Env2Vec: Accelerating VNF Testing with Deep Learning

Guangyuan Piao
Nokia Bell Labs
Dublin, Ireland
guangyuan.piao@nokia-bell-labs.com

Patrick K. Nicholson
Nokia Bell Labs
Dublin, Ireland
pat.nicholson@nokia-bell-labs.com

Diego Lugones
Nokia Bell Labs
Dublin, Ireland
diego.lugones@nokia-bell-labs.com

Abstract

The adoption of fast-paced practices for developing virtual network functions (VNFs) allows for continuous software delivery and creates a market advantage for network operators. This adoption, however, is problematic for testing engineers that need to assure, in shorter development cycles, certain quality of highly-configurable product releases running on heterogeneous clouds. Machine learning (ML) can accelerate testing workflows by detecting performance issues in new software builds. However, the overhead of maintaining several models for all combinations of build types, network configurations, and other stack parameters, can quickly become prohibitive and make the application of ML infeasible.

We propose `Env2Vec`, a deep learning architecture that combines contextual features with historical resource usage, and characterizes the various stack parameters that influence the test execution within an *embedding* space, which allows it to generalize model predictions to previously unseen environments. We integrate a *single* ML model in the testing workflow to automatically debug errors and pinpoint performance bottlenecks. Results obtained with real testing data show an accuracy between 86.2%-100%, while reducing the false alarm rate by 20.9%-38.1% when reporting performance issues compared to state-of-the-art approaches.

ACM Reference Format:
Guangyuan Piao, Patrick K. Nicholson, and Diego Lugones. 2020. Env2Vec: Accelerating VNF Testing with Deep Learning. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3342195.3387525

1 Introduction

Network operators are continuously adopting modern *DevOps* practices for virtualizing network functions (e.g., firewalls and load balancers) and accelerating software delivery and integration. These practices motivate short development cycles for creating and deploying new product features and reducing time-to-market. However, the change towards more aggressive product life-cycles is creating significant challenges for network engineers to test new software builds and assure carrier-grade quality. Several tools are available for testing automation, yet debugging software errors and discovering performance bottlenecks still require human expertise and manual interactions that limit the pace at which new features can be deployed to production.

Machine learning (ML) has the potential to accelerate VNF testing by automating the diagnosis of software defects and anomalous builds. This is complementary to recent proposals in the context of network operation [2, 19, 30, 43], where ML is used to enforce high-level policies to keep the network functioning in case of faults [9], or for recommending and automating tasks based on a centralized view of data [24].

Testing engineers are interested in promptly discovering whether new features in the current software build have resulted in any major performance deviation from previous ones. However, current practices require manual investigation of relevant key performance indicators (KPIs). Given the increasing complexity of cloud stacks, this is quickly becoming a daunting task – even for specialized teams dedicated to collaborating with developers in moving new software functionality into production environments, e.g., site reliability engineers (SREs) [11, 16, 40]. As a result, there is an increasing interest in applying ML techniques to characterize a large number of metrics and other monitoring information at scale.

Recent research has proposed ML to model VNF resources [21, 29, 44], such as CPU/Memory usage, or Disk I/O and Network counters, during previous non-problematic builds and used such models to detect defects or accomplish other decision making tasks [30, 43]. Current ML-based solutions, however, neglect the high-dimensional parameter space of clouds. This is a critical limitation that can yield such solutions ineffective upon changes in the hardware/software stack that are not representative of the environment used for data collection and model training. We argue, and demonstrate experimentally, that this parameter space is a core challenge in software testing as it necessitates the accurate characterization of VNF performance in many thousands of possible running environments. Here, the term ***environment*** refers to the choice among the dozens of deployments with diverse software builds, middleware versions, enabled features or

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387525

Table 1. Example of Environment Metadata (EM) organized according to their position in the hardware/software stack. The metadata consists of strings with either a numeric or textual value associated, which are used to construct an embedding space. The domain of values can vary significantly from tens in the lower layers of the stack (e.g., CPU clock 2.1 GHz to 4.0 GHz, 8 to 48 cores, etc.) to hundreds of possibilities at application layer (e.g., build type, configuration parameters, dependencies, etc.) or test-case layers (user/traffic profiles, multivendor environments, devices, etc.) – refer to § 4.2.1 for more details.

	Hardware	Virtualization	Operating System	Application/VNF	Test case
Metadata	CPU clock rate [GHz]	Hypervisor (e.g., ESXi 6.5)	Kernel (e.g., Linux 5.3.7)	Build (e.g. stable, 1.0.1)	Workload type (e.g. data)
	Number of cores [#]	Cluster size [#]	ulimits [list]	Runtime env. (e.g., JVM)	Traffic model (e.g., self-similar)
	RAM [GB]	DPDK [on/off]	FS/disk [ext4]	Features enabled [list]	Form Factor (e.g., surge)
	Disk size [GB]	SR-IOV [on/off]	Swap size [GB]	Service Chain [list]	System Under Test (e.g., DB)
	Hyper-Threading [on/off]	CPU pinning [on/off]	Page size [KB]	Slicing [#]	Test type (e.g., endurance)
	Number of thread [#]	vCPU [#]	CPU gov. (e.g., ondemand)	Elasticity [yes/no]	Fault injection [list]
	...	...	...	...	...

configuration, hundreds of test cases from endurance to regression, each running on a variety of testbeds composed of heterogeneous virtualized hardware.

At a high level, our aim is to incorporate a deep learning (DL) system in the testing workflow; however, we focus our design around three aspects that are critical for its adoption in practice. First, the system must be robust to environment variations across testbeds and test cases. Second, the ML models must be simple to maintain, avoiding any added overhead and complexity in the testing workflow. Third, the system must be trustworthy to extrapolate predictions to previously unseen environments and to minimize false alarms.

In this paper, we introduce `Env2Vec` (short for "*environment to vector*") a software testing system that abstracts out environment details using ***embeddings***; i.e., a fixed-length vector of real numbers that encode the high-dimensionality of cloud parameters leveraging environment metadata and large amounts of historical data collected during software testing activities. The idea is motivated by the recent success of embeddings in other domains such as natural language processing (NLP) [31] where word embeddings capture some of the semantics of the input (e.g., similar words are nearby in the embedding space), and can be reused across machine learning models for related tasks. In the context of VNF testing, we use the environment embeddings to automatically club together test executions with similar environment metadata, as well as to mix-and-match embeddings of different environments to detect performance problems when a test case is executed in a previously unseen environment, for which there is no historical data.

`Env2Vec` systematically models and predicts VNF performance using three inputs: i) VNF KPIs and traffic metrics from the live running test case; ii) historical resource usage from previous software builds and iii) environment metadata labels that describe the environment specific configuration; an example of concrete, but non-exhaustive, labels can be found in Table 1. The system discovers performance anomalies automatically by detecting KPIs in the new build that deviate from the baseline model, and reports the time interval of such deviation without further manual iterations.

The problem can be formalized as a *contextual anomaly detection* [17, 20]. More precisely, the resource characterization of VNFs is a time series prediction problem, in which the goal is to predict the resource usage y_p at a timestep p, given a set of contextual features $\{a_1^p, \cdots, a_f^p\}$, and a sliding window of historical values of both y and a at previous timesteps. We refer to the union of the resource time series and contextual features as *contextual time series*. Each time series belongs to a sequence of builds related to a specific testbed and test case combination, that we refer to as a *build chain*. A build chain may be used to test whether an updated VNF build could potentially cause problems for a particular telecommunication operator, by modelling the configuration and enabled features in some part of their network (via the testbed) and the types of traffic they receive (via the test case). From a workflow perspective, a tester may be responsible for some number of such testbeds and test cases, and must assure that software updates do not cause problems for their assigned setups. Thus, a build chain is tied to a particular environment (the testbed and test case), which is abstracted as a set of environment metadata (EM) values. The testbed can be represented by labels in the first four columns of Table 1, whereas the test case can be abstracted using labels from the last column. For example, the test case traffic model may be a typical daily load curve for the system under test, whereas a workload type may be a particular distribution of packet types and their lengths. For a particular build chain, `Env2Vec` therefore has the task of determining if the current time series indicate a performance anomaly with respect to the previous builds in the chain.

To achieve operational simplicity, our system creates and updates a single generic model that can be applied to VNF testing without overhead and complexity added to the engineers – contrary to creating models for each chain separately as proposed in previous work. As mentioned above, this is feasible by using environment embeddings to reduce the dimensionality of the environment variables.

In summary, our contributions are:

- `Env2Vec`, a ML-based VNF testing system that automates the detection of defects in new software builds across heterogeneous clouds. The core of the system is a novel deep learning architecture that combines gated recurrent units *(GRUs)* [8] to model the behavior of the resources over time, as well as a feedforward neural network *(FNN)* to model the contextual features automatically, and lookup tables with the environment embeddings to account for the high-dimensional parameter space of cloud stacks.
- A comprehensive evaluation with i) public benchmark datasets from three different types of VNFs, as well as, ii) real software testing scenarios with multiple builds across 125 testbeds and test case combinations. Compared to other ML approaches, `Env2Vec` performs $20.9\% - 38.1\%$ better in correctly identifying issues.
- A technique to create environment embeddings to reveal relationships in similar builds which tend to cluster together, and concrete examples that show how embeddings learned from historical data of other environments are used for detecting issues in a new previously unseen environment – this is not possible with previous proposals.

