UC Merced
UC Merced Electronic Theses and Dissertations

Title
Accelerating HPC Applications Using Machine Learning-based Approximation

Permalink
https://escholarship.org/uc/item/4nn1w1bj

Author
Dong, Wenqian

Publication Date
2022

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, MERCED

Accelerating HPC Applications Using Machine Learning-based Approximation

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Electrical Engineering & Computer Science

by

Wenqian Dong

Committee in charge:

Associate Professor Dong Li, Chair
Professor Shawn Newsam
Assistant Professor Shijia Pan
PNNL Research Scientist Gokcen Kestor

2022

Copyright
Wenqian Dong, 2022
All rights reserved.

The dissertation of Wenqian Dong is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

(Professor Shawn Newsam)

(Assistant Professor Shijia Pan)

(Associate Professor Dong Li, Chair)

University of California, Merced

2022

TABLE OF CONTENTS

Signature Page iii

Table of Contents iv

List of Figures viii

List of Tables x

Acknowledgements xi

Abstract xiii

1 Introduction 1
1.1 Research Chanllenges 2
1.2 Research Focus 2
1.2.1 Domain-Specific, NN-based Computation Approximation to HPC Applications 3
1.2.2 Automation of NN-based Approximation 5
1.3 Dissertation Roadmap 7

2 Adaptive Neural Network-Based Approximation to Accelerate Eulerian Fluid Simulation 9
2.1 Introduction 9
2.2 Background 12
2.2.1 Eulerian Fluid Simulation 12
2.2.2 Neural Network-Based Approximation 14
2.2.3 Motivation of Our Work 16
2.3 Overview 17
2.4 Approximate Model Construction 18
2.5 Offline Output-Quality Control 22
2.5.1 Construction of Training Samples 23
2.5.2 MLP Model Construction and Loss Function 24
2.5.3 Usage of MLP 25

2.6 Quality-Aware Runtime Design 26
2.6.1 Prediction of Simulation Quality Loss 26
2.6.2 Quality-Aware Model-Switch Algorithm 30
2.7 Evaluation 31
2.7.1 Model Speedup and Accuracy 32
2.7.2 Analysis on Runtime System 34
2.7.3 Evaluation of MLP Effectiveness 36
2.7.4 Sensitivity Study: Check Interval 37
2.7.5 Evaluation of Resource Usage 37
2.8 Conclusions 38

3 SmartPGsim: Using Neural Network to Accelerate AC-OPF Power Grid Simulation 39
3.1 Introduction 39
3.2 Background 43
3.2.1 Problem Formulation for AC-Optimal Power Flow . . . 43
3.2.2 Primal-dual Interior Point Solver 44
3.3 Related Work 45
3.4 Overview 46
3.5 Sensitivity Study 47
3.6 An interactive learning model 49
3.6.1 Multitask Learning 49
3.6.2 Domain-Specific Design 50
3.6.3 Details on Multitask Learning Model 52
3.7 Physics-Informed Learning 52
3.7.1 Embedding AC Nodal Power Balance Equations 53
3.7.2 Guarding Inequality Constraints 54
3.7.3 Optimization of Cost Function 55
3.7.4 Implying Lagrangian Conservation 55
3.8 Evaluation 56
3.8.1 Smart-PGsim Performance Evaluation 57
3.8.2 Performance Breakdown 59
3.8.3 Prediction Accuracy 59

3.8.4 Efficiency of Multitask Learning and Physical Constraints 60
3.8.5 Scalability Analysis on Multi-Node Systems 62
3.8.6 Comparison with Prior Work 64
3.9 Discussions 65
3.9.1 Generality of Proposed Approach 65
3.9.2 Analysis of Diverging Cases 66
3.10 Conclusions 67

4 Auto-HPCnet: An Automatic Framework to Build Neural Network-based Surrogate Model for HPC applications 68
4.1 Introduction 68
4.2 Background and Motivation 72
4.3 Data Acquisition 74
4.4 Input analysis 76
4.4.1 Autoencoders for Feature Reduction 77
4.4.2 Customized Design for Sparse Input 77
4.4.3 Workflow of Applying Autoencoding 78
4.5 2D Neural Architecture Search 79
4.5.1 Problem Definition 79
4.5.2 Hierarchical Bayesian Optimization 80
4.6 Implementation 81
4.6.1 Interaction with Users 82
4.6.2 Quality-Oriented Optimizations 82
4.6.3 Online Inference Invocation 84
4.7 Evaluation 85
4.7.1 AutoHPCnet Effectiveness 86
4.7.2 Comparison with Existing Work 89
4.7.3 Effectiveness of AutoHPCnet Components 90
4.7.4 Feasibility Analysis 92
4.8 Related Work 92
4.9 Conclusions 93

5 Other work 94

6 Conclusions 97
6.1 Summary of Contributions 97
6.2 Future Work 99

7 Related Work 103

LIST OF FIGURES

Figure 2.1: Distribution of quality loss for the Tompson's model with different input problems [165]. 16
Figure 2.2: Workflow of the proposed Smart-fluidnet. 17
Figure 2.3: Scatter plot of quality loss and time cost of different neural network models. 19
Figure 2.4: The network architecture of our MLP model. 22
Figure 2.5: Training losses of five MLPs. 24
Figure 2.6: Relationship between $CumDivNorm$ and Q_{loss}^{ts}. 27
Figure 2.7: An example to explain our runtime algorithm. 31
Figure 2.8: Performance (execution time) of the Tompson's model and Smart-fluidnet. 32
Figure 2.9: Variation of quality loss with various grid sizes for input problems. . . 33
Figure 2.10: Performance (execution time) for the Tompson's model and Smart-fluidnet. 34
Figure 2.11: Variation of simulation quality in different model candidates. 34
Figure 2.12: Success rate of reaching target quality with or without using MLP. . . 36
Figure 2.13: Impact of the check interval on the success rate. 37

Figure 3.1: Workflow of the proposed Smart-PGsim 46
Figure 3.2: Topology of the MTL. 51
Figure 3.3: Embedding AC physical laws in MTL training 54
Figure 3.4: Comparison of three aspects between MIPS and Smart-PGsim. 58
Figure 3.5: Execution time breakdown 58
Figure 3.6: Prediction accuracy of each feature used in the proposed MTL model. . 60
Figure 3.7: Performance comparison 61
Figure 3.8: Accuracy comparison. 61
Figure 3.9: Scaling across many GPUs 62
Figure 3.10: The asymptotic convergence of the tracking loss along the iterations . 66

Figure 4.1: AutoHPCnet's workflow 71
Figure 4.2: An example of applying the surrogate model. 72

Figure 4.3: An example of acquiring input and output variables. 75
Figure 4.4: The workflow of applying Autoencoder in AutoHPCnet 76
Figure 4.5: Speedup and prediction hit rate in AutoHPCnet. 88
Figure 4.6: Performance comparison of other representative methods 90

LIST OF TABLES

Table 2.1: Execution time and simulation quality loss of three models for solving the Poisson's equation. 16
Table 2.2: Percentage of input problems with which the simulation reaches the requirement on quality. 33
Table 2.3: Execution time distribution for the five neural network models used by Smart-fluidnet at runtime. 36
Table 2.4: Resource usage of different methods. 38

Table 3.1: **Ablation study on the input signals** 47
Table 3.2: Configurations in IEEE bus systems. 57
Table 3.3: Prediction Performance Comparison. 64

Table 4.1: Configurations in AutoHPCnet. 83
Table 4.2: Applications for Evaluation. 86
Table 4.3: Compare the performance of AutoHPCnet on GPU with the performance of the original code on GPU. The results are for AMG. 87
Table 4.4: Comparison of input compression ratio and model size. 91

ACKNOWLEDGEMENTS

My PhD is nowhere close to being easy, but because of these people, I have gained enough support and strength to get through and accomplish this journey.

First, I feel lucky to have Prof.Dong Li, who I truthfully respect to, be my academic advisor during my PhD. He guided me from the first class I took from him, to my every paper writing, to my dissertation day, and I believe to my future everyday. My understanding of scientific machine learning brought by Prof.Li. Learning about how machine learning models 'fire' the computation power fascinates me, and in retrospect, greatly contributes to the goal of my thesis to bring advanced AI techniques to scientific applications. My influence from Prof.Li started before I met him. He lets me see the joy of manipulating every paper, and inspires me to look beyond a single academic publication to exploit the interesting problems and the studying processes among different works. He never looks down on my naive academic ideas and he is always supportive to help me polish them well. Plus, he shows me what passion feels like from his teaching, presentations and even normal conversations. I hope I will be able to follow in his footsteps as a professor.

I also thank Gokcen Kestor for being on my mentor at PNNL, sharing her insights on systems for parallel computing and scientific applications. I want to thank for her steadfast support and mentorship. She is a great woman and a role model for me. I learnt kindness, patience, braveness, and wisdom from her, which encourages me to be confident in my research and personal life. I appreciate being given the flexibility to choose my own research direction and work on projects that I was passionate about. This was especially challenging when things did not go according to plan, but Gokcen always believed in me and taught me how to fail fast when necessary. I am thankful for her advice on research, technical writing, and connecting with members of the broader research community. I always felt I could talk to Gokcen about anything, professional or personal, without judgment; such confidence played a crucial role in my ability to succeed in graduate school.

I probably wouldn't have started my journey towards a Ph.D. if it hadn't been for mentors such as Roberto Gioiosa at PNNL, as well as Martin Foltin and Cong Xu formerly at HP Research Lab. They played a pivotal role in sparking my interest in machine learning, scientific applications, software engineering, and in pursuing a career in research. I'd like to thank them who gave me the opportunity to research scientific machine learning and who also did anything within their abilities to ensure that I had every possible resource needed

to succeed.

I would like to thank a number of faculty members at UCM for their time, support, and invaluable advice. First, I want to thank my committee members Prof.Shawn Newsam and Prof.Shijia Pan who gave me a lot of instructions in thesis writing and academic consulting. I also want to thank Prof.Hyeran Jeon for her patience to give me instructions for job research.

I am fortunate to have had fantastic coauthers, colleagues and friends who made me look forward to making the trek up to my research goal every day: Jie Ren, Jie Liu, Kai Wu, Zhen Xie, Anzheng Guolu, Shuangyan Yang, Dong Xu, Jiajia Li, Tong Geng, Xiuming Shang, Yuqiu Kong, Yiwen Chen, Sidi Zhao, Ting Liu, Haiyang Wang, Yuquan Chen, and YaoRong Fan.

Last but not least, I am grateful to my parents and friends for always being there through the ups and downs and encouraging me to keep going.

ABSTRACT OF THE DISSERTATION

Accelerating HPC Applications Using Machine Learning-based Approximation

by

Wenqian Dong
Doctor of Philosophy in Electrical Engineering & Computer Science

University of California Merced, 2022

Associate Professor Dong Li, Chair

Historically, numerical analysis has formed the backbone of supercomputing for decades by applying mathematical models of first-principle physics to simulate the behavior of systems from subatomic to a galactic scale. Recently, scientists have begun experimenting with a new approach to understanding complex systems using machine learning (ML) predictive models, primarily Deep Neural Networks (DNN), trained by the virtually unlimited data sets produced from traditional analysis and direct observation. Early results indicate that these "synthesis models" combining ML and traditional simulation, can improve accuracy, accelerate time to solution and significantly reduce costs.

In this thesis, we study how to enhance the usability of machine learning models to accelerate HPC applications. We first study an application, the Eulerian fluid simulation. The Eulerian fluid simulation is an important HPC application. The current methods that accelerate the fluid simulation with Neural Networks (NNs) lack flexibility and generalization. In this application, we tackle the above limitation and aim to enhance the applicability of NNs in the Eulerian fluid simulation. We introduce Smart-fluidnet, a framework that automates model generation and application. Given an existing NN as input, Smart-fluidnet generates multiple NNs before the simulation to meet the execution time and simulation quality requirement. During the simulation, Smart-fluidnet dynamically switches the NNs to make best efforts to reach the user's requirement on simulation quality. Evaluating with 20,480 input problems, we show that Smart-fluidnet achieves 1.46x and 590x speedup comparing with a state-of-the-art NN model and the original fluid simulation respectively on an NVIDIA Titan X Pascal GPU, while providing better simulation quality than the state-of-

the-art model.

Second, we explore another HPC application, the powergrid simulation. The basic powergrid simulation is an electricity generation model to minimize cost with generation constraints, line flow constraints, and bus voltage constraints. We use a machine learning model to generate a warm start startpoint solution for the power grid simulation, in order accelerate the simulation. We develop a framework, Smart-PGsim, which generates multitask-learning (MTL) neural network (NN) models to predict the initial values of variables critical to the problem convergence. MTL models allow information sharing when predicting multiple dependent variables while including customized layers to predict individual variables. We show that, to achieve the required accuracy, it is paramount to embed domain-specific constraints derived from the specific power-grid components in the MTL model. Smart-PGsim then employs the predicted initial values as a high-quality initial condition for the power-grid numerical solver (warm start), resulting in both higher performance compared to state-of-the-art solutions while maintaining the required accuracy. Smart-PGsim brings $2.60\times$ speedup on average (up to $3.28\times$) computed over 10,000 problems, without losing solution optimality.

