Towards Practical, Generalizable Machine-Learning Training Pipelines to build Regression Models for Predicting Application Resource Needs on HPC Systems

SWATHI VALLABHAJOYULA* and RAJIV RAMNATH*, The Ohio State University, USA

This paper explores the potential for cost-effectively developing generalizable and scalable machine-learning-based regression models for predicting the approximate execution time of an HPC application given its input data and parameters. This work examines: (a) to what extent models can be trained on scaled-down datasets on commodity environments and adapted to production environments, (b) to what extent models built for specific applications can generalize to other applications within a family, and (c) how the most appropriate model may change based on the type of data and its mix. As part of this work, we also describe and show the use of an automatable pipeline for generating the necessary training data and building the model.

CCS Concepts: • **Software and its engineering** → *Designing software*; • **Computing methodologies** → **Cost-sensitive learning**.

Additional Key Words and Phrases: automated data generation, ML, execution time, model scalability, model transferability

ACM Reference Format:
Swathi Vallabhajoyula and Rajiv Ramnath. 2022. Towards Practical, Generalizable Machine-Learning Training Pipelines to build Regression Models for Predicting Application Resource Needs on HPC Systems. In *Practice and Experience in Advanced Research Computing (PEARC '22), July 10–14, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3491418.3535172

1 INTRODUCTION

Problem Statement: The availability of shared high-performance computing (HPC) cyber-infrastructures (CI) such as those available at the Ohio Supercomputer Center [1] enables scientists to perform computationally intensive experiments such as simulations for weather prediction [2], or those using deep neural networks (DNN) at scale[3]. Multiple users share resources on these CI for simple to complex jobs with varying execution times and resource needs. Consumers are billed incrementally for nodes, processing time, and memory. Users run their applications on multiple environments as their research progresses, starting with explorations on their personal computers or a lab cluster, then moving to a HPC center or to new systems as they become available. Users have a limited understanding of their applications' resource needs; our investigations show that users habitually over-provision jobs to ensure that they execute till completion.

Over-provisioning resources increases cost and, counter-intuitively, increases turnaround time, as HPC schedulers wait for appropriately-sized windows in job queues. Better tuning resource requests would result in better utilization of shared resources, faster turnaround of jobs, and potentially reduced budgets, i.e. faster time-to-science at lower cost.

Data Requirements: The execution time of an application depends on **application-specific** features, such as its hyperparameters (e.g., batch size, epochs etc. for DNN) and the amount of input data. It also depends on the execution **environment-specific** features; that is, the characteristics of the cyberinfrastructure, such as the number and type of

*Both authors contributed equally to this research.
[1]https://www.osc.edu/about/mission
[2]https://www.olcf.ornl.gov/2019/05/07/mapping-climate-patterns/
[3]https://www.nvidia.com/en-us/data-center/resources/intersection-of-hpc-and-machine-learning/

Manuscript submitted to ACM

nodes, and their configuration (GPU, memory, processor speed, intra-node bandwidth etc.)[1, 7]. Extant research has explored models built using either application-specific features or environment-specific features [3, 6]. Our work does both together, i.e. builds ML models from application-specific and environment-specific elements taken together.

Model Evaluation: Regression models are typically evaluated using mean square error as the metric, with negative and positive errors treated identically. However, for execution-time prediction, models with negative errors (i.e., underpredictions) have a greater (and negative) impact because, for example, an application that has underestimated its resource needs will be aborted on shared batch-oriented systems. Thus, our models seek to reduce under-predictions. At the same time, our models cannot simply over-estimate every job because doing so would incur longer waits.[4]

Tractability: Machine-learning (ML) models are expensive in terms of computation and execution time. Minimizing data, training, and infrastructure needs for model building is essential for ML approaches to be cost-effective and practical, that is, tractable. To this end, we explored the tractability of model building. Specifically, we assessed to what extent models built on data from a small set of application runs on an minimal, compatible environment (such as a workstation or a single core HPC node) could be extrapolated or generalized. We examined multiple axes of generalizability: **(a)** to different target environments (such as an HPC center with multi-many cores that include GPUs, high-bandwidth communication libraries, and greater memory), and **(b)** to other applications within the same "family". This study also examined the effects of incrementally augmenting our models with data from runs on the target environment (essentially evaluating continuous learning over time).