The rest of the paper is organized as follows. In § 2, we motivate our work. § 3 provides details of `Env2Vec` and § 4 presents results. § 5 elaborates on related work, § 6 discusses limitations, and we conclude in § 7. Finally, Appendix A and Appendix B list formal definitions and abbreviations.

2 Motivation

Here we summarize the key limitations of current practices that motivate our design choices for `Env2Vec`, more details and a taxonomy of recent research is provided in Section 5.

Limited assurance. Current approaches are evaluated in terms of model accuracy and only consider how well the model represents the VNF resource usage. However, there is no assessment on how models improve the end-to-end testing workflow in detecting anomalies and software defects. This is a significant gap in the literature, since the real utility of performance modeling is its applicability to automate downstream tasks that can accelerate development cycles.

Neglected resource usage over time. Some techniques make use of contextual features for performance prediction, but only focus on the current state of the system usage and neglect the resource consumption over the recent past. This leads to inaccurate predictions as it does not properly characterize the non-linear behavior of software and makes the model inefficient to capture certain workload variations.

Tight coupling between model and environment. Typically, one model is learned for each environment, meaning that testing a VNF release across multiple testbeds and platforms results in a wide variety of models that depend heavily on the underlying testbed environment. Figure 1 (top) shows this issue. In the heatmap, each column represents a different build chain, and each row represents a contextual feature. For each build chain we trained a linear regression model to illustrate how the environment influences the learning process even when the model input (contextual features) and output (KPIs, CPU usage in this case) are fixed. That is, notice how the associated weights (shades of blue) of each contextual feature varies significantly for each build chain model. Moreover, the accuracy of each model varies widely, as shown in Figure 1 (bottom). Hence, models can be ineffective when the combination of contextual features and environments is sparse as there is insufficient training data available for some builds.

The virtualization or *softwarization* of network functions allows developers to use standard testing practices in high-availability, carrier-grade scenarios. Moreover, to expedite testing and onboarding new features to VNFs, network operators are adopting modern DevOps practices similar to those used by web-scale operators. However, there are operational challenges associated to VNF development that differ from generic software development. For example, the dependencies on accelerators and other hardware features –in some cases purpose-built– and the need for a separation between control and data planes for optimizing resource performance.

The above provides evidence that the environment metadata is crucial in the modelling process. Our hypothesis is that a single generic model improves the testing workflow over individual models if such additional metadata is used as input. This is possible if the metadata induces natural groupings over the build chains to increase the training data available compared to an individual model. Moreover, such a model would still apply to new environments for which there is no previous data, but that fall within these groupings. This is simply not possible using individual build models. In the next sections, we introduce our testing system and detail its components.

3 System Overview

Figure 2 shows the `Env2Vec` architecture and illustrates the testing workflow. In the following text, each step corresponds to a numbered arrow in the figure.

(1) Testbed data collection: The testing engineer schedules a test case execution of a VNF and spawns virtual machines (VM) for workload generation or trace replication. Both VNF and workload configurations are possible using the service API specific to the VNF under test, whereas the underlying resource configuration (e.g., VM flavor, user-data, allocation, etc.) is performed via the cloud API. Both APIs are, in turn, used to monitor the workload metrics (WMs), the VNF performance metrics (PMs), and the resource utilization (RU) metrics. Environment metadata (EM) is also collected, as shown previously in Table 1. The WM, PM, and RU metrics are linked to EM and pulled into a real-time time-series database (TSDB), in our case, Prometheus [35]. When a new test case is executed, we modify a service discovery configuration

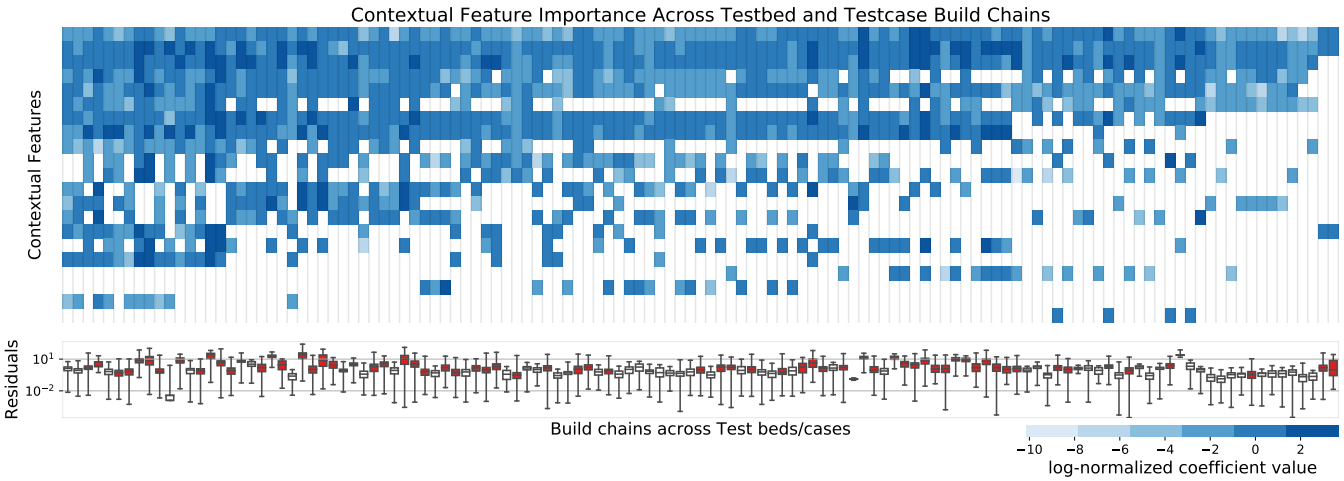


Figure 1. Top: A heatmap showing the weight, or importance, assigned to each contextual feature across a large number (125) of build chains, i.e., different testbed and test case executions. In the y-axis, each rectangular cell represents a different input metric, and in the x-axis each rectangular cell represents a different build chain. The color shade depicts the importance (i.e., symmetrically log-normalized coefficient) assigned to that metric and build chain for *predicting CPU utilization* based on a linear regression model. White cells have zero weight, which means that either the metric was unavailable on that testbed, or that it was not deemed important by the model. Bottom: for each column, we provide a boxplot capturing the absolute value of the residuals for the each model. Box plots highlighted in red indicate the occurrence of some residuals above 10%. In other words, the model prediction of the CPU was off by at least 10% on at least one sample of the test data, indicating poor performance.

JSON file for Prometheus, appending the endpoint for the metric collector along with a reference to the EM labels:

```
[..., {"targets": ["IP:PORT"],
"labels": {"env":"EM_record_id"}}]
```

(2) Model training: The model is updated daily using all the new data generated where no performance problem was flagged. Executions with true positive alarms are masked out from the training data, along with any false negative problems discovered independently by the testing engineers. We note that this is a best effort approach, and obviously some problems will be missed. Provided these are not sustained performance problems (e.g., covering a long duration and many samples) and overall only represent an extremely small fraction of the training data, we have found the model to still be highly accurate. After training completion, the model is available via HTTP.

(3) Prediction pipeline: Data of the running testbed is read from the TSDB into the `Env2Vec` ML pipeline. The data consists of the CFs and RU: recall that the union of WMs and PMs are referred to as contextual features, CFs. The prediction pipeline monitors the running VNF via Prometheus over HTTP, and constructs a dataframe from this monitoring data, appending the relevant EM. An example of such a dataframe can be found in Table 2. The running model receives the dataframe and compares the inferred RU to the observed value. If there is a significant deviation, then an anomaly is flagged. As discussed later in Section 3.2, the precise definition of what constitutes a significant deviation is user-configurable, and tuned based on the goals of the user.

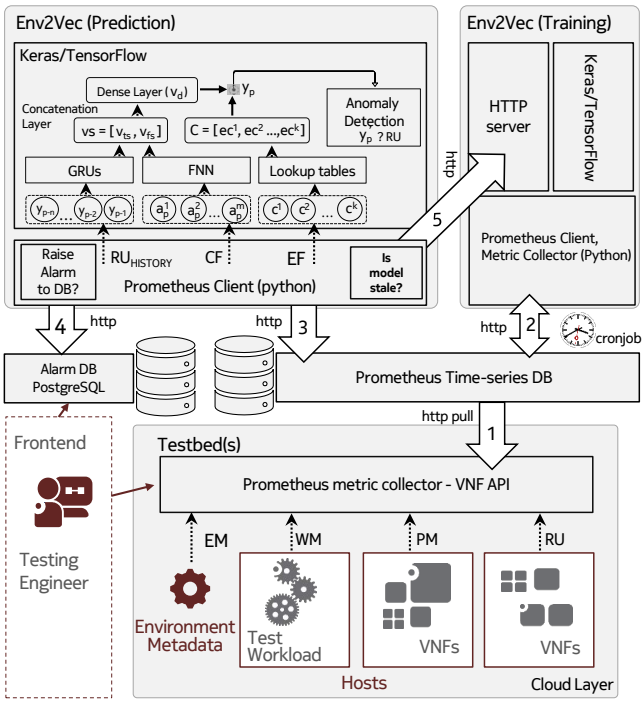


Figure 2. VNF testing workflow and system components .

Table 2. Dataframe example created from the TSDB. In the performance metrics (PMs) rows, success ratios indicate ratios of delivered-to-sent packets across various network interfaces, and response codes are counters tallying the number of egress server error response codes. For example, a 503 response code indicates a service is unavailable.