Third, we build a tool that can help the domain scientist automatically apply machine learning models to HPC applications. We introduce a framework, named AutoHPC-net, to democratize the usage of machine learning-based approximation. AutoHPC-net is the first end-to-end framework that makes past proposals for machine learning-based approximation practical and disciplined. AutoHPC-net introduces a workflow to address unique challenges when applying the approximation, such as feature acquisition and meeting the application-specific constraint on computation quality. Evaluating with a set of HPC applications that previously cannot run on GPU, we show that using AutoHPC-net, those applications can leverage NN and GPU to achieve $4.34\times$ speedup on average (up to $7.39\times$ speedup and with data preparation cost included) while meeting the application-specific constraint on computation quality.

As the future work, we propose the following two research tasks. First, we plan to exploit another important HPC application, i.e., a DFT-based ab initio quantum chemistry method(DQC). DQC is computation-intensive and involves frequent irregular memory access, which is promising to be benefited from NN based approximation. Second, we plan to implement a framework which can automatically identify and replace promising code re-

gions (which is time-consuming and frequently invoked), and automatically generate suitable NN models to approximate the code regions.

Chapter 1

Introduction

Large-scale scientific simulations drive scientific discovery across many disciplines. Those scientific simulations increasingly face performance problems, because of hardware heterogeneity, deep memory hierarchy, and massive thread-level parallelism. Addressing those problems often requires domain scientists to use sophisticated compiler and runtime techniques to optimize High performance computing (HPC) programs. However, domain scientists are often not skilled computer scientists, and may find program optimization time-consuming and daunting. In this project, we study how to use an alternative approach, machine learning (ML) to effectively improve performance of scientific simulation without losing simulation quality.

Machine learning, as a tool to learn and model complicated (non)linear relationships between input and output data sets, has shown preliminary success in some HPC problems. Using machine learning, scientists are able to augment existing simulations by improving accuracy and significantly reducing latency. For example, scientists working to detect neutrinos at Fermi National lab have realized a 33% improvement in neutrinos detection using a convolutional neural network [129]; Scientists achieve Bose-Einstein Condensates state in only 10-12 experiments using machine learning instead of 140 experiences using traditional models, which reduces the simulation time by 10 times [4]. Other successful examples of using machine learning for HPC include recognizing extreme weather events in large-scale climate simulations at Lawrence Berkeley National Lab (LBNL) [127], and precision medicine for cancer [3] at Argonne National Lab.

1.1 Research Chanllenges

The current methodology to apply machine learning to scientific simulations faces the following challenges. First, there is a lack of a systematic approach to ensure high simulation quality when using machine learning. To be used as a scientific methodology for common practice in scientific and engineering simulation, ML-based approximation must be robust and reliable.

Second, the current construction of ML models pays special attention to model accuracy, but the execution time of ML models does not attract enough attention. The ultimate goal of ML-based approximation is to reduce the execution time of HPC applications. How to minimize execution time without impacting computation accuracy is a challenge.

Third, the current ML-based approximation (especially neural network (NN)-based approximation) to accelerate HPC applications lacks a generalization ability. Typically, once an NN has been selected, this NN model is used for all input problems, which either leaves the performance opportunities on the table, have large simulation quality violations, or both. Research is needed to improve the coverage of input problems, such that the NN modeling can be generally applied to provide high quality approximation for various input problems with minimized execution time.

At last, there is no systematic approach to automate the application of NN models to scientific simulations. Traditionally, the NN model is manually constructed by computer scientists without domain knowledge and without the awareness of the requirements of domain scientists. Automating the model construction such that domain scientists can easily build the model without caring about NN topology and data transformation is very beneficial to enhance the usability of NN for HPC applications.

1.2 Research Focus

In particular, my research focuses on using NN-based computation approximation to accelerate HPC applications. NN-based approximation is used to bring significant performance improvement to the application without violating numerical simulation correctness and stableness, by replacing a solver or an execution phase (e.g., PCG [165] and FFT [88]) in the HPC application while using the same input/output as the solver or the execution phase. I design efficient ML algorithms and runtime systems. Those algorithms are inte-

grated into HPC applications to approximate computation at runtime with provable guarantees on computation quality. My research includes parallel system designs and computational complexity analysis. My work has given new insights on NN approximability and interpretability in the context of HPC applications. In particular, I work on the following research topics.

1.2.1 Domain-Specific, NN-based Computation Approximation to HPC Applications

By using NN as a tool to learn and model complicated (non-)linear relationships between input and output data sets, scientists have shown preliminary success in some HPC problems (e.g., detecting neutrinos [130], climate simulations [128], and fluid dynamic simulation [41]). Compared to domains such as image recognition and natural language processing (NLP), scientific HPC applications require a level of precision and robustness that may not be provided by most of the current ML methods employed in other domains. My work takes a principled approach toward domain-specific, NN-based computation approximation, and focuses on adaptive approximation and incorporating physical information to meet the precision and robust requirement of HPC applications. I depict the two focuses as follows.

- **Adaptive approximation.** The current method of applying NN-based approximation to HPC applications lacks reliability and flexibility. Given a simulation code, the current method usually generates and applies just one NN model to approximate computation. At runtime, this model is used throughout the whole execution to approximate some target computation. This method ignores the fact that replacing the target computation at different execution phases of the application can have different implications on the simulation quality. At some execution phase, using a different NN model may be able to generate higher simulation quality without losing performance. Hence, using multiple NN models instead of one is a better strategy. However, with just one single NN model, the current method lacks such flexibility.
- **Incorporating domain knowledge.** Traditionally, the NN model is manually constructed by computer scientists as a black box with limited or no domain knowledge and without considering domain requirements. Domain knowledge, such as some

physical laws (e.g. Nodal Power Balance law in power grid system), has been largely ignored during the construction of NN models. However, including this knowledge into the NN model can not only improve modeling accuracy but also improve interpretability of the model.

Based upon adaptive approximation and the strategy of incorporating domain knowledge, I studied two mission-critical HPC applications: the Eulerian fluid simulation and power grid simulation, depicted as follows.

Application 1: Adaptive NN-Based Approximation to Accelerate Eulerian Fluid Simulation. The current methods of using neural networks to approximate the Eulerian fluid simulation have fundamental limitations: Given a fluid simulation code with the NN model applied, there is no guarantee that the application can run the simulation to completion and meet the requirement on simulation quality for every input problem. Also, using a single NN model for all input problems either leaves the performance opportunities on the table, have large simulation quality violations, or both.

In our work [165], we tackle the above limitation and introduce Smart-fluidnet [41], a framework that automates NN model generation and application to the Eulerian fluid simulation. Given an existing NN as input, Smart-fluidnet generates multiple NNs before the simulation to meet the execution time and simulation quality requirement from the user. During the simulation, Smart-fluidnet dynamically switches the NNs to make the best efforts to reach the user's requirement on simulation quality. Evaluating with 20,480 input problems, we show that Smart-fluidnet achieves 1.46x and 590x speedup, compared with a state-of-the-art NN model and the original fluid simulation respectively on an NVIDIA Titan X Pascal GPU, while providing better simulation quality than the state-of-the-art model.

Application 2: Using NN to Accelerate AC-OPF Power Grid Simulation. In this work [42], we study the implication of using ML techniques to accelerate the power-grid simulations. We study the structure of the NN model to be used, the relative importance of the model features selected, and, most importantly, the impact of incorporating physics constraints on the performance of the NN model. Specifically, we develop a framework, SmartPGsim [42], which generates a multitask-learning (MTL) NN model to predict the initial values of variables critical to the convergence of the AC-OPF power grid simulation.

To study the importance of feature, we perform a sensitivity study to understand the

impact of the NN output accuracy on execution time and convergence, by applying precise or imprecise values to features. This sensitivity study provides guidance on choosing a correct and efficient NN topology. Based on sensitivity study, we build an MTL model. The MTL model includes layers shared between multiple prediction tasks (i.e., predicting the values of multiple initial variables with dependency), which allows information communication between tasks; The MTL model also includes separated layers to enable customized design for each prediction task. Also, to achieve the accuracy required by the simulation, Smart-PGSim embeds domain-specific constraints derived from the power-grid simulation into the MTL model. In particular, Smart-PGSim imposes those constraints into the training objective function (soft constraints) or into the last layer of the MTL model based on transformation of equality and inequality in the constraints (hard constraints).

SmartPGsim employs the initial values predicted by the MLT model as a high-quality initial condition for the power-grid numerical solver, resulting in both higher performance compared to state-of-the-art solutions while maintaining the required accuracy. SmartPGsim brings $2.60\times$ speedup on average (up to $3.28\times$) computed over 10,000 input problems, without losing solution optimality. This work is based on the collaboration with the Pacific Northwest National Laboratory (PNNL), and highlighted at the PNNL and DOE websites [125, 39]. This work leads to results of both theoretical and practical impacts.

1.2.2 Automation of NN-based Approximation

Although using NN-based approximation to accelerate HPC applications is promising, there is a lack of systematic tools that can democratize the use of NN-based approximation. In practice, once the domain scientist selects a code region in an HPC application to be approximated, he/she has to manually observe the code region, evaluate which variables in the code region are good to be taken as NN features, and select an appropriate NN topology based on selected features. This process is labor-intensive, and could be repeated multiple times before the NN model is finalized. Even worse, the domain scientist may not have sufficient knowledge on NN models. Specifically, the gap between domain scientist and automatic NN-based approximation can be summarized as follows.

- **Feature extraction.** Identifying the inputs/outputs of the NN model is difficult. In NN-based approximation, we determine the input/output variables of the approximated code region, and use them as inputs/outputs of the NN models. The input

variables of the code region are read inside the code region to update other variables; the output variables are updated in the code region and used after the code region. Manually examining the code region to identify those variables is difficult, because the number of variables could be large. There is no such a tool available to provide end-to-end support to simplify the variable identification. Also, to reliably and efficiently use the NN model in scientific applications, we require that the ML features have the ability to represent the *important but trimmed* attributes to guarantee the quality in the outcomes of scientific applications. The quality of outcomes include the ML prediction accuracy and cost. Hence, our goal is to find "good" features keeping important physical information to guarantee the simulation quality but removing redundant features to reduce the overhead of NN computation in HPC applications.

- **Coordination between feature extraction and NN topology construction.** The feature reduction and selection of NN model topology are tightly coupled, and how to coordinate the two processes to minimize execution time and maximize accuracy of the NN model is a challenge. The NN model topology refers to the number of network layers, the type of each layer (e.g., fully connected, convolution, deconvolution, or recurrent), and the number of neurons in each layer. Both the number of features and NN model topology impact model execution time and accuracy. On the one hand, the number of features determines the first layer in the NN model and impacts the design of following layers; on the other hand, the topology selection of NN model reflects feature eligibility. The existing Neural Architecture Search (NAS) methods [72, 10, 116] do not consider such interaction between feature reduction and NN topology.

Automatic feature extraction allows us to efficiently pave the way for domain scientists who want to build their ML models but have limited experience of feature engineering. since deciding the input/output feature is the first step toward ML model construction. We build a tool identifying the inputs/outputs of the NN model from the source code of scientific application. Specifically, the tool constructs a dynamic data dependency graph (DDDG) from an instruction trace. In the graph, edges are LLVM instructions (or operations) transforming input values into output values of variables. With DDDG, the root nodes represent inputs and leaf nodes represent outputs. Such a tool provides end-to-end support to simplify the variable identification process. Furthermore, we quantitatively ana-

lyze the sensitivity of output variables of the source code to simulation execution time and convergence. We introduce two data types, i.e., imprecise default data and precise simulation data, to study the impact of noisy feature to simulation quality and execution time. Based on the sensitivity analysis, we can remove some redundant output features (those that do not impact the final simulation coverage) and their related input features (those that have connections with redundant output features in DDDG).

To coordinate feature reduction and selection of NN model topology, we introduce a two-level hierarchical Bayesian optimization. This strategy is automated. At the first level (the higher level), this strategy uses a Bayesian optimization to decide the number of input features; at the second level (the lower level), this strategy uses another Bayesian optimization to decide the NN topology using an existing NN topology search mechanism (particularly AutoKeras [72]). The second level is based on the decision (the number of input features) of the first level. The two levels work iteratively and coordinately to consider the impact of both feature reduction and NN topology. Furthermore, we consider both execution time and correctness of the approximated application during the two-level Bayesian optimization. We use a user-given threshold as an application specific metric and incorporate this metric into the objective function of NN training, which uses the application metric as a soft constraint to improves the correctness of ML prediction.

