The Importance of Generalizability in Machine Learning for Systems

Varun Gohil, Sundar Dev, Gaurang Upasani, David Lo, Parthasarathy Ranganathan, and Christina Delimitrou

***Abstract*—Using machine learning (ML) to tackle computer systems tasks is gaining popularity. One of the shortcomings of such ML-based approaches is the inability of models to generalize to out-of-distribution data i.e., data whose distribution is different than the training dataset. We showcase that this issue exists in cloud environments by analyzing various ML models ~~used~~ to improve resource balance in Google's fleet. We discuss the trade-offs associated with different techniques used to detect out-of-distribution data. Finally, we propose and demonstrate the efficacy of using Bayesian models to detect the model's confidence in its output when used to improve cloud server resource balance.**

I. Introduction

In recent years, there has been growing interest in using machine learning (ML) to tackle computer systems challenges. This is motivated by the increasing complexity of modern systems and the effectiveness of ML in vision and natural language tasks with similar complexity. Prior work has demonstrated the effectiveness of using ML for scheduling, resource management, power management, debugging, memory allocation, and compiler optimizations [1], [2], [3], [4].

Despite the popularity of using ML in systems, there exist three major concerns that hinder its widespread adoption. The first is the interpretability or explainability of ML models. Many models act as black boxes, making it hard to extract useful insights from them and debug them when they do not work as intended. The second concern is scalability. As ML models become larger, their execution must scale to their respective system setting. The final concern is the generalizability of ML models. Generalizability refers to the model's ability to perform well on data that differs from its training data in its statistical properties, such as independence and identical distribution (ID). This is an issue in environments that change dynamically. One specific example is a cloud environment, where the system changes frequently due to high workload churn and increasingly heterogeneous hardware.

Interpretability and scalability have received significant attention from the systems community, with prior work evaluating their proposals for these metrics [1], [2], [5]. However, the generalizability of models has not received similar attention. ML models rely on the assumption that their training and testing data are independent and identically distributed (IID). When this assumption is violated, models perform poorly. Given that models deployed in a cloud environment often encounter data that violate the IID assumption, it is critical to consider their generalizability. Fortunately, this is well-studied by the ML community and is known as out-of-distribution (OOD) generalization [6].

Varun Gohil and Christina Delimitrou are affiliated with Massachusetts Institute of Technology (email: {varuncg, delimitrou}@csail.mit.edu)

Sundar Dev, Gaurang Upasani, David Lo and Parthasarathy Ranganathan are with Google. (email:{sundarjdev, gupasani, davidlo, parthas}@google.com)

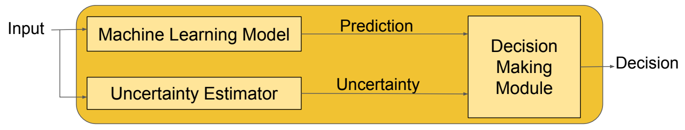

Fig. 1. Logical block diagram of an ideal uncertainty-aware system.

The most effective way to improve a model's performance on OOD data is to retrain it on the OOD data [7]. However, retraining is expensive and time-consuming, especially for larger models. Even though retraining helps generalizabilty, it can only be done retroactively, i.e., after observing that the model's prediction was inaccurate. The system relying on the model still uses the inaccurate prediction, which can be harmful, particularly in the systems domain, where errors can be costly. For example, a scheduler using an ML model with OOD data as input can hurt its Quality of Service (QoS).

Ideally, the system should ignore the model's prediction on OOD data and fall back to a heuristics-based approach. To achieve this one needs to include an uncertainty estimation that detects when the model cannot generalize. We refer to such a system as an *"uncertainty-aware"* system. Figure 1 shows a logical block diagram of such an ideal system.

We make the following contributions:

- We show that generalizability is critical when using ML in cloud systems by studying a representative example where ML has been extensively applied, namely resource provisioning in cloud environments.
- We also demonstrate the efficacy of using Bayesian models for OOD detection, which provides a quantitative confidence score for the model's output.

II. The Need for Generalizability

We now describe the setting we use as a running example to motivate the need for generalizable ML for systems.