		Dataframe Features	Data Type
CFs	WMs	Client UE	<int>
		Burst period	<float>
		Demand (Mbps)	<float>
		...	...
	PMs	Success_Ratio$_{<mod_id>}$	<float>
		Response_Code$_{ERR:50X}$	<int>
		Packet_cnt$_{<mod_id>}$	<int>
		...	...
EM		Build Version	<char>
		System Under Test	<char>
		Test Case	<char>
		...	...
RU Hist		cpu$_{t-1,t-2,\ldots,t-k}$	<list>
		mem$_{t-1,t-2,\ldots,t-k}$	<list>
		...	...
RU		cpu$_{usage}$	<float>
		net$_{TX<IF_{ID}>}$	<int>
		...	...

(4) Raising alarms: Upon detecting anomalies, `Env2Vec` pushes an alarm into a PostgreSQL database [34]. This alarm contains all the relevant information to allow a testing engineer who triggered the test case execution to pinpoint on which testbed the issue occurred, and during which time interval. Such alarms can also trigger automated actions, such as early termination of the test case execution.

(5) Updating the model: The `Env2Vec` prediction pipeline, fetches the latest model (essentially a weight matrix), before beginning execution, from the training pipeline HTTP server.

High-level overview of ML model The deep learning pipeline is implemented with Keras [23] and TensorFlow [41] (see Figure 2). This pipeline has the dual function of i) training a resource utilization model to infer VNF resource usage (y'_p) with respect to a set of contextual features and environment information; as well as ii) execute the model in real time to detect deviations between the inferred value and the actual observed resource utilization of the VNF (i.e., contextual anomaly detection: y'_p ? RU). In the next section we provide a detailed description of the deep learning architecture.

3.1 `Env2Vec` – Main Components

In this section, we describe each component of `Env2Vec`. To simplify the discussion, in the rest of the paper we use a tuple of four representative EM labels such as testbed, SUT, test case and build information to represent an environment, e.g., $< Testbed_{ID}, SUT_{Mod}, Testcase_{ID}, Build_{vers} >$ where $Testbed_{ID}$ refers to a testbed with a specified EM configuration (as in Table 1), SUT_{Mod} refers to the system under test or software module, $Testcase_{ID}$ indicates the test case which includes information such as workloads, duration, etc. Finally, $Build_{vers}$ denotes a combination of build type and version.

FNNs for capturing contextual features. Feedforward neural networks can model complex relations between CFs and RU over time, as shown by [29, 30], and reduce manual effort required for feature engineering: i.e., manual preprocessing to create new features by combining and transforming the CFs. FNNs learn a numeric weight matrix that captures important relations between the CFs that impact RU, and the result of multiplying this weight matrix with the input CFs (and applying an activation function) is a numeric vector $\mathbf{V_{fs}}$ (see Figure 2) that should reflect these relations.

GRUs for incorporating resource history. To improve the model predictions, we incorporate a window of n recent measurements of resource usage $\{y_{p-n}, \cdots, y_{p-1}\}$. As such data is sequential, we use GRUs [8] which are a type of Recurrent Neural Network (RNN), successfully adopted in other domains such as recommender systems [46] and time-series forecasting [6]. The input to the GRUs is a sequence of historical resource utilization values ($\mathtt{RU_{history}}$ in Figure 2), and the output is a vector $\mathbf{V_{ts}}$ that summarizes the expected impact of the observed historical RU.

Embeddings for environments. There are two possible extremes when training characterization models. One is that we assume build chains are completely different from each other, and we train a separate model for each, as in previous research. The other extreme is to treat all environments exactly the same and use all data from those environments to train a single characterization model. As one might expect, this assumption yields very inefficient models, if not useless, because of the high-dimensional space of environments, as we can notice from Figure 1 in Section 1. However, some environments will be similar to each other, especially those with certain overlap of EM labels, e.g., $< Testbed_{15}, SUT_{DB}, Testcase_{Regression}, Build_{s10} >$ and $< Testbed_{15}, SUT_{DB}, Testcase_{Endurance}, Build_{s11} >$. Intuitively, combining the data of similar environments can improve the modelling process in two aspects: 1) it provides models robust to environment changes, and 2) it improves the accuracy with more data from similar environments.

We provide an optimal solution between those two extremes using embeddings of the environment, which are numerical vectors that reduce the dimensional space and facilitate the detection of similarities between environments when implemented with lookup tables. For each environmental feature, there is a lookup table associated where each row corresponds to an embedding with respect to the feature value (e.g., $Testbed_{143}$). The input of lookup tables are the environment feature values $\{\mathbf{c^1}, \cdots, \mathbf{c^k}\}$, and the output are their

embeddings $\{\mathbf{ec^1}, \cdots, \mathbf{ec^k}\}$ as shown in Figure 2. Similar to handling unknown words in NLP, the lookup table also contains an additional *unknown vector/embedding* to deal with an unknown environment that has not appeared in the training data before.

The embeddings (values in each vector) are initialized before the training process with a dimension of 10. Afterwards, the embeddings are learned during the training process by minimizing the Mean Squared Error ($MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - y_i')^2$) loss function. In Section 4, we show how similar environments cluster together in the embedding space, and how these can be reused across testing tasks.

3.2 Putting All Together

We have described the different deep learning components of our approach and their output vectors with model coefficients learned automatically for dedicated types of input information. These vectors are:

- $\mathbf{v_{fs}}$ for contextual features,
- $\mathbf{v_{ts}}$ for previous resource utilization values, and
- a set of embeddings $\{\mathbf{ec^1}, \cdots, \mathbf{ec^k}\}$ for environmental features.

Next, we explain how to combine these vectors to infer VNF resource usage and detect anomalies at testing time.

Concatenation and dense layers. In this layer, the output from GRU and FNN layers are concatenated into a single vector $\mathbf{v_s} = [\mathbf{v_{ts}}, \mathbf{v_{fs}}]$. In addition, the embeddings with respect to different environments are concatenated into a single environment embedding C in this layer.

$$\mathbf{C} = [\mathbf{ec^1}, \mathbf{ec^2}, \cdots, \mathbf{ec^k}] \tag{1}$$

Furthermore, $\mathbf{v_s}$ is fed into a FNN in a dense layer to output $\mathbf{v_d}$, which has the same dimensionality of the concatenated environment embedding C. This is a requirement as $\mathbf{v_d}$ will eventually be multiplied by C, and it also enables discovery of further relations between $\mathbf{v_{ts}}$ and $\mathbf{v_{fs}}$.

Resource usage prediction. `Env2Vec` uses the dense feature representation $\mathbf{v_d}$ and the environment embeddings C to infer the resource utilization y_p' at timestep p, as follows:

$$y_p' = \sum \mathbf{v_d} \odot \mathbf{C} \tag{2}$$

As can be seen in Equation 2, we use the sum of the element-wise (Hadamard) product. This is inspired by previous work in recommender systems [46] where the element-wise product has proven effective to capture the preference score of an item with respect to a user [27, 36]. However, there are other choices for modeling the dense feature representation $\mathbf{v_d}$ and the environment embedding C. For example, the prediction can be done with an additional matrix R, i.e., $y_p' = \mathbf{v_d} \cdot \mathbf{R} \cdot \mathbf{C}$; or, it can be done by using additional neural network layers with the concatenated vector of $\mathbf{v_d}$ and C as an input. Both approaches require more parameters to learn but yield similar results.

Anomaly detection. The VNF resource model is used along with a common statistical approach [20] to automate anomaly detection. First, we construct a Gaussian distribution modelling the prediction error of normal (i.e., non-problematic) builds using the time series associated with previous builds in a particular build chain. This distribution has mean μ_{error} and standard deviation σ_{error}. Then, for the time series associated with the next build in the chain, we flag an anomaly at a given timestep when the prediction (y_p', eq. 2), provided by the model, deviates from the mean of the error distribution beyond a threshold $\gamma \times \sigma_{error}$. Here, γ is a positive real number that defines the normal area in the error distribution, and allows controlling the tradeoff between the precision and recall for detecting anomalies. A higher value of γ indicates a stricter criteria, which will result in a higher precision but lower recall. In other words, an error at a certain timestep which is larger than $\mu_{error} + 2 \times \sigma_{error}$ will be treated as an anomaly when $\gamma = 2$. γ can be set and adjusted by testing engineers empirically based on the testing data, and the subjective precision-recall tradeoff for detecting certain faults. Finally, we remark that this approach assumes that the prediction errors will follow a Gaussian distribution, and while this may be adequate in many cases, it is not necessarily always true. Thus, a more rigorous modelling of the prediction error for a particular VNF may be required in such cases.

4 `Env2Vec` Evaluation

We divide our evaluation methodology in three parts. In Section 4.1, we conduct a baseline comparison with state of the art approaches using open data from the networking community. The idea is to show the effectiveness of our proposal in modeling VNF performance in terms of model accuracy and simplicity. Then, in Section 4.2, we bring this model to the testing workflow and demonstrate its use to automate the anomaly detection in new software versions or system upgrades. In this case, we use proprietary testing data obtained from hundreds of different environments with diverse test cases related to *telecom* software running on VNFs. Last, we show, in Section 4.3, the performance of `Env2Vec` in discovering anomalies in previously unseen environments – without model retraining – by leveraging environment embeddings.

4.1 VNF Modelling using Benchmark Datasets

The goal here is to demonstrate (1) the comparative performance of the *single* model learned by `Env2Vec` when compared to *multiple* models learned for each environment running different VNFs in the benchmark datasets, and (2) whether using environment embeddings for training improves the characterization performance.

Table 3. KDN datasets split.