Putting together feature acquisition and two-level Bayesian optimization, we build Auto-HPCnet, a workflow relieving the domain scientist from labor-intensive work to apply NN-based approximation to HPC applications. Evaluating with a set of HPC applications that previously cannot run on GPU, we show that using AutoHPCnet, those applications can leverage NN and GPU to achieve $4.34\times$ speedup on average (up to $7.39\times$ speedup and with data preparation cost included) while meeting the application-specific constraint on computation quality.

1.3 Dissertation Roadmap

Chapters 2, 3, and 4 dive into the technical details of making scientific machine learning approachable from the three aspects – ease to use, interpretability, and flexibility. These techniques use machine learning to embed sensitivity study, physical information, online quality monitoring into model construction that can then be used to guide each task.

In particular, **Chapter 2** studies an adaptive neural network-based approximation method to accelerate the Eulerian fluid simulation **Chapter 3** studies another HPC application, the powergird simulation. **Chapter 4** introduces an automatic tool to easy the use of NN-based surrogate model for domain scientists. **Chapter 5** lists my collaborated works and **Chapter 6** summarizes my research contributions and proposes my future research topics.

Chapter 2

Adaptive Neural Network-Based Approximation to Accelerate Eulerian Fluid Simulation

2.1 Introduction

The fluid simulation aims to study the flow of fluid materials and has been widely applied to multiple disciplines such as chemical physics and material science [12, 26, 135]. However, the simulation of fluid dynamics usually requires prohibitively high computational resources [175, 155] and thus limits its application in the related fields.

The neural network-based machine learning model, as a tool to learn and model complicated (non)linear relationships between input and output datasets, has shown preliminary success in HPC problems (e.g., detecting neutrinos [129], developing Bose-Einstein Condensates state [4], and recognizing extreme weather events [127]). Using neural networks, scientists are able to augment existing simulations by improving accuracy and significantly reducing latency. The neural network has been applied to accelerate fluid simulation as well [77, 165, 179]. By replacing some execution phases with neural networks, the most recent work reports $14.6\times$ to $716\times$ performance improvement [165].

However, the current methods to accelerate the fluid simulation using neural networks have fundamental limitations. First, the current method to apply the neural work to the fluid simulation lacks flexibility. In particular, given the simulation code, the current method

usually generates just one neural network model. At runtime, this model is used throughout the whole execution to approximate some target computation. This method ignores the fact that replacing the target computation at different execution phases of the fluid simulation can have different implications on the simulation quality. At some execution phase, using another neural network model may be able to generate higher simulation quality without losing performance. Hence, using multiple neural network models instead of one is a better strategy. However, with just one single neural network model, the current method lacks such flexibility.

Second, the current method to apply the neural network to the fluid simulation lacks a generalization ability. In particular, given a fluid simulation code with the neural network applied, there is no guarantee that the application can run the simulation to completion and meet the requirement on simulation quality for every input problem. Using a single neural network model for all input problems either leaves the performance opportunities on the table (discussed in the last paragraph), have large simulation quality violations, or both.

Third, there is no *systematic* approach to construct and apply neural network models to the fluid simulation. How to construct neural networks to meet the user requirement on performance (execution time) and simulation quality is challenging. Currently, domain scientists build neural networks intuitively. There is no systematic approach to help them build and apply neural networks for HPC applications. The recent work on Auto-Keras [73] and AutoML [46] aims to automatically generate a neural network model with high accuracy. However, they lack concerns on high performance (execution time), and focus on image processing or natural language processing. Hence, they are not directly usable by HPC.

In this paper, we tackle the above limitations and aim to enhance the applicability of neural networks in HPC applications (particularly the Eulerian fluid simulation). Given an existing neural network model as input, our system uses a systematic approach to construct multiple neural network models and dynamically switches them at runtime during the execution of the fluid simulation to meet the user requirements on performance and simulation quality.

In order to tackle the above limitations, we must address three challenges. First, we must automatically generate multiple neural network models to enable high flexibility and better generality when applying neural networks. Given the user requirements on performance and simulation quality, we aim to generate multiple neural network models, each of

which has different topologies and different implications on performance and simulation quality. We should not expect the domain scientists to manually construct models.

Second, how to select neural network models at runtime to enable the best performance without violating the quality requirement is a challenge. We must have a method to predict the impact of applying a neural network model at a certain execution phase on the final simulation quality.

Third, a neural network model can approximate the fluid simulation with high accuracy for some input problems but not for all. How to construct neural network models to provide a high-quality approximation for a large number of input problems and ensure overall performance benefit is another challenge.

In this paper, we focus on a Eulerian fluid dynamic simulation code (mantaflow [163]) and introduce a framework (named "Smart-fluidnet") to address the above three challenges. Smart-fluidnet has three major components. (1) It includes a model construction plugin for Auto-Keras to extend its functionality to enable automatic construction of multiple neural network models. (2) Smart-fluidnet also includes a multilayer perceptrons model (MLP) to guide the model selection process to meet the requirement on performance and simulation quality. The neural network models selected by MLP is able to cover more input problems to ensure overall performance benefit. (3) Smart-fluidnet includes a runtime component integrated into mantaflow and dynamically switches the neural network models to make best efforts to meet the user requirement on simulation quality. The runtime component is based on a metric and a lightweight runtime algorithm that can predict the final simulation quality in the middle of the fluid simulation.

We summarize the major contributions of this paper as follows:

- A systematic approach and a framework (Smart-fluidnet) to accelerate the Eulerian fluid simulation; Our evaluation shows that using 20,480 input problems for the simulation, Smart-fluidnet achieves 46% performance improvement over the Tompson's model [165] (a state-of-the-art neural network model) on average and $590\times$ speedup over the original fluid simulation on average, while providing better simulation quality than the Tompson's model.

- A new methodology that constructs, selects, and applies multiple neural network models (instead of one) to address the fundamental limitation of model flexibility and generalization in the existing neural network-based approximation. We demonstrate

great potential of using this methodology to meet user requirements on the simulation quality and execution time.

2.2 Background

In this section, we provide background on the Eulerian fluid simulation and neural network-based approximation. In the rest of the paper, the term *performance* means execution time, not prediction accuracy in the machine learning field. We also use the terms *approximation model* and *neural network model* interchangeably.

2.2.1 Eulerian Fluid Simulation

The Eulerian fluid simulation, in essence, solves the Navier-Stokes equations. The Navier-Stokes equations describe the fluid movement under a continuous velocity field $\vec{u}$ and a pressure field p. When the fluid has zero viscosity, one uses the incompressible Navier-Stokes equations, which can be expressed as the Euler equations as follows:

$$\frac{\partial \vec{u}}{\partial t} = -\vec{u} \cdot \bigtriangledown \vec{u} - \frac{1}{\rho} \bigtriangledown p + \vec{g}, \tag{2.1}$$

$$\bigtriangledown \cdot \vec{u} = 0 \tag{2.2}$$

Equation 2.1 is a vector equation called “momentum equation”. This equation can make the velocity field stay divergence-free. Equation 2.2 is the incompressibility condition, which enforces fluid volume to remain constant throughout the simulation. In the above two equations, t is time, ρ represents fluid density, and $\vec{g}$ represents gravity.

Mantaflow performs fluid simulation by iteratively solving Equations 2.1 and 2.2. In this paper, we use MAC (marker-and-cell) grids [61] to discretize fluid flows, and use the finite difference (FD) method to calculate partial derivative on each grid [70, 182].

For each velocity component that borders a grid cell, the FD method iteratively applies updates on velocity and pressure. In a grid cell, the pressure is sampled at the grid cell center and the velocity is sampled at the centers of the vertical faces of the grid cell. The above method is common and can simplify the handling of solid-cell boundaries conditions.

To solve Equations 2.1 and 2.2, mantaflow uses a standard operator splitting method [158, 178, 28] to split up Equation 2.1 (Equation 2.2 is used as a constraint) into three parts. The

three parts are advection, adding external force, and pressure projection. Algorithm 1 depicts the implementation of the Eulerian simulation in mantaflow, which includes the above three parts.

Algorithm 1 Velocity Update in the Euler Equation

Require: Simulation time step N;
1: Start with an initial divergence-free velocity field $\vec{u}^0$
2: Determine a good time step $\triangle t$ to go from time t_n to time t_{n+1}.
3: **for** $n \leftarrow 1$ to N **do**
4: **Advection.** Set $\vec{u}^A$ = advect($\vec{u}^n$, Δt, q);
5: **Add body force.** $\vec{u}^B = \vec{u}^A + \triangle t\vec{f}$;
6: **Pressure projection.** set $\vec{u}^{n+1}$ = Project($\triangle t$, $\vec{u}^B$):
7: 1) Solve the Poisson eq. $\bigtriangledown \cdot \bigtriangledown \vec{p_n} = \frac{1}{\triangle t} \bigtriangledown \cdot \vec{u}^B$
8: //Use a PCG solver to to update $\vec{p}_n$.
9: Set initial guess $\vec{p_n}$=0 and residual vector r=d (if r=0, then return $\vec{p_n}$)
10: Set search direction $\vec{s} = ApplyPreconditioner(r)$;
11: **while** residual doesn't reach the convergence criteria
12: **do**
13: Set $\alpha = \frac{r^T r}{s^T A s}$;
14: Calculate the residual $r = r - \alpha A\vec{p_n}$;
15: Update the solution $\vec{p_n} = \vec{p_n} + \alpha\vec{s}$;
16: Update the conjugated direction $\vec{s} = r + \beta\vec{s}$;
17: **end while**
18: 2) Apply velocity update: $\vec{u}^{n+1} = \vec{u}^B - \Delta t\frac{1}{\rho} \bigtriangledown \vec{p_n}$;
19: **end for**
20: **return** 0

The Eulerian simulation in mantaflow includes N time steps. The first part (Lines 4-5) of each time step is to solve the momentum equation to get an auxiliary velocity field $\vec{u}^B$. $\vec{u}^B$ is a velocity approximation which is not divergence-free, and the pressure-gradient term ($\bigtriangledown p$) used during the solving process is computed in the previous time step. At Line 7, the divergence-free pressure $\vec{p_n}$ is computed by solving a Poisson's equation which includes the divergence of $\vec{u}^B$ and a scaled gradient of the pressure. At Line 18, a divergence-free velocity field, $\vec{u}^{n+1}$ is calculated, by subtracting off the pressure gradient from the approximate velocity field $\vec{u_B}$. In the above process, solving the Poisson's equation is the most crucial and time-consuming step to preserve the divergence-free constraint on the velocity and maintain simulation accuracy.

The process of solving the Poisson's equation in mantaflow (Line 7 in Algorithm 1) is based on the Preconditioned Conjugate Gradient (PCG) method, which involves large com-

putation that iteratively converges to meet a convergence criteria (Lines 8-17). Mantaflow uses a multi-grid approach [106] as a preprocessing step of the PCG method. The preconditioner (Line 10) applied in mantaflow is the Modified Incomplete Cholesky L0 preconditioner, called "MICCG(0)". In this paper and the existing work [165], neural networks are used to approximate this PCG method.

In this paper, we simulate a 2D smoke plume [52, 49]. The simulation output in mantaflow is a smoke *dense matrix* of a rendered smoke frame. The smoke dense matrix represents density blurring of the plume, which reflects fluid movement. After using neural networks to approximate the computation in the fluid simulation, the output dense matrix can be different from that in the original simulation (using the mantaflow's PCG-based solver), which means we have quality loss. The simulation quality loss (Q_{loss}) is formally defined by:

$$Q_{loss} = \frac{1}{N \times M} \sum_{i=1}^{N} \sum_{j=1}^{M} \left| \rho_{ij}^{*} - \rho_{ij} \right|, \tag{2.3}$$

where ρ refers to the smoke density matrix generated in the original simulation, and ρ^* represents the smoke density matrix generated after applying neural network-based approximation. ρ_{ij} and ρ_{ij}^* are matrix elements with the coordinate (i, j). In essence, Equation 2.3 calculates the average relative error of all matrix elements. After applying the neural network-based approximation, we want to avoid quality loss (i.e., we do not want to lose simulation accuracy).

2.2.2 Neural Network-Based Approximation

The neural network is a general-purpose method that can be used to learn and model complicated linear and non-linear relationships between input and output datasets. Hence, the neural network has been used to approximate some conventional algorithms in an application to improve performance [77, 165, 179]. The neural networks are expected to generate similar outputs as the conventional algorithms when fed with the same inputs as for the conventional algorithms.

A neural network can be represented as a directed acyclic graph where nodes of the graph are connected neurons. Embedded in the graph, there are a number of parameters (or "weights"). Those neurons and weights are organized as layers. The process of obtaining the values of those weights are called *training*. Once the neural network is trained offline

using training datasets, it can be used online within the fluid dynamic simulation to improve performance. During training, an *objective function* is used to calculate the model accuracy loss, so that the weights can be adjusted to reach better model accuracy.

