Summer 2020

Cloud Resource Prediction Using Explainable and Cooperative Artificial Neural Networks

Nathan R. Nelson
Missouri State University, Nathan0@live.missouristate.edu

CLOUD RESOURCE PREDICTION USING EXPLAINABLE AND COOPERATIVE ARTIFICIAL NEURAL NETWORKS

A Master's Thesis

Presented to

The Graduate College of

Missouri State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science, Computer Science

By

Nathan Nelson

August 2020

CLOUD RESOURCE PREDICTION USING EXPLAINABLE AND COOPERATIVE ARTIFICIAL NEURAL NETWORKS

Computer Science

Missouri State University, August 2020

Master of Science

Nathan Nelson

ABSTRACT

This work proposes a system for predicting cloud resource utilization by using runtime-assembled cooperative artificial neural networks (RACANN). RACANN breaks up the problem into smaller contexts, each represented by a small-scale artificial neural network (ANN). The relevant ANNs are joined together at runtime when the context is present in the data for training and predictions. By analyzing the structure of a complete ANN, the influence of inputs is calculated and used to create linguistic descriptions (LD) of model behavior, so RACANN becomes explainable (eRACANN). The predictive results of eRACANN are compared against its prototype and a single deep ANN (DNN). The DNN is shown to outperform eRACANN in terms of accuracy, though eRACANN shows specialized ANN topologies facilitate more specific LDs than singular DNNs.

KEYWORDS: cloud, resource prediction, machine learning, neural networks, explainable, cooperative agents, regression

CLOUD RESOURCE PREDICTION USING EXPLAINABLE AND COOPERATIVE ARTIFICIAL NEURAL NETWORKS

By

Nathan Nelson

A Master's Thesis
Submitted to the Graduate College
Of Missouri State University
In Partial Fulfillment of the Requirements
For the Degree of Master of Science, Computer Science

August 2020

Approved:

Ajay Katangur, Ph.D., Thesis Committee Chair

Siming Liu, Ph.D., Committee Member

Jamil Saquer, Ph.D., Committee Member

Julie Masterson, Ph.D., Dean of the Graduate College

ACKNOWLEDGEMENTS

I would like to thank Bitbrains IT Services Inc. for providing cloud utilization traces, the Grid Workloads Archive for hosting it, and S. Shen, V. v. Beek and A. Iosup for the analysis of that trace on which the prototype was trained. I would also like to thank Google for releasing cluster trace data and C. Reiss, A. Tumanov, G. Ganger, R. Katz, and M. Kozuch for the schema of that data on which the proposed system was trained.

I dedicate this thesis to my wonderful wife who had to deal with me throughout the process of realizing this work. I also dedicate it to the diligent and caring faculty of the Department of Computer Science at Missouri State University who also had to deal with me, however fortunately, in smaller quantities.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Predicting system resource utilization in cloud-based computing and software services, collectively "clouds," is an important area of study due to the desire for highly available online services from both consumers' and businesses' perspectives. Modern cloud services are expected to be accessible wherever the user is, leading to not just an increase in service demand, but increased stress on the physical and virtual resources used by those services. Service providers must allocate these resources ahead of time or suffer service degradation. They must also not leave excess resources idle or, especially for large systems, they are wasting large amounts of electricity. Resource utilization prediction is the area concerned with solving this problem.

The literature has many potential solutions. Some of these are simple, statistical models [1]. Others form an ensemble of statistical and machine learning-based (ML) approaches to try to get the best of both worlds or to correct for known biases [2]. ML, in particular, sees much application in and out of cloud computing to model complex systems, especially with artificial neural networks (ANN) [3]. However, ANNs' hidden layers abstract away the direct relationship between their inputs and outputs, making for models with a potentially accurately learned correlation but not for any particular reason. That is, these models are black boxes [4], [5]. With some systems being composed of large-scale deep ANNs (DNN), it quickly gets difficult or impractical to understand and implement them. In a word, these models are unexplainable, both in implementation and in result.

Because of this, there is a need for resource prediction by models with justifiable construction, which accurately forecast utilization, and can provide reason for their output. This

research introduces a system of runtime-assembled cooperative ANNs (RACANN), which is built from easy to understand components and uses common techniques found across ML literature in a novel way. Time is taken to not only justify each component of the system, but it is specifically done with respect to the properties and limitations of ANNs.

RACANN breaks up the problem of predicting resource utilization into contexts—categorical scopes of data—and represents each context with a cooperative agent; in this case, the agent is a small-scale ANN. They are cooperative in the sense that when the system is training or making predictions, only the agents that handle the characteristics of the current data are joined together to see it. This approach is initially explored using a simple prototype and tested on data with known usage trends. Its success leads to fully realizing RACANN and testing on very noisy data for robustness. The system is then extended with explainability (eRACANN) so its behavior can be described linguistically. The error of eRACANN is compared to a DNN trained on the same data to gauge its efficacy along with an empirical comparison of graphed predictions. The different restrictions on describing eRACANN and the DNN with generated linguistic descriptions (LD) is also elaborated.

Results show the error of the DNN outperforming eRACANN though their prediction outputs are comparably useful. The DNN, however, ends up being more limited when it comes to generating LDs for its behavior. Since the DNN trains over all the data, it has a global context, and one would have to exhaustively test the validity of every possible combination of descriptions to get anything more specific. Conversely, LDs for eRACANN inherit the context of the ANNs used to make the predictions, allowing them to be automatically more specific. Thus, eRACANN has more to offer in terms of generating context-specific LDs, but less in terms of predictive power.

The context for the problem and the necessary background information are covered in section 2. Section 3 summarizes similar work in the area of resource prediction in cloud computing and explainability. The simple model that eRACANN is based on is discussed in section 4 since it served as proof of concept. eRACANN is thereafter described and extended with explainability in section 5 with its predictive effectiveness analyzed and compared to the prototype and a DNN in section 6. Possible extensions to eRACANN are deliberated in section 7 and the contribution of the work is concluded in section 8.

2. BACKGROUND

To better illustrate the way all of these background components comprise the problem domain and influence the solution presented in this work, they will be presented with respect to a fictitious company, Report Corp. First, an overview of the services offered by this company is presented alongside the way they are provided to consumers and internal employees to set up an exemplary use case for eRACANN. The components of these services are then broken out to explore how they parallel typical cloud service models. Because eRACANN is built on ANNs, the background necessary to justify its construction is explored after that. Common terms used across cloud computing and related research are elaborated throughout to aid in understanding the related works and eRACANN, thereafter.

2.1. Cloud-Based Services

Consider the fictitious company Report Corp., which offers several services to internal employees over its local network and to external clients over the internet. Management is interested in having a special set of servers regularly generate and send various reports to their email addresses every morning. Some external clients pay to use a suite of online tools in a special web interface, typically called a portal. Other clients, still, are on a subscription plan where they rent digital resources like processing power and storage space. These are three different systems, each with a specific purpose and regular but distinct network traffic, or utilization patterns.

In the broadest scope, the collection of their services accessible over the internet comprise their cloud. It is also possible to identify each separate system as its own cloud, and the

collection thereof as a "cloud of clouds." When the cloud provides an interface to software, typically accessed via web browser or mobile application, this is called Software as a Service (SaaS, read "sass"). Report Corp. customers accessing the web portal are receiving SaaS. The customers renting computing power and storage space, e.g. in the form of a VM on Report Corp. hardware, are receiving Infrastructure as a Service (IaaS). In this way, there are many things that can be offered "as a Service." Without regard to any one service, one writes *aaS, as whatever service or services fit the context [6].

At Report Corp. there is an underpaid system administrator (sysadmin) who has been tasked with the additional responsibility of monitoring these three systems and provisioning more or fewer resource across them as demand fluctuates. A smart sysadmin would want to provision system resources so demand does not overwhelm its current capacity, but also so it does not sit largely un-utilized. The most obvious way to accomplish this is just turn physical machines on and off. There is also the option of activating virtual machines (VM) to make its portion of the physical system's CPU, RAM, disk, etc. available to consumers as part of a service, e.g. resulting in a more responsive portal. If the sysadmin noticed there were many resources sitting idle, VMs could be suspended to free the previously allocated system resources. Provisioning resources in anticipation of future need is known as a proactive strategy; the opposite of this, allocating resources after knowing demand, is a reactive strategy and is less desirable because at that point the consequences of having too few resources, however severe, have already occurred.

Over and under-utilization is a real problem. If many of a system's VMs are overwhelmed and there is a high degree of competition for the computing resources necessary to complete clients' jobs, the VMs are experiencing resource starvation. Starvation causes

performance degradation for all resident services, also called tenants, on the starved system. Degradation to the point of violating written agreements with customers defining acceptable service quality, called service level agreements (SLA), will cost Report Corp. their clientele, which is money walking out the door and over to the competition. The obvious but hasty reaction to this is to have many resources provisioned all the time so resource starvation becomes as improbable as possible. Doing this, however, will leave many systems under or even completely un-utilized for large periods of time, which translates directly into an unnecessarily massive electrical bill for Report Corp. Therefore, it would be best if the sysadmin could not only proactively allocate resource, but have a tool to guide the decision-making process.

2.2. Perceptrons and Learning

The classical Rosenblatt perceptron is a binary classification mechanism represented by a computational unit that receives stimuli (input) and produces either 1 or 0, interpreted to be indicative of the input's membership to one class [7], [8]. In its general form, however, it can be extended to identify multiple classes and calculate regressions. This is accomplished by computing a linear combination σ of the n-many inputs $\{x_1, x_2, \ldots, x_n\}$ multiplied by their associated weights $\{w_1, w_2, \ldots, w_n\}$ as in (1). The sum then serves as the input to a mapping function φ, more typically called the activation function, to produce the output of the perceptron, y as in (2). Perceptrons also usually have an additional bias input $b = 1$ with a weight $w_b = 1$ to add stability to the perceptron's learning process [9], but it is possible for either of those values to be updated as the perceptron is trained.

$$\sigma = \left(\sum_{i=1}^{n} w_i x_i \right) + w_b b \tag{1}$$

$$y = \varphi(\sigma) \tag{2}$$

The most appropriate φ is dependent on the application. A perceptron with one output uses a threshold function (3) to facilitate binary classification. If one replaces the threshold function with a continuous function, then the perceptron can fit a regression of the shape and degree of that function. By adding an extra output node, the perceptron is now suitable for modeling probabilistic classification for two classes. It is clear there are many configurations for a perceptron but not all are appropriate for every problem.

$$f(x) = \begin{cases} 1, & x > \text{threshold} \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

Having one set of inputs connected to one set of outputs is called a single layer perceptron (SLP). Therein, an individual unit (e.g., an input or output) is referred to as a node. Figure 1 illustrates this concept with the output node enlarged to show the steps of summation (Σ) and activation (φ) that result in the output value y. Note that inputs and outputs of SLPs are commonly described as being in their own "layers," but the "single layer" part of "SLP" is with respect to the number of layers of perceptrons.

Perceptron weights do not start magically tuned to values that make it produce the correct y. A common practice is to initialize each w using a random function with a range of $(0,1)$, but many other methods exist, such as gaussian and normal (distribution) functions. The process of making those w more correct is called training. This is implemented by supplying training data to the inputs, calculating how far off y was from the desired (correct) value d, and using that value to update the weights. The difference between y and d is called the error or the loss. Like the activation function, the most appropriate way to calculate the error is dependent on the application, but the goal is always to reduce it. Binary classifiers might use binary cross entropy

to measure loss; a regression perceptron could use the mean squared error (MSE), mean absolute error (MAE) (4), or the root mean squared error (RMSE) (5).

$$MAE = \frac{\sum_{i=1}^{n} |y_i - d_i|}{n} \tag{4}$$

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{(y_i - d_i)^2}{n}} \tag{5}$$

Once the error has been calculated, it can be used to update the weights between the input and output layer via backpropagation. The extent to which the weights are updated must be tempered, though, or the model may overcorrect and "learn" to be just as wrong as it was before, if not worse. The magnitude of correction is controlled by the learning rate α. This is typically a small value in the range $(0, 1)$ and close to 0 like 0.001. One of the most generally applicable ways to update weights based on the error is via gradient descent. This method backpropagates (sends backward) a portion of the error (credit) through each previous layer of the ANN by using partial derivatives. Gradient descent is overkill for a simple SLP, though, so (6) will suffice by computing Δw_i, the value to add to the weight of input i, which is the error tempered by the learning rate. If the error does not change for better or worse for a while during training despite updating weights, the model is said to have converged. This can also be thought of as the value of the error appearing to have reached an asymptote on a graph. The opposite of this is divergence.

$$\Delta w_i = \alpha(d - y) \tag{6}$$

The power of SLPs is limited by the chosen φ and that the output node is only capable of linear combinations. In the case of Rosenblatt classifiers, data that is not linearly separable cannot be accurately modeled. To illustrate, if one were to plot all the input data in two

dimensions, there would have to be a straight line capable of accurately separating it into two discrete spaces, that is, the two correct classes. This limit can be partly overcome by using a non-linear φ of an appropriate degree, but then the model is being told the appropriate shape of the line that separates the classes, called the decision boundary, rather than learning it. A classic foil for the Rosenblatt classifier is the XOR problem. By plotting the truth table for $A \oplus B$ (Figure 2) it becomes clear the two classes $True$ $(\boldsymbol{T})$ and $False$ $(\boldsymbol{F})$ cannot be separated by a single straight line.

2.3. Multilayer Perceptrons

A multilayer perceptron (MLP) is merely an extension to a SLP. Recall that a SLP has weighted inputs and an output layer. To make the SLP a MLP, one or more layers are added in between, called hidden layers. For any hidden layer, the inputs are the outputs of the previous layer and its outputs become inputs for the next layer (Figure 3). Activation takes place in hidden layers as well, so any output node computes the composition of all preceding φ.

One of the biggest advantages of MLPs over SLPs is their ability to estimate arbitrary continuous function; that is, they are universal approximators [10], [5]. The only provision to this is that φ is not linear, otherwise the computing power of the MLP collapses back down to a SLP as the composition of linear functions is a linear function. Because MLPs are universal approximators, one can model any degree polynomial but such is contingent on an appropriate ANN topology and convergence is not a guarantee. One such factor to consider in designing the ANN is the dimensionality of the input space. The number of inputs to an ANN indicates the number of dimensions required to represent the problem. This is most easily understood by starting back at the binary classification SLP with two inputs. The input space is two

dimensional (2D), enabling one to plot the inputs in a coordinate plane. The one output of the classifier produces a line (1D) that separates the classes, also easily plotted. In other words, the SLP models the function that maps the input space to the solution space. Typically, the solution space fits within the problem space, that is, it has fewer dimensions. This is seen in the previous example where the solution (a line) exists in fewer dimensions than the representation of the problem (a plane), but the location of the line may be different between solutions. Visualizing these things in the fourth dimensions and higher, though, requires projection into a lower dimension [5]. This explains the interesting shapes seen in decision boundaries that have been graphed for high-dimensional ANN solutions.

2.4. Deep Neural Networks

The term "deep" in DNNs simply refers to MLPs with more than one hidden layer (Figure 4). DNNs can be specialized by changing how data flows through the network and/or how individual nodes handle that data; some of these configurations have been given special names. Convolutional neural networks (CNN) and recurrent neural networks (RNN) are prime examples. The former has nodes that do much more than just calculate φ and the latter allows data to move both forward and backward through the network. DNNs are a popular choice for modeling complex problems because the extra hidden layers each add another layer of abstraction for internally representing the problem. This leads to some very accurate classifiers and regression networks, but makes it difficult to understand why the model is better. Typically, research is interested in accurate models, not minimal ones. So, the fact that DNNs generally outperform "shallow," two-layer MLPs is enough for most to abandon them in favor of deeper models.

2.5. Explainability

Explainable models have reasoning associated with either single predictions or overall behavior [4] and is one of the goals of explainable artificial intelligence (XAI) [11]. This can be accomplished by either designing a model so that it is explainable or via a process of extracting "knowledge" from one that already exists. Some motivations behind XAI include the inherent value of knowing why a model outputs something, like why driver-less car AI can be trusted or why a medical AI would suggest amputation. It could provide the scientific community with yet undiscovered insight or a trend the model discovered during its training phase that would have otherwise remained under the cover of the black box model. Also consider the inverse of providing positive insight; an XAI model could, by virtue of explaining its decision-making, reveal a flaw in the approach [12].

2.5.1. Layer-Wise Relevance Propagation. Aside from behavioral XAI, there is also value in a model being able to provide the importance of its inputs [11]. With that kind of information, input features' influence on the output can be determined so, e.g., a more accurate or minimal model can be made. In a SLP the weights between the input and output layers can be used as-is for determining feature contributions to the output. However, this direct relationship between features and outputs is lost when the ANN has hidden layers. So, other methods must be used to overcome the added abstraction.

Layer-wise relevance propagation (LRP) [13] is one such method that works by backpropagating node contributions, or relevance, from a layer $l + 1$ to the previous one l similar to the way gradient descent backpropagates error. A contrived, incomplete demonstration of LRP follows, given an ANN that approximates the function f of a set of input features $\{x_i\}$. The output node, which is the first and only node in layer M, produces $f(\{x_i\})$ and

its relevance R is equal to that value. The notation representing the relevance of the one node in the output layer M is given by $R_1^{(M)} = f(\{x_i\})$. The relevance of the previous layer j is $R^{(j)}$ and any particular neuron's relevance is $R_d^{(j)}$, where d is the d^{th} neuron in that layer, also read as the d^{th} dimension of layer j. $R_d^{(j)}$ is calculated using backpropagated messages composed of the decomposition of the function represented by the ANN at layer $l+1$ onto the neurons of layer l. This is generally written $R_{j\leftarrow k}^{(l,l+1)}$ [13]. By decomposing down to the input layer $l = 1$, $R_i^{(1)}$ can be realized, providing a function for individual input's relevance. The methods in [13] and [14], however, are for classification ANNs with ReLU activation in the hidden layer, so eRACANN will require new decompositions with respect to its logistic activation.

2.5.2. Fuzzy Sets and Logic. While having metrics indicating inputs' relevance to the output is useful, it is merely evidence in favor of the relationship rather than a description of its meaning. To illustrate, features like "number of cylinders," "engine capacity" and "zero-to-sixty time" to map cars to the output class, "gas-guzzler" are used by an exemplary binary classifier in [4]. Knowing that "engine capacity" is the most relevant input is useful but does not meaningfully describe its relationship with the dependent variable. Such would take the form of a description correlating one kind of engine capacity to what extent the car is a gas-guzzler; this is where fuzzy sets and fuzzy logic come in.

A fuzzy set, or class, is a space in which data points have a degree of membership. The domain of the class is called the universe U and the degree of membership has the range $[0, 1]$. A membership value of 0 indicates full exclusion from the class and value of 1 is full membership. These are called "crisp" values because they discretize membership into absolutes, yes or no. If 1 and 0 are the only possible membership values, the set itself is said to be crisp, or ordinary. The degree of membership in truly fuzzy sets, though, is a range, so a value between 0

and 1 describes partial or fuzzy membership. The function f that produces the degree of membership of a data point x in the fuzzy class A is expressed by the membership function $f_A(x)$ [15]. It is also commonly written $\mu_A(x)$ [4] [16].

The mathematical definition of f is relative to the application, and even then, is completely subjective. Let $U = \mathbb{R}$ and $A =$ "much greater than 1." The membership of $x \in U$ can be arbitrarily piecewise-defined by (7).

$$f_A(x) = \begin{cases} 0, & x \leq 1 \\ 0.1, & 1 < x < 10 \\ 1, & \text{otherwise} \end{cases} \tag{7}$$

Getting the degree of membership of x in A is called fuzzification. In fuzzy logic, A is called a fuzzy variable and the result of fuzzification is described linguistically. It is also the case that there exist multiple, partially overlapping classes to which x will have varying degrees of membership along the domain of U. If the classes do not overlap, the sets are crisp and there is no sense of fuzzy membership. As Type-1 fuzzy sets, these classes usually take one of a few simple shapes: most commonly, sigmoidal, triangular, or trapezoidal. In the case of triangular membership function $f_\triangle$, the domain of $\triangle$, a subset of U, can be described by a triple in the form (a, b, c). The first term a is where membership starts, but $f_\triangle(x) = 0$ at $x = a$. The second term b is the peak of the triangle where $f_\triangle(x) = 1$ and c mimics a but is where membership on the domain ends. Trapezoidal membership (a, b_1, b_2, c) is similar except the peak is constant from b_1 to b_2.

As an example, consider the universe of outdoor temperatures where U is bounded to $[0°\text{F}, 120°\text{F}]$ and membership functions are triangular (Figure 5). At some point, it definitely feels either hot or cold. But in between the two extremes are classes warm and cool for which the membership of the current temperature x varies depending on the person. The author, for

example, would describe the "warm" class with the triple $(65, 75, 85)$. However, it is possible for the lower end of "warm" to feel "cool," e.g. after a spring rain, so let "cool" be $(40, 55, 70)$. By comparing the fuzzified value for x in each of the two classes "warm" and "cool," one can ascertain which class the input most resembles. If the temperature is 68°F, $f_{\text{warm}}(68) = 0.3$ and $f_{\text{cool}}(68) = 0.13$. So, 68°F is considered warmer than it is cool. A temperature like 100°F, though, would be categorically "hot" and bear no resemblance to any other class.

2.6. Regression

The type of problem approached by this work is regression, that is, the goal is to design a MLP that learns to model a function that most accurately represents, or fits, a data set [17]. In this case the data set is resource utilization over time and the resulting model approximates a function that models the relationship between the two, however complicated. As stated before, SLPs are only capable of modeling regressions in the same degree or shape as their φ, which can be limiting. If one has a priori knowledge or perhaps a hunch that a data set contains a cubic trend and thus employs an ad hoc $\varphi(x) = x^3$, which would be considered abnormal, it will only decently converge if inputs actually conform to a third-degree function. Upgrading to MLPs unlocks arbitrary regressions provided the network is set up as described in the previous subsection "Multilayer Perceptrons," meaning the MLP will be capable of modeling the non-linear trends in the data without being told the shape of the trend in advance.

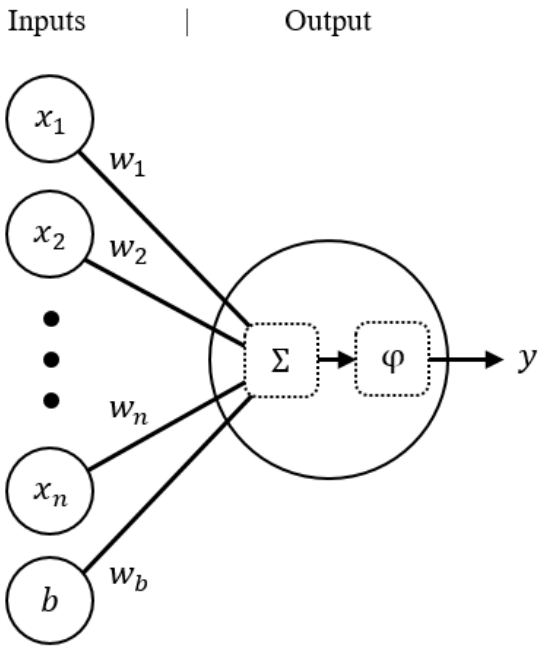

Figure 1. A SLP with n-many inputs, a bias, and one output node.