# of examples	Snort	Switch	Firewall
Total	1,359	1,191	755
Training	900	900	555
Validation	259	141	100
Test	200	150	100

4.1.1 Data Description

The KDN initiative [26] publishes a set of benchmark datasets to facilitate the comparison and evaluation of machine learning approaches for modelling the resource usage y_p of several VNFs provided a set of contextual features. The data has been widely used in previous studies [21, 29, 30] to evaluate and compare different ML-based networking solutions. The datasets contain the CPU utilization of three different VNFs operating with real traffic: 1) intrusion detection with *Snort* [38] configured with the default configuration [29][1]; 2) an SDN-enabled firewall, and; 3) an SDN-enabled switch. These VNFs were deployed in VMs (Ubuntu 14.04.1) running on VMware ESXi v5.5. The traffic is a replica obtained from a deep packet inspection (DPI) infrastructure and injected via tcpreplay. This traffic is represented by 86 features, in 20 second batches, which include data such as the number of packets, the number of different IPs/ports, and the number of 5-tuple flows. More details of the datasets can be found in the relevant related work [4, 26, 29, 30].

Our experiments follow the standard split of data into training, validation, and test sets. The training set is used for learning the coefficients at each layer of the proposed model, and for training the methods we compare to, which we will discuss later in this section. The validation set is used for tuning hyper-parameters in all methods, and for applying early stopping and optimizing training times. The test set is used for comparison purposes. Table 3 details the number of samples for training, validation, and testing for each VNF dataset.

4.1.2 Evaluation Metrics

We use Mean Absolute Error ($MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - y'_i|$) and Mean Squared Error (MSE), defined in Section 3.1, as target evaluation metrics for comparing the prediction performance achieved by different methods. N denotes the size of each test set, while y_i is the actual monitored resource utilization and y'_i is the value predicted by the machine learning model. A higher value of MAE or MSE indicates a lower accuracy. We compare all methods based on their performance on the test sets. Neural networks can find different local optimum, which results in different MAEs and MSEs for each run. Hence, we run up to 10 times the neural network models, e.g., `FNN`, `RFNN`, and `Env2Vec`, for consistency and report the average of these 10 runs when comparing to other methods. Finally, we use the *paired t-test* with a significance of 0.05 to draw meaningful conclusions when comparing means.

[1]We note that the performance of Snort is highly affected by its numerous configuration and ruleset specifications, and thus some important subset of the configuration options and rules, as identified by a domain expert, could be incorporated as environment metadata labels. However, we did not explore this direction in the present work as the Snort dataset only had the default configuration.

4.1.3 Methods to Compare

We compare `Env2Vec` to several state-of-the-art ML techniques and baseline approaches. These are:

`Ridge`: This is a baseline method that exploits the traffic features with a Ridge regression model, implemented with scikit-learn [37]. We search the parameter space of the regularization hyper-parameter α from $\{0.001, 0.1, \cdots, 1000\}$ using the validation set of each VNF dataset.

$\texttt{Ridge}_{\texttt{ts}}$: This approach exploits both the set of traffic features at timestep p and the resource utilization value of n previous timesteps, and uses Ridge regression for predicting the current resource utilization. Therefore, the set of features used in `Ridge(ts)` are the same than for `Env2Vec` but the complexity is different. Our intention with this comparison is to examine whether a complex model performs better than a linear model with the same set of features.

`RFReg`: This is a non-linear method that exploits the traffic features with a Random Forest Regressor, implemented with scikit-learn [37]. `RFReg` is an ensemble method which consists of a set of estimators (decision trees) for regression. We search the parameter space of the two important hyper-parameters *max_depth*: $\{3, 4, \cdots, 10\}$ and *n_estimators* (number of estimators): $\{10, 50, 100, 1000\}$.

`SVR` [21]: This approach uses support regression vectors as a prediction model, it is also implemented with scikit-learn. Here, the hyper-parameters to tune are: 1) regularization α: $\{0.001, \cdots, 1000\}$; 2) kernel function: $\{\texttt{linear}, \texttt{poly}, \texttt{rbf}\}$, which refers to a method of using linear models to solve a non-linear problem by transforming input data; 3) a margin of tolerance ϵ: $\{0.1, 0.2, \cdots, 1\}$ which sets a maximum allowed error margin.

`FNN` [29, 30]: This approach uses a feedforward neural network using one hidden layer with a set of traffic feature values as input. We use Keras [23] to implement this prediction model, and tuned two hyper-parameters; one is the number of neurons in the hidden layer, which is drawn from the following set of powers-of-two $\{32, 64, 128, 256, 512, 1024\}$, and the other is the dropout rate which is drawn from 0 to 0.9 in steps of 0.1. The number of neurons is set to 1,024 for all the three datasets, which provides the best performance on their corresponding validation sets. Dropout rates are set to 0.0, 0.6, and 0.1 for the Snort, Firewall and Switch datasets, respectively.

`RFNN`: This is a variant of `Env2Vec` using recurrent neural networks (GRUs) and FNNs but without using the embeddings of environments, and trained for *each environment separately*, e.g., we create three `RFNN` models for the KDN

Table 4. MSE and MAE results on the three different VNF datasets using different prediction approaches with the best-performing scores in bold. The mean and standard deviation of CPU utilization in each dataset are 196 ± 23 (Snort), 384 ± 46 (Firewall), and 448 ± 46 (Switch), respectively. [Shortened]

	Snort		*Firewall*		*Switch*	
	MAE	MSE	MAE	MSE	MAE	MSE
`Ridge`	5.72	49.83	13.68	661.37	11.34	211.60
$\texttt{Ridge}_{\texttt{ts}}$	5.35	44.52	12.03	339.21	**10.74**	**184.14**
`RFReg`	5.38	55.20	12.32	315.80	15.41	493.64
`SVR`	5.88	63.87	11.63	761.97	11.69	241.61
`FNN`	5.29 ± 0.08	45.18 ± 1.78	11.47 ± 0.20	398.20 ± 19.33	12.54 ± 0.08	333.45 ± 9.77
`RFNN`	4.81 ± 0.11	38.51 ± 3.32	10.47 ± 0.35	387.48 ± 34.78	11.15 ± 0.20	272.59 ± 8.06
$\texttt{RFNN}_{\texttt{all}}$	5.52 ± 0.24	57.78 ± 5.20	10.98 ± 0.51	415.89 ± 73.45	12.03 ± 0.34	312.72 ± 13.81
`Env2Vec`	**4.61 ± 0.12**	**36.08 ± 1.68**	**10.33 ± 0.49**	**288.85 ± 46.62**	10.90 ± 0.19	267.80 ± 18.38

datasets. Here, we reuse the best performing hyper-parameters of the FNN method. In addition, we tune the number of previous timesteps n to achieve the best performance. To this end, we draw n from 1 to 9 in steps of 1. The best performance on the validation set is achieved with $n = 1$ for Snort and SDN-switch and $n = 2$ for the Firewall data. We used TensorFlow [41] and Keras to implement `RFNN`.

$\texttt{RFNN}_{\texttt{all}}$: is a variant of `Env2Vec` without using the embeddings of environments, as above, but training a single model with all data from *all environments*. This allows to understand whether including environment embeddings actually improve performance. The prediction of $\texttt{RFNN}_{\texttt{all}}$ is made by the dense layer ($\mathbf{V_d}$) with regression.

`Env2Vec`: Contrary to other methods, which result in m models, where m is the number of environments, `Env2Vec` uses a single generic model to capture all environments. For the KDN benchmarks, $m = 3$ for different VNFs. However, for testing VNFs in production, the number is significantly higher (e.g., $m = 125$ for the data shown in Figure 1).

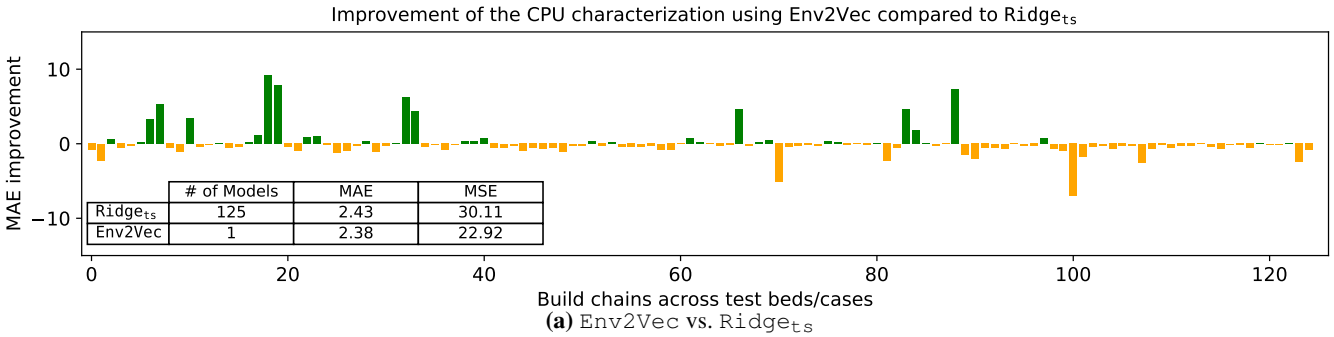
4.1.4 Results

Table 4 shows the MAE and MSE results of CPU usage prediction using aforementioned methods on each test set with respect to the three VNFs.