In this paper, we use Convolutional Neural Networks (CNN) to approximate computation (i.e., using the PCG solver to solve the Poisson's equation) in the Eulerian fluid simulation. This computation is the most time-consuming part of the Eulerian fluid simulation. Our profiling results reveal that this computation takes 70-80% of total simulation time.

Recent work [165] introduces an unsupervised learning framework to generate a CNN model with five stages of convolution and Rectified Linear Unit (ReLU) layers to approximate computation (the PCG solver) in the Eulerian fluid simulation. The inputs of the CNN are the divergence of the velocity field, denoted as $\nabla \cdot \ \vec{u}_t^*$, and the geometry field, denoted as g_{t-1}. The output of the CNN is the pressure field, denoted as $\hat{p}_t$. The mapping f_{conv} from input to output by this CNN model can be represented as follows:

$$\hat{p}_t = f_{conv}\Big(\nabla \cdot \ \vec{u}_t^*, g_{t-1}; W\Big), \tag{2.4}$$

where W is the CNN model parameters. The predicted $\hat{p}_t$ is used to update velocity $\vec{u_t}$ in Equation 2.1. In our study, this CNN (named as the Tompson's model) is used as input in Smart-fluidnet to generate new neural networks for online fluid simulation.

The objective function of the Tompson's model, i.e., $DivNorm$, is the sum of weighted L-2 norm of the divergence of the predicted velocity $\vec{u_t}$ over all fluid cells (mesh volumes) in the rendered smoke frame. $DivNorm$ is defined as follows:

$$DivNorm = \sum_i w_i \{\nabla \cdot \vec{u_t}\}_i^2, \tag{2.5}$$

where w_i is a weighting term for each fluid cell to emphasize the divergence of grids on geometry boundaries, i.e., $w_i = max(1, k - d_i)$. d_i is the distance field. It takes the value 0 for solid cells or the minimum Euclidean distance to the nearest solid cell for fluid-cells.

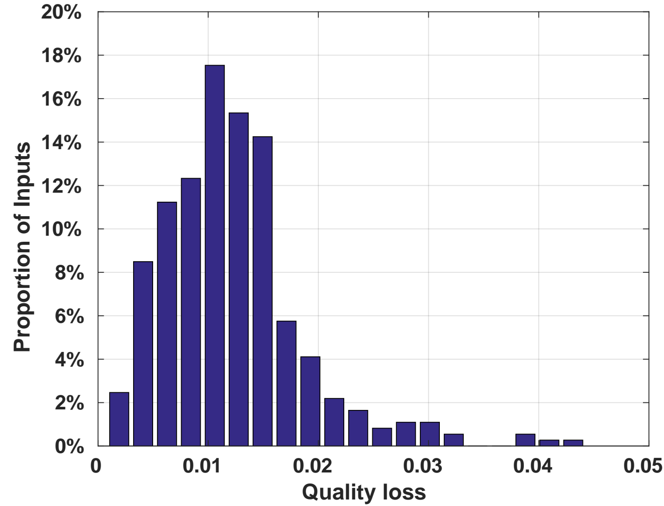

Figure 2.1: Distribution of quality loss for the Tompson's model with different input problems [165].

Table 2.1: Execution time and simulation quality loss of three models for solving the Poisson's equation.

Method	Execution Time (ms)	Avg. Quality Loss
PCG	2.34×10^8	$--$
Tompson [165]	7.19×10^4	1.3×10^{-2}
Yang [179]	3.20×10^4	4.9×10^{-2}

2.2.3 Motivation of Our Work

The existing work to solve the Poisson's equation includes the PCG solver in mantaflow and two neural network models (Tompson [165] and Yang [179]). We study the implications of the three methods on simulation quality and execution time. To study the impact of each model, we evaluate 20,480 different input problems of the fluid simulation and report the average simulation quality loss and execution time. The simulation quality loss Q_{loss} is calculated by comparing the simulation output and the real physical measurements on fluid flow. Table 2.1 and Figure 2.1 show the results.

Table 2.1 shows that PCG achieves the highest simulation quality as an exact solution but the worst performance (the longest execution time). On the other hand, the Yang's model performs $10^4\times$ faster than PCG but causes about $10^2\times$ loss in the simulation quality. There is a clear trade-off between simulation quality and performance.

Figure 2.1 shows the distribution of quality loss for various input problems when we

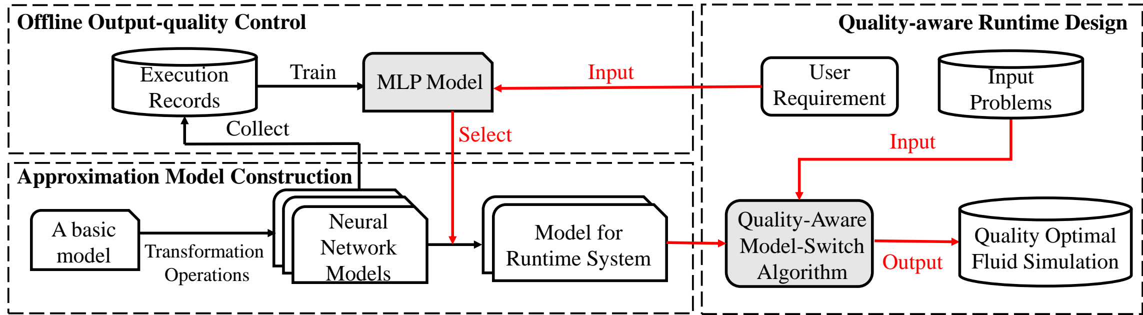

Figure 2.2: Workflow of the proposed Smart-fluidnet.

use the Tompson's model. The figure reveals that given various input problems, the model generates different simulation quality. For most input problems, the simulation quality loss is between 0.01 and 0.02. Given a user-defined quality requirement (e.g., the quality loss should be less than 0.01), the simulation may not meet the quality requirement for most input problems (e.g., around 65.42% input problems can not meet user requirement when the requirement is 0.01). We have the same observation for the Yang's model.

The above results reveal that it is imperative to use multiple models to explore the trade-off between performance and simulation quality and maximize the possibility of reaching the user requirement on the simulation quality for various input problems.

2.3 Overview

Figure 2.2 gives the workflow of Smart-fluidnet. The workflow includes offline and online phases. During the offline phase, Smart-fluidnet takes an existing neural network model as an input and constructs a set of neural network models by model transformation. We introduce four operations (deleting, narrowing, pooling and dropout) to transform the input neural model into multiple neural network models. Then we choose neural network models that are promising for high performance improvement and high quality based on the Pareto optimality analysis.

After model construction, Smart-fluidnet further chooses models based on the user requirement about simulation quality. Given various input problems of the fluid simulation, we introduce an MLP-based model to predict the probability of each model to reach the user requirement on the simulation quality. Considering the possible cost of restarting the simulation when the simulation quality does not meet the user requirement, we choose

those models that have a sufficiently high probability to benefit the performance improvement. After the above offline phase, we have a handful of neural network models ready for online approximation. At runtime, given an input problem of the Eulerian fluid simulation, Smart-fluidnet dynamically switches neural network models with the most promising neural network to meet the user requirement on the simulation quality. The model switching is based on a metric and a linear regression model to predict whether the current model can reach the requirement on the simulation quality at the end of the simulation.

The Smart-fluidnet framework consists of three main modules: approximation model construction (Section 2.4), offline output-quality control (Section 2.5), and quality-aware runtime design (Section 2.6). Their relationships are depicted in Figure 2.2. We explain them in detail as follows.

2.4 Approximate Model Construction

Given an input neural network, we transform it to construct multiple neural network models with different network architectures.

The new neural network models can be more accurate or more efficient (i.e., using less execution time) than the input neural network; Having such a mixture of different models provides flexible computation approximation during the online fluid simulation.

To generate more accurate neural network models, the user can use an existing framework such as Auto-Keras to generate and train models. Auto-Keras uses the Bayesian optimization to generate a single model with the best accuracy. We change Auto-keras to generate and train five models with the better accuracy. We generate five models, because generating more than five highly accurate models often causes more than five models to be selected after applying MLP (Section 2.5), which causes large runtime overhead when making the decision on switching models; While generating less than five models will lead to insufficient candidates after the model selection (Section 2.5).

Besides the above, we also aim to generate less accurate but faster models. We introduce new transformation operations into Auto-Keras to simplify the neural network architecture, which will shorten the execution time. We describe our transformation operations as follows.

Operation 1: deleting a layer of the neural network. This operation is denoted as $shallow(G, L)$,

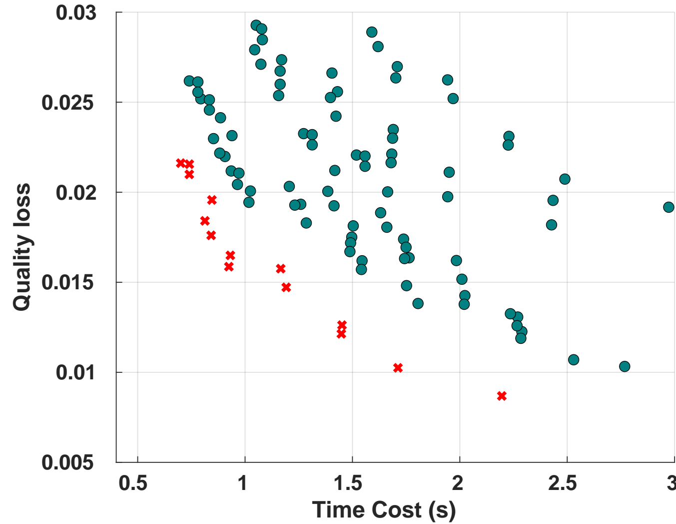

Figure 2.3: Scatter plot of quality loss and time cost of different neural network models.

where G is the neural network graph of the input neural network and L is the layer to be deleted. This operation not only shortens the depth of neural network but also reduces memory consumption, thus makes the fluid simulation time shorter.

Operation 2: narrowing a layer of the neural network. This operation reduces neurons in an intermediate layer. This operation is denoted as $narrow(G, L, r)$, where r is the number of neurons to be reduced at the layer L, and L can be either a fully-connected layer or a convolutional layer.

Operation 3: pooling. This operation, denoted as $pooling(G, L, m)$, downsamples a layer L using a pooling matrix m. We can apply two pooling strategies, i.e., $maxpooling$ and $averagepooling$. A special case of m is a 2×2 matrix which can discard 75% neurons in the intermediate layers.

Operation 4: dropout. This operation, denoted as $dropout(G, L, p)$, drops neurons at a layer L with a given probability p. It offers a more flexible way to reduce the number of neurons, compared with the second operation by controlling the value of p. This operation is useful to increase the generalization capability of the model.

Based on the above four operations, we transform the input neural network into new ones. Given an input neural network, we first apply a $shallow(G, L)$ operation on each of the intermediate layers. We do not apply the operation more than twice in the input model, and hence do not delete more than one intermediate layers. After applying $shallow(G, L)$, we generate five new models.

Second, we apply $narrow(G, L, r)$ operation on the five new models. A big value of r means a large number of neurons will be reduced. Based on our experimental results, the simulation quality loss can be large (more than 20%) for most of the input problems, if $r > \frac{|L|}{2}$, where $|L|$ is the total number of neurons. Therefore, we empirically use $r = \frac{|L|}{10}$. For each new model, we randomly choose r neurons to apply the $narrow(G, L, r)$ operation; Furthermore, for each new model, we apply $narrow(G, L, r)$ ten times, each of which generates a new model. In our case, in total, we have 55 new models (five new models after applying $shallow(G, L)$ and 50 more after applying $narrow(G, L, r)$).

Third, for the 55 new models, we apply $pooling(G, L, m)$ operations. In particular, for each new model, we randomly replace any of two neighbor-neurons with a new neuron using max pooling. The total number of neurons to be replaced is constrained to be 10% of the total neurons. After applying $pooling(G, L, m)$, we have 55 more new models (110 models in total).

At last, to enrich our neuron network models, we randomly select 18 out of the 110 models to apply the $dropout(G, L, p)$ operation. In particular, in each of the 18 models, we randomly drop out neurons. The total number of neurons to drop is limited to 10% of the total neurons. After applying $dropout(G, L, p)$, we have 18 more new models. In total, we have 128 models.

We apply the four operations in the above order, because the operation that tends to reduce more neurons than other operations will be performed earlier. This method allows us to efficiently generate new models. Using a different order can take longer time to generate models or be prone to generate less accurate models.

After the above model generation and in combination with the accurate models generated by Auto-Keras (five models), we have 133 models in total. Afterwards, we use *Pareto optimality* to reduce the number of models for online approximation. We select models that have the lowest time cost, the lowest quality loss, or both.