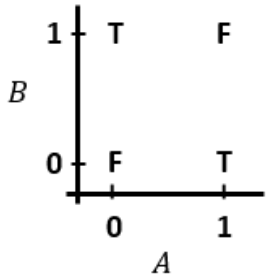

Figure 2. The truth table of $A \oplus B$ expressed in two dimensions.

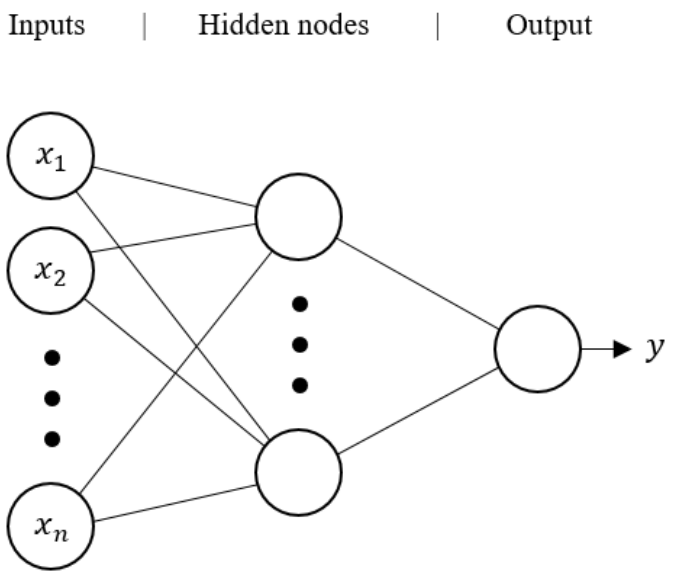

Figure 3. A MLP with an input, hidden, and output layer.

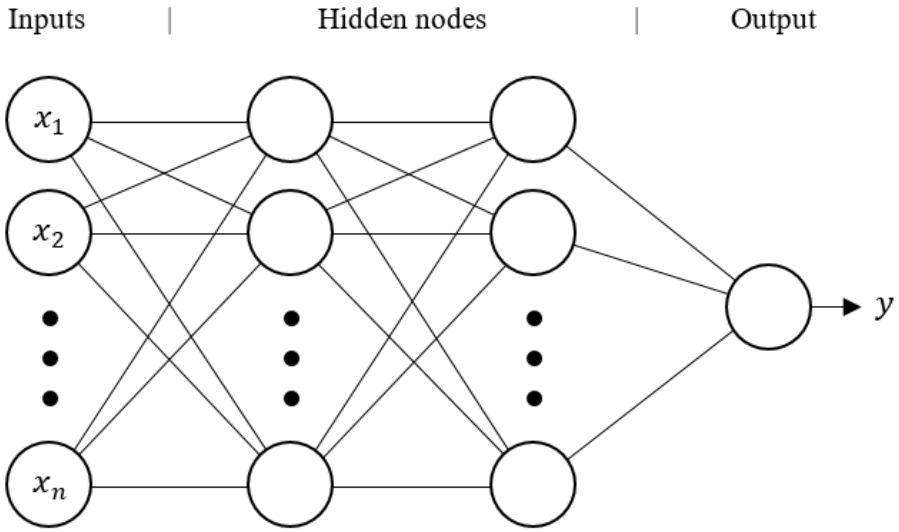

Figure 4. A DNN with two hidden layers.

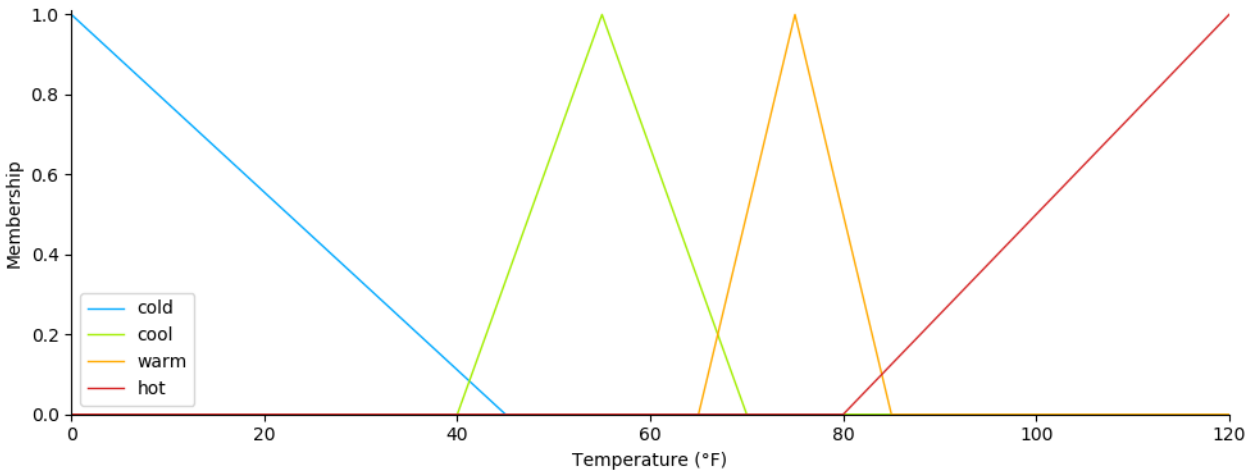

Figure 5. Outside temperature described by four fuzzy variables.

3. RELATED WORK

3.1. Resource Prediction in Cloud Computing

One of the primary motivators of this work is [3], the last in a series of three papers in which a system of five interconnected modules is proposed. Each system has a specific purpose contributing to the larger picture of autonomic resource provisioning in clouds. Therein, each module is composed of individual "units" or "components" responsible for a portion of the module's work. The first module is responsible for taking workload traces and predicting the expected volume of incoming requests at the next time step. The second module takes the output of the first and determines the resources necessary to accommodate such a workload (number of requests). These first two modules, combined, serve the same purpose as eRACANN, but are broken apart in DEARS to further discriminate and allocate sub-problems to modules' individual components. Subsequent modules take this metric, compare it to currently allocated resources, add (allocate) or subtract (deallocate) resources, and consolidate fragmented resources so fewer machines are running. The last module then verifies that these changes do not violate agreements with clients regarding this like service availability or performance, i.e. SLAs.

Of these modules, the most relevant one is the workload prediction module. It uses a DNN, an ANN with more than one hidden layer, to determine the prediction for the next time step. In fact, the employed DNN has 50 input nodes going into four hidden layers. The inputs are connected to 50 long short-term memory modules (LSTM) in the first hidden layer, 250 LSTMs the second, 250 regular nodes in the third (a "flattening" layer), and 2 regular nodes in the fourth. The output layer of the system also has 2 regular nodes representing the number of expected requests and the expected size of the response in megabytes per second. The output of

each LSTM layer is also followed by a dropout layer, the point of which is to "remove the potential strong dependency on one dimension so as to prevent overfitting" by providing a chance to disregard the output of the previous LSTM node. DEARS used a dropout mask/value of 0.2, so each LSTM output had a 20% chance to not feed its data forward.

The data set used in [3] was the FIFA world cup 1998 traces [18], which spans 3 months and has 1.35 billion requests. Preprocessing included feature extraction and normalization. The number of requests N and the size of the response R in megabytes per second were calculated to serve as inputs to each of the 50 LSTM inputs along with the next time step t so inputs took the form of a triple, $\{N, R, t\}$. Labels (desired outputs) for t were simply calculated as $\{N, R\}$ at the previous time step $t - 1$.

After preprocessing, the transformed data set contained 7.405×10^6 records and was split into 80% training and 20% validation data. Training used a batch size of 500,000 over 400 epochs. The resulting RMSE of the system depends on the perspective as the window of time in which the results are viewed changes the apparent magnitude of the error. On the two busiest days from the FIFA traces there were 17,217 and 35,575 requests with a RMSE on the predicted number of requests at 30.700 and 34.080, respectively. These errors were called "quite satisfactory" since an average error of about 30 requests compared to many thousands is, indeed, quite good.

In [19] an approach to resource prediction with a similar setup to this work is explored. Predictions are presented as being at the task-level and then at the resource-level, that is, how much of a resource will be needed and for long by a task. The experiment overlaps with this one in that a small MLP is used to solve the problem. Models consisted of 2 inputs, 10 hidden nodes in one layer, and 1 output node. The work also describes resource prediction as a problem that

can be broken into sub-problems. Here, the sub-problems are identifying tasks and correlating the amount of necessary resources to those tasks, whereas eRACANN compartmentalizes with regard to time and correlates utilization trends to discreet time units.

The data set used in the experiment is composed of 1 million continuous integration (CI) job traces from Travis CI and GitHub. These included information like the name of the project being built, the commit ID, build duration, and others. The features extracted from the data were repository, file count, and total size of the repository in bytes. The label for inputs these models is the build duration in seconds. With the build process for each repository considered a different task, the system built a ML model for each repository to predict the resources necessary to build the project. The data set was split into 70% training and 30% validation data. Models were trained with a batch size equal to the size of the data set over 250 epochs with a learning rate of 0.01.

The measure of success reported in [19] used an error ratio of the root mean square deviation (RMSD) (same formula as RMSE) of the MLPs to their base of comparison, linear regression. The values used to calculate the RMSD, however, were not normalized. With the linear regression having a RMSD of 10.4 seconds, the results of task-based MLPs were presented in three classes. The worst 5% have an error ratio of 2.30, that is, they performed 2.3 times worse than linear regression. The median error ratio was 0.80 and the best 5% came in at 0.11. The claim that this method "can reduce the prediction error by 20% in a typical use case, and improvements above 89% are among the best cases" is technically true. But again, the base of comparison is linear regression. As described in section 2.6 above, and even acknowledged in [19], linear regression is known to be underpowered, so the value of the described success is overstated.

Others have also approached this problem using classification instead of regression. The work of [2] is similar to [19] and the systems presented in sections 4 and 5 in that more than one model is used to solve the problem. The issue with classification, though, is that it adds additional hyperparameters to the experiment; in this case, those are what and how many classes or labels are appropriate. Resultant classes have to be either self-explanatory or the ANN must learn how to map data points to predefined "bins" that clearly identify what they contain. The classifier in [2] is said to result in three such bins: "high," "medium," and "low" usage, but the meaning is not defined, resulting in a lack of quantification of the descriptive power of the system. That is, what "medium" means with respect to resource use is not established. The reported loss of the system is a RMSE of 0.37 and an accuracy of 86.56%.

Among the easier methods to understand are statistical ones, such as [1]. The work proposed double exponential smoothing to make resource utilization predictions and, similar to this one, considers windows of time in which statistics are calculated. The issue with double exponential smoothing is "it is suitable for time series with linear trend [*sic*]," so complex usage patterns, if any, cannot be picked up by this method.

3.2. Explainability in ML Using Fuzzy Logic

As introduced in section 2.5, methods of explainability seek to offer rationale for model behavior or output and combats the idea of black boxes. Recently, [4] combined LRP with concepts from fuzzy logic and linguistic summarization (LS) to describe model behavior using LDs. The model in the work is a deep classification ANN with ReLU hidden activation and softmax in the output layer making LRP as presented in [13] directly or easily applicable. In order to present the parts of [4] most relevant to eRACANN more clearly, the explainability

methods used are divided into three parts: calculating relevance scores for inputs, establishing fuzzy variables, and generating the LDs.

The relevance scores in [4] are calculated using the same backpropagated message method presented in [13]. The whole ANN approximates a classification function $f(x)$ and per-neuron relevance is computed at each layer, moving backward to the input features. Only features with positive relevance are considered and normalized, with the resulting normalized value being called the feature's "influence." This maps a feature's influence to the associated input's domain [4 Fig. 3] and describes it in terms of the fuzzy variables "low," "medium" and "high."

The next step is defining the actual input features in terms of fuzzy variables. Again, the number of fuzzy variables and their membership domain bounds are subjective. The experiment in [4] was performed using NSL-KDD, a dataset used for testing network intrusion detection systems. Each input, e.g. "dst_host_rerror_rate," "srv_error_rate," etc., got the fuzzy variables "low," "medium" and "high." The combination of fuzzy input variables as single antecedents to fuzzy output classes as consequents yields the set of possible LDs. These single antecedent-single consequent (SASC) LDs take the form "IF *input-variable* IS *input-class* THEN *output-variable* IS *output-class,*" where "*x* IS *y*" is called a summarizer. The whole LD is also referred to as a summary or a rule. Multiple-antecedent, multiple-consequent (MAMC) rules are also possible. They take form of a summarizer followed by a logical operator and another summarizer, e.g. "IF *input-variable-1* IS *input-term-1* AND *input-variable-2* IS *input-term-2*" has multiple antecedents.

With the set of possible LDs computed, it is then possible and necessary to strip out the ones that do not faithfully represent the data set. For this purpose, there five metrics for evaluating the knowledge described by LDs from [20]:

1. *Validity:* This property pertains to the significance of the knowledge that has been discovered.
2. *Novelty:* This describes the degree to which the discovered pattern(s) deviate from prior knowledge.
3. *Usefulness:* This relates the findings of the knowledge discovery to the goals of the user, especially in terms of the impact that these findings may have on decisions to be made. This is strongly related to the notion of "interestingness."
4. *Simplicity:* This is primarily concerned with the aspects of syntactic complexity of the presentation of a finding. Greater simplicity promotes significant ease of interpretation.
5. *Generality:* This entails the fraction of the population of data to which a particular finding refers.

Formulae for these measures with specific regard for Type-1 fuzzy set SASC LDs were proposed in [16]. Generalized formulae for MAMC are also described, but such is not considered in this work or [4], and so, is omitted for brevity. In the following metrics, M is the number of objects in the data set and S_n is the summarizer (a fuzzy variable) for the value v_n^m of the n^{th} attribute for the m^{th} object from the data set (a vector of properties). The validity of a rule is the value of its "degree of truth" DT:

$$DT = \frac{\sum_{m=1}^{M} \min\left(f_{S_1}(v_1^m), f_{S_2}(v_2^m)\right)}{\sum_{m=1}^{M} f_{S_1}(v_1^m)} \tag{8}$$

Generality is found by the "degree of sufficient coverage" DC. Calculating DC first requires the coverage ratio r_c:

$$r_c = \frac{\sum_{m=1}^{M} t_m}{M} \tag{9}$$

where

$$t_m = \begin{cases} 1, & f_{S_1}(v_1^m) > 0 \text{ and } f_{S_2}(v_2^m) > 0 \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

Now r_c is mapped to DC, an S-shaped function tuned by hyperparameters r_1 and r_2:

$$DC = f(r_c) = \begin{cases} 0, & r_c \le r_1 \\ 2\left(\frac{r_c - r_1}{r_2 - r_1}\right)^2, & r_1 < r_c < \frac{r_1 + r_2}{2} \\ 1 - 2\left(\frac{r_c - r_1}{r_2 - r_1}\right)^2, & \frac{r_1 + r_2}{2} \le r_c < r_2 \\ 1, & r_c \ge r_2 \end{cases} \tag{11}$$

The "degree of usefulness" DU is simply the minimum between DT and DC. Novelty can be interpreted as unexpectedness, and so, is represented by the "degree of outlier" DO:

$$DO = \begin{cases} \min(\max(DT, 1 - DT), 1 - DC), & DT > 0 \\ 0, & DT = 0 \end{cases} \tag{12}$$

The "degree of simplicity" is simply based on length and not a relevant metric when only SASC rules are considered. The rules extracted by this process can be then filtered by thresholds on DU and DO to present only the most relevant or applicable ones to the user. For a thorough background on the notation and rationale behind these formulae along with some simplifications, see [16].

4. THE SIMPLE SYSTEM

The vast majority of the characteristics of eRACANN are shared by, or derived from, its prototype, the simple system (TSS). Both are referred to as systems rather than just a model because they are indeed systems of multiple simple models rather than one large, complex model. Since eRACANN is an extension to TSS, and the design decisions behind them are the same, TSS is introduced first. The setup for these systems is formally expressed in section 4.1. Section 4.2 defines the notation used to describe individual models from TSS (simple models). The architecture of simple models is elaborated in section 4.3 and provides rationale based on the limitations and characteristics of SLPs and MLPs presented in section 2. The implementation of the preprocessing, training, and prediction stages are presented in section 4.4 with the results of experimentation in section 4.5.

4.1. Setup

Recall the setup in section 2.1 where a sysadmin has the need for a tool to predict the amount of resources needed for the various systems at Report Corp. This involves a set of systems S. A single $s \in S$, for the purpose of this research, is defined to be a collection of heterogeneous physical or virtual resources that can be allocated to processes or jobs. A single resource $r \in s$ is a finite number of units of something required by a computer process. All r have a maximum amount that can be utilized by s at any one time and two distinct s do not share any r. If they do, they are considered part of the same s.

To apply this concept to Report Corp., their $S = \{s_{\text{reporting}}, s_{\text{SaaS}}, s_{\text{IaaS}}\}$. One particular system $s = s_{\text{SaaS}}$ is known to have regular CPU-bound work and handles many web requests.

Its set of resources is $R_s = \{r_{\text{CPU}}, r_{\text{RAM}}, r_{\text{network}}\}$ and Report Corp. has utilization histories for these R.

It is also the case that Report Corp. sees more or less demand (load) on their S depending on the time. When a system experiences patterns in utilization over time the load on s is said to be seasonal or to exhibit seasonality. If, for example, many academic institutions use their SaaS portal for reporting, Report Corp. will consistently see more load on s_{SaaS} when it is time to report on students who have filed for financial aid, on staff that have enrolled in benefits, or perhaps on the bookstores' quarterly losses due to students being clever about where their books come from, with less demand on s_{SaaS} in between.

Provided trace data for the utilization of $r \in R_s$, $s \in S$ over time (time series), a MLP can learn to model the utilization graph for r because this fits the definition of a regression problem. In particular, this is a polynomial regression with unknown degree, making MLPs the perfect tool.

4.2. Describing a Simple Model

As mentioned before, TSS is a collection of MLPs. Each has one hidden layer, one output, and is designed for use in a specific context C. The C for a simple model M is a triple composed of a resource r for which predictions are being made and the two time-limited scopes t_m and t_d during the intersection of which it is applicable. That is, M_C is a specific simple model designed for $C = (r, t_m, t_d)$ and only that C.

The two time-limited scopes are for handling windows of seasonality, thereby adding two hyperparameters to M so its application can be more targeted than just to some r. This approach takes the problem of resource prediction and breaks it into sub-problems, or subtasks, that are

described by C and handled by the corresponding M_C. This technique is similar to the concept of cooperative agents in reinforcement learning, where multiple models encounter different parts of a problem and share the experience with each other so the system can learn faster [21]. So, two lengths of time must be chosen for dividing the problem space.

It is known a system s will see different usage over time. Since people and businesses typically operate at designated times of the day depending on the day of the week, a weekday d (which includes weekends) is a good candidate for the time-limited scope t_d; that is to assert the problem space contains usage patterns relative to the day. By itself, however, this scope is naïve. It is too limited to consider regular events like holidays, seasonal business processes, and peak shopping times like Black Friday. So, to give the model some additional perspective, t_m will cover the utilization over a month m. To summarize and quantify $C = (r, t_m, t_d)$, TSS operates within the claim that it is possible to partition the problem into subtasks described by a cloud resource r, usage patterns existing over a given month t_m and usage patterns existing over a given weekday t_d. The generic nature of these time-limited scopes allows for further arbitrary partitioning, e.g. to include usage patterns over a year or some other distinct window of time as a part of C. The most limiting factor there is one must have time series data for the duration of that time scope.

To be specific, the month-limited scope t_m takes a month ordinal argument m and the weekday-limited scope t_d takes a day ordinal argument d. To coordinate with the Python datetime module used in all proceeding implementations, months are 1-based starting with January ($m = 1$) and weekdays are 0-based starting with Monday ($d = 0$). To keep notation more readable, the tuples (r, t_m, t_d) and (r, m, d) describe the same C, e.g. $(r_{\text{CPU}}, t_{m=8}, t_{d=0})$ and $(\text{CPU}, 8, 0)$ are equivalent and the latter format will be used going forward. The simple

model for making CPU predictions in August on Mondays can therefore be expressed $M_{(\text{CPU},8,0)}$. Weekday was chosen over the actual date of the day because associating utilization patterns with the date would create 28 to 31 different M due to t_d alone. Additionally, not all of those would even be useful in a given month leading to an imbalance in how much the date models were trained. It is also the case that the scope of t_m is large enough to account for regular events on specific dates anyway. So, if d were a date then that time of the month would be doubly accounted for. Hereafter, any reference to the "day" should be interpreted as the 0-based weekday ordinal, e.g. $d = 5$ is always Saturday.

4.3. TSS Architecture

4.3.1. The Input Layer. The choice of inputs, or features, is incredibly important as it determines how the problem is represented to the model. One obvious choice for inputs are those found via statistical methods as the features are necessarily representative of the problem space. The literature has many purely statistical models for predictions of varying kinds, but these are generally outperformed by ML models. The choice for statistics-based features in TSS and eRACANN synergizes these methods to leverage the appropriate contribution of both approaches.

The inputs for any simple model M_C are listed in Table 1, where $\hat{r}$ denotes resource utilization has been normalized to $[0, 1]$ and W is a window of time, e.g. 30 minutes. The total number of inputs is 28, implying the problem space for predicting utilization has 28 dimensions per this configuration; this can also be interpreted as 28 features are necessary to faithfully represent the problem. The number of inputs that are necessary for a problem is not always obvious or even can be known a priori, but ablation, or systematic removal, is one way to verify

the number is sufficient. In TSS it was found via experimentation that each added input increased a model's predictive accuracy. These findings are exemplified by their corresponding prediction graphs in the results discussion, section 4.5. It is for this reason that inputs highly correlated to the desired output are asserted to make better models. Conversely, it was also found that removal of single feature caused a decrease in predictive accuracy, regardless of which one it was. Therefore, no inputs were ultimately removed.