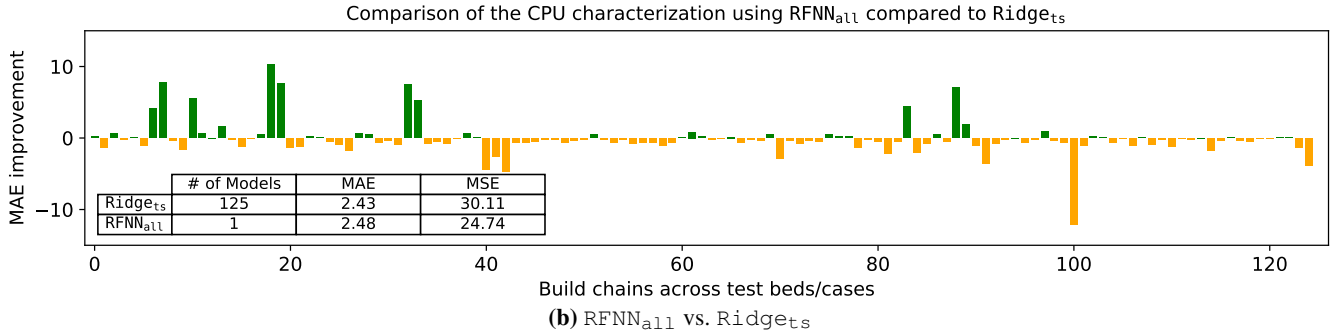
Single model vs. Multiple models. First, we discuss whether training a single model using `Env2Vec` outperforms training separate models for each dataset. On the *Snort* dataset, we observe that `Env2Vec` outperforms other baseline methods significantly followed by `RFNN`, `FNN` and $\texttt{Ridge}_{\texttt{ts}}$. $\texttt{Ridge}_{\texttt{ts}}$, which incorporates n previous timesteps to the Ridge regression model also improves the prediction performance significantly in terms of both MAE and MSE. Different from the results in [21], `SVR` does not perform well on Snort or the other datasets. This might be because the evaluation metric used in [21] is different from MAE and MSE, and does not reflect the prediction performance as well. Similar to the results on the *Snort* dataset, `Env2Vec` provides the best performance in terms of MAE and MSE on the *Firewall* dataset followed by `RFNN`. On the *Switch* dataset, we observe different performance trends than in other datasets. The best performing method is $\texttt{Ridge}_{\texttt{ts}}$, which is a linear model with n previous CPU utilization values and the set of features at the current timestep. Consistent with the results on *Snort* and *Firewall* datasets, $\texttt{Ridge}_{\texttt{ts}}$ outperforms `Ridge` significantly. Overall, our proposed approach provides a competitive characterization performance compared to all techniques across all datasets despite using a single characterization model.

Importance of environment embeddings. Secondly, we discuss the impact of using environment embeddings by comparing the performance achieved by `Env2Vec` and $\texttt{RFNN}_{\texttt{all}}$. As we can see from the table, the MAE and MSE results for $\texttt{RFNN}_{\texttt{all}}$, which does not use embeddings, are worse than using `Env2Vec` across all VNF datasets. Without embeddings, $\texttt{RFNN}_{\texttt{all}}$ essentially treats all data, from the three different VNFs, identically for training the characterization model, which leads to a poorer performance due to the differences across VNF datasets. This is another extreme compared to methods such as `RFNN` which assumes each VNF dataset has no relationship or similarity and should be trained separately. The improvement of `Env2Vec` over $\texttt{RFNN}_{\texttt{all}}$ demonstrates that capturing the similarities between environments with embeddings is crucial for training a single characterization model with all data from those environments.

The implications of these results are twofold: (1) Despite training and using a single model, `Env2Vec` can achieve either the best or competitive performance in predicting resource usage of different VNFs when compared to training models for each VNF separately. The results are promising since in reality there is a large number of VNFs to model, so maintaining a model per VNF/environment combination can be a daunting task; (2) Incorporating environment embeddings, is critical to generalize the model when combining training data from multiple environments.



(a) Env2Vec vs. $\texttt{Ridge}_{ts}$



(b) $\texttt{RFNN}_{all}$ vs. $\texttt{Ridge}_{ts}$

Figure 3. (a) The improvement of CPU characterization in terms of MAE for the set of build chains in Figure 1 using Env2Vec compared to using $\texttt{Ridge}_{ts}$. Despite using a single model, Env2Vec provides competitive characterization performance and better MAE and MSE results as shown in the table at the bottom left of the current figure. (b) The comparison of using a variant of Env2Vec without using environment embeddings ($\texttt{RFNN}_{all}$) and using $\texttt{Ridge}_{ts}$. The results indicate the importance of using environment embeddings in order to achieve the best characterization performance with a single model.

4.2 Env2Vec in the Software Testing Workflow

We now incorporate Env2Vec in the testing workflow end-to-end to detect performance defects in software builds across hundreds of different environments. This approach can be used for detecting performance problems across many types of resources such as CPU, memory and disk, or other VNF specific KPIs such as response time, network jitter, etc. Here, we characterize VNF usage of specialized network cards with interest of detecting abnormal CPU usage of the network card for different test executions of upgraded builds.

4.2.1 Carrier-grade VNF Testing

We use data of a virtualized telecom software product, which incorporates several test scenarios for different software upgrades. The testing activities include over 600 environments. The dataset contains 125 build chains for multiple combinations of testbed, build type, SUT, and test case, and consists of about 400,000 timesteps/data points measured at 15 minute intervals. In particular, there are nearly one hundred testbeds, several types of SUT, and hundreds of test cases and builds. The number of test executions continuously increases with number of testbeds, test cases, and new builds. Each time series describes a set of contextual features and the CPU usage of the network function during a certain testing period for each new build. The contextual features are workload and performance metrics defined and collected by testing engineers including standard event counter data associated with different network elements and functions. For the purposes of training and testing our ML models, we treat the time series associated with the current (or most recent) build in each build chain as the test case, and those associated with the previous builds as the training/cross-validation data. As one might expect, manual investigation of the results for all test cases of each new build is not scalable, and requires deep domain knowledge. Also, training one model for the set of time series of each environment separately can result in hundreds of models to be retrained and maintained.

Next, we first show the CPU characterization results given a set of traffic features in order to investigate whether the results are similar to those for the KDN datasets. Second, we use the characterization model for contextual anomaly detection in Section 4.2.2, where the goal is to detect whether a build upgrade has caused some issue during the test.

Env2Vec – Model Accuracy. In this case, we evaluate over the same set of build chains from which we obtained Figure 1 (in Section 1). Figure 3a shows the improvement of using Env2Vec for the entire set of chains compared to using separated $\texttt{Ridge}_{ts}$ on each build chain. Notice, again, that the

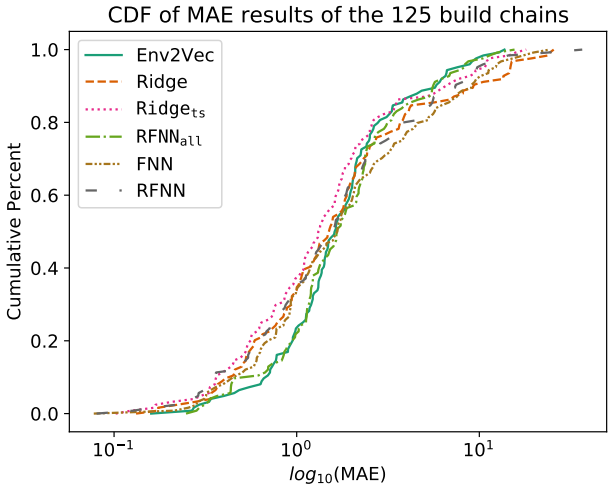


Figure 4. Generalized results. MAE CDF over all 125 build chains and all methods. Logarithmic scale is used to clearly show MAE differences between approaches.

single Env2Vec VNF model provides competitive accuracy compared to 125 different models for each environment. The table in the bottom left of Figure 3a shows the detailed results of the average of MAE and MSE values across all 125 build chains. We also observe that without embeddings, RFNN$_{all}$ performs worse than Env2Vec in terms of both MAE and MSE and also has higher MAE compared to Ridge$_{ts}$ in Figure 3b. This confirms our hypothesis that embeddings are necessary for training a single generic model with all environment data. These results are consistent with the ones obtained in Section 4.1. The highest MAE improvements of using Env2Vec and RFNN$_{all}$ are obtained due to the limit of Ridge$_{ts}$ as a linear model. This is consistent with the findings in Mestres *et al.* [29, 30] which show that neural networks are needed to model complex resource usage.

We generalize the results above in Figure 4, which shows the MAE cumulative distribution function (CDF) across the 125 build chains for all techniques under evaluation. Noting the log-scale on the x-axis, we make the following observations: 1) Env2Vec has slightly worse MAE scores than other methods when the MAE for the test case is small (e.g., the MAE of the CPU prediction is $\leq$ 1%), however; 2) Env2Vec has significantly better MAE scores than other methods when the MAE scores are high (e.g., $>$ 5%). For the most difficult 10% of the cases, that have the highest MAE scores, Env2Vec has the best performance over all methods. This indicates that Env2Vec is not overfitting to small CPU fluctuations, and is also more robust in difficult cases.

4.2.2 Automating Anomaly Detection in New Builds

Here, we compare Env2Vec to a state-of-the-art anomaly detection approach: HTM-AD [1], to evaluate the benefits of the VNF characterization model end to end. The objective is to automate the detection of CPU performance problems. We also include Ridge, Ridge$_{ts}$ in the comparison.

HTM-AD is an unsupervised anomaly detection method that does not consider any contextual features. Rather, it only uses the target resource consumption (in this case CPU) as input to determine whether an anomaly has occurred. Our proposed approach, as well as the other methods discussed in Section 4.1.3, use additional time series metrics as contextual features. Hence, we include HTM-AD to quantify the benefits of incorporating contextual features for detecting performance problems, compared to a naïve approach that only considers a single time series.