To explain the idea of Parento optimality, we use Figure 2.3 to illustrate it. Figure 2.3 shows the result of our model selection approach. In this figure, each point represents a model; The red points are those selected model for further analysis (Section 2.5); The green points are discarded models. In Figure 2.3, the quality loss and execution time for each model are collected during the model construction. This is a common method to collect the model information in AutoML work [157, 87]. Section 2.7 gives details on the

hardware platform and input datasets to collect the results in Figure 2.3. We can observe that those models located in the leftmost part of Figure 2.3 either have the lowest time cost, the lowest quality loss, or both. Those models (14 models) are selected based on the Pareto optimality method (we name them "model candidates" in the later discussion).

Sensitivity Study. The above process of constructing neural networks involves a couple of parameters. We summarize them as follows and change those parameters to study their impact on the simulation quality. In our study, we use 100 input problems. Using the simulation quality of PCG as the baseline, we calculate the average quality loss of all input problems when using the Tompson's model. This average quality loss is used as the user requirement for quality loss in our sensitivity study.

(1) The number of layers to prune (Operation 1). Our current method prunes one layer at most. For sensitivity study, We prune more than one layer, but find that it leads to a large quality violation (20% quality loss on average), which is not good.

(2) The percentage of neurons to apply pooling (Operation 3). Our current method applies pooling to 10% of total neurons. We set the percentage of neurons to apply pooling as 5%, 20% and 30%, and evaluate the impact of this parameter on the simulation quality. Our experiments show that using 20% and 30%, the fluid simulation has a large quality violation (35% and 50% quality loss on average, respectively), which is not good. However, using 5%, the fluid simulation has the similar quality loss on average as using 10%. In addition, since using 10% can lead to better performance than 5%, we choose 10% in our study.

(3) The dropout rate (Operation 4). Our current method drops out 10% of total neurons. We also try 5% and 15% as the dropout rate. Our experiments show that 5% and 10% outperform 15% in terms of the simulation quality loss. In particular, the quality losses with 5% and 10% as the dropout rate are 0.156 and 0.164 respectively, while the quality loss with 15% as the dropout rate is higher (0.239). Since 5% and 10% has the similar quality loss and using 10% leads to less execution time, we adopt 10% as our dropout rate.

(4) The number of neural network models to apply the dropout operation. Our current method chooses 18 models. Our study reveals that choosing 15-20 models are enough for the Pareto optimality and MLP (see Section 2.5) to generate 2-5 models. Using less than 15 models to apply the dropout operation, however, we could generate no qualified model after applying MLP. Using more than 20 models, we could have more than 5 qualified models

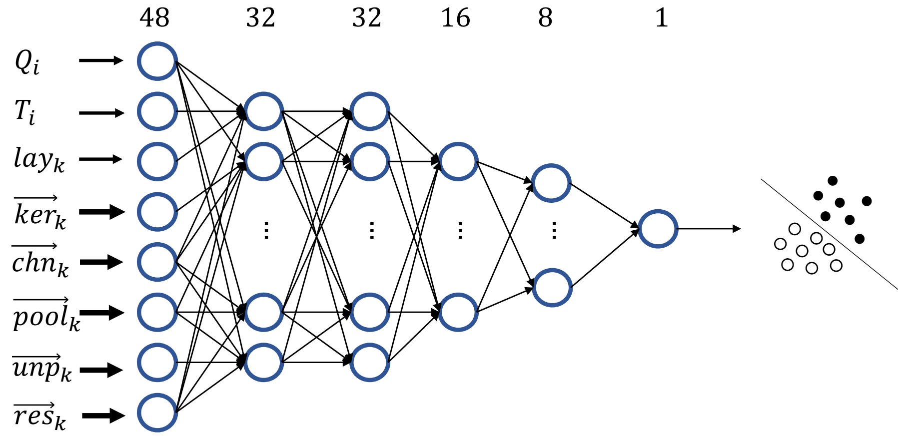

Figure 2.4: The network architecture of our MLP model.

after applying MLP. Having more than 5 models means that the runtime system may suffer from large runtime overhead for selecting a model to use. As a result, we choose 18 (in between 15 and 20) as our parameter.

2.5 Offline Output-Quality Control

In many fluid simulation cases, users can have specific requirement on the simulation quality and execution time. We use the notation, $U(q, t)$, to represent the user requirement, where q and t are the user requirement on the quality loss and execution time respectively. The final quality loss and execution time of the fluid simulation should be less than q and t, respectively. The quality control should be aware of the *success rate* of neural network models, namely the ratio of those input problems with which the fluid simulation can reach the simulation quality and time requirement to the total number of input problems.

The awareness of the success rate can be developed from statistical knowledge by executing neural network models on various input problems. In this section, we design a non-linear MLP model to develop such awareness. In the following subsections, we first introduce the construction of training samples for training the MLP model and then give details on how to construct and apply the MLP.

2.5.1 Construction of Training Samples

We collect execution records (i.e., simulation quality and execution time) for the 14 neural network models after the construction of those models (Section 2.4). Based on the execution records, we generate training samples to train the MLP model.

Collection of Execution Records. For each of the 14 neural network models, we get N execution records by running N input problems ($N = 20,480$ input problems in this paper). Each of the N execution records includes the simulation quality q_n^k and execution time t_n^k, where $n \in [1, N]$ and k refers to a neural network NN_k from the 14 neural network models. Each execution record is represented as ER_n^k. Such execution record is collected during the model construction (Section 2.4). Note that our model construction process, similar to other AutoML work [157, 87], includes not only changing model architecture but also training models. Hence we can collect execution records during the model construction.

Sample Generation. Given N execution records, we can generate samples to train the MLP. Each sample is represented with a feature vector using Equation 2.6.

$$F = \left(q, t, l_k, \overrightarrow{ker}_k, \overrightarrow{chn}_k, \overrightarrow{pool}_k, \overrightarrow{unp}_k, \overrightarrow{res}_k\right) \tag{2.6}$$

In Equation 2.6, the feature vector F includes quality and execution time requirements (q and t), and architecture information for the neural network NN_k (i.e., l_k, $\overrightarrow{ker}_k$, $\overrightarrow{chn}_k$, $\overrightarrow{pool}_k$, $\overrightarrow{unp}_k$, and $\overrightarrow{res}_k$, representing the number of layers, kernel sizes, channel number, pooling size, unpooling size, and residual connection of each layer respectively). Each of the last five architecture information is a vector composed of nine components that indicate properties of each layer of the neural network NN_k. Therefore, each input feature vector has $3 + 5 * 9 = 48$ components in total.

Given a neural network NN_k, we generate a sample by randomly picking up a user requirement (q and t), and then use Equation 2.6 to build the feature vector of the sample. Based on the N execution records and the user requirement, we calculate that how many of N execution records meet the user requirement. The ratio of those execution records to N, denoted as $r_{k,q,t}$, is the label of the sample. By choosing different combinations of q and t, we can generate as many samples as possible.

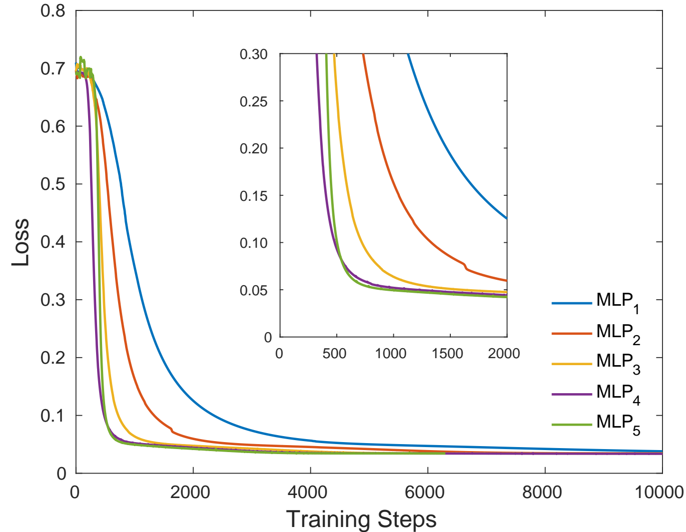

Figure 2.5: Training losses of five MLPs.

2.5.2 MLP Model Construction and Loss Function

Given a neural network NN_k and user requirement q and t, we can build a feature vector F using Equation 2.6. Our MLP model (see Equation 2.7) takes such a feature vector as input and generates an output $\hat{r}_{k,q,t}$, which is a floating point number indicating the probability that NN_k meets the user requirement for any input problem.

$$\hat{r}_{k,q,t} = f_{MLP}(F_{k,q,t}) \tag{2.7}$$

Figure 2.4 shows the network topology of our MLP. It includes six hidden layers and a 48-neuron input layer. The numbers of neurons in the six hidden layers are 32, 32, 16, 16, 8 and 8 respectively. All the neurons in the hidden layers use ReLU as activation to increase the non-linearity of the model. The neurons in the last hidden layer uses a sigmoid function as activation. Using the samples constructed in Section 2.5.1 to train the MLP, we aim to minimize the loss between the model output $\hat{r}_{k,q,t}$ and the ground truth label $r_{k,q,t}$.

Alternative MLP Topologies. Besides the above MLP topology, we try four alternative MLP topologies, in order to find one for best accuracy. Among the four MLPs, two of them are deeper than our current MLP, while the other two are shallower. When building an alternative MLP, we follow the rule that in the topology of a neural network, a deeper layer generally has less number of neurons than shallower ones (i.e., the number of neurons gradually decreases across layers). Many discriminative neural networks, such as Alex-

Net[81], VGG-Net[153], are constructed, following this rule. The architectures of the four models plus our current MLP model are briefly described as follows:

- MLP_1 has 4 layers with 48, 32, 16 and 1 neurons;
- MLP_2 has 5 layers with 48, 32, 16, 8 and 1 neurons;
- MLP_3 (our current MLP model) has 6 layers with 48, 32, 32, 16, 8 and 1 neurons;
- MLP_4 has 7 layers with 48, 64, 32, 32, 16, 8 and 1 neurons;
- MLP_5 has 8 layers with 48, 64, 64, 32, 32, 16, 8 and 1 neurons.

Figure 2.5 presents the training loss curves of the above five MLPs (MLP_3 is our current MLP model). We find that the convergence speed of MLP_3 is faster than those of MLP_1 and MLP_2, and offers lower training loss (i.e., higher prediction accuracy). Compared with MLP_3 model, MLP_4 and MLP_5 do not have significant advantages in terms of convergence speed and loss, although they have deeper topologies. Hence, MLP_3 exhibits a balanced trade-off between prediction accuracy and model size, and is thus chosen as our MLP model in this paper.

2.5.3 Usage of MLP

Given a user-specified simulation quality (q) and time cost (t), we use MLP to calculate $\hat{r}_{k,q,t}$ for a given neural network model NN_k. A larger value of $\hat{r}_{k,q,t}$ represents a higher success rate. In other words, it is highly possible that NN_k can meet the simulation quality and time requirement on an input problem.

Given the user-specified simulation quality (q) and time requirement (t), the neural network NN_k and MLP prediction result ($\hat{r}_{k,q,t}$), we use the following method to decide if NN_k should be selected for the runtime system for the fluid simulation. In particular, considering the probability that the user requirement on the simulation quality is violated and the user has to re-run the simulation without using any neural network, the simulation time is calculated based on Equation 2.8.

$$T_{total} = \hat{r}_{k,q,t} \times T_{M_k} + (1 - \hat{r}_{k,q,t}) \times T', \tag{2.8}$$

where T' is the execution time without using any neural network and T_{NN_k} is the execution time using the neural network NN_k. We compare T_{total} with the user requirement on the execution time t. Only those neural networks that have T_{total} less than t is selected.

The above selection method considers the impact of violating the simulation quality requirement on the simulation time, and ensure that if NN_k is repeatedly employed for many input problems, there is performance benefit.

2.6 Quality-Aware Runtime Design

After applying MLP, multiple neural networks are selected. We use a runtime technique to schedule those neural networks to optimize performance and meet the simulation quality requirement. In order to determine which neural network should be used at runtime, we need to evaluate the model being used in terms of the final quality loss, and switch to a suitable one if necessary. However, without running the simulation to completion, we cannot know the final quality loss. Thus we construct a metric called $CumDivNorm$ (defined in Equation 2.9), which is used to set up a bridge between $DivNorm$ (see Equation 2.5) measurable at runtime and the final simulation quality loss Q_{loss} (Section 2.6.1). Based on $CumDivNorm$ and the predicted final quality loss, we introduce a quality-aware model-switch algorithm (Section 2.6.2) to select the best neural network to accelerate the fluid simulation.

2.6.1 Prediction of Simulation Quality Loss

The objective function $DivNorm$ provides a goal that our neural network aims to achieve. Using $DivNorm$, we can know how the neural work performs in terms of prediction accuracy. However, there is a missing link between the prediction accuracy of the neural network and the final simulation quality loss Q_{loss}.