Model Selection: We examined several regression models before settling on the two models - a one-layer neural network and a decision tree - for evaluation. We noted that some parameters in the data were numeric, while others were categorical, which could influence model selection. These explorations, combined with exploring tractability, serendipitously led us to an interesting insight concerning model selection. As we included more data, the most effective model type changed.

Contributions: The contributions of this paper centered on demonstrating the tractability and generalizability of our approach, that is: **(a)** Demonstrating an automated pipeline for tractable model development - i.e., collecting data and training the model - easily configurable to accommodate different target applications and ML techniques. The code for this automated pipeline is available in GitHub[5]; **(b)** Demonstrating how predictive models built with data on one target environment can be extrapolated to other target environments; **(c)** Demonstrating to what extent predictive models built with data on a specific application can be extrapolated to other applications in the same "family"; and **(d)** Insights on model selection, in particular, demonstrating that the best model depends upon the type and the scale of the data available, but that models specially designed for scale, such as DNN, may not always be the best.

2 RESEARCH PIPELINE

This section describes the components of our research pipeline - automated data collection, the target applications, execution environments, and datasets.

Automated Data Collection Each target application was incorporated into the framework and treated as a black box that exposes tunable parameters through which their behavior was modified systematically to generate the training data. This training data was then pipelined into the model training. The following are the tools and steps used to generate training data: (**a**) We used the **Cheetah** Experiment Harness a Campaign Management System [5] to run a target application with different training executions configured using a campaign file into a single submission.

[4]https://www.osc.edu/supercomputing/batch-processing-at-osc/job-scripts
[5]https://github.com/manikyaswathi/PreditingExecutionTime

(**b**) The outcomes, such as its run time, memory requirements, and application-specific output, were analyzed using the Tuning and Analysis Utilities (**TAU**) profiler [8]. (**c**) A post-execution script gathers the features and generates application-specific training datasets. (**d**) We build the selected regression models after running *correlation* analysis and principal component analysis (*PCA*) on the generated data.

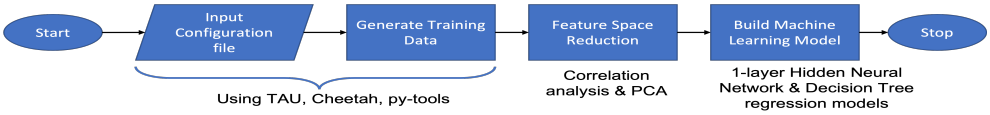

Fig. 1. Visualizing the Automated Data Collection Pipeline and Model Generation

Target Applications: We chose three target applications that spanned a range of behavior and had a different portfolio of resource needs: (a) Gray Scott Simulator [4] is a chemical diffusion simulator that is computationally intensive. **(b)** Trimmomatic [2] is used to pre-clean raw genome sequence raw archives (SRA) to get a higher assembly quality and is both compute and I/O intensive. **(c)** DNNs for Image Classification is a family of deep neural network models - TensorFlow's image classification models: VGG16, ResNet50, and InceptionV3 - with a few thousand learnable parameters. The memory and execution requirements change based on the number of images and batch size, making them both computationally and memory intensive.

Model Features and Execution environments: Features that impact the execution time of an application are shown in Figure 2. Our experiments were run on the following systems: **(a)** Linux CPU with 8-cores and 16 GB Memory and **(b)** CPUs and GPUs hosted on Owens and Pitzer cluster at OSC [6] to generate training data.

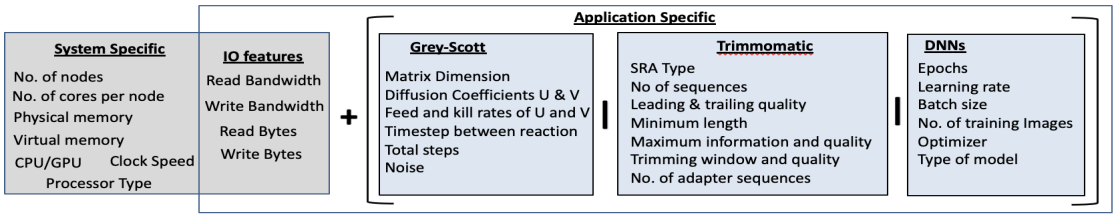

Fig. 2. The system-specific and application specific features used in our experiments