A. Problem Description and Relevance

Over the past few years, the heterogeneity in datacenters has increased dramatically, making performance optimizations more challenging. One such challenge is *balanced resource provisioning*. Balance is achieved when the amount of provisioned resources matches the amount of resources required to execute a workload. This ensures no resources are stranded at the target utilization. Resources include the server's memory and network bandwidth, cache, and storage and compute capacity. Balance is important to optimize performance per cost. If a resource is under-provisioned, it hurts performance. On the flip side, if a resource is over-provisioned it will be underutilized, which hurts cost. Below we show an analysis of system balance from Google's fleet, which shows that one of the platforms is imbalanced in terms of CPU and memory bandwidth. We collect CPU and memory bandwidth utilization data over a 5 min interval for over 10 days on each machine in the fleet and use the median value per machine.

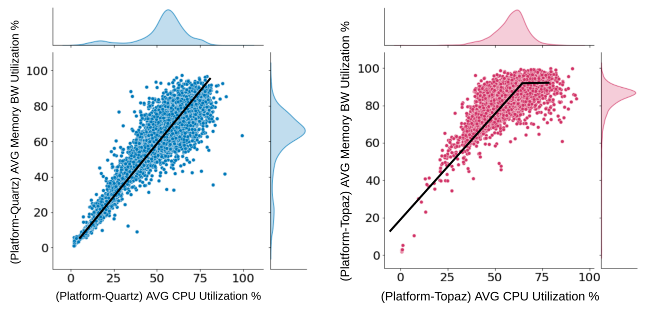

Fig. 2. Fleet-wide memory bandwidth utilization versus CPU utilization. Each datapoint represents the median CPU and bandwidth utilization per node. Distributions along the axes show the histograms of CPU (horizontal) and memory bandwidth utilization (vertical).

Figure 2 shows the fleet-wide memory bandwidth utilization vs CPU utilization for two platforms. For the platform code-named Quartz (left), memory bandwidth utilization scales linearly with CPU utilization. However, for the platform code-named Topaz we see a roofline plot. Here, memory bandwidth utilization plateaus after reaching 60% CPU utilization. Figure 2 illustrates the imbalance in platform Topaz due to under-provisioned memory bandwidth.

Such imbalanced resource provisioning affects multiple policies deployed in Google's datacenters, including job scheduling, and Quality-of-Service-based (QoS-based) eviction. Incorporating such imbalances across all resource dimensions in various system level policies is difficult due to the increasing server heterogeneity and the high job variation frequency. Owing to the problem's complexity, ML becomes an attractive candidate. Specifically, we develop an ML model that predicts the memory bandwidth utilization of a given workload mix on a platform to assess its resource imbalance.

This model can help platform designers decide the memory bandwidth needed for a new platform to be balanced. Second, it can be used during scheduling to ensure that the memory bandwidth utilization on a given machine remains within an acceptable range after new workloads are scheduled on it.

Our model takes the platform configurations and workload mix characteristics as inputs, and predicts the 90^{th} percentile memory bandwidth utilization. The platform configurations include various resources, such as CPU and memory capacity, network bandwidth, cache capacities, etc. Workload mix characteristics include percentile vectors of resource utilization for the mix like CPU, memory, and network utilization along with scheduler hints for cache and NUMA affinity.

We focus on workload mixes rather than individual jobs because platforms employ colocation to improve utilization. Further, since Google has a diverse set of applications, it is not possible to manually select a few workload mixes that are representative of the fleet-wide behavior. Hence, we abstract away the individual workloads within a mix by representing mixes by their resource utilization and scheduling scores.

B. Methodology

We focus on four platform types deployed in Google's fleet, namely, Quartz, Jade, Topaz and Opal. The platform names are anonymized to preserve confidentiality. This list includes platforms developed by different vendors, as well as multiple platforms developed by the same vendor. These four platforms execute the majority of cycles in Google's fleet. We collect data across five representative clusters, each consisting of tens of thousands of servers. We obtain the platform configurations from the platform specification documentation. We collect resource metrics for each workload for a 24 hour period using Google-Wide Profiling (GWP) [8]. Our final dataset consists of 1.2 million data samples, each of which is a unique combination of a workload mix and platform configuration.

We train six different models including random forests, decision trees, histogram gradient boosters, neural networks, a bagging regressor, and a linear regressor. Table I shows the description of each model and its hyperparameters.