***CumDivNorm*: A Metric for Runtime Quality Control.** To explore the relationship between $DivNorm$ and final simulation quality Q_{loss}, we calculate $CumDivNorm$ (i.e., the accumulation of $DivNorm$) and Q_{loss} at *each simulation time step* (denoted as Q^{ts}_{loss}). The accumulation of $DivNorm$ over n time steps is defined in Equation 2.9.

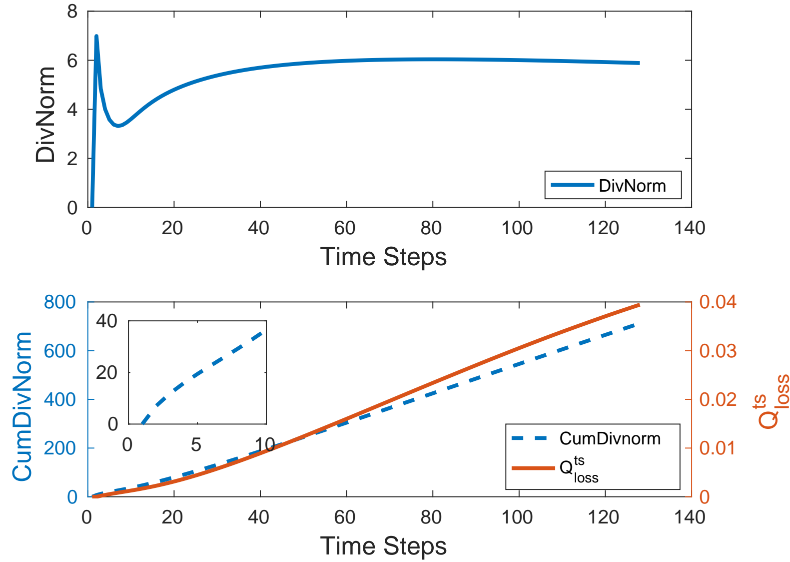

Figure 2.6: Relationship between $CumDivNorm$ and Q^{ts}_{loss}.

$$CumDivNorm = \sum_{i=1}^{n} DivNorm_i. \tag{2.9}$$

In order to observe how these varibles are correlated, Figure 2.6 depicts how $DivNorm$, $CumDivNorm$, and Q_{loss} vary across all time steps of the fluid simulation using an input problem with the grid size 1028*1028. We have the following observations. These observations are valid for other input problems as well.

- **Observation 1**: $DivNorm$ dramatically increases at the first few time steps and then gradually converges to a stable value;
- **Observation 2**: Q^{ts}_{loss} and $CumDivNorm$ have similar increasing tendency (except the first few time steps).

The above observations indicate that $CumDivNorm$ and Q^{ts}_{loss} calculated at each time step are correlated. In order to quantify the relationship between $CumDivNorm$ and Q^{ts}_{loss} at each time step, we use the Pearson's product moment correlation coefficient (rp) [113] and the Spearman's rank correlation coefficient (rs) [62] to statistically reveal the correlation between the two variables.

The two coefficients are defined as follows. Given two input vectors x and y (the vector length is n), the calculation of rp and rs to quantify the correlation between x and y

is formulated as follows:

$$rp = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2(y_i - \overline{y})^2}}, \tag{2.10}$$

$$rs = 1 - \frac{6\sum d_i^2}{n(n^2-1)}. \tag{2.11}$$

where x_i and y_i are the values of x and y for the i-th component, and d counts the pairwise disagreement (i.e., x_i is not equal to y_i) between the two vectors (see [8] for a more detailed description). In general, the coefficients that belong to [0.10-0.29] represent weak association, (0.29-0.49] represent medium association, and above 0.49 represent strong association [27].

In our study, we use 20,480 input problems, each of which will have 128 simulation time steps. The $CumDivNorm$ and Q_{loss}^{ts} of each time step will be calculated to build the two input vectors. Using Equations 2.10 and 2.11, we have $rp = 0.61$ and $rs = 0.79$, which indicates a strong correlation.

Based on the above discussion, it is possible to use $CumDivNorm$ to predict Q_{loss}^{ts} in the final time step (i.e., the final quality loss Q_{loss}). Note that we cannot calculate Q_{loss}^{ts} at runtime, because it involves PCG, which is too expensive. In the following discussion, we first discuss how to predict $CumDivNorm$ in the final time step ($CumDivNorm^{final}$), and then we discuss how to predict Q_{loss} based on the predicted $CumDivNorm^{final}$.

Predicting *CumDivNorm*^{final}. We introduce a lightweight approach to predict the bridge variable $CumDivNorm^{final}$. Our prediction approach is based on Figure 2.6. In this figure, it is obvious that $CumDivNorm$ quickly grows at first, and then the growth rate remains stable. This trend is general across all 20,480 input problems we test.

The stable growth rate of $CumDivNorm$ makes it possible to predict $CumDivNorm^{final}$ in the middle of the fluid simulation. In particular, we use five time steps to build a linear regression model ($f_k(x) = ax + b$), by the least square method, where x is the time step and $f_k(x)$ is the predicted $CumDivNorm$. Note that the data used to build the linear regression model must be collected *after* the growth rate becomes stable. Hence, we skip the first five time steps and build the regression model after each five steps. Also, in each five time steps (a check interval) to build the model, we skip the first two to make sure the trend is stable and only use the remaining three to build the model. The above model-building process happens every five time steps, and the model is used to predict and *check* Q_{loss} (see

Algorithm 2 The quality-aware model-switch runtime algorithm

Require: The user requirement $U(q, t)$.
1: Choose a neural network model M_k with the highest success rate according to MLP.
2: **while** t does not reach the final time step **do**
3: Send M_k to predict the final simulation quality.
4: **Prediction of quality loss:**
5: 1) Build a linear regression model with $DivNorm$ values measured in the last five time steps;
6: 2) Predict $CumDivNorm^{final}$ by the linear regression model;
7: 3) Predict Q'_{loss} of the current neural network model by the KNN algorithm;
8: **Model Switch:**
9: **if** Q'_{loss} is close to q **then**
10: Continue using the current neural network model for L steps;
11: **else if** Q'_{loss} less than q **then**
12: Switch to a faster (less accurate) neural network model;
13: **else if** Q'_{loss} is larger than q **then**
14: Switch to a slower (more accurate) neural network model;
15: **else if** Cannot find any model **then**
16: Restart by the PCG method;
17: **end if**
18: $t \leftarrow t + L$; //L is the check interval.
19: **end while**
20: **return** 0

the following discussion). Hence, we fix the check interval to be five time steps in the rest of the paper. In Section 2.7.4, we study the impact of the check interval on the simulation quality.

Predicting Q_{loss} based on $CumDivNorm^{final}$. We use a method based on the k-nearest neighbor (KNN) algorithm to predict Q_{loss}. Our method includes offline and online phases. During the offline phase, we test the neural network models selected by MLP with 128 small input problems. For each test, we collect a pair of data ($CumDivNorm^{final}$, Q_{loss}) and put them into a historical database. The offline phase is fast, because we use small input problems. During the online phase, at a time interval, to predict Q_{loss}, we check $CumDivNorm^{final}$ in the database and find k pairs whose $CumDivNorm^{final}$ are the closest to the predicted $CumDivNorm^{final}$ in the current time interval. We use the average of Q_{loss} in the k pairs as the predicted Q_{loss} in the current time interval. In our evaluation, we use different values for k, but find that $k \in [4, 6]$ is usually sufficient to give accurate prediction, hence we choose $k = 4$ to reduce runtime overhead.

For example, assuming that the predicted $CumDivNorm^{final}$ in a time interval is

108. To predict Q_{loss} for this time interval, we select four pairs from the database, i.e., (101, 0.09), (112, 0.11), (105, 0.10), and (109, 0.11), whose $CumDivNorm^{final}$ is closest to the given $CumDivNorm$ (108). Then the predicted Q_{loss} for this time interval is 0.1025 $\big((0.09 + 0.11 + 0.10 + 0.11)/4\big)$. We organize all data pairs as a binary search tree, such that finding the four pairs is cheap.

2.6.2 Quality-Aware Model-Switch Algorithm

With the ability to predict the simulation quality loss, we introduce a quality-aware model-switch algorithm. Algorithm 2 depicts this runtime algorithm. After applying MLP, we have several promising neural network models and their probabilities to meet the user-specified requirement. We also know the execution time (i.e., the inference time) of each network model. During the simulation, the neural network with the highest probability to meet user-specified requirement is selected as the first model to approximate computation in the fluid simulation. Then we calculate $CumDivNorms$ in the first check interval, build a linear regression model, calculate $CumDivNorm^{final}$ using the regression model, and predict Q_{loss} by the KNN algorithm. After that, we compare the predicted Q_{loss} (annotated with Q'_{loss} in the rest of the paper) with the user requirement q. If Q'_{loss} is close to q, we predict that the current neural network model can meet the user requirement. The runtime algorithm continues to use the current neural network model. But if Q'_{loss} is larger (or smaller) than q, then the runtime algorithm chooses a accurate (or fast) model with better (or worse) accuracy. If all the neural network models cannot meet q, we restart the simulation and use the traditional simulation method (i.e., the PCG method). The above model switch process happens periodically (the period is the check interval). We calculate $CumDivNorms$ at the end of every check interval to determine if the model switch is necessary.

An Example. Figure 2.7 gives an example to further explain our runtime algorithm. In this example, we have five neural network models and the user requirements on the simulation quality loss and execution time are 0.013 and 6.64s respectively.

During the offline phase, five neural network models are constructed by the model transformation (Section 2.4) and selected by MLP (Section 2.5); We record the possibility and execution time of the five neural network models (shown as Step 1 in Figure 2.7). At runtime, we use the first neural network ($M1$) which has the highest probability (91%)

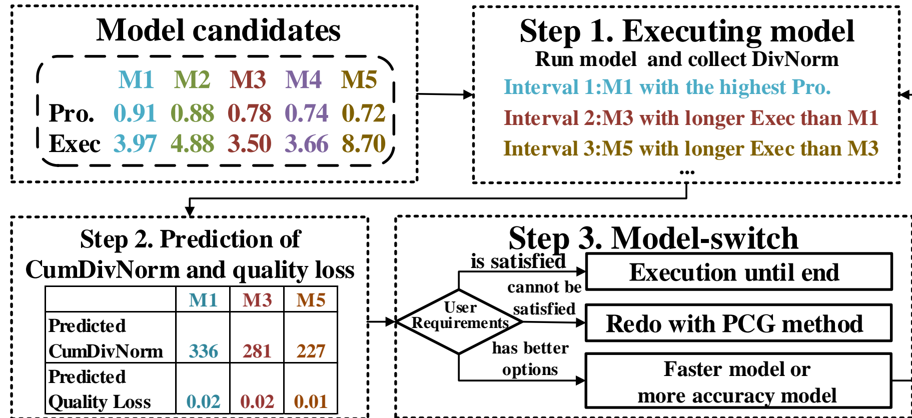

Figure 2.7: An example to explain our runtime algorithm.

to meet the user requirement on the simulation quality. Then we skip the first five steps and fit the values of $DivNorms$ in the first check interval into a linear regression model, and predict that the $CumDivNorm$ closes to 395 at the final time step. Using the KNN method, we predict Q_{loss} as 0.019, which is much larger than the user requirement (0.013). So we switch to a more accurate neural network model, i.e., $M3$ (a model with higher accuracy than $M1$ and the highest probability among remaining neural network models). After using $M3$ for another five time steps (one check interval), we predict Q'_{loss} of $M3$ as 0.015, which is still larger than the user requirement. So we switch to another more accurate neural network model, i.e., $M5$. We predict Q'_{loss} of $M5$ as 0.013, which can meet the user requirement, we then use $M5$ during the next check interval.

Compared with using a single neural network model, our runtime algorithm introduces additional computation (i.e., predicting $CumDivNorm$ and applying the KNN method). However, the computation of the simple linear regression algorithm to predict $CumDivNorm$ and the traversal of the binary tree to apply the KNN method are lightweight. Such overhead can be easily overweighed by the performance benefit introduced by adaptive neural network-based approximation. We evaluate performance (including the runtime overhead) in Section 2.7.2.

2.7 Evaluation

We evaluate our framework to examine its impact on performance and simulation quality of the Eulerian fluid simulation.

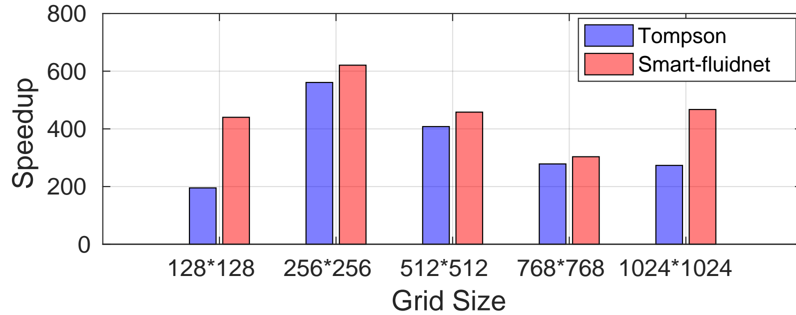

Figure 2.8: Performance (execution time) of the Tompson's model and Smart-fluidnet.