4.3.2. The Hidden Layer. At least one hidden layer with non-linear φ is necessary to introduce non-linearity to a ML system. Adding more hidden layers to a model increases its capacity for modeling problems with a higher dimensionality problem domain. The issue with doing so is legitimizing the addition. Evidence that the problem demands the added capacity for more layers of abstraction would be convincing, but the goal is usually just a lower error. Trying random tweaks to ANN configurations in hope for that lower error is, indeed, just programming by permutation. Implementations born from programming by permutation lose any sense of interpretability, if it had any to begin with. Even if more hidden layers do improve performance, it will not be clear why.

A similarly vague principle in architecting ANNs is how many nodes should be in a given hidden layer. An appropriate number is typically found via experimentation, but given more than one input node and at least one hidden node, a system can model non-linear data. To help keep a model minimal, i.e. only as complex as is necessary, one can start with a large number of nodes, e.g. 64, and decrease the count until model accuracy obviously suffers.

Because $\hat{r}$ is normalized and the model output is the predicted, normalized resource utilization, one should use a φ with similar limits. The sigmoid function is ubiquitous in ML and has the ideal range $(0,1)$. This is more appropriate than other frequented activation functions

such as $tanh$ because its range is $(-1,1)$ and values less than 0 are not meaningful in this application.

4.3.3. The General-Specific Tradeoff. There is typically a tradeoff when building any system designed to learn. A more generalized model is applicable to more settings but will be less accurate compared to a model made more specifically for the same setting. The latter, though, will not be as useful outside of its intended application. This architecture balances generalizability and specificity by composing a general system out of parameterized simple models limited in applicable context, e.g. with different r, t_m, or t_d.

4.4. Implementation

TSS was implemented in Python 3.7 using TensorFlow 2.1 (TF) with the Keras backend. It follows a process very typical of ML systems. First, training data is preprocessed so it fits the models' inputs without requiring additional transformation. Next, the system trains the simple models on the data, recording the loss as it goes. Lastly, the system produces a series of predictions for a "live" system.

4.4.1. Model Configuration. The M_C used in TSS experiment has 28 inputs, a hidden layer of 12 nodes, and a single output node, all fully connected (Figure 6). The hidden layer started with 64 nodes but was gradually reduced in size until the quality of predictions obviously suffered. The activation function in the hidden layer is sigmoid. The identity function is used for φ in the output node because even though sigmoid's range is $(0, 1)$, it is possible for resource demand to exceed what is available, i.e. $\hat{r}$ can and does exceed 1.0 in the trace files. Since $f(x) = x$ has the range $(-\infty, +\infty)$, though, the individual M_C are expected to adapt the estimation functions they represent to the appropriate shape. This way, even if they output

negative numbers or values much greater than one, they can be clamped without loss of generality. The ReLU function was also tested as the φ in the output node, but the benefit of having the activation function forcibly clip values to $[0, +\infty)$ was outweighed by how it hindered learning. The lower bound of the ReLU's range causes prediction values less than 0 to be misrepresented as 0, so the calculated loss misrepresents the actual error and weight updates are not as significant as they should be.

4.4.2. Preprocessing. The purpose of this stage is to get the training data into a form more easily processed by a model. This can include adding removing, and transforming data. The "fastStorage" data set [22, 23] was preprocessed to match the inputs expected by a M_C. Notable transformations include normalizing resources' utilization to $[0, 1]$ where it was not already and changing the format of the timestamp. CPU utilization was provided already "normalized," except that utilization sometimes exceeded 1.0, as mentioned before; these values were not clamped to 1.0 as part of preprocessing.

The fastStorage data set is a time series of system resource utilization for a collection of VMs from mid-August to mid-September. To isolate testing to a single month, part of preprocessing was shifting the data so it started at the beginning of the month and did not spill over into the next. The alternative to this was having half as much data between two months, which would then be split all the more unevenly between weekdays—and less training data is not ideal for ML. All of the time series was also shifted backward by 30 minutes so the model would learn what utilization looks like 30 minutes after the time stamp of the features present at that time.

The other inputs expected by a model were computed using the Pandas Python module and added to the data. For example, input 8 from the Table 1 is calculated by the `pct_change`

function, which takes the aforementioned time window $W = \text{"30Min"}$, a Pandas Period, as an argument. This means the minimum, maximum, percent change, etc., for $\hat{r}$ was with respect to the next 30 minutes rather than the current time. The system, therefore, learns to model behavior happening a duration of $W = 30$ minutes after the current time stamp.

Inputs 11 and 12 which are twelve and six additional input nodes, respectively, use an alternative $W' = jW$ before the time stamp for each input node, where j is the 0-based index of the input node. This helps the model learn to make predictions from historical trends or patterns that occur over multiple time steps. For example, the third node for input 12 is the percent change of the 60 minutes prior to the current time step when $W = 30$ minutes. Preprocessing outputs these transformations to "feed files" (raw input data) and "rolling files" (rolling statistical data) as the time series on which the models are trained.

It is generally a good idea for all ANN inputs to be normalized otherwise the weights of the normalized ones can diminish or get overcorrected to compensate for the backpropagation of an error magnified by the non-normalized input. Since the `pct_change` function has the range $(-\infty, +\infty)$, the relevant inputs (8 and 12) from Table 1 needed to be clamped and normalized. To this end, a maximum magnitude must be chosen, adding another hyperparameter. This experiment clamped percent change to $[-100, 100]$ before normalizing because any slope greater than 100 between two data points' utilization was empirically indifferentiable. In addition to facilitating normalization, clamping also prevents infinite values from occurring in the training data, which happens when the percent change is between 0 and anything else.

4.4.3. Training. Training starts by creating an input pipeline from the feed files using TF and a Python dictionary to cache simple models. It is important to note that these models are note created until the context calls for it. Recall the definition of a context from section 4.2, a

triple of (r, m, d). Once training starts, there will exist the first C demanding the first model be created by the "Model Factory." The data set starts on August 1, 2013, which is a Thursday. If the first resource trained on is r_{CPU}, the Model Factory will see that $M_{(\text{CPU},8,3)}$ does not yet exist, so it will create it, cache it, and return it to the trainer. The next time $C = (\text{CPU}, 8, 3)$, i.e. on August 8, the Model Factory will be able to provide the model right away. This also keeps the solution minimal.

The experiment was conducted with 2000 epochs and $a = 0.001$ using the Adam optimizer because the standard stochastic gradient descent optimizer did not converge as quickly to approximately the same error metrics. The entire training set was also used as one batch so models got to experience every data point before adjusting their weights.

4.4.4. Predicting. The whole point of having a system for prediction is to be able to get demand at the next time step or even create extended forecasts. Because the models are trained on data shifted backward by 30 minutes, a live system s can continually feed into TSS to get live predictions for 30 minutes out from the current time. This approach is simulated to create the prediction graphs shown in all subsequent results sections. The implementation described here is also able to rely on its own prediction abilities to feed itself for predictions that are further out than the next time step, however with greater variance.

4.5. Results

There are many metrics with which to evaluate ANNs, but the most appropriate choice is dictated by the purpose of the ANN. The most popular evaluation metrics for regression included MSE, MAE, and RMSE. It is also important to consider what is used in others' research so the results are comparable. Since many researchers report their models' RMSE, this

one will also. Nonetheless the MAE is also included since it is easier to interpret than the RMSE.

Originally, only a single $C = (\text{CPU}, 8, 0)$ was used to see if breaking up the problem via context resulted in low enough error to warrant generalizing to the other t_d, which was found to be true. The simple models all trained and recorded the MAE and RMSE after each epoch on both the training data (80% of the feed files) and the validation data (remaining 20%). The results of the experiment using the configuration in section 4.4.3 are presented in Table 2. The progression of the loss under the same hyperparameters is graphed in Figure 7; most training and validation plots overlap due to the errors being similarly close to zero.

It was found each added input from Table 1 increased the models' predictive accuracy. In particular, adding input 8 benefited the models by better correlating frequent and strong "spikes," or sudden increases in utilization, to the desired output. Predictions by TSS without this input are presented in Figure 8. It was also found after adding input 12 that TSS become more sensitive to "plateaus," or relatively consistent use past W; it could also be said input 12 smoothed out some or the more random or exaggerated spikes that appeared when there should have been a plateau or at least less "spiky" usage. Predictions by TSS without this input (but with input 8) are presented in Figure 9. The inputs being highly correlated with the desired outputs made for more accurate models. It was also found that ablating any single feature caused a decrease in predictive accuracy, regardless of which one it was. Therefore, no inputs where deemed unnecessary and removed.

By numbers alone, this appears to be a reasonably accurate system at a minimum, but error graphs and numbers are not inherently indicative of much without quantification. The predictive power of TSS is displays in Figure 10 with a $W = 30$ minutes. To observe the effect of

changing hyperparameter W, Figures 11 and 12 show the same experiment with $W = 1$ hour and $W = 60$ hours, respectively; the error for these two additional experiments are unperceivably different from the $W = 30$ minutes experiment, and so are omitted for brevity. It is also observed that the most accurate prediction Ws are 30 minutes and 6 hours, but it is not clear why when W is the only thing that changed. This hyperparameter is, therefore, expected to yield better results after being tuned specifically for different data sets and ANN topologies at a minimum. To compensate for this and if better accuracy is desired with a smaller W, recall from section 2.2 that MLPs can estimate arbitrary continuous functions. Then, even if W is larger, say 6 hours, the prediction resolution can be turned down to, say, 1 hour, because the function estimated by TSS models is continuous regardless of W. Figure 13 portrays this concept.

The graphs (Figures 10 – 13) are limited to the first week in August for readability, but it is clear that the low errors reported in Table 2 coincide with an accurate system, especially considering how small and simple the individual models are. The fact that they are this powerful though small serves as part of the reason for extending it, exploring whether limited additional complexity further improves on predictive power.

Table 1. Inputs to simple models.

	Description	Example Value
1	Month elapsed, normalized.	August 6 = 6/31 = 0.1935
2	Day elapsed, normalized.	1:30 PM = 13.5/24 = 0.5625
3	Minimum $\hat{r}$ in W.	0
4	Maximum $\hat{r}$ in W.	1.0151
5	Standard deviation of $\hat{r}$ in W.	0.0144
6	Average $\hat{r}$ in W.	0.9711
7	Median $\hat{r}$ in W.	0.9690
8	Percent change for $\hat{r}$ in W.	-0.6201
9	Average $\hat{r}$ in t_m.	0.5337
10	Average $\hat{r}$ in t_d.	0.4332
11	The last 12 time-steps' $\hat{r}$ as 12 additional inputs.	[0.75, 0.24, 0.14, . . ., 0.94]
12	The percent change between each of the last 6 time-steps' $\hat{r}$ and the current time step as 6 additional inputs.	[-0.21, 0.8, -0.6, . . ., 1.0]

Table 2. Error of simple models.

C	MAE		RMSE	
	Training	Validation	Training	Validation
$(\mathrm{CPU}, 8, 0)$	0. 0261	0.0290	0. 0628	0.0627
$(\mathrm{CPU}, 8, 1)$	0.0210	0.0235	0.0571	0.0495
$(\mathrm{CPU}, 8, 2)$	0.0124	0.0143	0.0504	0.0526
$(\mathrm{CPU}, 8, 3)$	0.0175	0.0293	0.0577	0.0629
$(\mathrm{CPU}, 8, 4)$	0.0141	0.0419	0.0359	0.0533
$(\mathrm{CPU}, 8, 5)$	0.0159	0.0496	0.0517	0.0680
$(\mathrm{CPU}, 8, 6)$	0.0236	0.0329	0.0775	0.0955

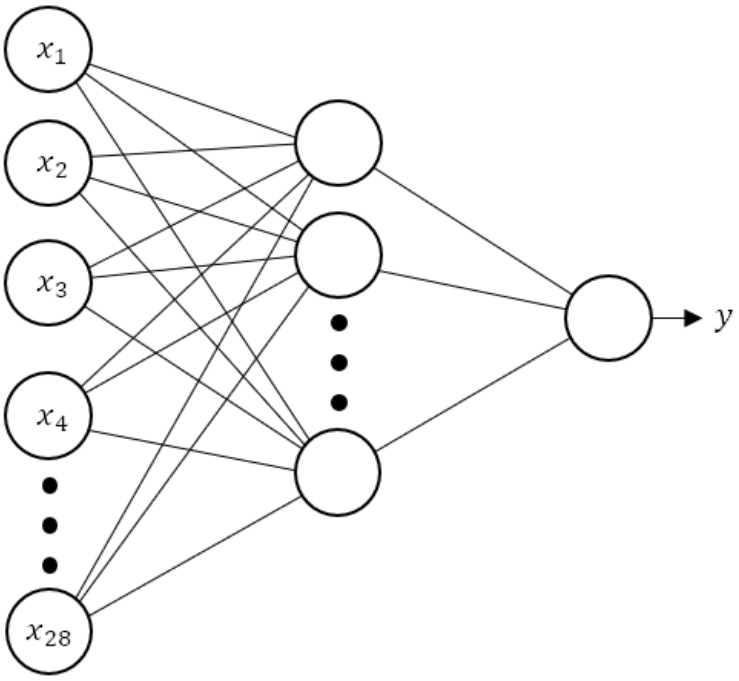

Figure 6. The topology of a simple model.

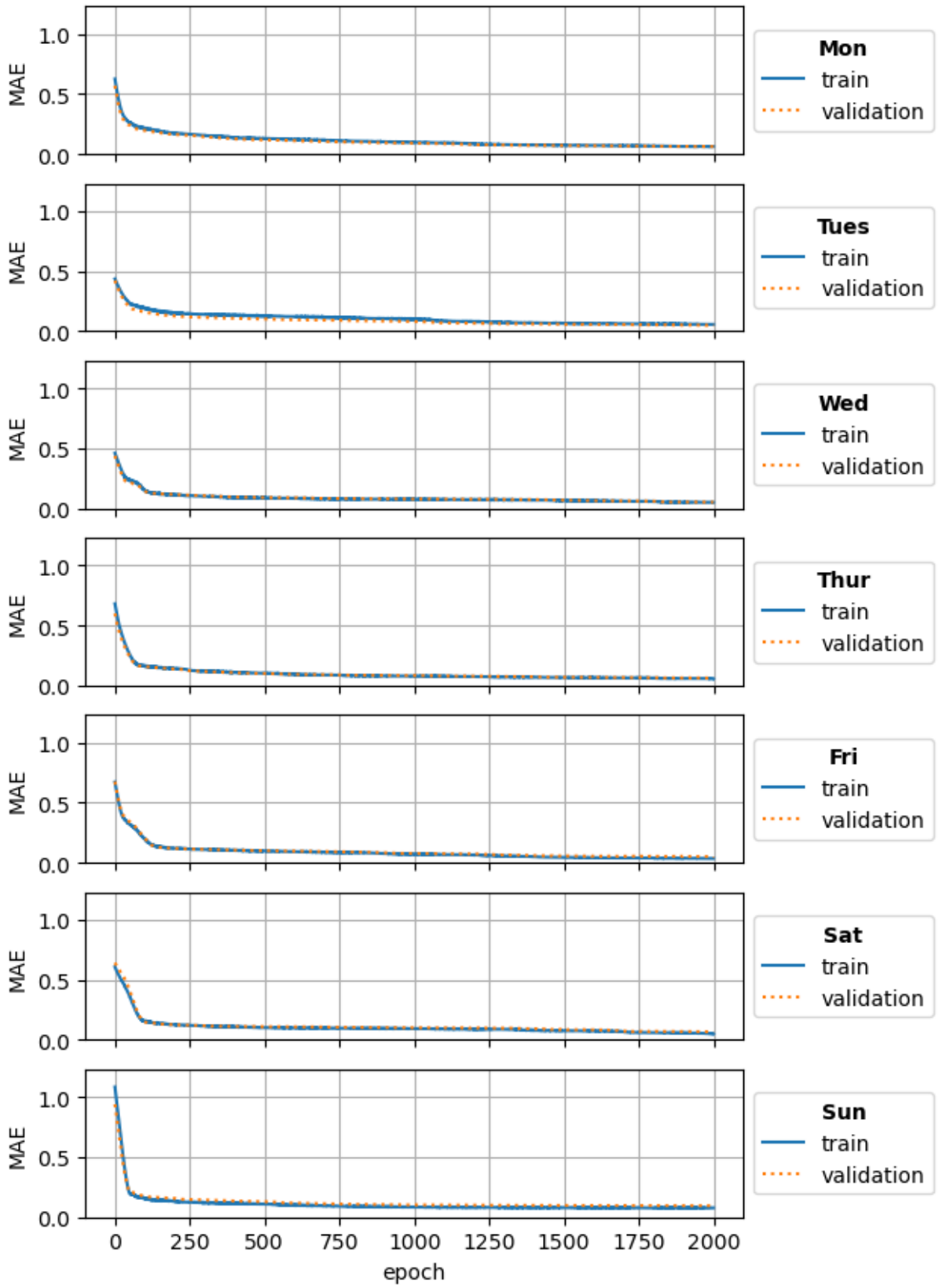

Figure 7. The MAE of each trained day model for August over 2000 epochs.

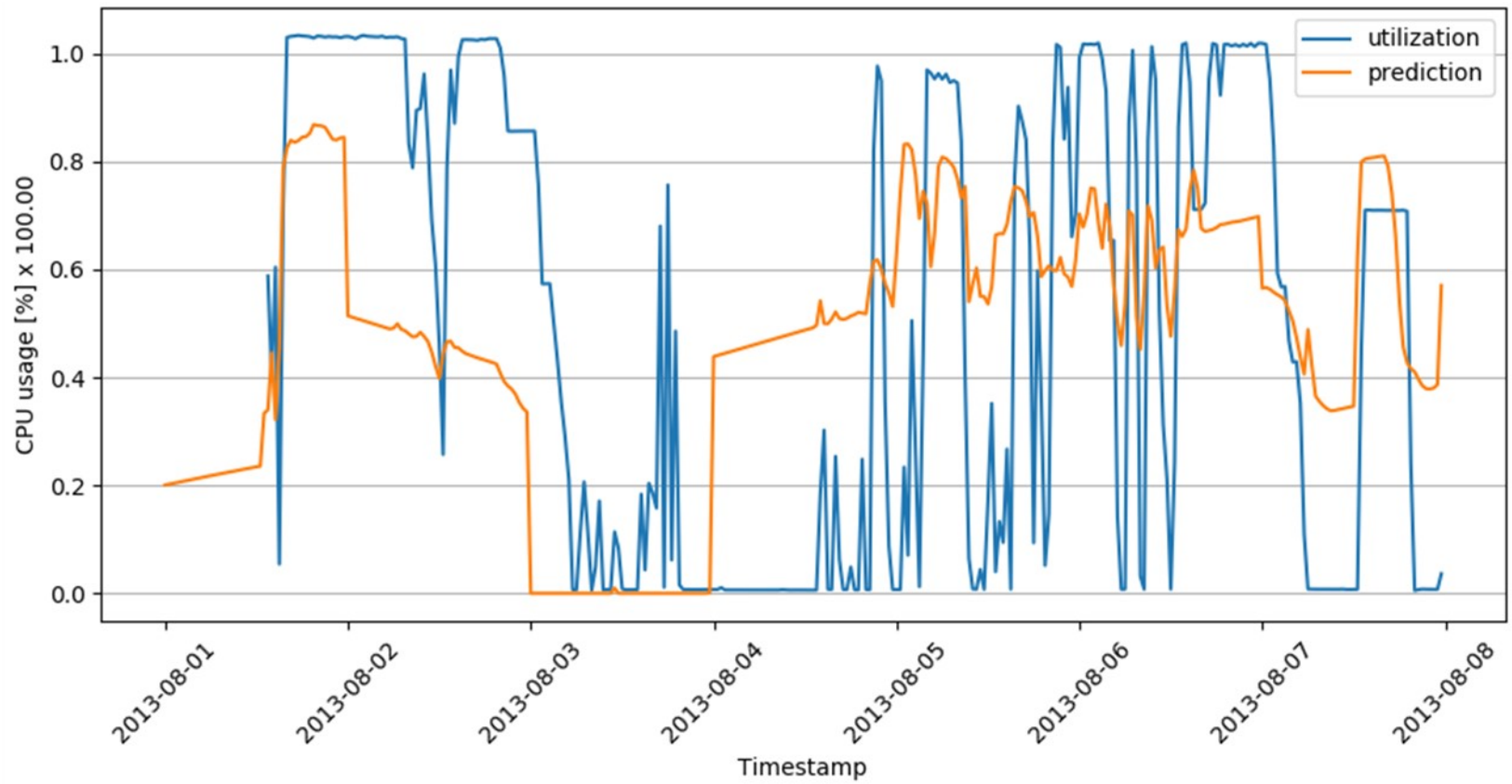

Figure 8. TSS CPU predictions for $W = 30$ minutes without input 8 from Table 1.

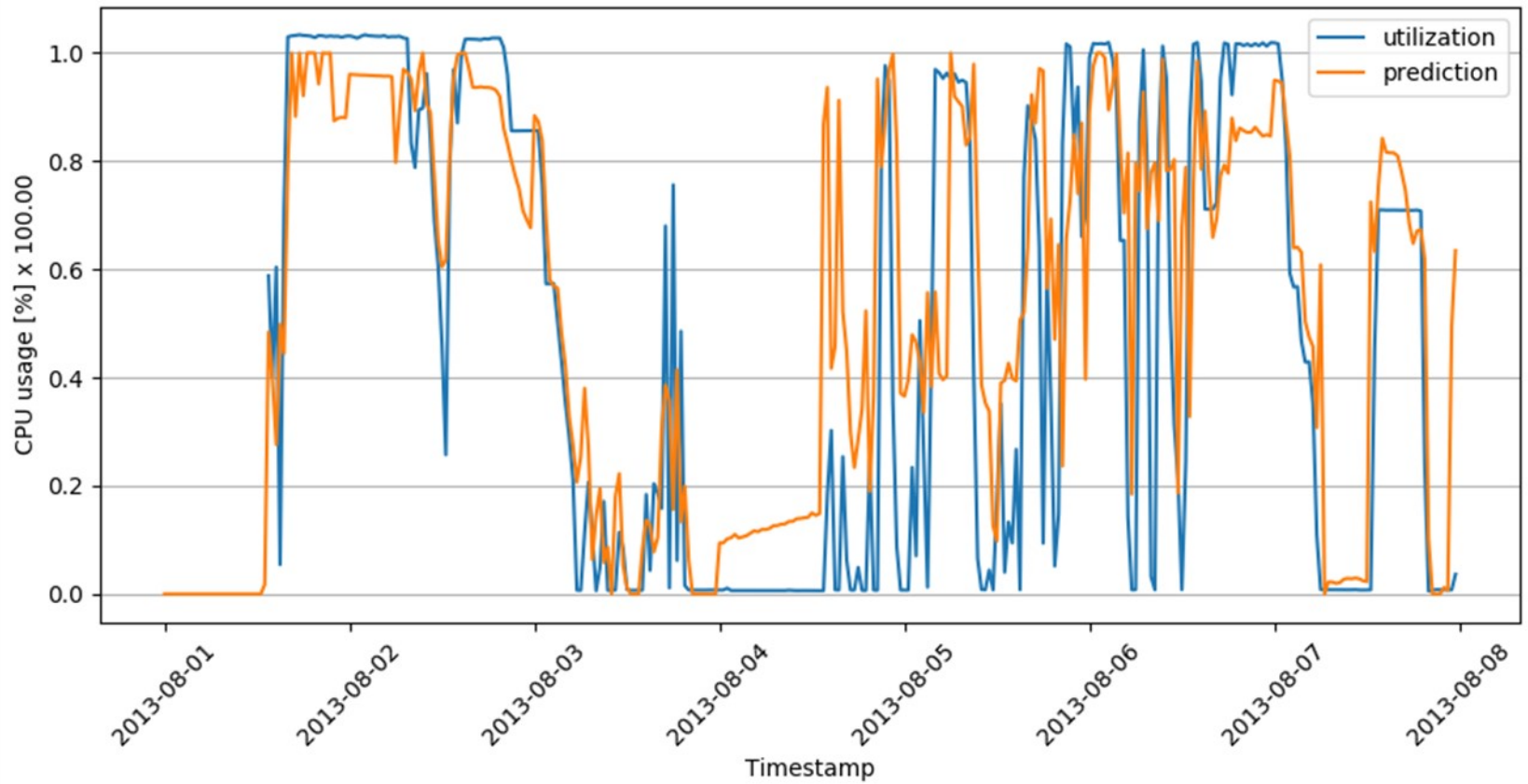

Figure 9. TSS CPU predictions for $W = 30$ minutes without input 12 from Table 1.

