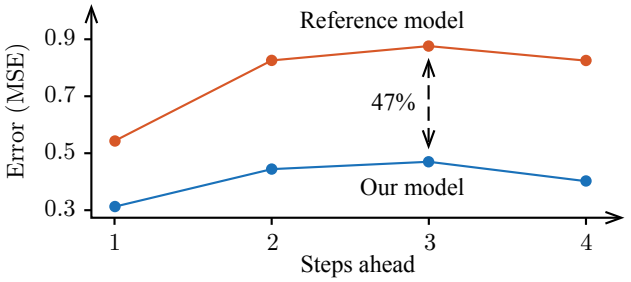

Figure 5. Accuracy of predictions up to four steps ahead by our model (blue) and by the reference one (red) with respect to the testing set X_3.

($0.1 \times 2 \times 10^6 = 2 \times 10^5$ traces). The best trained model is found to have the following hyperparameters: $c = 3$, $u = 1600$, and $p = 0$. In general, deeper and wider architectures tend to outperform shallower and narrower ones (the depth and breadth are measured by c and u, respectively), which is expected. In these experiments, the dropout mechanism does not have much impact on the resulting accuracy, which is likely due to the amount of data being enough for regularizing the model.

Table I also shows an estimate of the memory required by each configuration, including the model's trainable parameters and internal state. It can be seen that, if the memory usage is a concern, one could trade a small decrease in accuracy for a considerable memory saving. For example, the fourth best configuration requires 75% less memory than the first one.

After the exploration stage, the best trained model is taken to the testing stage (see Sec. VI-C), which is undertaken using X_3 ($0.2 \times 2 \times 10^6 = 4 \times 10^5$ traces). At this stage, the model is extensively assessed by predicting the resource usage of individual tasks multiple time steps ahead at each step of the testing traces in X_3. In these experiments, we predict four steps into the future ($h = 4$ in Sec. III). This elaborate sequential testing procedure takes around 18 hours from start to finish.

In order to assess better the accuracy of our model, we employ also an alternative one, which we shall refer to as the reference model. The reference model is based on random walk. It postulates that the best prediction of what will happen tomorrow is what happens today plus an optional random offset, which we set to zero. In other words, the next value of a resource-usage trace is estimated to be the current one, which results in four identical predictions at each time step.

The results of the testing stage can be seen in Fig. 5, which shows the MSE of our model (the blue line) as well as the one of the reference model (the red line) with respect to X_3. The magnitude of our model's errors suggests that the amount of regularity present in the data is not sufficient to make the resource-usage predictions highly accurate. Nevertheless, it can be seen in Fig. 5 that, relative to the reference model, our model provides an error reduction of approximately 47% at each of the four future time moments. This observation indicates that a certain structure does exist, and that it can be identified and utilized in order to make educated predictions.

VIII. Conclusion

We have presented our experience of working with the state-of-the-art recurrent neural networks in the context of fine-grained long-range prediction of the resource usage in a computer cluster. Our workflow—which starts from making the data readily available for learning and finishes by predicting the resource usage of individual tasks multiple time steps into the future—has been described in detail and applied to a large data set of real resource-usage traces of a computer cluster.

The experimental results suggest that the considered fine-grained traces possess a certain structure, and that this structure can be extracted by advanced machine-learning techniques and subsequently utilized for making educated predictions. This information can be of use to such a crucial component as the resource manager of the computer cluster in question, allowing the manager to more intelligently orchestrate the cluster.

Table I
Validation results (top 10 configurations)

Rank	c	u	p	Error (MSE)	Memory (MB)
1	3	1600	0.00	0.3148	198
2	4	1600	0.00	0.3154	277
3	3	1600	0.25	0.3158	198
4	2	800	0.25	0.3194	30
5	5	200	0.00	0.3205	6
6	2	1600	0.05	0.3207	119
7	5	800	0.00	0.3251	90
8	3	800	0.00	0.3257	50
9	1	1600	0.25	0.3278	40
10	2	800	0.00	0.3316	30

References

[1] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: Data mining, inference, and prediction*, 2nd ed. Springer New York, 2009.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[3] A. Coskun, T. Rosing, and K. Gross, "Proactive temperature management in MPSoCs," in *International Symposium on Low Power Electronics & Design*, 2008, pp. 165–170.

[4] P. Kumar and D. Atienza, "Neural network based on-chip thermal simulator," in *IEEE International Symposium on Circuits and Systems*, May 2010, pp. 1599–1602.

[5] D.-C. Juan, S. Garg, and D. Marculescu, "Statistical peak temperature prediction and thermal yield improvement for 3D chip multiprocessors," *ACM Transactions on Design Automation of Electronic Systems*, vol. 19, no. 4, pp. 39:1–39:23, August 2014.

[6] M. Dabbagh *et al.*, "Energy-efficient resource allocation and provisioning framework for cloud data centers," *IEEE Transactions on Network and Service Management*, vol. 12, no. 3, pp. 377–391, September 2015.

[7] S. Ismaeel and A. Miri, "Using ELM techniques to predict data centre VM requests," in *IEEE International Conference on Cyber Security and Cloud Computing*, November 2015.

[8] J. Cao *et al.*, "CPU load prediction for cloud environment based on a dynamic ensemble model," *Software: Practice and Experience*, vol. 44, no. 7, pp. 793–804, July 2014.

[9] (2017, September) Source code, configuration files, and input data. Embedded Systems Laboratory at Linköping University. [Online]. Available: https://www.ida.liu.se/~ivauk83/research/PRU

[10] C. Reiss, J. Wilkes, and J. Hellerstein, "Google cluster-usage traces: Format + schema," Google, Tech. Rep., November 2011. [Online]. Available: https://github.com/google/cluster-data

[11] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, November 1997.

[12] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," *CoRR*, vol. abs/1409.2329, 2014. [Online]. Available: http://arxiv.org/abs/1409.2329

[13] L. Li *et al.*, "Efficient hyperparameter optimization and infinitely many armed bandits," *CoRR*, vol. abs/1603.06560, 2016. [Online]. Available: http://arxiv.org/abs/1603.06560

[14] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2016. [Online]. Available: http://arxiv.org/abs/1603.04467

[15] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980