We use 75% of the dataset for training and the remaining 25% for testing. To decide on the models' hyperparameters, we use 20% of the training data for validation. We use mean square error loss for training all models.

C. Effectiveness and Challenges of Using ML

We use Mean Absolute Percentage Error (MAPE) to evaluate each model's inference performance, in Table I.

The Random forest model performs the best, with 6.6% MAPE on the test data. The second and third best-performing models are the histogram gradient booster and the bagging regressor, both with 6.9% and 7.0% MAPE. The results indicate that these models are effective at predicting the memory bandwidth utilization of a workload mix on a given platform.

We further test these models on data collected from an unseen platform. Here, an unseen platform refers to a platform whose data was not used to train the model. Here, we have trained the models on data collected on platforms Quartz, Jade, Topaz and Opal. We test these models on data collected from unseen platform Amber. We follow the methodology described in Section II-B to collect data from the Amber platform.

The last column in Table I shows the results on the unseen platform. All top-performing models like RF, gradient boosters, and NNs have significantly higher MAPE on the unseen platform. The RF still performs the best with a MAPE of 56%. Gradient booster and Decision Tree are the next best performing with MAPE of 57% and 57.2% respectively.

The high MAPE values demonstrate the models' inability to make high-accuracy predictions on unseen platforms. If they were used in production, they would lead to memory bandwidth contention and QoS violations. Alternatively, if such a model is used to provision a new platform's memory bandwidth, it would lead to either excess cost or poor performance. This inability to generalize prevents deploying them in production, where they would encounter unseen data.

III. Poor performance on unseen platforms

Since unseen platforms have different resources than seen platforms, the model would have to extrapolate from seen

TABLE I
ML MODEL PERFORMANCE WHEN TESTED ON ID AND OOD DATA.

Model	Hyperparameters	MAPE on data from Quartz, Jade, Topaz, Opal (seen or ID)	MAPE on data from Amber (unseen or OOD)
Random Forest	Max depth = 40, Features used for splitting = 50%	6.6%	56%
Hist. Grad. Boosters	Base Estimator = decision tree, Max depth = 15, L2 regul. = 0.7	6.9%	57%
Bagging Regressor	Base Estimator = decision tree, Features by each estimator = 95%	6.9%	60.9%
Neural Network	2 hidden layers with 200 neurons each using ReLU activation	7.0%	94%
Decision Tree	Max depth =14	7.4%	57.2%
Linear Regression	Standard features, removing the mean and scaling to unit variance	8.5%	73.2%

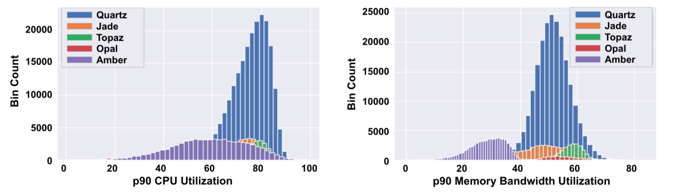

Fig. 3. Histogram of 90^{th} percentile CPU utilization and 90^{th} percentile memory bandwidth utilization on different platforms.

TABLE II
RESULTS OF BAYESIAN NEURAL NETWORK MODEL

Test on data from	MAPE	Uncertainty
Bayesian NN trained on platforms Quartz, Jade, Topaz, and Opal		
Quartz, Jade, Topaz, Opal	8.7%	1.2
Amber	47.7%	15.6
Bayesian NN trained on Quartz, and Topaz		
Quartz, Topaz	8.0%	1.0
Amber	56.0%	15.0
Jade	12.8%	1.9
Opal	11.8%	3.4

datapoints, leading to high inaccuracy. On the other hand, the distribution of workload mix characteristics across platforms can also differ depending on the policies used by the cluster scheduler. Figure 3 shows the distributions of the 90^{th} percentile memory bandwidth and 90^{th} percentile CPU utilization on different platforms. Distributions on platforms Quartz, Jade, Topaz and Opal are similar, however, the distribution for platform Amber is significantly different. The distributions differ because of different hardware characteristics of the platforms and differences in workload mixes.

The models perform worse on unseen platforms because data collected from seen and unseen platforms are not identically distributed. Since data from platform Amber violate the IID assumption, they can be classified as out-of-distribution (OOD), emphasizing the need for generalizable ML models.