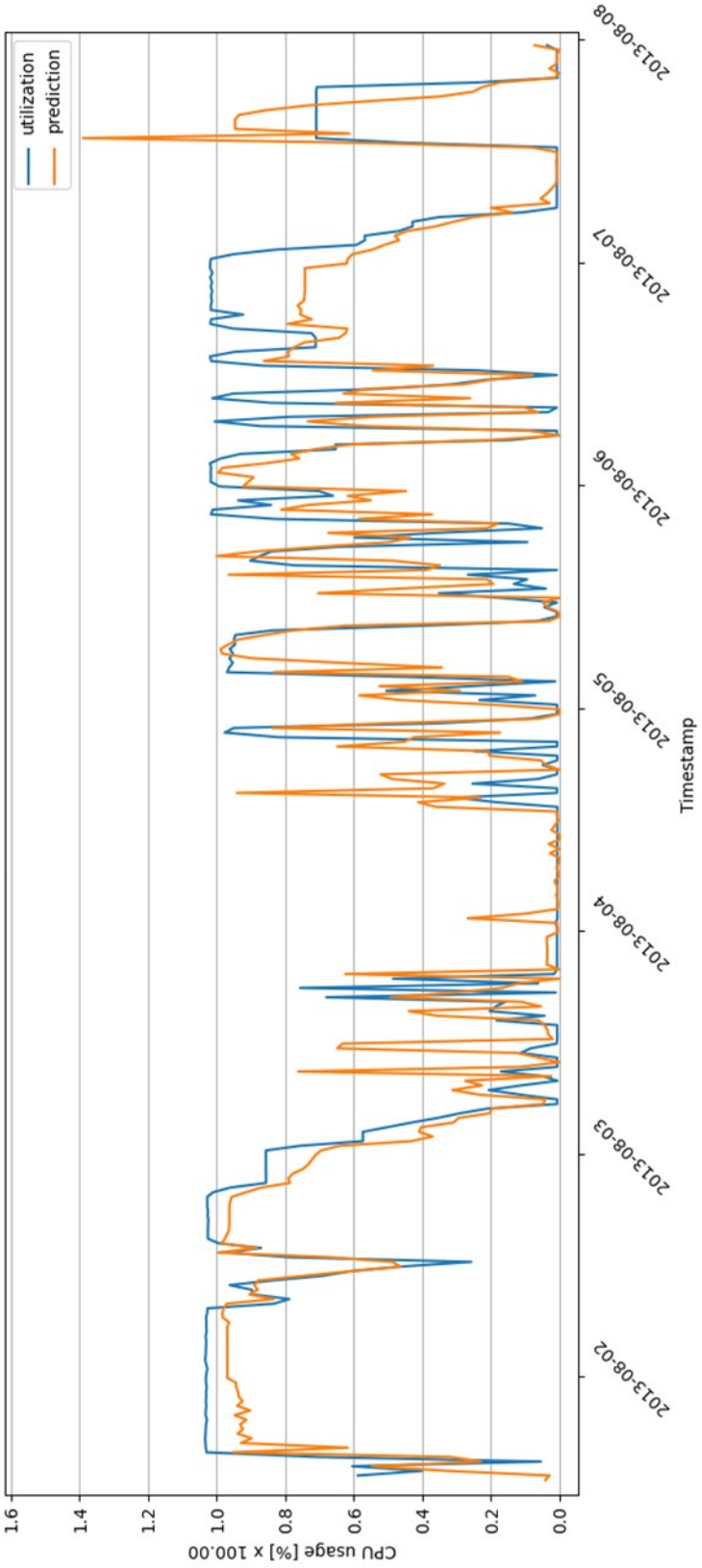

Figure 10. TSS CPU predictions for $W = 30$ minutes.

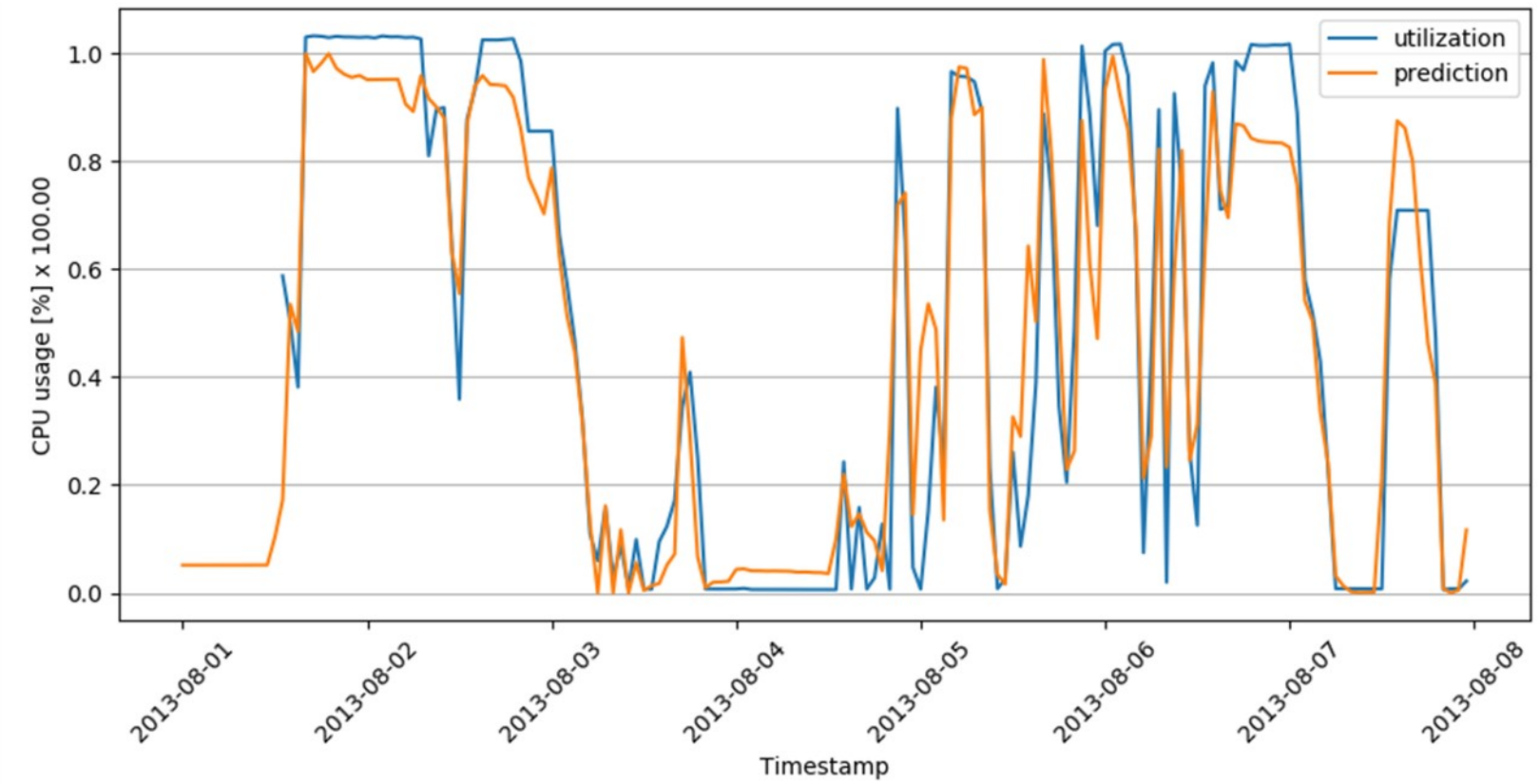

Figure 11. TSS CPU predictions for $W = 1$ hour.

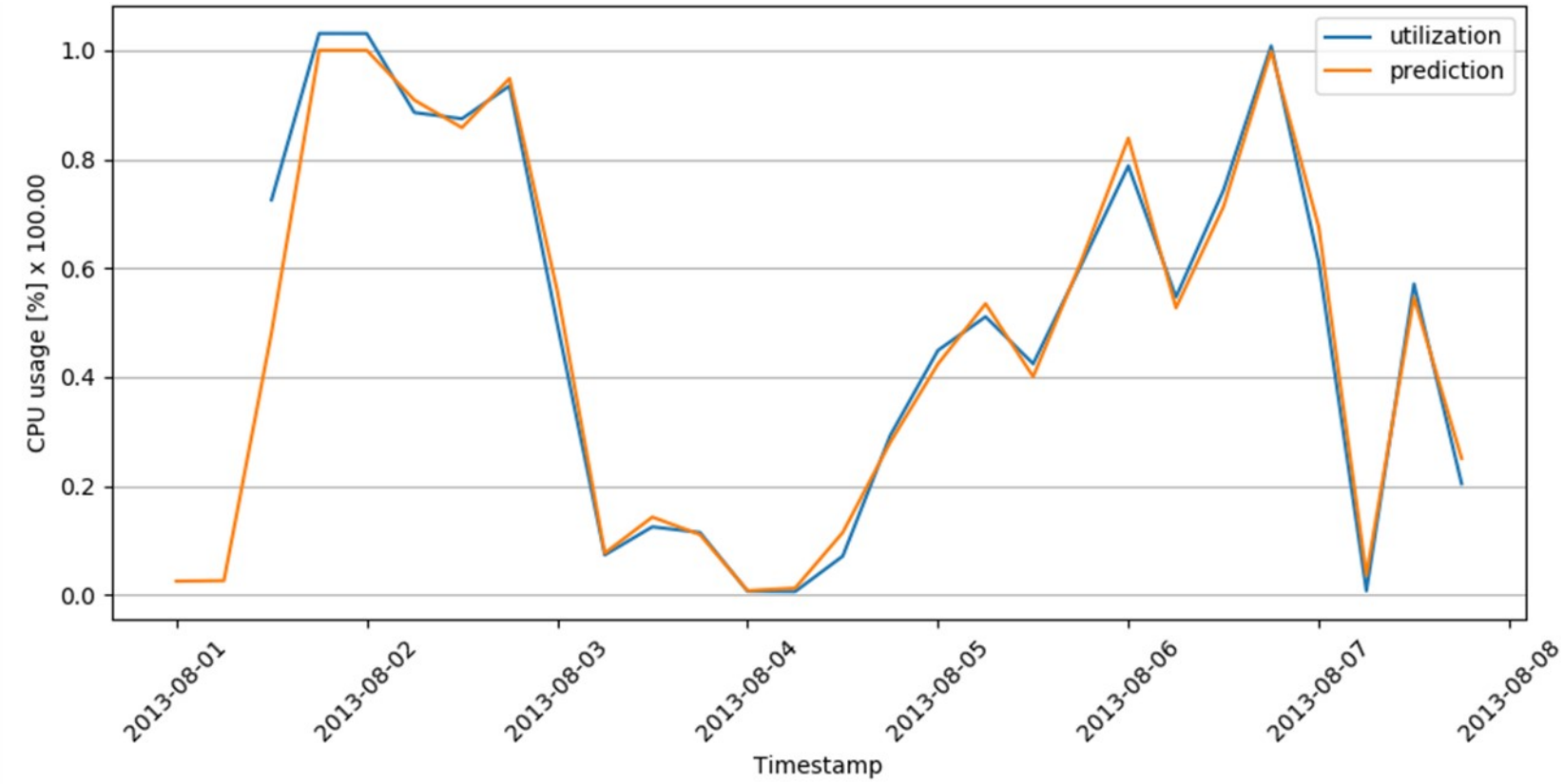

Figure 12. TSS CPU predictions for $W = 6$ hours.

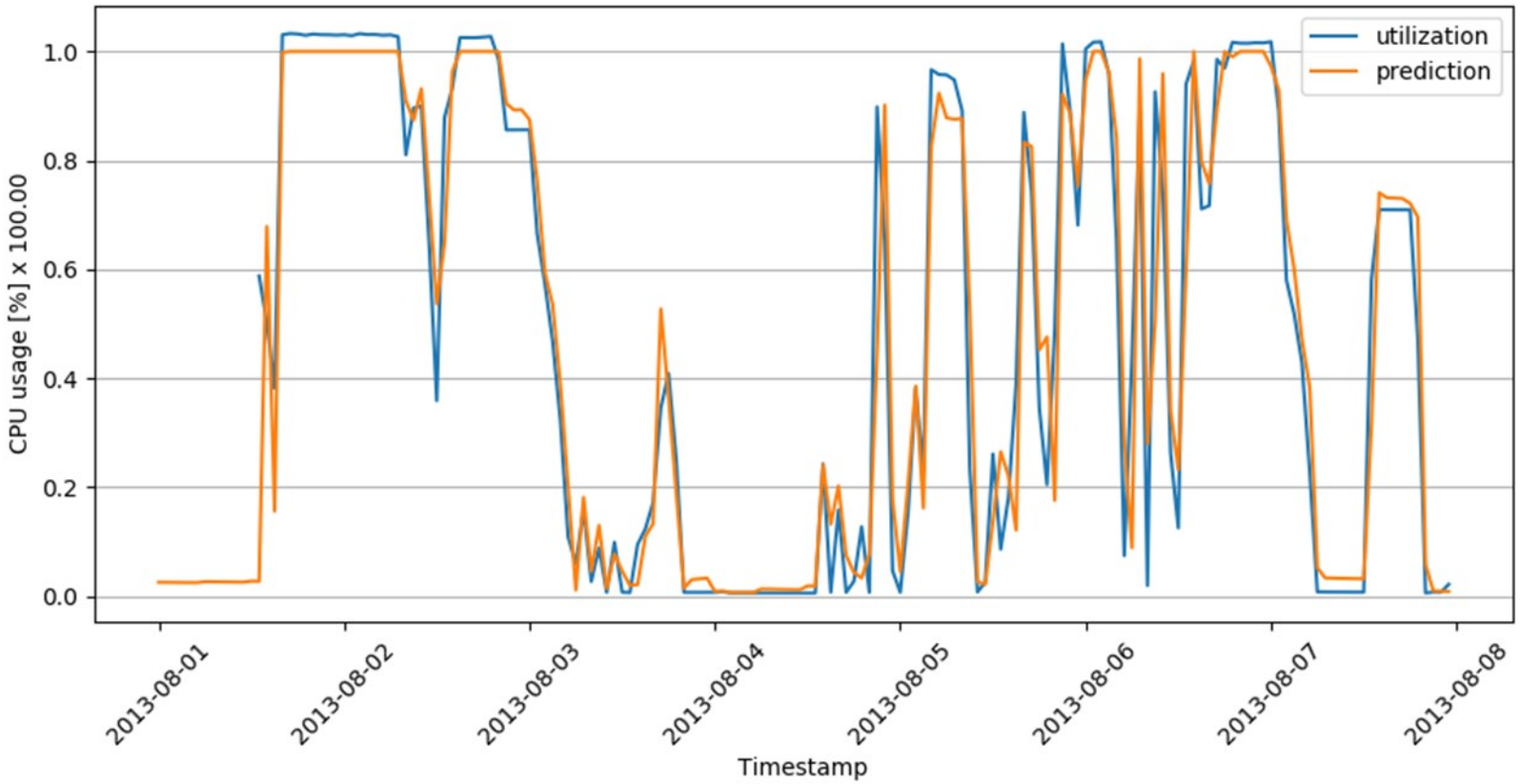

Figure 13. TSS CPU predictions for $W = 6$ hours, but at a 1-hour resolution.

5. RACANN

eRACANN borrows much from its design of TSS. The setting is also identical. The difference between eRACANN and TSS is now the contexts are broken up further into more specific parts and it gets extended into an explainable system, eRACANN. It is still a system of MLPs, but a unique take on them.

5.1. Describing a Model

Recall from section 4.2 where a model M from TSS could be described by a context $C = (r, m, d)$. eRACANN breaks C up into two parts: a month context $C_m = (r, m)$ and a day context $C_d = (r, d)$. What makes this approach unique is that because C is two distinct sub-contexts, they must be represented by two sub-models: the month model M_m and the day model M_d. These two models are brought together at training and prediction time in order to fully represent the complete context $C = C_m + C_d$. Thus, M_C is redefined as $M_C = M_m + M_d$ where M_m has $C_m = (r, m)$ and M_d has $C_d = (r, d)$. The notation $M^R_{(r,m,d)}$ will be used to describe the composite model representing the two joint contexts, with the superscript R being used to differentiate eRACANN models from TSS models (no superscript). The effect of this redefinition is that M^R_C is now a runtime assembly of sub-models, not just a list of models representing all possible contexts. By tying them together, it has the same cooperative aspects of TSS. However, these agents solve more specific parts of the problem space, and so, there are more of them.

5.2. Architecture and Configuration

The topology of a complete eRACANN model is brought about by joining the last layer of the month model and the last layer of the day model to a single output node (Figure 14). The individual sub-models mimic TSS models, and likewise, are only trained on the data belonging to their contexts. They each also maintain the same hidden node count of 12, hidden layer φ as the sigmoid function, and linear output activation. The only difference is that rather than sub-models having their own outputs, they are joined together at one output node, the output of the model representing the complete context.

Something that makes this approach interesting is how the error is backpropagated. Gradient descent credits each layer with its contribution to the error via partial derivatives and that credit is tempered by the learning rate to update the weights of the layer. This is less straight forward in this approach because not all layers are fully connected. Nonetheless, the derivative of the output layer with respect to the hidden layer is no different because these two layers are fully connected. It is just a matter of making sure the partial derivative from the hidden layer backward is considerate of only its own inputs. The good news is TensorFlow takes care of these details by itself. It is still a relevant concept, though, as the partial derivatives respecting the appropriate inputs is key to calculating input relevance, to be introduced in section 5.3.3.

5.3. Implementation

5.3.1. Preprocessing. The steps necessary to apply eRACANN to [23] or the Google cluster traces data set [24] are almost identical to TSS, though extracting a subset of data from the data set is necessary and the output format of the feed files differ for the purpose of

input alignment. Due to the volume of data in [24], only some machine's utilization histories are used. In order to have about the same amount of training and validation data per month as TSS, this experiment pulled 396 machine traces and divided it using an approximately 80/20 split. Those traces were interpolated into 5-minute resolution resource files (the same resolution as [23]) so the sum of many simultaneously running tasks could be accounted for. The result was also realigned so they covered the entire year, starting over again on January 1 of the same year if realignment would have spilled into a new year. This way there is about the same amount of data for every month and there is enough data to be able to train month models multiple times over a different set of data. This also adds an element of noise, which can help test the robustness of the approach. Before the eRACANN experiment, it was not known whether [24] exhibited any seasonality or not.

5.3.2. Training. Unsurprisingly, the training process also follows TSS. The aforementioned feed files are identified via glob pattern by a TensorFlow input pipeline and three Python dictionaries cache month sub-models, day sub-models, and composite models. Again, no sub-models are created until the context calls for it. Once training starts, the first C with come to pass demanding the Model Factory provide a M_C^R. The training data starts January 1, 2019, which was a Tuesday. If the resource being trained on is CPU, the Model Factory will see $M_{(\mathrm{CPU},1,1)}^R$ does not yet exist in the composite model cache. So, it will check the sub-model caches for the components that comprise $C = (\mathrm{CPU}, 1,1)$: $C_m = (\mathrm{CPU}, 1)$ and $C_d = (\mathrm{CPU}, 1)$. If neither one of these exist either, sub-models consisting of inputs and a hidden layer are created for the month and day, and are cached. Now that these exist, the factory can join them together into an eRACANN model by connecting the two sub-models' outputs (what will be the hidden layers) to an output node. This realizes a full composited eRACANN model.

When January 2 comes up and $C = (\text{CPU}, 1, 2)$, the Model Factory will just pull the existing January month model from the cache, create the day model for Wednesday, and join them into a new eRACANN model. In this way, the January model can contribute to every days' learning as long as the month context is still January. The same is true when training hits February 1, a Friday. By that point, the Friday model will exist, but the February model will not. So, the Model Factory will join the Friday model to the newly created February model, allowing February-Fridays to inherit the learning that the Friday model already got in January. With an entire years' worth of models, this starts to resemble a lookup table of contexts, where any column-row (month-day) combination is the output of an eRACANN model. Figure 15 exemplifies this perspective with a lookup on the outputs available for the month of September. Again, and to be clear, note that there are only seven total day models and twelve total month models, but there is a unique singular output node for each combination of these.

Experiments were run for eRACANN models on two data sets, Google cluster traces and the fastStorage data set. The latter of the two allows eRACANN to be compared to TSS and was also observed to contain seasonality, thus satisfying an underlying assumption of both systems. The experiments initially ran with a $W = 30$ minutes and for 2000 epochs with $\alpha = 0.001$ using the Adam optimizer. The entire training and validation data sets were considered one batch. eRACANN appeared to over-fit on the data at 2000 epochs, though. So, for the fastStorage data set it was scaled down to 10 epochs and a larger $\alpha = 0.01$ to compensate for the lesser training time. Any relatively small increase to the number of epochs over 10 drastically and negatively affected predictive accuracy; this is exemplified in the discussion on the results in section 5.4. It was only scaled down to 500 epochs on the Google data set, though, and α was left at 0.001 to

compensate for what turned out to be very unseasonal data, thus requiring the additional time and gentler learning rate.