Dataset Generation: Our goal concerning data generation was to explore the tractability of the model building process. Generating training data at large using full input or training data is expensive. Thus, we sought to assess to what extent a model built from data collected from a small set of runs of a single application on a chosen environment could then be extrapolated or generalized. We start by generating "scaled-down" (**SD**) data on minimally expensive compatible execution environments (workstations or single-core HPC node) where the target application could run till completion. We slowly add more full-scale data as it becomes available when we run the application in a production environment (say by using an entire node on an HPC). We expanded the SD data by adding these "full-scale" (**FS**) instances. Essentially, the goal is to see how a regression model can learn to scale before and as "real" data becomes available. The configurations required to generate SD and FS data depend on the target application and execution environment. We generated 8088:398, 3234:128, and 1313:210 SD:FS samples for Gray-Scott, Trimmomatic, and Family of DNNs, respectively.

[6]https://www.osc.edu/services/cluster_computing

3 REGRESSION MODELS AND MODEL SELECTION CRITERIA

Regression models and Metrics: We tried several regression models - Linear Regression, Support Vector Machines, Decision Tree Regression, Random Forest, and Neural Networks. The accuracy of these models varies based on the distribution of data and the types of features. For example, Linear regression predicted negative execution time, Decision trees and random forests seem to perform equally well on our data sets, but decision trees were faster to train. We also experimented with neural network models using different hyperparameters as they could generalize to different applications and work well with non-linear data. We used two different regression models throughout our experiments: (a) One Layer Hidden Neural Network (**1LHNN**) with Adam optimizer, trained with 0.01 learning rate on a 20 batch size for 300 epochs (best-identified configuration); (b) Scikit-learn's [7] Decision Tree Regressor (**DTR**) with default configurations. For there models, we measure: **(a)** Mean Absolute Error (**MAE**) and Mean Percentage Error (**MPE**) between the predicted and actual values (rounded to nearest integer); and (b) Under Prediction Percentage (**UPP**): The percentage of test set jobs with $predicted < actual$ execution times.

Model Selection: The goal of model selection is to choose a model that tries to reduces the number of under-predictions while retaining a lower MAE. When comparing two regression models, A and B, we select a better-performing model based on the following heuristic criteria: **(a)** select a model with lower MAE and lower UPP; or **(b)** When $UPP(A) > UPP(B)$ and $MAE(A) < MAE(B)$, select A if the relative change between the models for MAE is greater than or equal to UPP (i.e. $|MAE(B) - MAE(A)|/MAE(A) >= |UPP(B) - UPP(A)|/UPP(A)$). If this relation doesn't hold, select B. (This rule is applicable in vice-versa case)

4 EXPERIMENTS AND RESULTS

We generated scaled-down (SD) and full-scale(FS) data with a systematic sweep of application parameters and execution environments per application using Cheetah. We then split our FS data into 50:25:25 training, validation, and test sets. Given our primary goal of reducing under-predictions in a controlled manner, we reduced the number of under-predictions made by the regression models by multiplying the model predictions with an Underprediction Adjustment Multiplier (UPAM), computed per model and dataset using the FS validation dataset. We set a UPAM value that corrected at least 50% of the underpredictions from the validation set. We then applied this UPAM to the test predictions by adjusting them to $executiontime = UPAM * prediction$. **Note that experiments 1 and 2 report scores on adjusted predictions**.

Baseline: We use FS datasets to train the models and predict execution time without (checkUPAM) and with UPAM (BaselineFS) adjustments. The UPAM adjustment reduced the number of underpredictions on the FS test dataset (checkUPAM to BaselineFS in Table 1).

4.1 Experiment 1: Scaling models with respect to FS data

We start by training the model only on the SD data while testing it on FS data to see if it could scale the predictions (NoFS model). We gradually augmented the SD training data with samples from FS data and trained two additional models (25%FS and 50%FS). The 25%FS and 50%FS respectively added 25% and 50% of the FS samples to the SD training data.