IV. USING BAYESIAN MODELS FOR OOD DETECTION

Our observations suggest that a model can be used on data that is identically distributed but not on out-of-distribution data. Hence, one can fall back to a non-learning approach for OOD data and only use the model for the remaining data inputs. Unfortunately, detecting OOD data is not trivial, as it requires access to ground truth to check the prediction accuracy, which is not available during deployment.

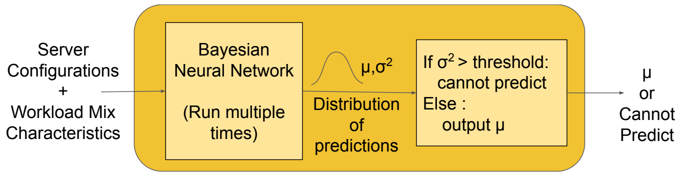
Server Configurations + Workload Mix Characteristics
Bayesian Neural Network (Run multiple times)
μ,σ²
Distribution of predictions
If σ² > threshold: cannot predict Else : output μ
μ or Cannot Predict

Fig. 4. Logical block diagram showing how we use Bayesian neural network.

We propose using uncertainty estimates provided by Bayesian models to detect OOD data samples. For our running example, we use Bayesian neural networks that learn weight distributions instead of specific weight values. During inference, the network samples from this weight distribution to produce a prediction. Inference involves probabilistic sampling and hence produces a different output for the same input for each inference. By running the Bayesian neural network x times, one obtains x different predictions. These x predictions together define a distribution of predictions. The larger the standard deviation of this distribution, the more uncertain the network is. Hence, we define the uncertainty of predictions as the standard deviation of the prediction distribution. Figure 4 shows that the Bayesian model acts as both the ML model performing the prediction and the uncertainty estimator.

We train a 2-layered Bayesian NN on data collected from platforms Quartz, Jade, Topaz, and Opal. Once trained, we test the network on data collected from unseen platform Amber. While testing, we run the model 100 times to obtain a distribution of predictions. We use the standard deviation of this distribution as uncertainty. Further, to obtain accuracy, we compare the test label with the mean of the prediction distribution. Table II shows the MAPE values obtained when testing the Bayesian model. When tested on seen platforms, the model has a MAPE of 8.7% with uncertainty of 1.2. When tested on an unseen platform (Amber), the Bayesian model has a MAPE of 47% and an uncertainty of 15.6. The Bayesian model is similar to previously mentioned models in that it does not perform well on OOD data. However, it differs in that it also indicates when it has a high uncertainty.

Next, we train a Bayesian neural network only on data collected on platforms Quartz and Topaz. Table II shows the results when the Bayesian network is tested on data collected from platforms Jade, Opal, and Amber. As expected, the model performs well on test data collected from seen platforms Quartz and Topaz. It does not perform well on data collected from platform Amber, which is OOD, where the model does give a high uncertainty of 15. Next, we test the model on data coming from platforms Jade and Opal. Both Jade and Opal are unseen platforms, since their data is not used to train the Bayesian neural network. However, the distribution of data collected on Jade and Opal is similar to that of data collected on Quartz and Topaz. Our Bayesian neural network has 12.8% MAPE and 11.8% MAPE on platforms Jade and Opal respectively. It also provides low uncertainty estimates of 1.9 and 3.4 for them. This indicates that for unseen servers, our Bayesian neural network provides predictions with low uncertainty estimates, as long as the data has a similar distribution to the data coming from a seen server.

Using Bayesian NNs as an uncertainty estimator helps us realize a system which knows when to not trust the model's prediction. When using such a Bayesian model in a scheduling system, the model will provide a prediction with high uncertainty on Amber. The scheduler decides a threshold of tolerable uncertainty above which it ignores the model's prediction and uses other heuristics to make scheduling decisions. If the model was used for memory bandwidth provisioning on a new platform, the server design can check the model's uncertainty and refrain from using its prediction to guide the design process when uncertainty is high.

The workflow in Fig. 4 requires a domain expert to tune the uncertainty threshold. Since uncertainty equals the standard deviation of predictions, it has the same units as the prediction target. In our case, an uncertainty of 1 means that the model's memory bandwidth prediction is off by $\pm$1GBps. A domain expert needs to analyze the criticality of the task and determine an acceptable uncertainty level. Additionally, one also needs to decide how many times to run the Bayesian model. The approach we used is to measure the uncertainty after each run and stop when the change in uncertainty is below 5%.