5.3.3. Explainability. Now RACANN is formally extended with features enabling explainability. The approach taken is very similar to [4]. Once all models have been trained, it is possible to calculate relevance and influence scores for the inputs. eRACANN deviates from the assumptions necessary to apply LRP as presented in [13], most notably in that it is not a classification ANN and does not use ReLU as its hidden activation function. The following adaptations are made to facilitate getting relevance scores for sub-model inputs.

LRP as presented in [13] looks for the relevance of neurons in terms of their contribution to the desired output class. To exemplify this, consider a binary classifier. Input neurons will either contribute positively to the identification of the output class. Phrased differently, their features either support for the detection of the class or provide evidence against it. The results of LRP in this instance is a heatmap. Heatmaps, however, do not apply to regression ANNs; the contributions of the regressors (inputs) require a different interpretation. For a regression ANN, all features contribute to the output, a real number. The fact that their influence is positive or negative is not indicative of anything other than the ANN having learned to compensate for the features' effects. Because the traditional summation of layer-wise relevance would lead to negative relevance scores cancelling out positive relevance scores, i.e. diminishing measured contribution, eRACANN will treat all relevance as positive by using the absolute value function.

It is also the case that the presented backpropagation formulae are based on activation functions with roots, where it's possible to determine with what data an activated neuron will output 0. If this is true, the output of a neuron with respect to non-root inputs can be compared to the output with respect to a root input. The result is the magnitude of the difference in the

output, similar to computing sensitivity or getting the slope of the difference on the solution surface. The problem with applying that to eRACANN is the sigmoid function has no roots. It can be given one artificially by shifting it down by 0.5, but that would require maintaining and backpropagating an error that may end up being larger than input nodes individual contributions. This would result in the desired value of calculated relevance being absorbed by error.

So, eRACANN must compute relevance scores without access to roots or the slopes born from being able to compare non-roots against them. The way this is handled is a sort of fusion between [13] and gradient descent. Gradient descent uses the chained, partial derivatives of the output with respect to each previous layer to attribute a portion of the error to that layer. This can also be seen as determining how much a layer (wrongly) contributed to the output. So, rather than determine the contribution to the error, eRACANN determines the contribution to the actual output. This is accomplished by using the derivative of the output value with respect to each input node, thus demanding the same chained, partial derivative technique of gradient descent.

Since the last calculation in eRACANN is the identity function in the output node, it has the ultimate degree of contribution to the output value. Conveniently, the derivative of the identity function is 1 and is not otherwise dependent on any other values. So, the contribution of the output node can actually be said to be zero since any derivatives before it multiplied against it will remain the same. This matches up the intuition of the identity function; nothing changes. The derivative must then be found with respect to the pre-activation value of the output layer, the activation functions in the hidden layer, etc., down to the input nodes themselves. This becomes an interesting problem since an eRACANN model is not fully connected. To handle this, the calculations must be derived with respect to only the nodes in its own sub-model. The values

involved in determining a relevance score using this method are given by (13) without mathematically expanding the expression for brevity. The subscripts i, j, and k are the input, hidden, and output layers, respectively. The z_l terms are pre-activation values, which serve as the input for the φ in the following node of the layer l; w terms are weights and x terms are the outputs of their respective nodes. The logical location of these values is depicted in Figure 16.

$$R_i^1 = \frac{\partial y}{\partial x_i} = \frac{\partial y}{\partial z_k}\frac{\partial z_k}{\partial x_j}\frac{\partial x_j}{\partial z_j}\frac{\partial z_j}{\partial x_i} \tag{13}$$

Once each input node i has a relevance score $R_i^{(1)}$, they are normalized to the range $[0, 1]$ as in [4] to get their influence scores, I. This grants the methodology two things. First, inputs can be evaluated in terms of influence with respect to each other. Second, now LDs can be made for the most influential input features so what the model learned can be described with words like [16]. eRACANN accepts generated rules into a per-model rule bank if their DU meets a configurable threshold and if their DO does not pass another configurable threshold; these thresholds are expected to vary between data sets and ANN applications, but this experiment required $DU(x) \geq 0.6$ and $DO(x) < 0.2$. The result of admitting per-model rules to a rule bank is now eRACANN can provide trends with respect to the specific $C = (r, m, d)$ that M_C^R was trained on. The data set used to determine DU and DO is the model's own predictions, so rules describe model behavior, not the training or validation data sets.

The presence of x_i in (13) makes it appear that the relevance of an input is dictated by the actual value of the feature. However, this is not the case. Since R_i^1 is dependent on the derivatives of the functions and weights of an already learned model, the relevance of all other input nodes will increase and decrease accordingly, so the normalized influence scores do not

change. This is evidenced by running the prediction step of the system multiple times, during which a random input is selected to use for R, but the rule banks do not change.

5.4. Prediction Results

Like TSS, eRACANN trained and recorded the MAE and RMSE after each epoch on both the training data (80% of the feed files) and the validation data (remaining 20%). The results of the experiment using the configurations described in sections 5.2 and 5.3.2. Due to the large number of models comprising the system, the results for only one month (August) and only the first week are graphed, as in section 4.5. Because feature ablation was explored on simple models and eRACANN inherits its design from them, no additional ablation was performed. Experiments were performed using a few different W, which changes the way data is preprocessed and therefore model behavior. So, results are provided for W values of "30 minutes," "1 hour," and "6 hours," also like section 4.5. As mentioned in section 5.3.2, fastStorage experiments' duration was 10 epochs and $\alpha = 0.01$ as a greater number of epochs was found to greatly diminish predictive accuracy. As a demonstration, eRACANN on the fastStorage data set with $W = 30$ minutes, $\alpha = 0.001$, and over 2000 epochs is graphed in Figure 17. There is only slight change for the better with 25 epochs (Figure 18), but 10 epochs yield results sufficient for comparison. Google trace experiments used $\alpha = 0.001$ and 500 epochs because the data is very noisy and lacks seasonality. Subsections 5.4.1 – 5.4.3 describe the error and prediction graph figures for the different W; prediction graphs were clamped to $[0, 1]$. Subsection 5.4.4 discusses these results.

5.4.1. $W = 30$ Minutes. Figure 19 graphs the loss over time for eRACANN on the Google traces data set and Figure 20 graphs the loss over time on the fastStorage data set. The

final errors for the August-based models for each data set are presented in Table 3. The predictions for eRACANN on Google data are graphed over a VM trace in Figure 21 and the same for the fastStorage data is graphed in Figure 22.

5.4.2. $W = 1$ Hour. The predictions for eRACANN on Google data are graphed over a VM trace in Figure 23 and the same for the fastStorage data is graphed in Figure 24. The error graphs have been omitted because they differ unperceivably from Figures 19 and 20. The final errors for the August-based models for each data set are presented in Table 4.

5.4.3. $W = 6$ Hours. The predictions for eRACANN on Google data are graphed over a VM trace in Figure 25 and the same for the fastStorage data is graphed in Figure 26. The error graphs have been omitted because they differ unperceivably from Figures 19 and 20. It may appear that larger W cause the predictions to smooth out, but because the models are approximating a continuous function, the resolution of the predictions can be turned up or down. Figure 27 shows the same data as Figure 26, but with the same 1-hour resolution as Figure 24. The final errors for the August-based models for each data set are presented in Table 5.

5.4.4. Discussion. The predictions graphed in Figures 17 – 25 exhibit eRACANN's ability to pick up on the seasonality in the seasonal data, but make its struggle with noisy data evident. As hinted at in section 4.5, different W have led to better or worse predictions and are different still between data sets. The both the Google and fastStorage experiments saw the best predictions with W at 6 hours, though the 1-hour fastStorage experiment was also good. The system seemed to generally struggle at $W = 30$ minutes. It is possible the resolution there is too high and so the data is too noisy for something designed to handle seasonality. The larger W,

though, seem to be at a sufficiently "low" resolution that eRACANN finds a way to internally represent the utilization patterns. This is supported by the fact the predictions on the noisiest data, the Google traces, does not become acceptably good until $W = 6$ hours.

These findings seem to directly oppose what is expressed in terms of loss alone. The MAE for eRACANN on fastStorage is consistently high; nonetheless the graphs look decent. It seems reasonable the true effect of the error is lost by clamping the predictions. Though, a sysadmin would be more frustrated by a system with a reportedly low error that predicts poorly than the opposite. The implication of this is eRACANN produces the desired shape, but with greater amplitude than desired, so the clamping is not considered inappropriate.

5.5. Knowledge Extraction

In this implementation, LDs are created as a part of the prediction process. The process of extracting LDs, however, does not need to be repeated until the model changes due to the fact that influence scores are not affected by inputs, as described in section 5.3.3. Furthermore, as a result of the topology of eRACANN models, rules produced by this process are local to the context of the model that produced them. Since rules can relate either input features or time units to the output, these are broken up into Tables 6 and 7, respectively; they list the feature and time summarizers for the August-Monday model trained on Google data, e.g., when $W = 1$ hour, along with their DU. Tables 8 and 9 do the same for the August-Monday fastStorage model. In cases where features or time summarizers appear in more than one rule, those SASCs may joined together with logical OR without changing their meaning.

Feature names have been replaced by square bracketed number representative of the numbered input from Table 1. In the case of inputs 11 and 12 which are, themselves, a series on

inputs, an additional 1-based parenthetical number is added inside the brackets to indicate which input in the series is being referred to. For example, "[11 (1)]" is the first in the series of 12 features added by input 11 in Table 1. The rules themselves have been presented in SASC form as described in section 3.2 and, again, are with respect to only the context the corresponding M_C is for. So, in Table 6, rule 2 indicates that for Mondays in August, predicted utilization will be low when the second input feature (day elapsed) has membership in the fuzzy class "later." The fuzzy membership boundaries for time-based and non-time-based input features are listed in Tables 10 and 11, respectively. These are relevant for interpreting the feature-based rules (Tables 6 and 8) and are triangular in shape. Table 12 lists the fuzzy membership boundaries for the "daytime" universe used in Tables 7 and 9; "daytime" memberships are trapezoidal.

Table 3. Error of August eRACANN models when W = 30 minutes.

	Google traces				fastStorage			
	MAE		RMSE		MAE		RMSE	
C	Train.	Valid.	Train.	Valid.	Train.	Valid.	Train.	Valid.
$(\mathrm{CPU}, 8, 0)$	0.0021	0.0047	0.0045	0.0059	0.2317	0.2446	0.2845	0.3047
$(\mathrm{CPU}, 8, 1)$	0.0020	0.0109	0.0034	0.0111	0.5464	0.4612	0.6494	0.5507
$(\mathrm{CPU}, 8, 2)$	0.0012	0.0088	0.0025	0.0090	0.0958	0.0820	0.1658	0.0129
$(\mathrm{CPU}, 8, 3)$	0.0010	0.0019	0.0194	0.0195	0.1081	0.1258	0.1551	0.1570
$(\mathrm{CPU}, 8, 4)$	0.0014	0.0075	0.0024	0.0079	0.1301	0.1893	0.1833	0.2283
$(\mathrm{CPU}, 8, 5)$	0.0014	0.0115	0.0028	0.0118	0.1899	0.1768	0.2190	0.2111
$(\mathrm{CPU}, 8, 6)$	0.0015	0.0029	0.0029	0.0036	0.1123	0.1070	0.1651	0.1624

Table 4. Error of August eRACANN models when $W = 1$ hour.

	Google traces				fastStorage			
	MAE		RMSE		MAE		RMSE	
C	Train.	Valid.	Train.	Valid.	Train.	Valid.	Train.	Valid.
$(\text{CPU}, 8, 0)$	0.0030	0.0038	0.0056	0.0059	0.2319	0.2446	0.2845	0.3047
$(\text{CPU}, 8, 1)$	0.0032	0.0048	0.0064	0.0065	0.5464	0.4512	0.6494	0.5507
$(\text{CPU}, 8, 2)$	0.0024	0.0039	0.0051	0.0055	0.0958	0.0820	0.1658	0.1285
$(\text{CPU}, 8, 3)$	0.0017	0.0033	0.0034	0.0043	0.1081	0.1258	0.1551	0.1700
$(\text{CPU}, 8, 4)$	0.0016	0.0049	0.0033	0.0057	0.1301	0.1893	0.1833	0.2283
$(\text{CPU}, 8, 5)$	0.0015	0.0076	0.0023	0.0081	0.1899	0.1768	0.2190	0.2111
$(\text{CPU}, 8, 6)$	0.0017	0.0033	0.0035	0.0042	0.1123	0.1070	0.1651	0.1624

Table 5. Error of August eRACANN models when $W = 6$ hours.

	Google traces				fastStorage			
	MAE		RMSE		MAE		RMSE	
C	Train.	Valid.	Train.	Valid.	Train.	Valid.	Train.	Valid.
$(\text{CPU}, 8, 0)$	0.0053	0.0311	0.0463	0.0550	0.3023	0.2238	0.3669	0.2890
$(\text{CPU}, 8, 1)$	0.0030	0.0243	0.0246	0.0476	0.2030	0.1425	0.2823	0.2091
$(\text{CPU}, 8, 2)$	0.0031	0.0250	0.0338	0.0478	0.3590	0.2404	0.4502	0.3106
$(\text{CPU}, 8, 3)$	0.0032	0.0182	0.0223	0.0443	0.1957	0.1964	0.2390	0.2380
$(\text{CPU}, 8, 4)$	0.0033	0.0101	0.0266	0.0344	0.1576	0.1253	0.2041	0.1518
$(\text{CPU}, 8, 5)$	0.0024	0.0478	0.0228	0.0531	0.3063	0.2171	0.3669	0.2717
$(\text{CPU}, 8, 6)$	0.0026	0.0433	0.0176	0.0495	0.1246	0.1030	0.1835	0.1695

Table 6. Feature-based rules for August-Mondays ($W = 1$ hour, Google data set)

Rank	SASC	DU
1	IF [4] IS low THEN utilization IS low	0.8704
2	IF [2] IS later THEN utilization IS low	0.8122
3	IF [2] IS end THEN utilization IS low	0.8076
4	IF [12 (12)] IS low THEN utilization IS low	0.7914
5	IF [5] IS minimal THEN utilization IS low	0.7841
6	IF [2] IS early THEN utilization IS low	0.7764
7	IF [4] IS medium THEN utilization IS low	0.7737
8	IF [11 (1)] IS low THEN utilization IS low	0.7707
9	IF [11 (1)] IS minimal THEN utilization IS low	0.7560
10	IF [11 (8)] IS minimal THEN utilization IS low	0.7556
11	IF [11 (8)] IS low THEN utilization IS low	0.7528
12	IF [11 (2)] IS minimal THEN utilization IS low	0.7411
13	IF [2] IS middle THEN utilization IS low	0.7364
14	IF [2] IS beginning THEN utilization IS low	0.7118
15	IF [11 (8)] IS high THEN utilization IS low	0.7037
16	IF [11 (1)] IS high THEN utilization IS low	0.6807
17	IF [11 (1)] IS medium THEN utilization IS low	0.6756
18	IF [11 (12)] IS medium THEN utilization IS low	0.6745
19	IF [11 (12)] IS high THEN utilization IS low	0.6539
20	IF [11 (8)] IS medium THEN utilization IS low	0.6516

Table 7. Time-based rules for August-Mondays ($W = 1$ hour, Google data set)

Rank	SASC	DU
1	IF daytime IS afternoon THEN utilization IS low	0.8738
2	IF daytime IS evening THEN utilization IS low	0.8488
3	IF daytime IS morning THEN utilization IS low	0.6533
4	IF daytime IS early THEN utilization IS low	0.7209

Table 8. Feature-based rules for August-Mondays ($W = 1$ hour, fastStorage data set)

Rank	SASC	DU
1	IF [10] IS minimal THEN utilization IS minimal	0.8551
2	IF [1] IS beginning THEN utilization IS minimal	0.7914
3	IF [1] IS end THEN utilization IS minimal	0.6533
4	IF [1] IS early THEN utilization IS minimal	0.6234

Table 9. Time-based rules for August-Mondays ($W = 1$ hour, fastStorage data set)

Rank	SASC	DU
1	IF daytime IS evening THEN utilization IS very high	1.0000
2	IF daytime IS afternoon THEN utilization IS very high	0.9952
3	IF daytime IS night THEN utilization IS very high	0.9717
4	IF daytime IS early THEN utilization IS very high	0.9353
5	IF daytime IS morning THEN utilization IS very high	0.8572

Table 10. Fuzzy membership boundaries for normalized time-based input features.

Fuzzy Variable	Start	Peak	End
Beginning	0	0	0.22
Early	0	0.22	0.33
Middle	0.22	0.5	0.77
Later	0.66	0.77	1
End	0.77	1	1

Table 11. Fuzzy membership boundaries for normalized non-time input features.

Fuzzy Variable	Start	Peak	End
Minimal	0	0	0.25
Low	0.1	0.25	0.5
Medium	0.3	0.5	0.7
High	0.5	0.75	0.9
Very high	0.75	1	1

Table 12. Fuzzy membership boundaries for the “daytime” descriptor.

Fuzzy Variable	Start[1]	Peak Start[1]	Peak End[1]	End[1]
Early	0	0	0.21	0.25
Morning	0.21	0.25	0.50	0.51
Afternoon	0.50	0.51	0.67	0.75
Evening	0.71	0.75	0.83	0.86
Night	0.83	0.86	1	1

[1]Values are fractions of a 24-hour day.

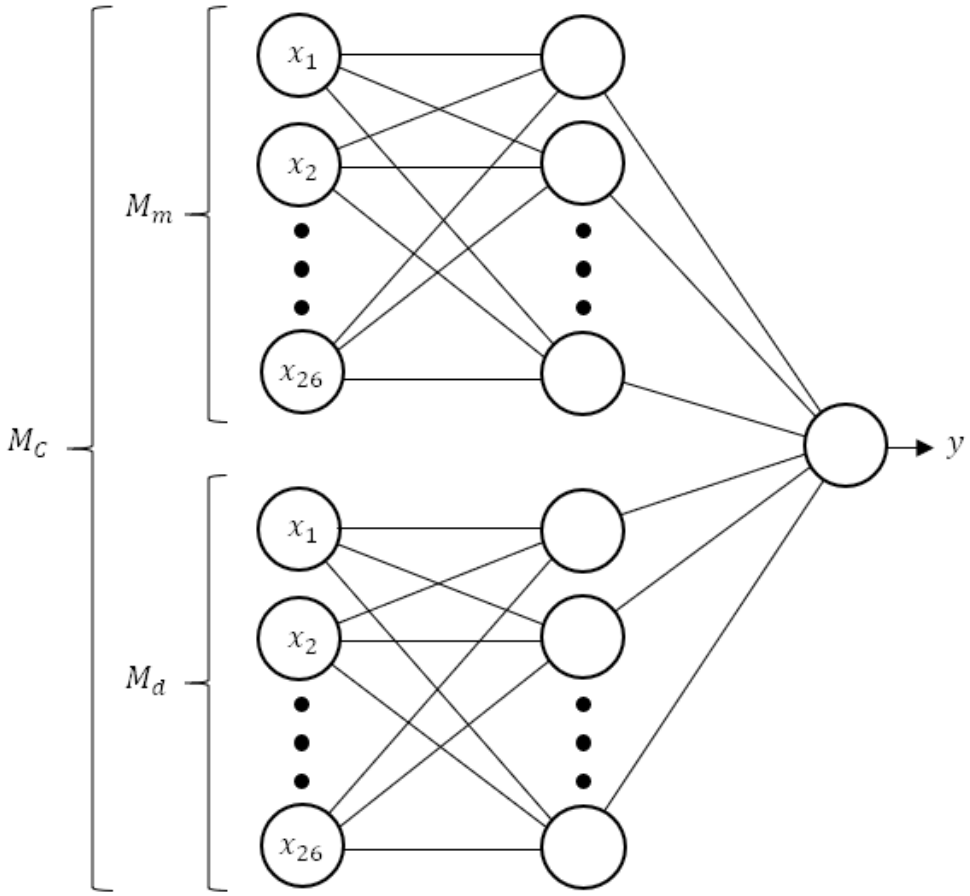

Figure 14. The topology of an eRACANN model $M_C = M_m + M_d$.

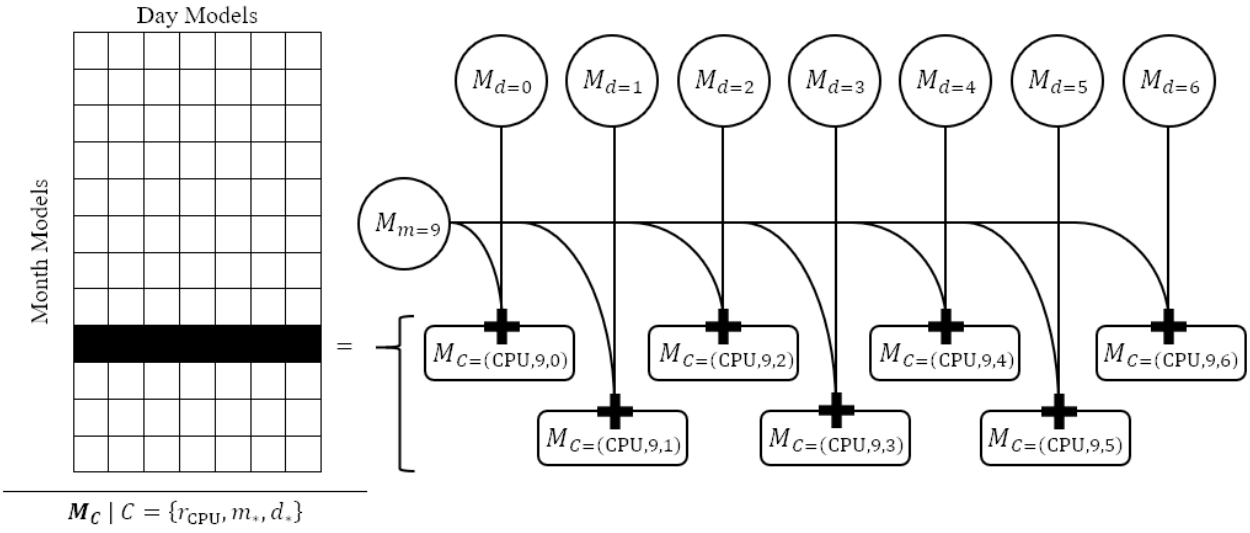

Figure 15. The set of all possible $\boldsymbol{M_C}$ for r_{CPU} as a lookup table.