When automating anomaly detection, it is critical to avoid raising many false alarms that would incur testing engineers unproductive efforts and likely cause them to abandon the system. To this end, we further filter predicted anomalies to only include those where the difference, in CPU utilization, between the predicted and observed values not only exceeds γ standard deviations, but also has absolute value exceeding 5%. This additional filtering is a common practice to reduce false alarms in the literature [7].

Evaluation metrics. Manually checking each timestep for anomalies across all testbeds and build types is infeasible. Instead, we requested testing engineers provide labels only for the set of alarms flagged by at least one of the different implemented approaches. To measure quality of the alarms flagging performance problems, we use true and false alarm rate for evaluation: $A_T = \frac{N_{tp}}{N_{tp}+N_{fp}}$ and $A_F = 1 - A_T$, respectively. N_{tp} and N_{fp} denote the counts of the true positive and false positive alarms, respectively. So, an ideal detector should flag performance problems with $A_T = 100$ and $A_F = 0$.

Detected performance problems. Table 5 shows the results of performance problem detection for all new build tests on a certain date across 11 different test executions. During the test executions, a variety of different problematic inputs and scenarios (e.g., increased latency on certain interfaces) are simulated in the network, often overlapping in time and affecting different components of the system under test. Crucially, the vast majority of these simulated problems do not lead to any noticeable impact on the collected metrics. As discussed, we pooled all alarms raised by the union of all approaches, and among these alarms, 35 were confirmed to be true positive cases, where a significant impact on the metrics was observed. Due to the data collection process, we do not have a measure of the false negative cases (i.e., an alarm should have been raised but was not).

HTM-AD provides anomaly scores ranging from 0 to 1, and we only considered when the anomaly score is equal to 1 to generate alarms. Despite using the highest anomaly score to detect performance problems, HTM-AD provides poor results

Table 5. Performance problem detected for different γ values. In total, there were actually **35** performance problems based on the feedback from testing engineers.

	# of alarms	correct alarms	A_T	A_F	Note
HTM-AD	42	16	0.381	0.619	
Ridge	32	20	0.625	0.375	$\gamma = 1$
Ridge(ts)	26	17	0.653	0.347	
$RFNN_{all}$	18	16	0.889	0.111	
Env2Vec	29	25	0.862	0.138	
Ridge	25	16	0.64	0.36	$\gamma = 2$
Ridge(ts)	24	16	0.667	0.333	
$RFNN_{all}$	15	14	0.933	0.067	
Env2Vec	18	18	1.0	0.0	
Ridge	20	11	0.55	0.45	$\gamma = 3$
Ridge(ts)	21	13	0.619	0.381	
$RFNN_{all}$	8	8	1.0	0.0	
Env2Vec	13	13	1.0	0.0	

compared to other approaches that leverage contextual features, which shows the importance of considering contextual features account for in this context.

Table 5 shows the results when using different values of γ which can be empirically set by testing engineers. The accuracy (A_T) increases with higher values of γ while the number of detected problems decreases. Testing engineers can set γ based on their priorities. For example, if the priority is detecting more performance problems automatically with reasonable accuracy, Env2Vec with $\gamma = 1$ can detect the highest number of problems (25) with A_T of 0.862. When the priority is raising highly accurate alarms and no false positives, Env2Vec also provides the best performance with more alarms (18) and A_T of 1.0 when $\gamma = 2$. Results indicate that Env2Vec outperforms other VNF models trained separately on each environment for different $\gamma's$.

4.3 Testing Unseen Environments, Embeddings Reused

Previously proposed approaches train one model for each environment, which requires the samples from each environment to be available prior to training. These approaches cannot be used for testing in unseen environments as data must be collected *a-priori* for training. Here, if there is a new previously unseen environment, e.g., $e =< Testbed_{13}, SUT_F, Testcase_{Endurance}, Build_{S01} >$, several tests need to be run to gather enough data for training a model. This happens despite that there can be a lot of unexplored data of some similar environments in the historical data, such as $e =< Testbed_{13}, SUT_F, Testcase_{Load}, Build_{S03} >$ and $e =< Testbed_{08}, SUT_A, Testcase_{Endurance}, Build_{S01} >$.

In contrast, Env2Vec can learn the embeddings of environments, given the time series data in a similar environment,

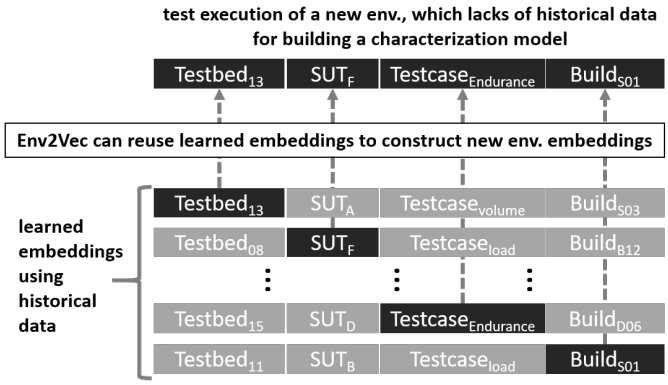


Figure 5. Example of reusing environment embeddings from historical data to construct environment embedding for a new unseen environment. This allows Env2Vec to be applicable for detecting VNF performance anomaly without delay, while other approaches still need to collect new training data.

or a portion of it. These embeddings can be reused for constructing the environment embeddings of a new unseen environment as illustrated in Figure 5. The idea of reusing learned environment embeddings is motivated by its use in natural language processing where the main purposes of learning word embeddings is to discover semantics or relationships, and then use them to infer meaning, especially when there is not enough data for supervised learning.

Next, we discuss how embeddings learned with Env2Vec have semantic meanings or relationships, and show how to reuse them for modelling RU in unseen environments by combining EM subsets.

Learned environment embeddings. Figure 6 shows the concatenated embeddings of environments for each test execution in the telecom dataset, where the dimensionality has been reduced to 2-dimensional space using principal component analysis [22] for simplifying visualization. These environment embeddings are clustered based on their similarities. We notice that each cluster with different colors in the figure denotes different build types (e.g., S, B, D, etc.). That is, environment embeddings belong to the same build type are close to each other in their embedding space. In each cluster, more overlap of the environments with respect to the testbeds, SUT, and test cases, indicates a higher degree of similarity. More spread out clusters, such as the grey one, indicate environments running the same build type but have some differences with respect to their testbeds, SUT, or test cases. This shows that the embeddings learned by Env2Vec capture the similarities between environments, and therefore, it is reasonable to construct the embeddings for a new unseen environment when there is a lack of data for such environment.

Performance problem detection. To evaluate performance problem detection in unseen environments, we reuse the 11 test executions from the telecom testing datasets discussed

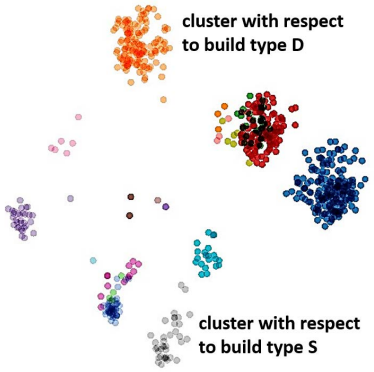


Figure 6. Embeddings of environments concatenated for each test case, visualized in a 2-dimensional space. Different colors in the figure denote different types of builds (e.g., build type D (debug), T (test), S (stable) etc.), and we can notice that environment embeddings associated to similar build types (e.g., $Build_{D01}$, $Build_{D02}$, $\cdots$) are close to each other.

in Section 4.2.2 but blind out their available history of time series data to treat those as unseen environments. We use the rest of the data which does not contain any historical time series associated with each target test execution for training `Env2Vec`, and use it for detecting performance problems. As there is no previous prediction error distribution associated to a test execution in an unseen environment, we apply the user-defined γ to the prediction error distribution computed for all timesteps in the test execution. To justify the need for a model that can adapt to a new environment, note that there are around 800 timesteps for those 11 test executions, which require a significantly time-consuming manual investigation.

Table 6 shows the results of performance problem detection for the unseen environments and varying γ values. Characterization models such as `Ridge` and $\texttt{Ridge}_{\texttt{ts}}$ are not applicable (N/A) here since there is no historical data associated to each unseen environment for training. `HTM-AD` does not perform well since it does not take those contextual features into account and only uses the resource-usage time series for detecting anomalies. Overall, `Env2Vec` outperforms $\texttt{RFNN}_{\texttt{all}}$ with different values of γ. For example, `Env2Vec` raised 12 correct alarms out of 19 raised alarms with $A_T = 0.632$ while $\texttt{RFNN}_{\texttt{all}}$ also raised 12 correct alarms out of 26 alarms $A_T = 0.462$. The improvement compared to using $\texttt{RFNN}_{\texttt{all}}$ again shows the importance of leveraging environment embeddings in the context of detecting performance problems. These results show that `Env2Vec` can detect performance problems in unseen test executions, which is useful during the period of gathering enough data and constructing prediction error distributions while other alternative approaches do not perform well. Despite a good precision of detected alarms (e.g., with $\gamma = 3$), the number of detected performance problems is smaller than the case with historical data in Section 4.2.2. This problem is resolved by retraining `Env2Vec` incrementally with the new data from the environment.

Table 6. Results of performance problem detection for unseen environments based on different numbers of standard deviation (γ) for detecting anomalies. `Ridge` and $\texttt{Ridge}_{\texttt{ts}}$ are not applicable (N/A) due to the lack of data in unseen environments.