On an A100 GPU, the Bayesian model takes 600ms for inference with batch size of 512. This overhead limits its use for tasks like microsecond-scale scheduling. However, it is useful for resource partitioning that runs for a few seconds, as well as for resource provisioning during server design.

V. Discussion of trade-offs

While we use a Bayesian NN for uncertainty estimation, other uncertainty estimators can also be used. The choice of estimator offers trade-offs between the ability of the model to detect when it cannot generalize and the model's scalability.

Our Bayesian neural network uses the standard deviation of the prediction of distributions as uncertainty. However, generating this distribution requires running the Bayesian network multiple times. While these runs can be parallelized, the total resource consumption is greater than running the model once, limiting scalability. The exact trade-off is contingent on the number of runs of the Bayesian network. The higher the number of runs the more accurate the prediction distribution, but the higher also the resource consumption. Hence, the Bayesian network is good for detecting when the model can generalize but it is not optimal for scalability.

Another way to deal with OOD data is to model the training data distribution and to measure the distance of any new data sample from the training distribution. Here the uncertainty estimator would produce this distance as the uncertainty estimate. This method is scalable, since the distance calculation requires few vector operations. This uncertainty estimator requires defining a threshold distance for a high-dimensional distribution consisting of multiple features. Defining such a threshold is not intuitive, and defining it incorrectly can limit the system's ability to detect when the model cannot generalize. The Bayesian neural network avoids this issue because one only needs to define the threshold uncertainty in the target label space which tends to be one-dimensional for most systems tasks, and hence is easier to reason about.

Alternatively, one can use a Bayesian neural network that produces the parameters of a distribution as the output, instead of point estimates. In our case, the model would produce the mean and variance that define the distribution of memory bandwidth utilization. With such a network, one only needs to perform one inference to obtain the prediction distribution. However, when trying such a model we found that its memory usage exceeded the capabilities of a single GPU, due to the size of the weight and activation matrices for the intermediate layers needed to learn the representation for the distribution. Here again there is a trade-off between the ability to detect generalizability and the model's scalability, with the latter being limited by the memory and not the CPU usage.

Given these trade-offs, one should decide the estimator by analyzing the properties of the specific use case. If the input feature space is low-dimensional, using a simple distance based approach can be sufficient. If one has a high-dimensional feature space, using a Bayesian model would be a better approach. We plan to perform a thorough quantitative analysis of the trade-offs offered by different uncertainty estimators and their effects of the task in future work.

VI. Conclusion

We highlight the importance of generalizability of ML models when using them for systems tasks. We showcase that models deployed in cloud environments can have poor generalizability and propose estimating uncertainty via Bayesian models to identify scenarios where models do not generalize.

References

[1] Y. Zhang, W. Hua, Z. Zhou, E. Suh, and C. Delimitrou, "Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices," in *Proceedings of ASPLOS*, April 2021.

[2] S. Chen, A. Jin, C. Delimitrou, and J. Martinez, "ReTail: Opting for Learning Simplicity to Enable QoS-Aware Power Management in the Cloud," in *Proceedings of HPCA-28*, February 2022.

[3] M. Maas, D. G. Andersen, M. Isard, M. M. Javanmard, K. S. McKinley, and C. Raffel, "Learning-based memory allocation for c++ server workloads," in *Proceedings of ASPLOS*, 2020, p. 541–556.

[4] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A hierarchical neural model of data prefetching," in *ASPLOS*, 2021.

[5] N. P. Jouppi, C. Young, N. Patil, D. Patterson, and et al., "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017.

[6] Z. Shen, J. Liu, Y. He, X. Zhang, R. Xu, H. Yu, and P. Cui, "Towards out-of-distribution generalization: A survey," *CoRR 2108.13624*, 2021.

[7] I. Gulrajani and D. Lopez-Paz, "In search of lost domain generalization," *arXiv:2007.01434*, 2020.

[8] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt, "Google-wide profiling: A continuous profiling infrastructure for data centers," *IEEE Micro*, vol. 30, no. 4, pp. 65–79, 2010.