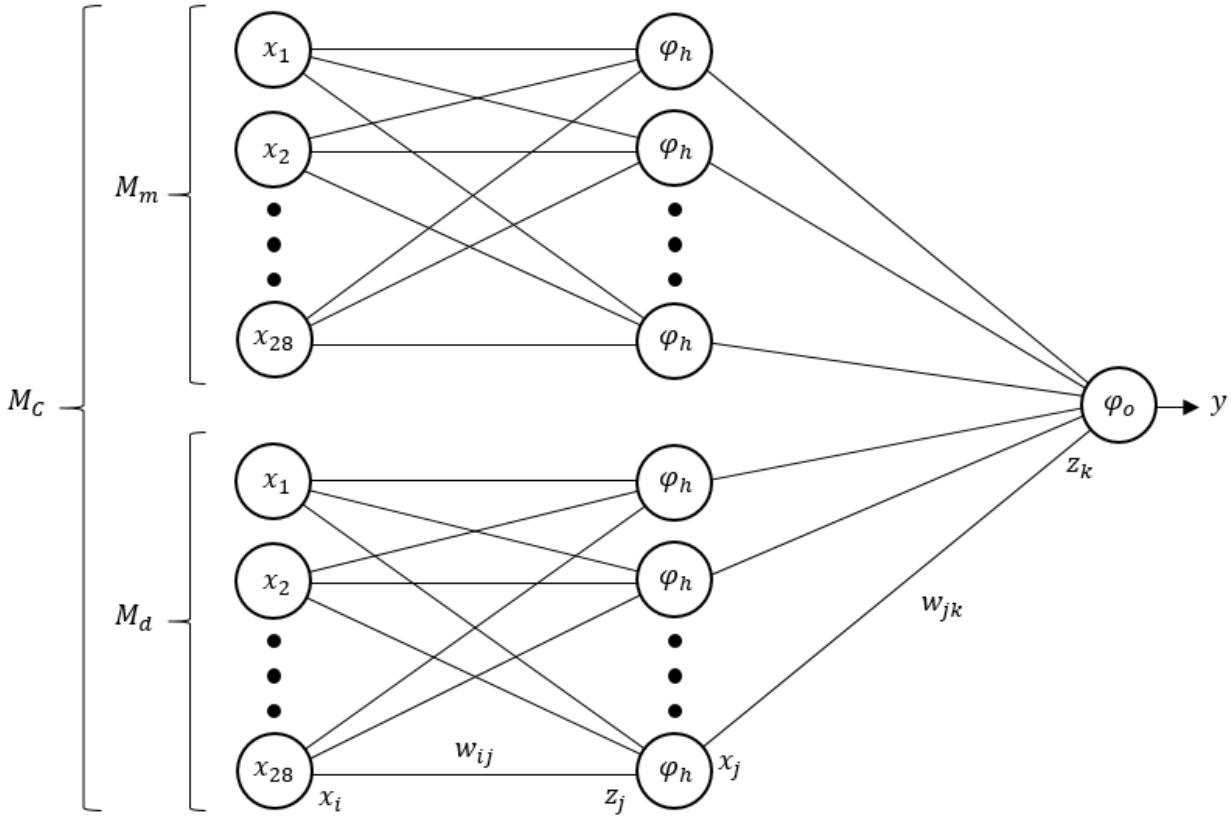

Figure 16. The logical location of the values used in (13).

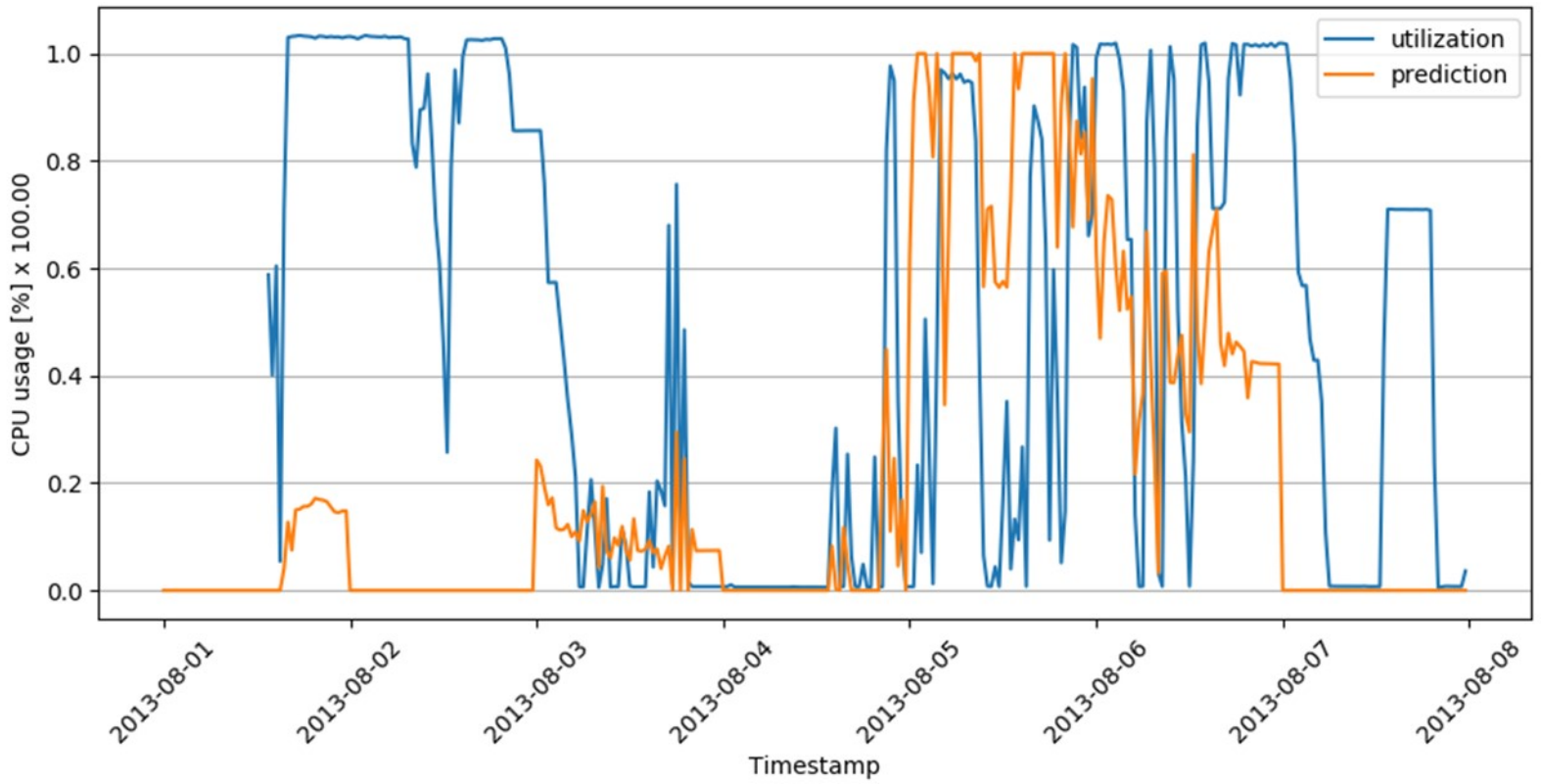

Figure 17. FastStorage eRACANN CPU predictions for $W = 30$ minutes, trained over 500 epochs.

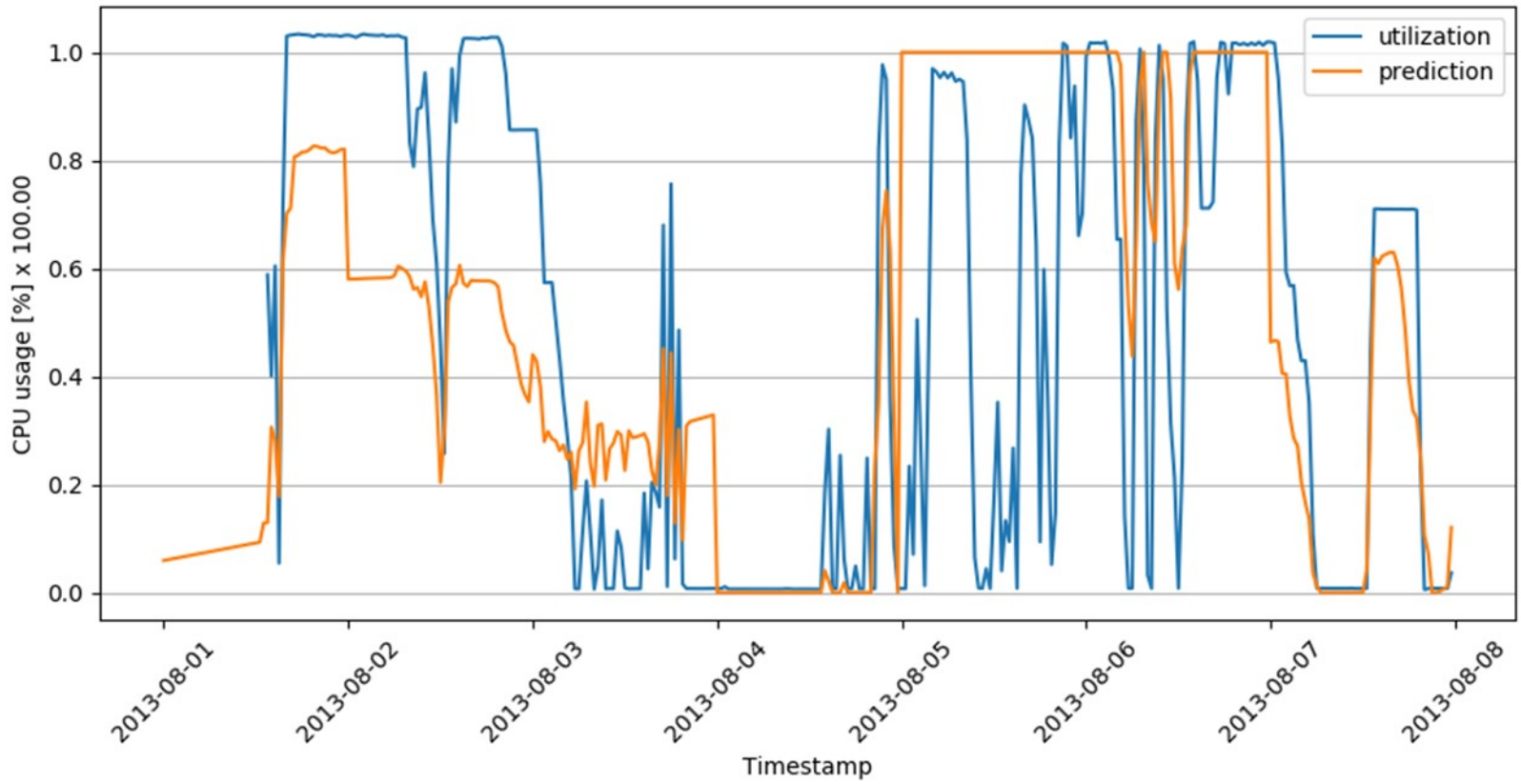

Figure 18. FastStorage eRACANN CPU predictions for $W = 30$ minutes, trained over 25 epochs.

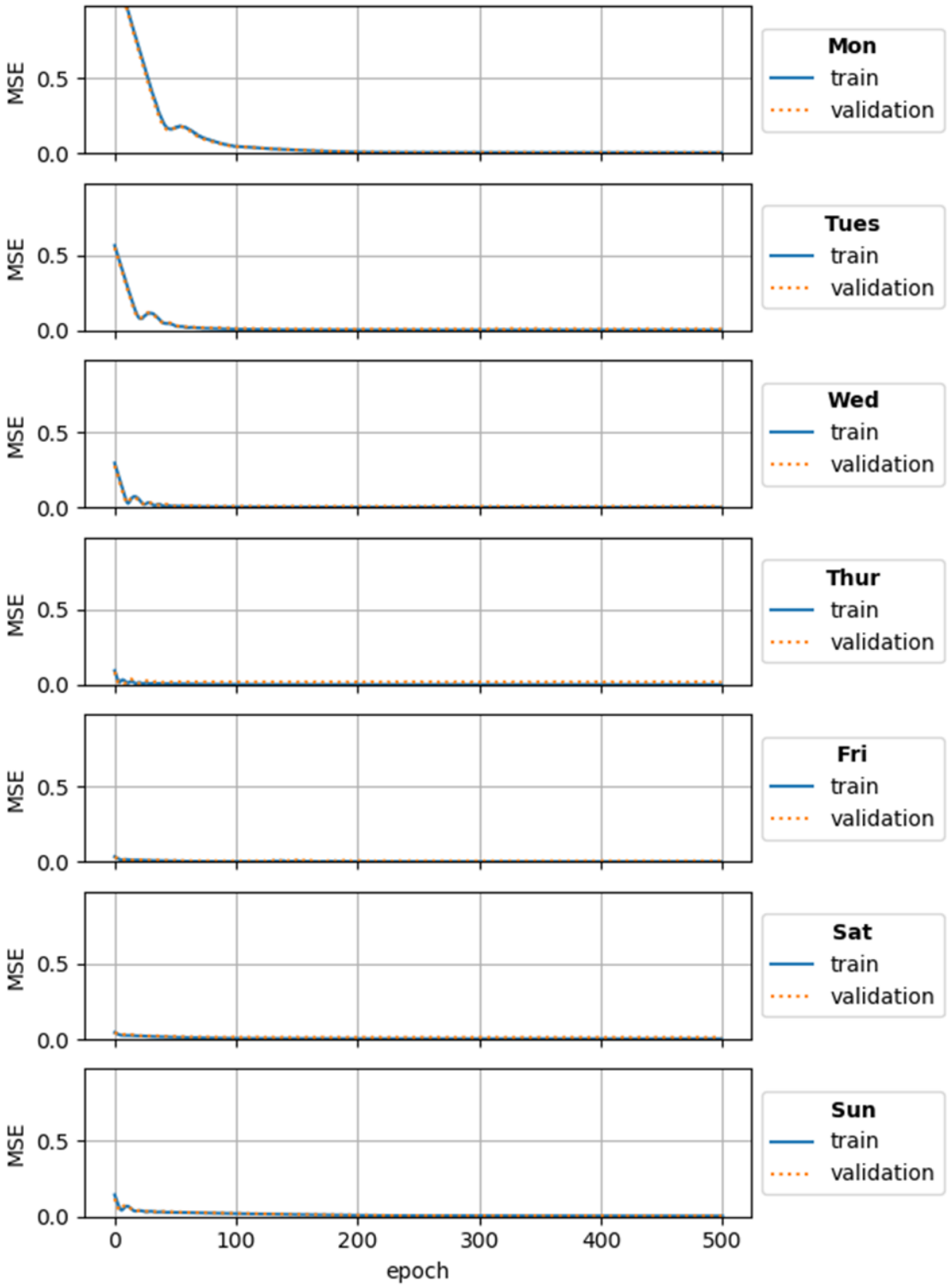

Figure 19. The MAE for eRACANN on the Google trace data for $W = 30$ minutes.

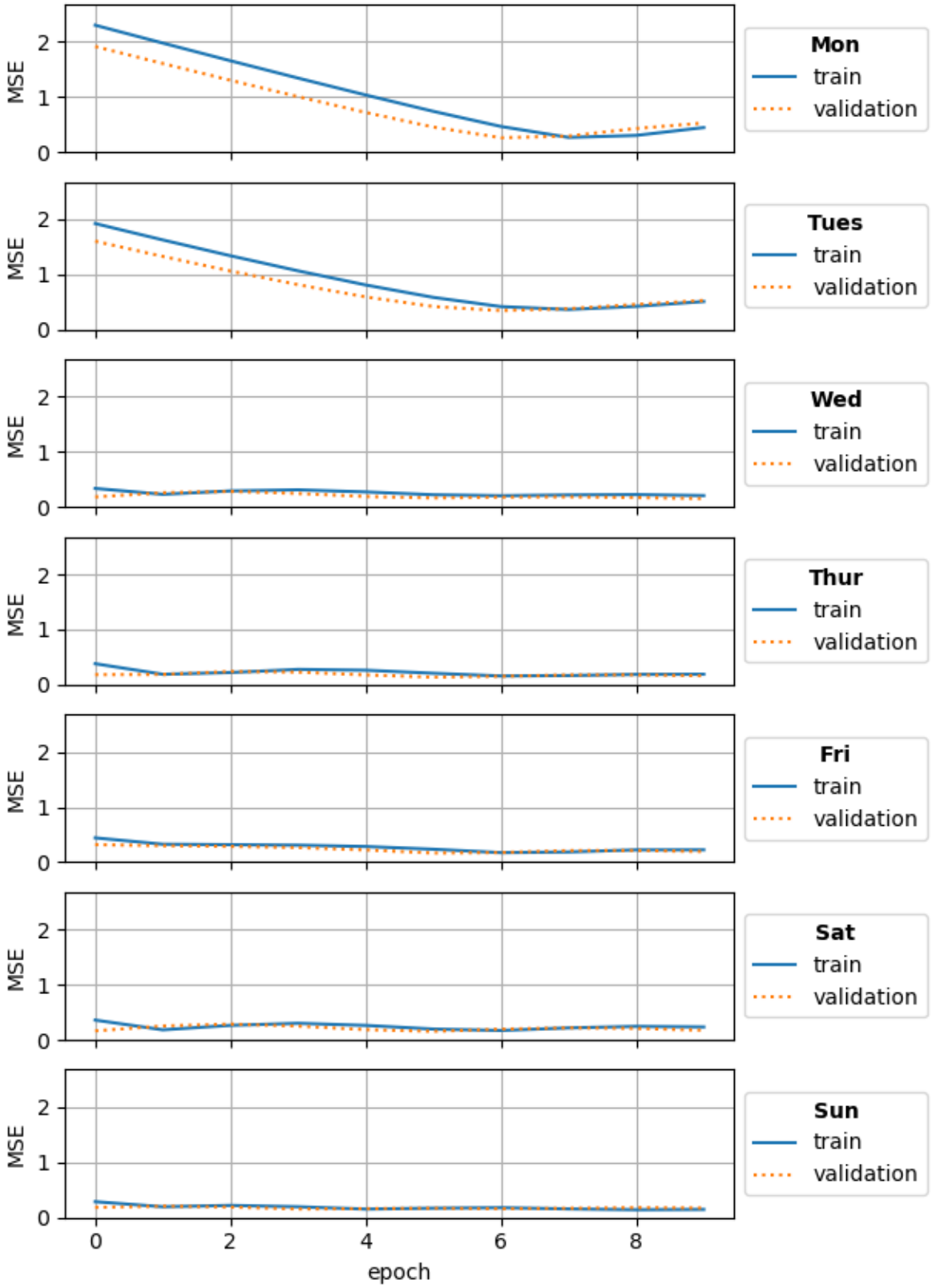

Figure 20. The MAE for eRACANN on the fastStorage data for $W = 30$ minutes.

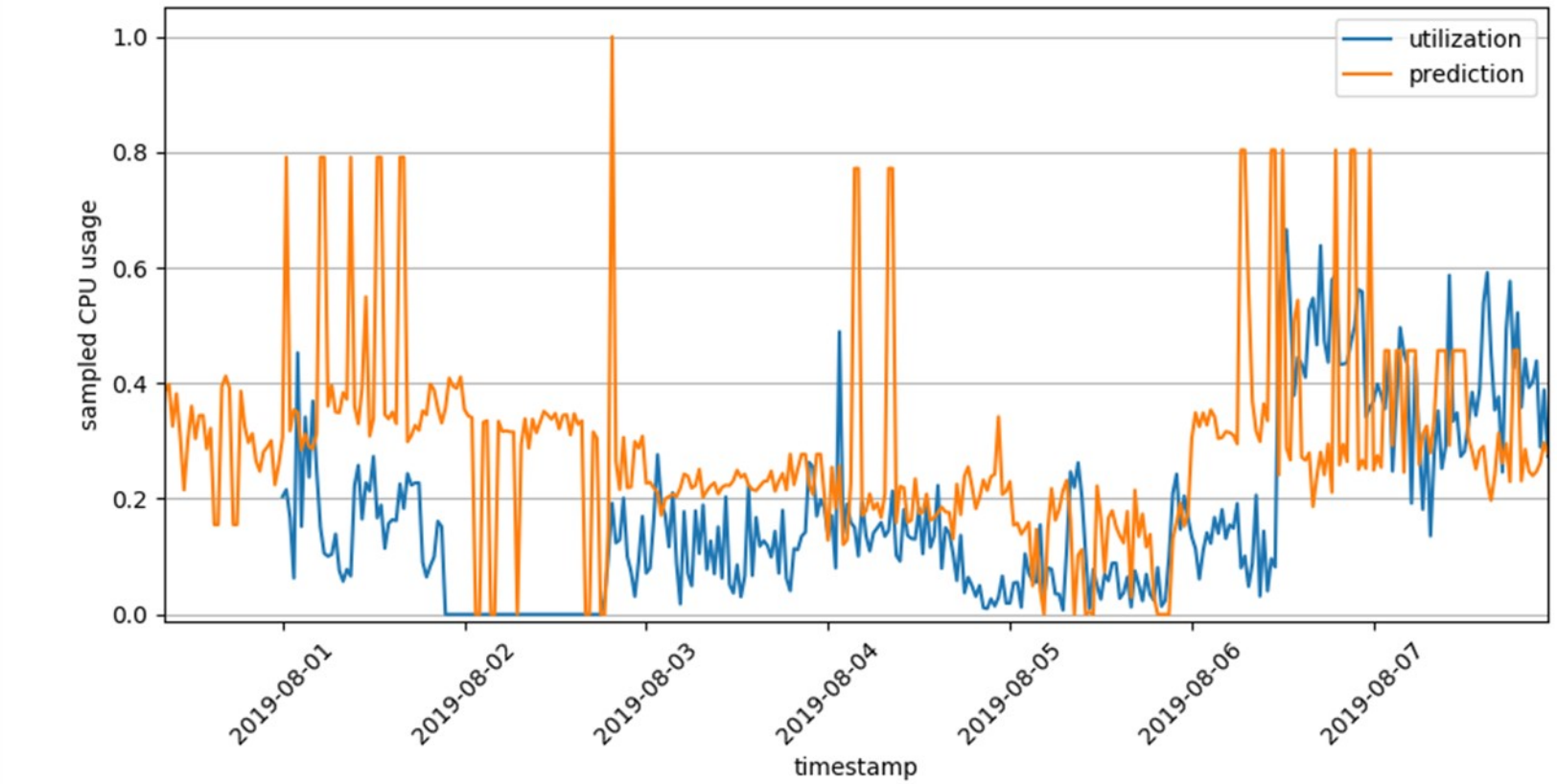

Figure 21. Google trace eRACANN CPU predictions for $W = 30$ minutes.