Platform. We conduct all experiments on a high-end server with 24 Intel Xeon E6-2760 v3 CPU cores running at 2.30GHz. The server is equipped with an NVIDIA Titan X (Pascal) GPU. We use cuDNN 5.0 on this GPU to run neural networks.

Fluid Simulation. During the fluid simulation, we run 128 time steps (the default number of time steps in mantaflow) for each input problem. To comprehensively evaluate the performance, we use multiple grid sizes for each input problem, including $128*128$, $256*256$, $512*512$, $768*768$, and $1024*1024$.

Input Datasets. We generate training and evaluation datasets by mantaflow, which is an open-source framework for the fluid simulation. The training dataset is used to optimize the parameters of the neural network models and MLP model, while the evaluation dataset is used to evaluate the performance of the online algorithm during the fluid simulation. Each of the two datasets contains 20,480 input problems. There is no overlapping between the training and test datasets. To generate the total 40,960 problems, we initialize velocity by a pseudo-random turbulent field [78], and generate occupancy grids with the border wall by introducing some objects in the simulation domain. Those objects are from the NTU 3D Model Dataset [126].

Neural Networks. We use Auto-Keras [73] with extension (Section 2.4) to search qualified convolutional neural network architectures. Auto-Keras constructs neural networks using Python, but the fluid simulation is implemented in Lua/Torch7. Hence we re-implement and optimize the neural networks with the Torch7 package.

2.7.1 Model Speedup and Accuracy

We conduct experiments to measure performance. We take the PCG solver as the baseline method. PCG is the traditional method used in the Eulerian fluid simulation and

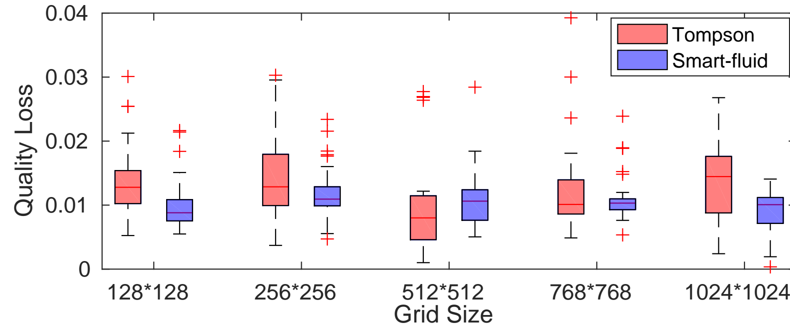

Figure 2.9: Variation of quality loss with various grid sizes for input problems.

Table 2.2: Percentage of input problems with which the simulation reaches the requirement on quality.

Grid size	128*128	256*256	512*512	768*768	1024*1024
Tompson	68.22%	67.16%	85.27%	71.06%	46.38%
Smart-fluidnet	88.27%	87.14%	91.36%	86.47%	91.05%

does not include any neural network. Compared with neural network-based approximation, PCG has the highest simulation quality but the performance is very bad. All performance (execution time) reported in this section is shown as "speedup" with respect to the performance of PCG.

Figure 2.8 shows the results for Smart-fluidnet and the Tompson's model with different grid sizes. The Tompson's model represents the state-of-the-art neural network to accelerate the Eulerian fluid simulation. In all test cases, Smart-fluidnet performs better than the Tompson's model and is $1.46\times$ better on average. The largest improvement over the Tompson's model is $2.25\times$. Besides the execution time, we also study the simulation quality of the Tompson's model and Smart-fluidnet. We use 20,480 input problems to evaluate each method. We use the simulation quality of PCG as the ground truth and study the quality loss of the Tompson's model and Smart-fluidnet. Since Smart-fluidnet requires the user to specify a requirement on the quality loss, we use the average quality loss of all input problems when using the Tompson's model, as the user requirement (the target).

Figure 2.9 presents boxplots[1] to show the results of the quality loss for all input problems with five selected grid sizes. We draw two observations from Figure 2.9: (1) The outputs of Smart-fluidnet are closer to the target value than those of the Tompson's model;

[1]In the boxplots, the boxes are bounded by 25th and 75th percentiles of the quality loss; The central marks of the boxes indicate the median; The '+' markers outside the boxes indicate the extreme outliers [33].

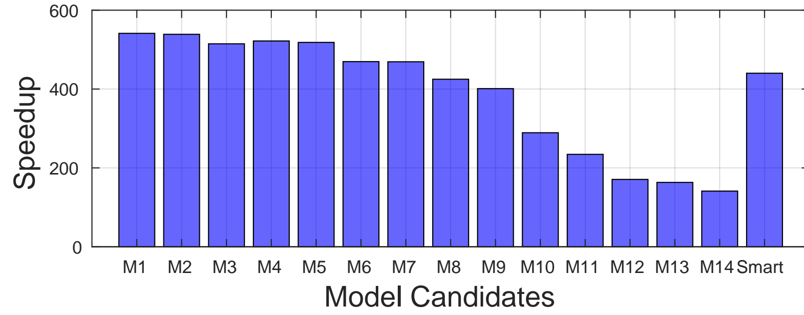

Figure 2.10: Performance (execution time) for the Tompson's model and Smart-fluidnet.

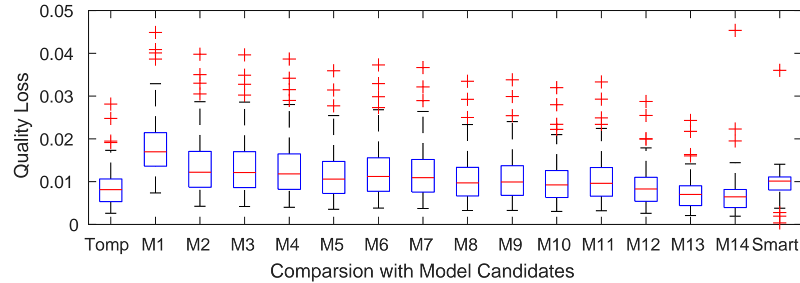

Figure 2.11: Variation of simulation quality in different model candidates.

(2) The variances of Smart-fluidnet are smaller than those of Tompson's model. These two observations reveal that Smart-fluidnet can give more *consistent simulation quality* than the Tompson's model, which is crucial for dealing with largely diversified input problems.

To further study the consistency of simulation quality with various input problems, we study how many input problems can lead to the simulation with satisfiable simulation quality. Table 2.2 shows the results. Table 2.2 reveals that Smart-fluidnet leads to a larger percentage of a high-quality simulation than the Tompson's model in all cases. The difference between Smart-fluidnet and Tompson's model is as large as 44.67% (when the grid size is 1024*1024).

2.7.2 Analysis on Runtime System

In this section, we analyze the effectiveness of our runtime system. Taking the grid size of 1024*1024 as an example, the average quality loss and execution time of the Tompson's model are 0.013 and 6.64 seconds respectively. We take this quality loss and execution time as the target (i.e., the user requirement) of Smart-fluidnet.

Speedup. We show the performance (the speedup of execution time) of running the 14 neural network models alone without model switching. We use the performance of PCG

as the baseline to calculate speedup. We also show the performance of Smart-fluidnet and compare it with the 14 individual models. Figure 2.10 shows that the performances of the 14 neural network models are quite different, with the speedup ranging from $541.25\times$ to $141.17\times$. The performance of Smart-fluidnet is close to the median performance (440.1 $\times$) of the 14 neural network models. This is the result of dynamically using different neural network models at runtime.

Quality. We compare the 14 neural network models, the Tompson's model, and Smart-fluidnet, in terms of quality loss. We calculate the quality loss using the method in Section 2.7.1. Figure 2.11 shows the results. Similar to Figure 2.9 in Section 2.7.1, the figure shows the distribution and variation using the boxplots. Figure 2.11 reveals that the variation of the quality loss in Smart-fluidnet with various input problems is much smaller than any of the 14 neural network models alone. With Smart-fluidnet, 91.05% of the input problems' simulation quality meet the user requirement. With the shortest and longest models (among the 14 neural network models), 12.52% and 92.71% of the input problems' simulation quality meet the user requirement.

Figures 2.10 and 2.11 include the results for using only the fastest model $M1$ or the most accurate model $M14$ throughout the simulation. $M1$ is $1.18\times$ faster than Smart-fluidnet, but achieves the user requirement on the simulation quality in only 12.52% of the input problems (for Smart-fluidnet, it is 91.05); $M14$ achieves the user requirement on the simulation qualtiy in 92.71% of the input problems, which is close to Smart-fluidnet, but the performance of $M14$ is $3.12\times$ worse than Smart-fluidnet.

Table 2.3 shows the time distribution of five neural network models used by Smart-fluidnet for all input problems. The second row of the table shows the probability of reaching the target when using each neural network model alone, which is predicted by MLP; The third row shows the percentage of execution time of the fluid simulation for the five neural network models (i.e., the time distribution). The table shows that the model with the highest probability, $M7$, takes 50.56% of the total execution time, which is the longest execution time among the five models. We also notice that $M5$, which is the fastest model among the five neural network models, takes 18.1% of the total execution time (the second longest execution time among the five models). These indicate that Smart-fluidnet makes best efforts to reach the user requirement on the simulation quality *and* execution time.

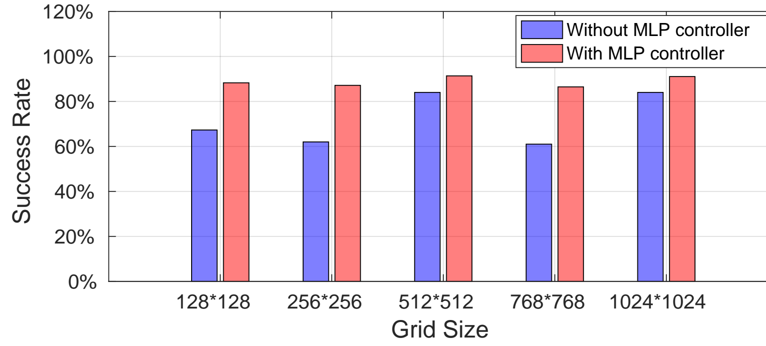

Figure 2.12: Success rate of reaching target quality with or without using MLP.

Table 2.3: Execution time distribution for the five neural network models used by Smart-fluidnet at runtime.

Grid size	$M7$	$M5$	$M10$	$M2$	$M13$
Prob.(MLP)	86.12%	82.16%	79.43%	74.60%	70.38%
Time Distr.	50.56%	18.10%	11.12%	4.07%	16.15%

2.7.3 Evaluation of MLP Effectiveness

In this section, we evaluate the effectiveness of MLP. We compare the *success rate* of our runtime system with and without MLP. The success rate means using all input problems for tests (20,480), how many of them reach the simulation quality requirement with our runtime system. Similar to Section 2.7.1, we use the average quality loss when using all input problems for the fluid simulation with the Tompson's model, as the user requirement. Without MLP, we have 14 neural network models to be used by the runtime before the fluid simulation, while with MLP, we have five.

Without MLP, we use the fastest neural network model (but less accurate) in the beginning and then switch to more accurate models until we find a model that can reach the user requirement on the simulation quality. We use this model in the remaining of the fluid simulation. Figure 2.12 shows the results. The figure reveals that Smart-fluidnet with MLP causes higher success rates than without MLP. The success rate with MLP is 88.86% on average and can be up to 91.36%. This result shows that without MLP, the runtime system can use those neural network models that have a lower possibility to reach the quality target, in order to have better performance. With MLP, we avoid applying those models, hence improving the success rate. We also compare performance with and without MLP. For the grid sizes of 128∗128, 256∗256, 512∗512, 768∗768, and 1024∗1024, the corresponding performance when we use MLP, which is normalized by the performance without MLP, is

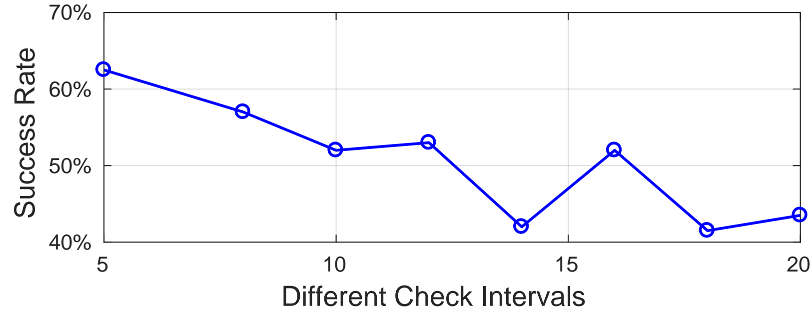

Figure 2.13: Impact of the check interval on the success rate.

97%, 84%, 92%, 79% and 83% respectively. With MLP, we perform better in all cases.