	# of alarms	correct alarms	A_T	A_F	Note
`HTM-AD`	42	16	0.381	0.619	
`Ridge`	N/A	N/A	N/A	N/A	
`Ridge(ts)`	N/A	N/A	N/A	N/A	
$\texttt{RFNN}_{\texttt{all}}$	99	18	0.182	0.818	$\gamma = 1$
`Env2Vec`	35	14	0.4	0.6	
$\texttt{RFNN}_{\texttt{all}}$	26	12	0.462	0.538	$\gamma = 2$
`Env2Vec`	19	12	0.632	0.368	
$\texttt{RFNN}_{\texttt{all}}$	13	8	0.615	0.385	$\gamma = 3$
`Env2Vec`	9	7	0.778	0.222	

5 Related Work

In Jmila et al. [21], SVR was proposed for RU prediction for the same KDN benchmark datasets used in this paper. Other work on the same benchmark datasets suggest using FNNs, but without revealing a comparative performance with different models [29, 30]. Importantly, these approaches do not incorporate historic resource usage and embeddings that can reduce the high-dimensionality of environments for the multiple possible testing use cases.

In the context of datacenters [10, 45] propose supervised models for detecting system compliance with target service level objective. Similarly, `Mercury` [28] can detect performance issues caused by software upgrades after they are deployed in operation. A signature-based approach has been proposed [7] for detecting different types of previously-seen performance anomalies of servers. In contrast, our system can detect contextual anomalies of upgrades in the testing phase, i.e., *before* deployment, and thereby it uses a non-continuous set of time series for each test execution with different builds to achieve a more representative model. Another difference is that our learning approach is unsupervised in terms of the anomalies in the dataset, while previous work [10, 13] treat the problem as a supervised learning with SLA (Service Level Agreement) compliance labels. Having an unsupervised approach is important because anomalies in build upgrades are difficult or at least time-consuming to label. `Env2Vec` also complements approaches that detect functional anomalies through static analysis [12, 33], yet our focus is on detecting performance anomalies without access to the actual code.

Finally, we consider representation learning [5] techniques that leverage embeddings and have been recently studied in the context of networking. For example, `Net2Vec` [14, 15]

aims to learn the embeddings of network traffic to enable user profiling and personalization. Similarly, COBANETS [47] applied embeddings to video traffic features to improve traffic performance and quality of experience, QoE. We consider these approaches complementary to Env2Vec.

6 Discussion

In this section, we discuss current limitations requiring further research, and elaborate on our plans to address them in the near future.

First, training a single model using all the data from all environments requires more training time versus training a separate model (such as Ridge) for each environment. For example, Ridge and Ridge_{ts} take less than 1 second to train per build chain, on commodity hardware. Therefore, such regression models can be trained on the fly and then used immediately for detecting anomalies. Compared to those methods, Env2Vec and RFNN_{all} require about 30 minutes training time on commodity hardware. Therefore, they must be periodically trained and stored. Overall, our Env2Vec model requires less than 10MB storage space, for a file containing the environment embeddings and the DL model.

Another limitation of Env2Vec is that limited coverage of some EM in the training dataset can result in reduced performance as the embeddings cannot be properly learned for those EM. For example, we noticed that the Env2Vec results of A_T in Table 5 when $\gamma = 1$ is caused by one under-performing case out of the 11 cases, with the other 10 having $A_T = 1$. A detailed investigation shows that the under-performing case has much lower coverage with only 17 examples in the training data for the corresponding testbed compared to other 10 cases as shown in Table 7. This shows that the coverage of an environment within the training data is an important issue that affects the performance of Env2Vec, and can result in unsatisfactory performance for test executions from underrepresented environments. In addition, it is worth noting that the unseen environments in Section 4.3 refer to those can be constructed by known environment embeddings (i.e., embeddings covered in the training data). For instance, it is challenging to apply Env2Vec in a new environment with a new testbed which has not been appeared in the training data before since we cannot construct the corresponding environment embeddings. This also leads us to suggest test case executions by testing engineers to be as balanced as possible, especially in terms of the underlying testbeds.

Regarding anomalies, we evaluated Env2Vec based on the labels from testing engineers for the set of alarms flagged by at least one of the implemented approaches. However, it is also important to understand whether there are certain types of anomalies that are out of the scope of Env2Vec, or whether Env2Vec is able to flag unknown anomalies (i.e., which have not appeared before). Such an evaluation would require a complete set of manual annotations of all anomalies (not just those flagged by an algorithm) for test executions from different environments.

Table 7. The under-performing test execution vs. the rest in the 11 test executions when $\gamma = 1$ with the information of the number of examples covering the testbed of a target execution and the coverage (in %) of the testbed over all training data.

	Under-performing case	The remaining cases
A_T	0.5	1.0
# of examples	17	$12,313 \pm 5,097$
Coverage (%)	0.004	3.15 ± 0.014

Despite the limitations discussed above, these promising results for software testing with DL open many interesting future research directions. One example would be incorporating the attention mechanism [3, 42], which allows a DL model to focus on the certain relevant parts of the input. This could be useful to learn relationships between metric values from previous timesteps. In addition, a deeper analysis of the contributions of different groups of CFs or different EM could help to reduce the complexity of Env2Vec. For example, starting with the complete Env2Vec model and using a "hold out" strategy to remove a set of CFs or EM to investigate how the performance changes [18].

Finally, we mention that more automated ways of mapping VNF configuration options to environment metadata would be an interesting direction for future research. For example, as discussed in Section 4.1.1, a VNF like Snort has many possible configurations and ruleset options that could be incorporated as environment metadata. However, the current process would require a Snort expert to manually select which such options are important to performance. Simply taking all possible configuration options would likely lead to an embedding space that is too large for practical purposes, and therefore likely run into issues of data sparsity.

7 Conclusions

In this paper, we have introduced a deep learning architecture in the software testing workflow of virtual network functions, VNFs. The key advantage of this approach is to automate the detection of defects and bugs in new software builds by identifying performance degradation and informing the engineer. A central contribution relates to using environment embeddings to abstract the stack deployments from the machine learning models, which allows to 1) cope with the high-dimensionality of cloud parameters, 2) create a unique easy-to-maintain universal model and 3) extrapolate such model to previously unseen environments. Env2Vec, uses sophisticated neural networks to characterize the resource usage of the VNFs, and contextual anomaly detection to pinpoint performance issues in new builds. We have shown, in the evaluation, that a single model combined with environments embeddings can achieve state-of-the-art accuracy in modeling resources and

detecting performance issues in software updates – even when compared to multiple models, each specifically crafted for the running environment. This result is key as it shows that `Env2Vec` is scalable and feasible to adopt by testing engineers. By treating the different combinations of testbeds and software builds as embeddings, which are learned at training time with a centralized view of data, `Env2Vec` can also be used extrapolate predictions to unseen deployments proactively as shown in Section 4.3. In the future, we will investigate more sophisticated prediction methods and extend the embeddings with more testing types and VNF KPIs.

A Appendix – `Env2Vec` Formal Definitions

FNNs for capturing contextual features. The FNN part of `Env2Vec` consists of hidden layers formulated as follows: $\mathbf{q_t} = \sigma(\mathbf{W^{(q)}a_t} + b_q)$, where σ denotes the *sigmoid* function: $s(x) = \frac{1}{1+e^{-x}}$, $\mathbf{W^{(q)}}$ is a weight matrix for the input $\mathbf{a_t}$, and b_q denotes a bias term. In this paper, the evaluation is performed with the FNN component having one hidden layer. The input vector $\mathbf{a_t} = [a_1^t, \cdots, a_m^t]$ consists of the CF values.

GRUs for incorporating resource history. A GRU consists of two inputs: $\mathbf{h_{t-1}}$, denoting the information (output) from the previous timestep t-1, and $\mathbf{y_t}$, denoting the value at the current timestep. The output of a GRU is the information to be passed to the next timestep. GRUs have several vectors so called *gates* for controlling the information to be passed through the network. The first one is the ***update gate*** $\mathbf{z_t}$ for timestep t, which can be defined as follows: $\mathbf{z_t} = \sigma(\mathbf{W^{(z)}y_t} + \mathbf{U^{(z)}h_{t-1}})$, where σ denotes the *sigmoid* function, $\mathbf{W^{(z)}}$ and $\mathbf{U^{(z)}}$ are weight matrices for the input $\mathbf{y_t}$ and the output from the previous timestep $\mathbf{h_{t-1}}$. The *update gate* determines how much information from previous timesteps should be considered.

The second gate is called ***reset gate***. In contrast to the *update gate*, the *reset gate* determines how much information from the past should be forgotten with the following formula: $\mathbf{r_t} = \sigma(\mathbf{W^{(r)}y_t} + \mathbf{U^{(r)}h_{t-1}})$, where $\mathbf{W^{(r)}}$ and $\mathbf{U^{(r)}}$ are weight matrices of the reset gate. $\mathbf{r_t}$ is then used to calculate the current memory content $\mathbf{h'_t}$ with the following formula: $\mathbf{h'_t} = f(\mathbf{W^{(h)}y_t} + \mathbf{r_t} \odot \mathbf{U^{(h)}} \cdot \mathbf{h_{t-1}})$, where $f(\cdot)$ is an activation function (we empirically adopt the *ReLu* [32] activation function based on the model performance on training and validation datasets), $\mathbf{W^{(h)}}$ and $\mathbf{U^{(h)}}$ are weight matrices, and $\odot$ denotes the element-wise (Hadamard) product of two vectors. As $\mathbf{r_t}$ ranges from 0 to 1, $\mathbf{r_t} \odot \mathbf{U^{(h)}} \cdot \mathbf{h_{t-1}}$ denotes how much information from the previous timesteps should be removed. Finally, the current memory content $\mathbf{h'_t}$, the previous hidden state $\mathbf{h_{t-1}}$ and the update gate are used to determine the current hidden state $\mathbf{h_t}$: $\mathbf{h_t} = (1 - \mathbf{z_t}) \odot \mathbf{h'_t} + \mathbf{z_t} \odot \mathbf{h_{t-1}}$.