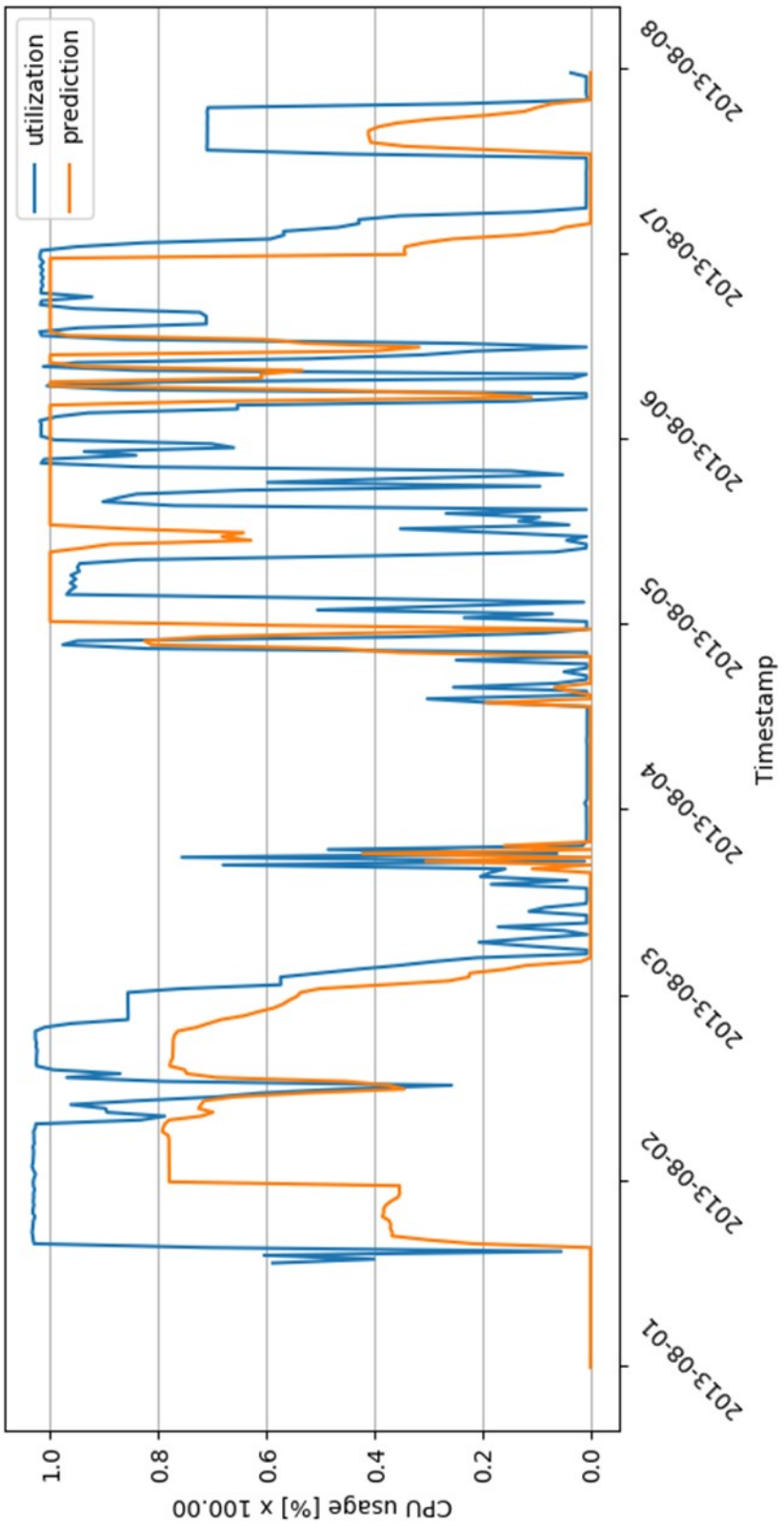

Figure 22. FastStorage eRACANN CPU predictions for $W = 30$ minutes.

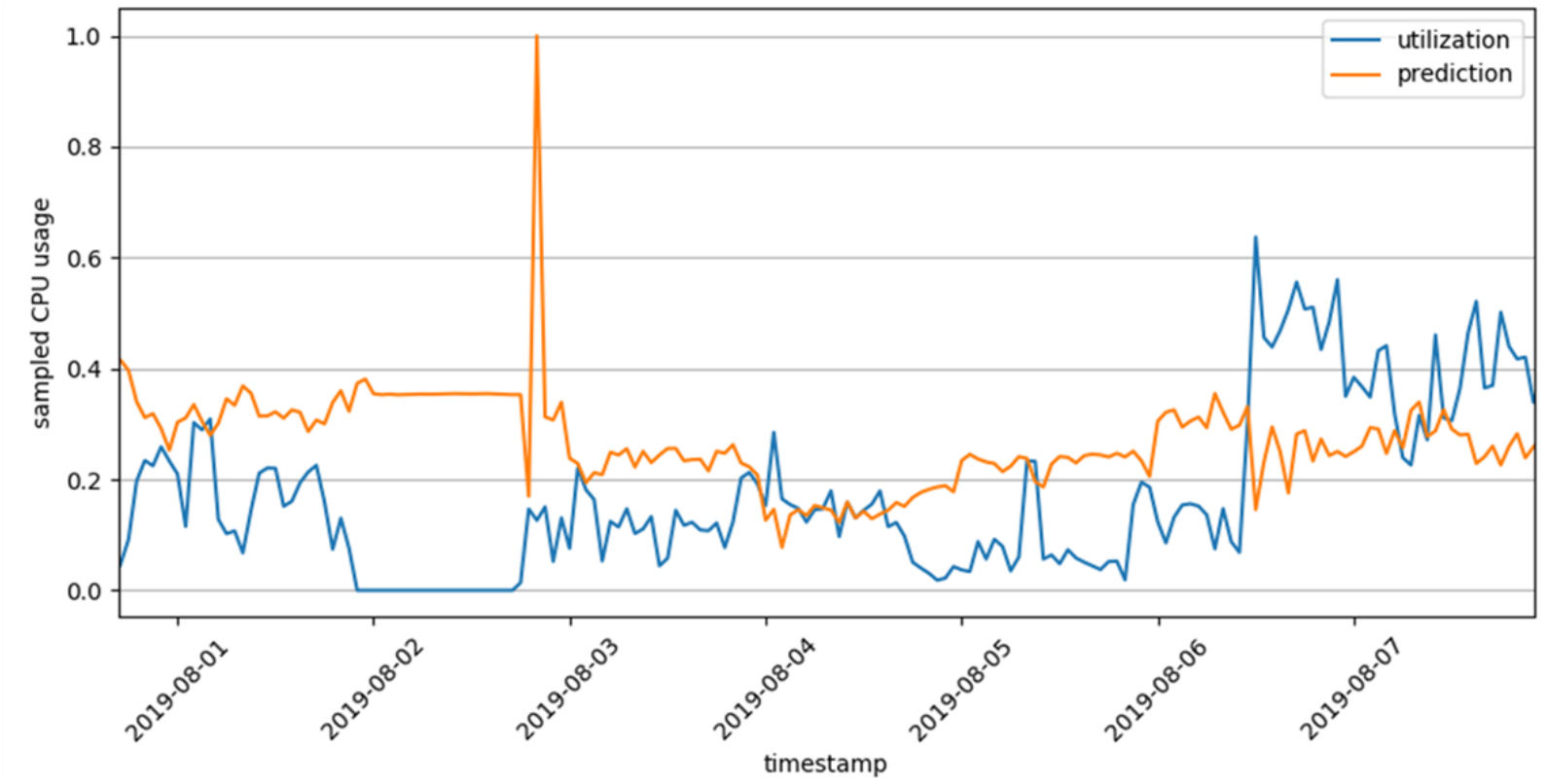

Figure 23. Google trace eRACANN CPU predictions for $W = 1$ hour.

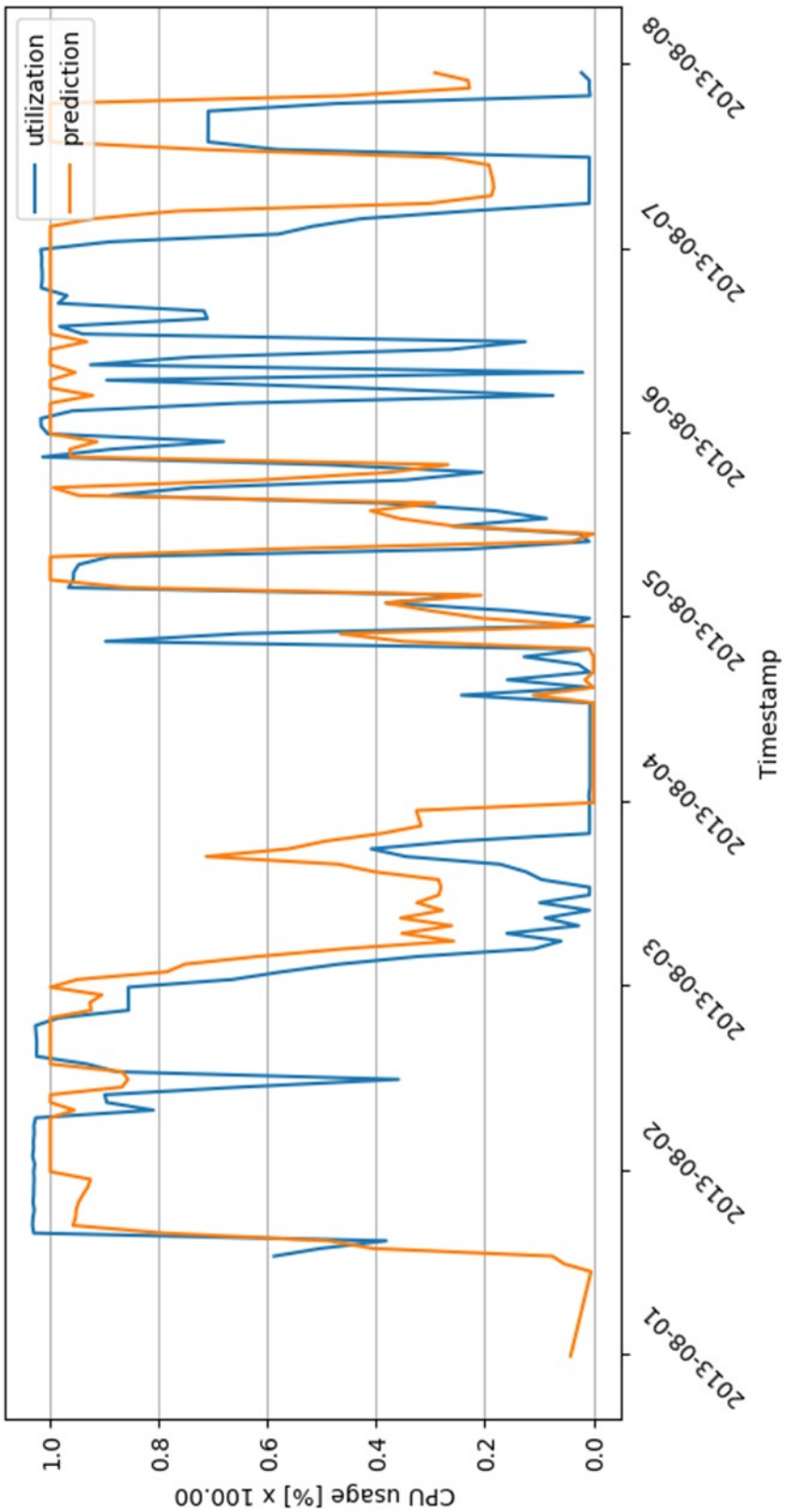

Figure 24. FastStorage eRACANN CPU predictions for $W = 1$ hour.

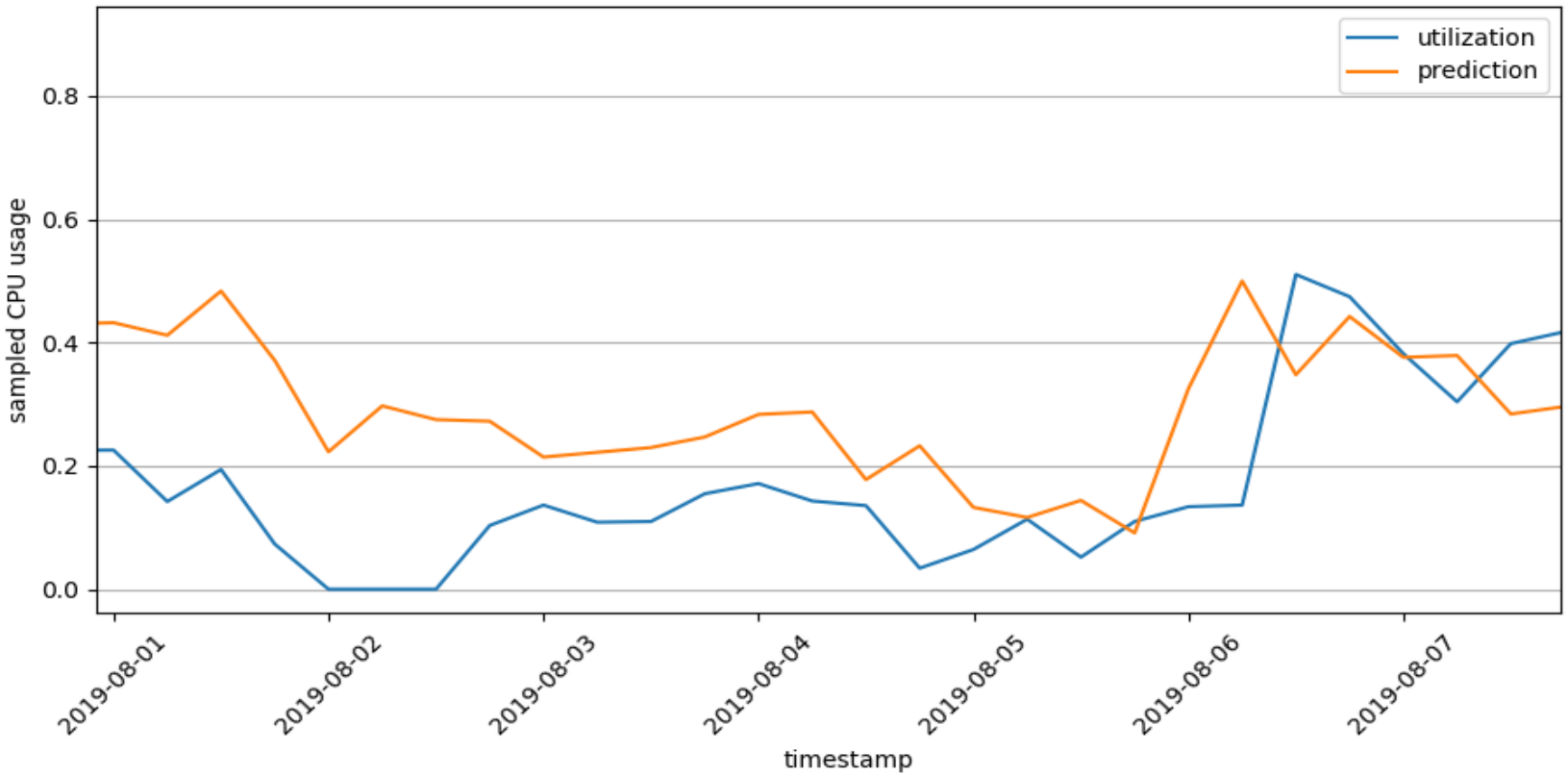

Figure 25. Google trace eRACANN CPU predictions for $W = 6$ hours.

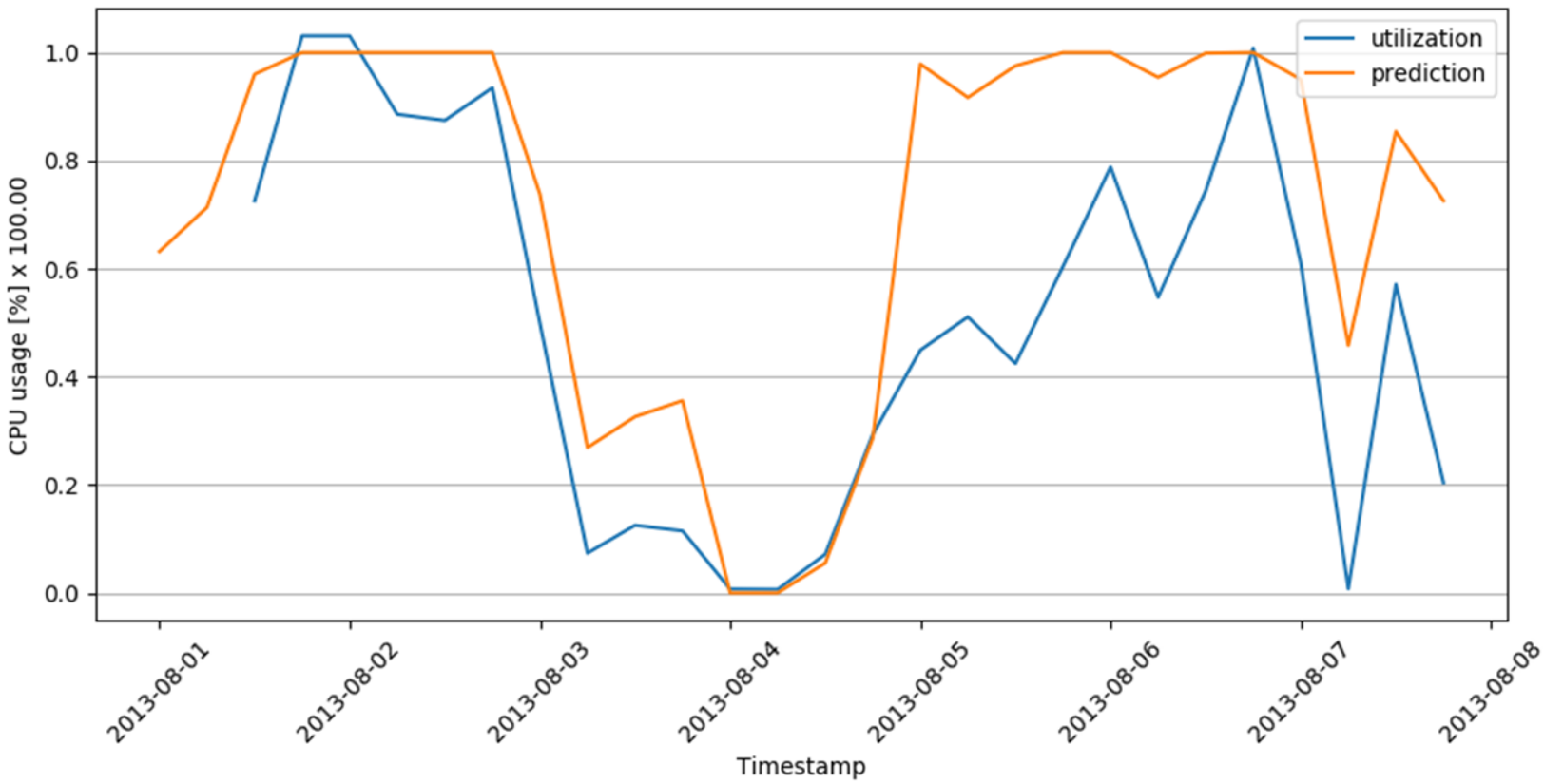

Figure 26. FastStorage eRACANN CPU predictions for $W = 6$ hours.

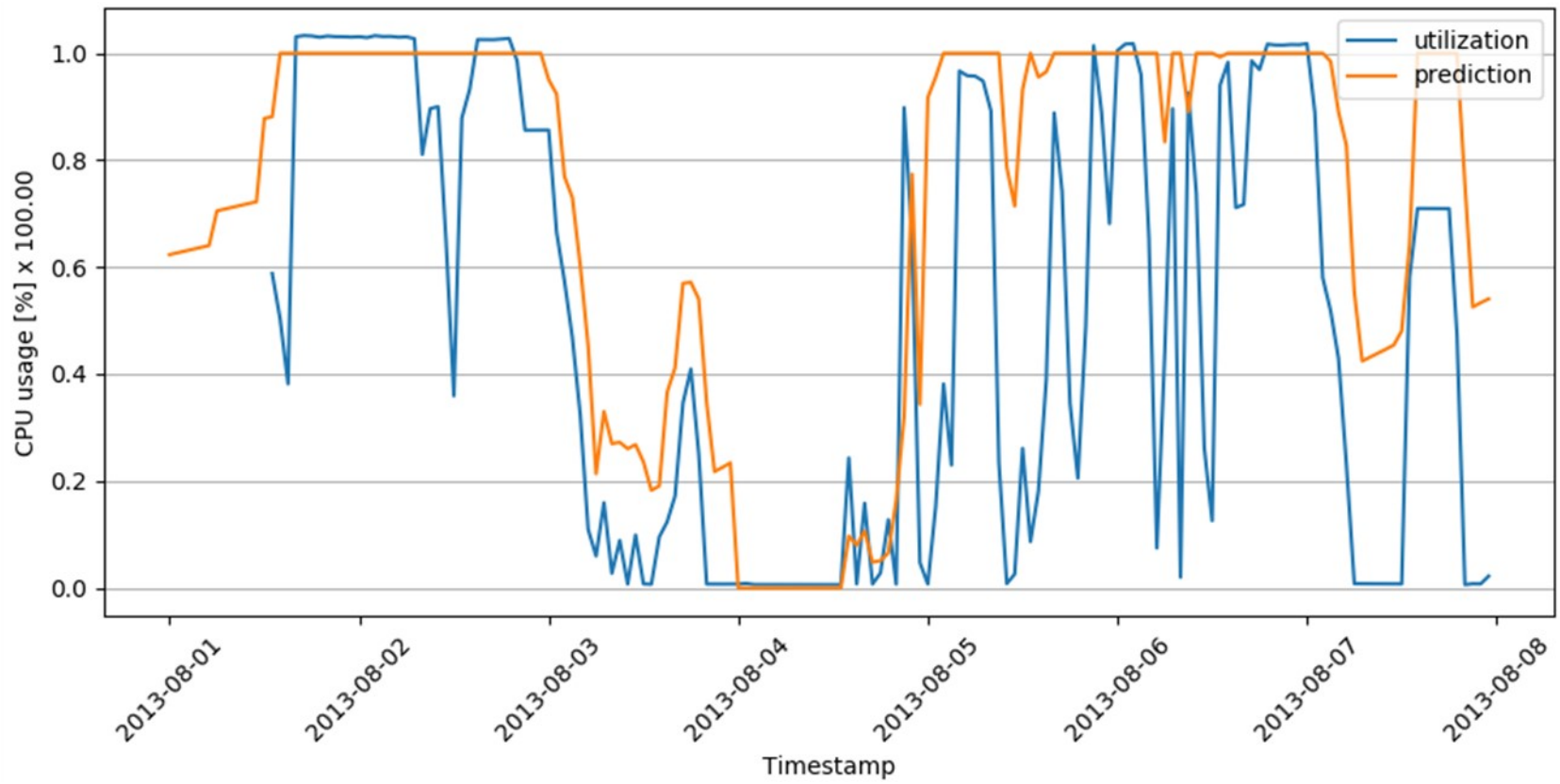

Figure 27. FastStorage eRACANN CPU predictions for $W = 6$ hours at 1-hour resolution.