From Table 1, we can see that 1LHNN performs better for all applications when we have no additional FS training data (NoFS) compared to the baseline (built on FS data), where DTR was getting better accuracies. As we add more

[7] https://scikit-learn.org/stable/index.html

		checkUPAM		BaselineFS		NoFS		25%PS		50%FS	
		MAE	UPP	MAE	UPP	MAE	UPP	MAE	UPP	MAE	UPP
		MPE	UPAM	MPE	UPAM	MPE	UPAM	MPE	UPAM	MPE	UPAM
Gray-Scott	1LHNN	255.30	47.52	317.2	9.90	**94.16**	**11.88**	82.78	17.82	**167.61**	**3.96**
		215	1x	269	1.2x	**88**	**1.2x**	78	1.4x	**162**	**1.7x**
	DTR	35.33	19.80	**47.23**	**7.92**	58.89	39.60	413.05	30.69	267.78	16.83
		37	1x	**48**	**1.1x**	52	2.1x	294	1.2x	522	1.1x
Trimm-omatic	1LHNN	15.49	66.66	44.57	7.69	**225.64**	**64.10**	305.07	64.10	435.88	61.53
		4	1x	10	1.1x	**50**	**38.2x**	68	30.1x	119	74.9x
	DTR	12.30	43.58	**48.32**	**2.56**	1125.13	43.58	61.77	7.69	**46.86**	**5.12**
		3	1x	**11**	**1.1x**	1344	328.6x	12	1.1x	**10**	**1.1x**
Family of DNNs	1LHNN	223.02	55.55	337.37	27.77	**708.43**	**1.85**	**292.42**	**11.11**	511.30	5.55
		58	1x	81	1.3x	**517**	**1.4x**	**136**	**1.5x**	445	1.2x
	DTR	178.17	35.18	**226.90**	**14.81**	922.70	12.96	1811.05	9.25	599.84	14.81
		40	1x	**49**	**1.1x**	856	1.9x	518	2.3x	333	1.3x

Table 1. Experiments to show the scalability of modules when the prediction is adjusted with respect to the FS validation data

Source of Training data	Target Application											
	VGG16				RN50				InsV3			
	1LHNN		DTR		1LHNN		DTR		1LHNN		DTR	
	MAE	UPP	MAE	UPP	MAE	UPP	MAE	UPP	MAE	UPP	MAE	UPP
	MPE	UPAM	MPE	UPAM	MPE	UPAM	MPE	UPAM	MPE	UPAM	MPE	UPAM
VGG16	**1043**	**10.5**	**1272**	**26.31**	1141	0.00	597	21.05	3283	0.00	305	52.63
	741	**1.4x**	**369**	**1.3x**	322	1x	569	4.3x	1249	1x	252	2.1x
RN50	3224	5.26	253	21.05	**1097**	**0.00**	**1546**	**10.52**	1295	15.78	1197	5.26
	1191	2.1x	78	2.4x	**962**	**1.1x**	**297**	**3.6x**	216	3.8x	465	4.6x
InsV3	51229	0.00	426	36.84	3075	21.05	245	36.84	**1041**	**0.00**	**1058**	**0.00**
	20378	5.0x	179	3.7x	2264	1.1x	80	1.6x	**603.08**	**1.1x**	**545.24**	**2.4x**
Others All	951	0.00	242	31.57	266	31.57	747	26.31	1988	0.00	752	0.00
	262	2.0x	59	2.0x	95	1.2x	903	1.7x	184	3.8x	217	1.5x
Others +25%FS	**139**	**10.52**	1733	10.52	**439**	**0.00**	769	10.52	524	0.00	1246	5.2
	81	**1.5x**	509	1.8x	**184**	**1.4x**	217	1.8x	123	2.0x	808	1.6x
Others +50%FS	371	15.78	727	15.78	506	0.0	606	10.52	**222**	**31.57**	485	10.52
	265	1.4x	237	1.3x	206	1.6x	129	1.8x	**53**	**1.9x**	365	1.1x

Table 2. Experiments to show the transferability of models within a family of applications of regression modules

training runs from FS data, we can see a shift in the "selected model" as we change the training data from NoFS to 25%FS or 50%FS for Trmmomatic.