A.1 Model Training

Here, we describe the training details of `Env2Vec` including the loss function, regularization details for preventing overfitting, and how the model is maintained.

Loss function. In order to learn the parameters of proposed model such as $\mathbf{W^{(z)}}$ and $\mathbf{W^{(r)}}$, we need to have sample data and a predefined loss function. The objective of training is to learn those parameters automatically in order to minimize the loss on the sample data. [29, 30]. We use Mean Squared Error ($MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - y'_i)^2$) as our loss function to train our proposed model. where N denotes the number of total training examples, y_i denotes a resource utilization value, and y'_i denotes the predicted value by our model associated with y_i. To learn the parameters of our proposed approach for minimizing the loss, we use the Adam update rule [25] to train the model on the training set.

Regularization. Overfitting is a crucial problem in training neural networks, which denotes a learned neural network model fits a training set well but is failing to fit unseen data (e.g., examples in a separate test set) [39] and does not generalize well. To prevent overfitting, we adopt the widely used regularization techniques of dropout [39], where randomly selected neurons are ignored during training, and an early stopping strategy, which stops the training if there is no improvement on a validation set.

B Appendix – Abbreviations

This section lists the various abbreviations and notation used throughout the paper, refer to for more details Table 8.

Concepts	CPU	Central Processing Unit
	DB or TSDB	Database or Time-series database
	HTTP	Hyper-text transfer protocol
	JSON	JavaScript object notation
	KDN	Knowledge defined networking
	KPI	Key performance indicators
	NLP	Natural language processing
	VNF	Virtual network function
Metrics	CF	Contextual Features
	EM	Environment Metadata
	PM	Performance Metrics
	RU	Resource Utilization
	WM	Workload Metrics
Methods	DL	Deep Learning
	FNN	Feed-forward Neural Network
	GRU	Gated Recurrent Neural Networks
	ML	Machine Learning
	RNN	Recurrent Neural Networks
	SVR	Support Vector Regression
Evaluation	MAE	Mean absolute error
	MSE	Mean squared error
	CDF	Cumulative Distribution Function
	A_T	True Positive Alarms
	A_F	False Positive Alarms
	γ	Tunable confidence threshold

Table 8. Abbreviations, acronyms and notation.

References

[1] S. Ahmad, A. Lavin, S. Purdy, and Z. Agha. Unsupervised real-time anomaly detection for streaming data. *Neurocomputing*, 262:134–147, 2017.

[2] S. Ayoubi, N. Limam, M. A. Salahuddin, N. Shahriar, R. Boutaba, F. Estrada-Solano, and O. M. Caicedo. Machine Learning for Cognitive Network Management. *IEEE Communications Magazine*, 56(1):158–165, 2018.

[3] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[4] P. Barlet-Ros, G. Iannaccone, J. Sanjuàs-Cuxart, and J. Solé-Pareta. Predictive resource management of multiple monitoring applications. *IEEE/ACM Transactions on Networking (TON)*, 19(3):788–801, 2011.

[5] Y. Bengio, A. Courville, and P. Vincent. Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013.

[6] F. M. Bianchi, E. Maiorino, M. C. Kampffmeyer, A. Rizzi, and R. Jenssen. An overview and comparative analysis of recurrent neural networks for short term load forecasting. *arXiv preprint arXiv:1705.04378*, 2017.

[7] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: automated classification of performance crises. In *Proceedings of the 5th European conference on Computer systems*, pages 111–124. ACM, 2010.

[8] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[9] D. D. Clark, C. Partridge, J. C. Ramming, and J. T. Wroclawski. A knowledge plane for the internet. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 3–10. ACM, 2003.

[10] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *OSDI*, volume 4, page 16, 2004.

[11] N. M. Comcast. From engineering operations to site reliability engineering. San Francisco, CA, 2017. USENIX Association.

[12] T. Dai, J. He, X. Gu, S. Lu, and P. Wang. Dscope: Detecting real-world data corruption hang bugs in cloud server systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 313–325. ACM, 2018.

[13] J. Gao, G. Jiang, H. Chen, and J. Han. Modeling probabilistic measurement correlations for problem determination in large-scale distributed systems. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 623–630. IEEE, 2009.

[14] R. Gonzalez, A. Garcia-Duran, F. Manco, M. Niepert, and P. Vallina. Network Data Monetization Using Net2Vec. In *Proceedings of the SIGCOMM Posters and Demos*, SIGCOMM Posters and Demos '17, pages 37–39, New York, NY, USA, 2017. ACM.

[15] R. Gonzalez, F. Manco, A. Garcia-Duran, J. Mendes, F. Huici, S. Niccolini, and M. Niepert. Net2Vec: Deep Learning for the Network. In *Proceedings of the Workshop on Big Data Analytics and Machine Learning for Data Communication Networks*, Big-DAMA '17, pages 13–18, New York, NY, USA, 2017. ACM.

[16] D. R. Google. Building successful SRE in large enterprises—one year later. Santa Clara, CA, 2018. USENIX Association.

[17] M. A. Hayes and M. A. M. Capretz. Contextual anomaly detection framework for big sensor data. *Journal of Big Data*, 2(1):2, 2015.

[18] L. Hong, A. S. Doumith, and B. D. Davison. Co-factorization machines: modeling user interests and predicting individual decisions in twitter. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 557–566, 2013.

[19] J. Hyun, N. V. Tu, and J. W. Hong. Towards knowledge-defined networking using in-band network telemetry. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–7, 2018.

[20] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth. Performance anomaly detection and bottleneck identification. *ACM Computing Surveys (CSUR)*, 48(1):4, 2015.

[21] H. Jmila, M. I. Khedher, and M. A. El Yacoubi. Estimating VNF Resource Requirements Using Machine Learning Techniques. In *International Conference on Neural Information Processing*, pages 883–892. Springer, 2017.

[22] I. Jolliffe. Principal component analysis. In *International encyclopedia of statistical science*, pages 1094–1096. Springer, 2011.

[23] Keras. The python deep learning library. "https://keras.io/", 2019.

[24] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, 2015.

[25] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[26] Knowledge-Defined-Networking. Training datasets. "http://knowledgedefinednetworking.org/", 2019.

[27] Y. Koren, R. Bell, and C. Volinsky. Matrix Factorization Techniques for Recommender Systems. *Computer*, 42(8):30–37, aug 2009.

[28] A. A. Mahimkar, H. H. Song, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and J. Emmons. Detecting the Performance Impact of Upgrades in Large Operational Networks. *SIGCOMM Comput. Commun. Rev.*, 40(4):303–314, aug 2010.

[29] A. Mestres, E. Alarcón, and A. Cabellos. A machine learning-based approach for virtual network function modeling. In *2018 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, pages 237–242, 2018.

[30] A. Mestres, A. Rodriguez-Natal, J. Carner, P. Barlet-Ros, E. Alarcón, M. Solé, V. Muntés-Mulero, D. Meyer, S. Barkai, M. J. Hibbett, G. Estrada, K. Ma'ruf, F. Coras, V. Ermagan, H. Latapie, C. Cassar, J. Evans, F. Maino, J. Walrand, and A. Cabellos. Knowledge-Defined Networking. *SIGCOMM Comput. Commun. Rev.*, 47(3):2–10, sep 2017.

[31] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[32] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[33] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 902–912. IEEE, 2015.

[34] Postgresql. The worldś most advanced open source relational database. "https://www.postgresql.org/", 2019.

[35] Prometheus. From metrics to insight. "https://prometheus.io/", 2019.

[36] S. Rendle. Factorization Machines with libFM. *ACM Trans. Intell. Syst. Technol.*, 3(3):57:1—-57:22, may 2012.

[37] Scikit-Learn. Machine learning in python. "http://scikit-learn.org", 2019.

[38] Snort. Snort. "https://www.snort.org/", 2019.

[39] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[40] A. T. Tenable. Breaking in a new job as an SRE. Santa Clara, CA, 2018. USENIX Association.

[41] Tensorflow. An end-to-end open source machine learning platform. "https://www.tensorflow.org/", 2019.

[42] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[43] M. Wang, Y. Cui, X. Wang, S. Xiao, and J. Jiang. Machine Learning for Networking: Workflow, Advances and Opportunities. *IEEE Network*, 32(2):92–99, 2018.

[44] C. M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In L. C. Briand and A. L. Wolf, editors, *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, pages 171–187. IEEE Computer Society, 2007.

[45] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 644–653, 2005.

[46] S. Zhang, L. Yao, and A. Sun. Deep Learning based Recommender System: A Survey and New Perspectives. *CoRR*, abs/1707.0, 2017.

[47] M. Zorzi, A. Zanella, A. Testolin, M. D. F. De Grazia, and M. Zorzi. Cognition-based networks: A new perspective on network optimization using learning and distributed intelligence. *IEEE Access*, 3:1512–1530, 2015.