6. RESULTS ANALYSIS AND COMPARISON

6.1. Observations

This section serves to reflect and expand on the data sets used in these experiments and the performance of models. The first observation is fastStorage traces exhibit seasonality between VMs. Whether the VMs were all tenants on the same physical system or not, their utilizations over the span of the trace are almost identical. The set of all traces could probably be put in about three bins, were each bin is seasonal, just in a different way. This makes the fastStorage data set an ideal testing environment for the assumption that TSS and eRACANN make about the data they perform regression on. The Google traces, though, do not appear to exhibit seasonality at all; that is, they are very noisy. This means the data goes against the assumption of seasonality that these systems rely on. It also means if eRACANN performs well on it anyway, the system is robust. If it fails spectacularly then it is highly sensitive to its assumptions. As shown by the results in section 5, it did okay on the noisy data once the W was wide enough. It also appears to have needed a W of at least 1 hour to find the seasonality of the fastStorage data; at $W = 6$ hours it starts to compete with TSS, but in way fewer epochs.

Another point worth mentioning is about the errors reported in Tables 3, 4 and 5. When eRACANN tested over the data sets for 2000 epochs, they were concerningly low, $(0 < error \ll 10^{-4})$, which raised a red flag the models are probably overfit. Looking at the prediction graphs for this setup indicated as much by the way the prediction went from 0 to 1 and back every other time step. But, due to the noisy nature of the Google traces, the steep slopes were just accidentally very close to the validation data. This goes to show that while error metrics are useful as a means of comparison, they must also be quantified in another way to give

them more substantial meaning. So, even though these are concerningly high, the prediction graphs are empirically accurate and most importantly, useful.

6.2. Baseline for Comparison

To aid in quantifying the efficacy of eRACANN, the baseline for comparison is one of the most ubiquitous forms of ANN found across the literature, a DNN. It is made more comparable to the systems presented thus far by sharing much of TSS's configuration. The DNN has the same 28 inputs, two hidden layers of 12 nodes each, and one output node. The hidden activation functions are sigmoid with the identity function in the output layer. Training used an $\alpha = 0.001$ over 100 epochs, $W = 30$ minutes, and a batch size of the entire data set. Loss was optimized using Adam. To provide a single, simple common ground that adheres to the assumptions of TSS and eRACANN, evaluation of the DNN is performed with respect to the fastStorage system only. Figure 28 graphs the loss over time for the DNN. Figure 29 graphs the prediction for the first week of August. The errors for the DNN are in Table 13 where they are compared to TSS and eRACANN.

6.3. Comparison

The first means of comparison is the error. Since TSS and eRACANN are systems of many models, the average error among the August models is used. Table 13 continues to show that error metrics demand quantification. Even though the DNN beats TSS and eRACANN in MAE, TSS actually has a lower RMSE. Regardless of this, though, the DNN reaches this error in only a fourth as many epochs as TSS. Suffice it to say eRACANN's error alone is not ideal.

The rules extracted from the DNN using the same methodology of eRACANN are presented in Tables 14 and 15, which are its feature-based and time-based rules, respectively. The rules generated for the DNN did not meet the DU threshold used in section 5.5, so the top 10 were selected instead. The same fuzzy variables are used in these tables with the addition of two more descriptors, the "weekday" and the "week-time." Simply, the weekday is one of the days of the week and the week-time is either literally the value "weekday" in the case of Monday through Friday or "weekend" otherwise. The week-time was intended to help identify regular work-week versus weekend trends, provided sufficient seasonality presented itself. The membership shape for these descriptors is trapezoidal and lie on the domain $[0, 7]$; the "weekday" is listed in Table 16 and Table 17 describes the "week-time."

The DNN gains access to these larger-scope descriptors because the model itself is not limited in scope; that is, it can be said it has a global context since it must train on all the data. So even though the DNN has the advantage in quick learning for accurate predictions, this is actually a place where eRACANN has a unique advantage. Since each model and sub-model in eRACANN only operates in a specific context, it only has access to a subset of the data. This means for any eRACANN model, its generated rules are local to that context and the trends in its data, enabling a different set of input features to be more important on Mondays versus Fridays. It is possible for the DNN to try to localize rules to time-units by training on everything and holding the month and day inputs constant for predictions. In fact, this would work. However, the same cannot be done, directly, for the features. If month and day inputs have global scope, the only way to verify features are more or less important with respect to a time is to use multiple antecedent-single consequent (MASC) rules. Then one must choose what comprises a MASC rule, e.g. how many antecedents, which without guidance, devolves into brute force

combinatorics on the set of all possible fuzzy variables, a value that quickly explodes. eRACANN gets relationships between month-day pairs and the input features "for free" by virtue of the fact that the data it can access is restricted to that month-day.

6.4. Application

Bringing Report Corp. back into the picture, the entire above system, eRACANN is plugged into their environment. The underpaid sysadmin chose the desired prediction window and fed utilization histories through it, which built new models local to the context of the new data as it was encountered. Once training completes, initial predictions for r_{CPU} and r_{RAM} on s_{SaaS} and s_{IaaS} are generated so resources can be allocated or deallocated ahead of time. As this happens, system utilization is still occurring; that is, the course of business generates new data that eRACANN can be trained on. This is not an unrealistic thing to do since individual data points, or at least any set smaller than the initial training data, will not take anywhere near as long to update model weights. The author's particular implementation of eRACANN was designed to be "always-on," so at any point, a new batch of data can be trained on. This would require influence scores be recalculated, but eRACANN models are small, so the process is not all that computationally expensive.

The prediction and rule components are expected to be always-on as well. With utilization histories available and the rolling files used during training, the prediction component can receive a stream from a live system and just output new predictions as time progresses. In the event the sysadmin knows something eRACANN does not, it can always be ignored. Conversely, one may see that there is a "highly useful" (high DU) rule that is generally true but

is not coming through on the next prediction for whatever reason, and so that rule is used to guide modifying available resources.

Table 13. Errors of TSS, eRACANN, and the DNN for August when $W = 30$ minutes.

	MAE		RMSE	
System	Training	Validation	Training	Validation
TSS	0.0174	0.0315	**0.0550**	**0.0635**
eRACANN	0.2020	0.1981	0.2603	0.2324
DNN	**0.0082**	**0.0083**	0.0907	0.0914

Table 14. Feature-based rules for DNN, August ($W = 30$ minutes, fastStorage data set)

Rank	SASC	DU
1	IF [3] IS minimal THEN utilization IS low	0.5290
2	IF [5] IS minimal THEN utilization IS low	0.4831
3	IF [10] IS medium THEN utilization IS low	0.4167
4	IF [10] IS high THEN utilization IS minimal	0.4167
5	IF [10] IS high THEN utilization IS low	0.4167
6	IF [10] IS minimal THEN utilization IS minimal	0.3495
7	IF [3] IS minimal THEN utilization IS minimal	0.3322
8	IF [10] IS minimal THEN utilization IS low	0.3125
9	IF [5] IS minimal THEN utilization IS minimal	0.2704
10	IF [10] IS medium THEN utilization IS minimal	0.2301

Table 15. Time-based rules for DNN, August ($W = 30$ minutes, fastStorage data set)

Rank	SASC	DU
1	IF week-time IS weekday THEN utilization IS low	0.4275
2	IF weekday IS Sat THEN utilization IS low	0.4144
3	IF daytime IS early THEN utilization IS minimal	0.3125
4	IF daytime IS afternoon THEN utilization IS low	0.2500
5	IF daytime IS afternoon THEN utilization IS minimal	0.2500
6	IF daytime IS early THEN utilization IS low	0.2500
7	IF weekday IS Sat THEN utilization IS minimal	0.2364
8	IF week-time IS weekday THEN utilization IS minimal	0.2311
9	IF daytime IS evening THEN utilization IS low	0.2083
10	IF daytime IS evening THEN utilization IS minimal	0.2083

Table 16. Fuzzy membership boundaries for the "weekday" descriptor.

Fuzzy Variable	Start	Peak Start	Peak End	End
Mon	0	0	0.99	1.01
Tues	1	1.01	1.99	2.01
Wed	2	2.01	2.99	3.01
Thurs	3	3.01	3.99	4.01
Fri	4	4.01	4.99	5.01
Sat	5	5.01	5.99	6.01
Sun	6	6.01	7	7

Table 17. Fuzzy membership boundaries for the “week-time” descriptor.

Fuzzy Variable	Start	Peak Start	Peak End	End
Weekday	0	0	4.99	5.01
Weekend	5	5.01	7	7

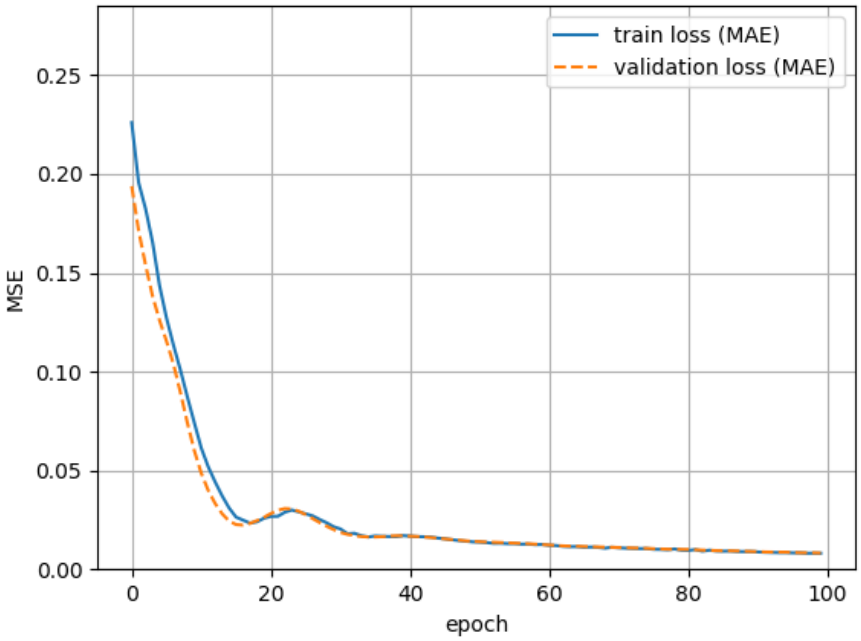

Figure 28. The MAE for the DNN on the fastStorage data.

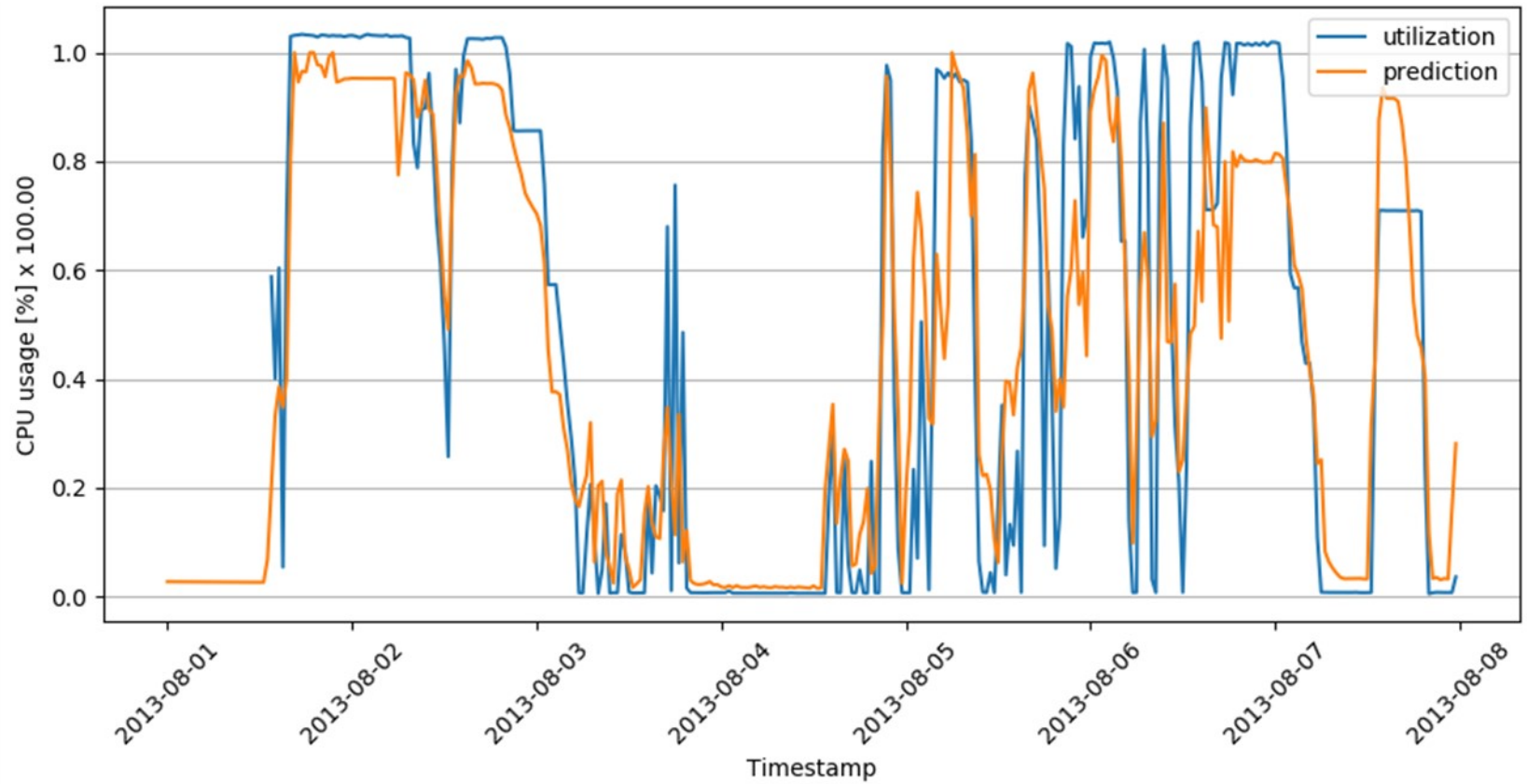

Figure 29. DNN FastStorage CPU predictions.

7. FUTURE WORK

In this section, ways eRACANN can be improved or extended are briefly explored based on the results found by the preceding experiments. After observing the efficiency of the DNN with respect to eRACANN, the elephant in the room is representing sub-models with DNNs rather than two-layer MLPs. Or, perhaps, TSS had it right and cooperative agents should not be represented by split ANNs; that is, for maximally powerful ANNs, they should be fully connected. There is also the potential for LSTMs to pay off here as they are known for being especially good at timeseries predictions [3], though that would make calculating input influence much more difficult. So, two avenues of research are building eRACANN models with different kinds of MLPs instead, or a deep simple system with one deep model per context.

Another facet of eRACANN potentially worth exploring, or exploiting, is the generic extensibility of the concept of model contexts. With particular regard to time-based rules, it is easy to see how rules that regard the week of the year or which are sensitive to holidays could be useful especially to, say, e-commerce. One could investigate how eRACANN benefits from access to more contexts. This would take the form of an additional MLP for each context. So, to test the aforementioned additional time units, eRACANN would need a year model, a week model, etc. Just as before, these would all be joined together at runtime to form the full context. It would also then be able to provide even more kinds of guidance since now LDs can be generated from more contexts. The year model, for example, could aim to generate rules for making predictions around common yearly events and holidays like the Super Bowl or Black Friday.

One could also consider contexts other than time-based ones or perhaps in addition to them. Indeed, any unit that can be measured and has limited scope can serve as a context as long as there is data for it. One way that contexts could be redefined to include non-time units is, say, demarcating when the weather was sunny in addition to the month; that would probably help predict hot dog stand sales better than system utilization, though. Alternatively, the time contexts can take different forms like which season it is or which of the four years it is in between United States presidential elections. Large-scale time units combined with LSTMs or DNNs could potentially yield some very long-term, but accurate LDs given their ability to represent and abstract complicated time series-based problems.

One of the biggest problems with approaching large scale data with time-based contexts is researchers must be able to find enough data spanning a suitably long time to test their ideas. The Google set, for example, is very large, but only actually spans a month. There was also no indication it contained seasonality or any other qualities conducive to making eRACANN look better. So, time spent either collecting or synthesizing this kind of data would be beneficial to many.

8. CONCLUSION

This work studied the idea of explainable, runtime-assembled, cooperative artificial neural networks: eRACANN. This was initially explored by representing each discrete context with a single ANN in TSS with promising results. Explainability was added to the extended system in such a way that it was not sensitive to input values. Rather, the internal weights and activation functions determined the relative contribution of each input node. This made for a stable way of finding the influence of features so their effect on prediction could be described linguistically. eRACANN did not compete with the error metric obtained by a DNN, though both achieved reasonably accurate and useful predictions at a minimum. Nonetheless, eRACANN alone did show that its unique architecture facilitates knowledge extraction that is local to a model's context. This has some interesting implications and a definitive application to cloud environments where sysadmins can receive insight on why the system is suggesting allocations and deallocations to their *aaS.

REFERENCES

[1] J. Huang, C. Li, J. Yu, Resource prediction based on double exponential smoothing in cloud computing, 2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet), Yichang, 2012, pp. 2056-2060, doi: 10.1109/CECNet.2012.6201461.

[2] T. Mehmood, S. Latif, S. Malik, Prediction of cloud computing resource utilization, 2018 15th International Conference on Smart Cities: Improving Quality of Life Using ICT & IoT (HONET-ICT), Islamabad, 2018, pp. 38-42, doi: 10.1109/HONET.2018.8551339.

[3] M. Hassan, H. Chen, Y. Liu, DEARS: a deep learning based elastic and automatic resource scheduling framework for cloud applications, 2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom), Melbourne, Australia, 2018, pp. 541-548, doi: 10.1109/BDCloud.2018.00086.

[4] K. Amarasinghe, M. Manic, Explaining what a neural network has learned: toward transparent classification, 2019 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), New Orleans, LA, USA, 2019, pp. 1-6, doi: 10.1109/FUZZ-IEEE.2019.8858899.

[5] W. Duch, Coloring black boxes: visualization of neural network decisions, Proceedings of the International Joint Conference on Neural Networks, 2003., Portland, OR, 2003, pp. 1735-1740 vol.3, doi: 10.1109/IJCNN.2003.1223669.

[6] Y. Duan, G. Fu, N. Zhou, X. Sun, N. C. Narendra, B. Hu, Everything as a service (XaaS) on the cloud: origins, current and future trends, 2015 IEEE 8th International Conference on Cloud Computing, New York, NY, 2015, pp. 621-628, doi: 10.1109/CLOUD.2015.88.

[7] W. S. McCulloch, W. Pitts, A logical calculus of the ideas imminent in nervous activity, Bulletin of Mathematical Biophysics*,* vol. 5, 1943, pp. 115-133.

[8] F. Rosenblatt, A probabilistic model for information storage and organization in the brain, Cornell Aeronautical Laboratory, Psychological Review, vol. 6, 1958, pp. 386-408.

[9] J. Brownlee, Clever algorithms, revision 2, 2011, pp. 234-335. Accessed on: May 27, 2020. [Online]. Available: https://github.com/clever-algorithms/CleverAlgorithms.

[10] B. Csaji, Approximation with artificial neural networks, M.S. thesis, Eötvös Loránd Univ., Hungary, 2001. Accessed on: May 27, 2020. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.2647&rep=rep1&type=pdf.

[11] M. Gevrey, I. Dimopoulos, S. Lek, Review and comparison of methods to study the contribution of variables in artificial neural network models, Ecol. Modell., vol. 160, no. 3, 2003, pp. 249-264.

[12] J. Townsend, T. Chaton, J. M. Monteiro, Extracting relational explanations from deep neural networks: a survey from a neural-symbolic perspective, in IEEE Transactions on Neural Networks and Learning Systems, 2019, pp. 1-15, doi: 10.1109/TNNLS.2019.2944672.

[13] S. Bach, A. Binder, G. Montavon, F. Klauschen, K-R. Müller, W. Samek, On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation, PLoS ONE, vol. 10, no. 7, 2015, doi: 10.1371/journal.pone.0130140.

[14] A. Binder, G. Montavon, S. Lapuschkin, K-R. Müller, W. Samek, Layer-wise relevance propagation for neural networks with local renormalization layers, in: Villa A., Masulli P., Pons Rivero A. (eds) Artificial Neural Networks and Machine Learning – ICANN 2016, ICANN 2016, Lecture Notes in Computer Science, vol 9887, pp. 63-71, Springer, Cham, doi: 10.1007/978-3-319-44781-0_8.

[15] L. Zadeh, Fuzzy sets, Inf. Control, vol. 8, no. 3, 1965, pp. 338-353.

[16] D. Wu, J. M. Mendel, J. Joo, Linguistic summarization using if-then rules, International Conference on Fuzzy Systems, 2010, pp. 1-8.

[17] D. Poole, A. Mackworth, Artificial intelligence: foundations of computational agents, 2nd ed., Cambridge: Cambridge University Press, 2017. Accessed on: May 27, 2020. [Online]. Available: https://artint.info/2e/html/ArtInt2e.Ch7.S3.html.

[18] M. Arlitt, T. Jin, A workload characterization study of the 1998 world cup web site, in IEEE Netw 14, 2000, pp. 30-37.

[19] M. Borkowski, S. Schulte, C. Hochreiner, Predicting cloud resource utilization, in 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC), Shanghai, 2016, pp. 37-42.

[20] K. Hirota, W. Pedrycz, Fuzzy computing for data mining, in Proceedings of the IEEE, vol. 87, no. 9, 1999, pp. 1575-1600.

[21] M. Tan, Multi-agent reinforcement learning: independent vs. cooperative agents, Proceedings of the Tenth International Conference on Machine Learning, 1993, pp. 330-337.

[22] S. Shen, V. v. Beek, A. Iosup, Statistical characterization of business-critical workloads hosted in cloud datacenters, 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Shenzhen, 2015, pp. 465-474, doi: 10.1109/CCGrid.2015.60.

[23] Bitbrains IT Services Inc., GWA-T-12 fastStorage trace, The Grid Workloads Archive, 2015. Accessed on: Mar. 21, 2020. Available: http://gwa.ewi.tudelft.nl/datasets/gwa-t-12-bitbrains

[24] C. Reiss, J. Wilkes, J. Hellerstein, Google cluster-usage traces: format + schema, Google, 2013. Accessed on: May 20, 2020. [Online]. Available: https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md.