4.2 Experiment 2: Using existing models trained on other applications in same family

We choose VGG16, ResNet50(RN50), and InceptionV3(InsV3) from the Family of DNNs. To analyze the transferability of a model within the family of applications, we predict execution time for a target application by training the following modules: **(a)** OneVsOne - train models on all applications in the family one at a time including target application (**BaslineFS**); (**b**) OtherAll - train the model on samples of all applications except the target; **(c)** Others+25%FS & Others+50%FS- train these models on all samples from all applications except the target, augmented with 25% or 50% of FS samples from the target application.

From Table 2, we notice that models built on training data of one application in the family give better predictions for the target applications than its baselines (OthersAll for VGG16 and RN50), signifying the **transferability**. The UPAM factor could indicate which training dataset could fit a target application. A higher scalability factor could result in

high over predictions (as in the case of InsV3 to VGG16). We could gradually add FS training samples from the target application to improve the predictions. An increase in the UPAM factor after adding a few additional FS training samples could cause higher over predictions, thus reducing the model accuracies. The newly generated FS runs tend to have a narrow distribution of features compared to SD data, thus causing a higher UPAM factor. *When we look at the MPE, we could see that the model selection criterion is picking a model with a lower value (while still keeping the under-predictions lower as a priority).*

5 CONCLUSION AND FUTURE WORK

We have explored the feasibility of building a framework that can generate training data for a given application through this work. Using an existing experimental harness called Cheetah, we automatically generated training data and captured both application and system-specific features with minimal human intervention. The proposed models were able to scale the predictions if either the execution environment or application-specific configurations were changed. These models were able to transfer within a family. When we try to apply a model generated on the family of experiments for a new application in the family, we need to include the adjustment percentage that can aid in scaling a model to the new application.

We propose future work in three areas: **(a)** Model selection: From the results (Experiments 1 & 2), it is evident that choosing a regression model specific to the application does not depend on one metric (like MAE or UPP). We plan to evaluate other heuristics for model selection. **(b)** Dealing with missing values: The current models predict execution time for the test data and not inference data. Since the inference data do not have runtime features like I/O bandwidth, we need to fill in the missing values for these features before predicting execution time. **(c)** Cost models: To choose a regression model, we need to compute a cost metric that measures the trade-off between successful execution of an application due to over-estimation (expecting higher wait times) and failure to execute an application due to under-prediction.

REFERENCES

[1] Marcos Amarís, Raphael Y. de Camargo, Mohamed Dyab, Alfredo Goldman, and Denis Trystram. 2016. A comparison of GPU execution time prediction using machine learning and analytical modeling. In *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*. 326–333. https://doi.org/10.1109/NCA.2016.7778637

[2] Anthony M Bolger, Marc Lohse, and Bjoern Usadel. 2014. Trimmomatic: a flexible trimmer for Illumina sequence data. *Bioinformatics* 30, 15 (2014), 2114–2120.

[3] Ling Huang, Jinzhu Jia, B. Yu, Byung-Gon Chun, Petros Maniatis, and Mayur Naik. 2010. Predicting Execution Time of Computer Programs Using Sparse Polynomial Regression. 883–891.

[4] Jeff S. McGough and Kyle Riley. 2004. Pattern formation in the Gray–Scott model. *Nonlinear Analysis: Real World Applications* 5, 1 (2004), 105–121. https://doi.org/10.1016/S1468-1218(03)00020-8

[5] Kshitij Mehta, Bryce Allen, Matthew Wolf, Jeremy Logan, Eric Suchyta, Jong Choi, Keichi Takahashi, Igor Yakushin, Todd Munson, Ian Foster, and Scott Klasky. 2019. A Codesign Framework for Online Data Analysis and Reduction. In *2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. 11–20. https://doi.org/10.1109/WORKS49585.2019.00007

[6] Tudor Miu and Paolo Missier. 2012. Predicting the Execution Time of Workflow Activities Based on Their Input Features. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 64–72. https://doi.org/10.1109/SC.Companion.2012.21

[7] Farrukh Nadeem and Thomas Fahringer. 2013. Optimizing Execution Time Predictions of Scientific Workflow Applications in the Grid through Evolutionary Programming. *Future Gener. Comput. Syst.* 29, 4 (jun 2013), 926–935. https://doi.org/10.1016/j.future.2012.10.005

[8] Sameer S Shende and Allen D Malony. 2006. The TAU parallel performance system. *The International Journal of High Performance Computing Applications* 20, 2 (2006), 287–311.