2.7.4 Sensitivity Study: Check Interval

We study the impact of the check interval on execution time and success rate. We change the check interval, and Figure 2.13 shows the results. Figure 2.13 shows that the success rate decreases when the interval increases. Such decrease is because the model switching is too slow to achieve high simulation quality. We also observe an unusual increase of the success rate when the interval changes from 14 to 16. We attribute such an increase to the statistical variance of using the linear regression method to make the prediction. Although using 16 seems to be useful to increase prediction accuracy for our input problems, using 5 achieves the highest success rate.

Therefore, we use 5 as the check interval throughout our evaluation. We do not use a check interval smaller than 5, because we have to skip the first two time steps to ensure that the growth rate of $CumDivNorm$ is stable and using less than three time steps to build the linear regression model cannot give high prediction accuracy.

2.7.5 Evaluation of Resource Usage

We evaluate resource usage (FLOP and GPU memory consumption) by PCG, the Thompson's and Smart-fluidnet with the grid size of 512*512. Since all input problems for such a grid size has the same resource usage, we randomly choose an input problem and present the evaluation results in Table 2.4.

We find that Smart-fluidnet requires much less FLOP than PCG and the Tompson's, which explains why Smart-fluidnet has better performance. On the other hand, Smart-

Table 2.4: Resource usage of different methods.

Methods	FLOP (single step)	GPU Memory
PCG	$\sim$1,250 M	332 MB
Tompson	243.79 M	299 MB
Smart-fluidnet	110.97 M	1,069 MB

fluidnet consumes more memory than PCG and the Tompson's, because Smart-fluidnet uses five neural network models on GPU (but not running them simultaneously). However, the memory consumption of Smart-fluidnet is still smaller than the GPU memory capacity (12 GB). If GPU memory is not sufficient, then we may either use fewer neural network models or run them on CPU.

2.8 Conclusions

Using machine learning (especially neural networks) to approximate computation in HPC applications and improve performance has shown preliminary success recently. However, using this approach faces fundamental limitations due to the lack of model flexibility and generality. In this paper, we focus on a specific HPC application and introduce a systematic approach to address the above limitation. In particular, we introduce a framework (Smart-fluidnet) that automatically uses multiple neural network models at runtime to approximate computation and make best efforts to meet the user requirements on simulation quality and execution time. The framework includes a series of techniques to construct and select neural network models. Based on the comprehensive evaluation, we show that Smart-fluidnet is $1.46\times$ and $590\times$ faster than a state-of-the-art neural network model and the original fluid simulation respectively on an NVIDIA Pascal GPU, while providing better simulation quality than the state-of-the-art model.

Chapter 3

SmartPGsim: Using Neural Network to Accelerate AC-OPF Power Grid Simulation

3.1 Introduction

Artificial Intelligence (AI) and Machine Learning (ML) techniques are revolutionizing the way researchers approach scientific and engineering problems. By employing reverse-engineering and automatic learning methodologies it is often possible to solve complex, unstructured problems with a fraction of the computing power and execution time required by traditional direct and first-principle methods. ML provides researchers with a powerful tool to learn the structure of physical phenomenon directly from Nature, rather than having to explain the causal relationships through direct application of physics law. Many research and engineering fields, from image recognition to autonomous driving, from health to natural language processing (NLP), have experienced a tremendous boost in performance and efficiency over the last few years. Many problems that seemed impossible to be solved, can now be tackled thanks to the use of ML methodologies.

The use of ML methodologies in scientific and engineering applications has been, somehow, limited. By using Neural Network (NN) as a tool to learn and model complicated (non-)linear relationships between input and output data sets, scientists have shown preliminary success in some HPC problems (e.g., detecting neutrinos [129], climate simu-

lations [127], and fluid dynamic simulation [41]). With NN, scientists are able to augment existing scientific simulations by improving simulation accuracy and significantly reducing latency [133, 4, 103, 7, 180, 95, 94]. However, although there have been successful studies of applied ML to scientific applications, these fields have not experienced the double- or triple-digit improvements seen in other domains. The reason for such discrepancy is the fundamentally different characteristic of scientific and engineering applications compared to domains such as image recognition and NLP: scientific applications require a level of precision and robustness that may not be provided by most of the current ML methods employed in other domains.

In this work we study the implication of using ML techniques to accelerate the power-grid simulations, the structure of the ML model to be used, the relative importance of the features selected, and, most importantly, the impact of incorporating physics constraints on the performance of the application. The power-grid simulation [91] is a complex nonlinear optimization problem for the management of power flow and is critical to the power industry in electricity dispatch scheduling, reliability analysis, and maintenance planning for power and generators [50, 19]. The alternating current optimal power flow (AC-OPF) simulation is the most fundamental and time-consuming part of the power grid simulation. The problem size of AC-OPF is generally large, in which the scale of the generator node can vary from 10^3 to 10^6 [66, 59, 111]. Despite the large problem size, the AC-OPF simulation requires near real-time updates during power scheduling. In a typical scenario, power grid operators repeatedly solve the optimal power flow problem multiple times within every minute throughout a day, every day of the year [143, 124, 38], for decades, to ensure that the power grid system is operating reliably and safely. The high requirement on the simulation latency and frequent usage of the simulation make the power grid simulation a mission-critical application under active development in the HPC community and within the U.S. Department of Energy (DOE) and the DOE Exascale Computing Project (ECP).

NN has been applied to solve the optimal power flow problem in the past [57, 6, 183, 35, 114]. However, existing efforts have focused on improving performance by entirely replacing the simulation solver with an approximated NN model or facilitating existing solvers by identifying active constraints. While these approaches provide considerable speedups, NN provides only an *approximation* of the optimal solution or approximates computation in the simulation. As a result, these approaches may not provide the desired

precision for the solution or may provide a non-optimal solution. In the context of power-grid simulation, the first case results in an infeasible solution (e.g., not being able to provide the required power to satisfy the user demand) while the second case may results in a large economic loss (i.e., solving the problem at a much higher cost.)

In this paper, we introduce a new method to apply NN to the AC-OPF simulation. Unlike the existing studies, we employ NN to generate an initial solution and then inject it to the AC-OPF solver. Because of the high quality of the initial solution and guidance of other outputs generated by the proposed NN, the simulation can run faster (or converge faster) without losing the solution optimality.

There are several challenges to apply our method to the AC-OPF simulation. First, deciding which variables in the AC-OPF simulation should be used as NN output and quantifying the sensitivity of simulation execution time and convergence to those variables is a challenge. The AC-OPF simulation involves a set of variables, including power grid information and multiple variables critical for the computation convergence. We cannot use all of them as NN output because that largely increases network complexity and puts high requirements on training efficiency and sufficiency of training samples. On the other hand, using only the solution of the AC-OPF as NN output, we often lose simulation robustness because of the limited guidance for the simulation from the initial solution. Furthermore, understanding the sensitivity of simulation time and convergence to those variables is useful for deciding NN topology and generating high-quality initial solutions.

Second, how to apply NN to the AC-OPF simulation without disturbing the simulation robustness is a challenge. Due to the non-convex and nonlinear nature of the AC-OPF problem, the simulation process itself is at the risk of a failed convergence with the use of iterative numerical methods. The simulation must be robust enough to handle various power flow cases with computation convergence. Using NN to generate an initial solution, we must make sure that the initial solution makes sense and does not impact the computation convergence in the original simulation.

Third, how to impose physical constraints on NN to ensure the validness of NN prediction. Traditionally, the NN model is manually constructed by computer scientists as a black box with limited or no domain knowledge and without considering domain requirements. Although NN models can be adjusted as a nonlinear tool box to accommodate a change of inputs and generate some approximation, the understanding of the model is lost. Instead

of blindly trusting that the data mining algorithm will produce a correct model, we seek for what variables physically mean and which physical laws are driving the interpretable evolution of the analysis paradigm.

To address the above challenges, we introduce, Smart-PGsim, a framework that facilitates the construction of a NN model to accelerate the AC-OPF simulation. Smart-PGsim is based on the following design principles. First, it generates an NN model that uses power grid components as inputs and variables critical for the simulation convergence as the model output. By using Smart-PGsim, we perform a sensitivity study to understand the impact of the output accuracy on execution time and convergence, by using precise or imprecise data for some variables. This sensitivity study provides guidance on choosing a correct and efficient NN topology.

Second, Smart-PGsim uses a novel multitask-learning NN model to accelerate the AC-OPF simulation. The model topology allows information sharing when predicting multiple dependent variables while including customized layers for each variable. This multi-task model improves the model accuracy, compared with the traditional single-task model, while simplifying the training process.

Third, Smart-PGsim allows embedding physical constraints from the original formulation of the AC-OPF problem into the NN model and imposes those constraints into the training objective function or the last layer based on transformation on equality and inequality in the constraints. We summarize our major constitutions as follows.

- A systematic approach and a framework (Smart-PGsim) to accelerate optimization problems in general and the AC-OPF power grid simulation in particular;

- A set of techniques to construct NN models for robust, accurate, and high-performance numerical solvers;

- We show that Smart-PGsim achieves $2.60\times$ speedup on average (with the consideration of NN cost) and up to $3.28\times$ over the original AC-OPF simulation method (computed over 10,000 problems as the simulation input), without losing the optimality of the final solution.

3.2 Background

In this section, we review the problem formulation and the primal-dual interior-point method in the AC-OPF problem.

3.2.1 Problem Formulation for AC-Optimal Power Flow

The AC-OPF problem aims at minimizing an objective function by optimizing the power dispatch and transmission decisions. The objective function calculates the cost of power generation, subjecting to physical, operational, and technical constraints including Kirchhoff's laws, operating limits of generators, voltage levels, and loading limits of transmission lines [74]. The standard AC-OPF problem is formulated as:

$$\min_X f(X) \tag{3.1a}$$

$$s.t.\ G(X) = 0 \tag{3.1b}$$

$$H(X) > 0 \tag{3.1c}$$

$$X_{min} \leq X \leq X_{max}. \tag{3.1d}$$

where $f(X)$ is the cost function to be minimized, and X is an optimization vector as the simulation solution.

Eqn. 3.1b builds an equality constraint, which sets up power balance incorporating variable bounds. The formulation 3.1c is an inequality constraint that sets up branch flow limits. The optimization vector X is bounded by X_{min} and X_{max} which introduces the constrains on reference bus angles, voltage magnitudes, and generator injections. The optimization vector X consists of four variables, $X = \{V_a; V_m; P_g; Q_g\}$, i.e., voltage angles V_a, voltage magnitudes V_m, generator real power injections P_g and reactive power injections Q_g.

In power grid simulation, $G(X) = 0$ is an AC nodal power balance equation and enables the AC-steady conditions of the power system, which can be split into real and reactive parts:

$$P_i(C_g, P_g) = P_d + P_{bus}(Y_{bus}, V_a, V_m) \tag{3.2a}$$

$$Q_i(C_g, Q_g) = Q_d + Q_{bus}(Y_{bus}, V_a, V_m). \tag{3.2b}$$

In Eqn. 3.2, C_g is the generator connection matrix reflecting generator locations in a power grid network. Y_{bus} is the bus admittance matrix including all constant impedance elements. P_i and Q_i refer to power real and reactive injection for the power system. P_d and Q_d are power loads. P_{bus} and Q_{bus} are power consumption of transmission lines.

3.2.2 Primal-dual Interior Point Solver

The primal-dual interior point method [107, 171] is an efficient algorithm to solve the non-convex optimization problem for AC-OPF. Matpower [185] is a widely used framework for solving power flow and optimal power flow problems. Matpower uses a solver, called MIPS, to solve those problems.

To solve the AC-OPF problem, MIPS first converts the inequality constraint in Eqn. 3.1c into an equality constraint with a vector Z, $H(X) + Z = 0$ where Z is a vector of positive slack variables. MIPS further uses a barrier function $\ln(Z)$ to bound Z. Based on that, MIPS uses a Lagrangian formulation to formulate the AC-OPF problem as follows.

$$L^{\gamma}(X, Z, \lambda, \mu) = f(X) + \lambda^{\intercal} G(X) + \mu^{\intercal}(H(X) + Z) - \gamma \sum_{m=1}^{n_i} \ln(Z_m) \tag{3.3}$$

where λ is called the equality Lagrangian multiplier, μ is called the inequality Lagrangian multiplier, and γ is called the perturbation parameter. During the solving process, γ is approaching zero. If $\gamma = 0$, the solution to this Lagrangian formulation equals to that of the original form (Eqn. 1).

Matpower uses Newton method to solve Eqn. 3.3, which iteratively converges to a set of convergence criteria (particularly four terminate conditions) [185]. The Newton Method is computationally intensive and requires constant updates of input and output variables: the method firstly updates X and λ, then Z based on X, and, finally, μ based on X and Z. As we will see in the next Sections, this structure introduces internal dependencies on the variables that our model exploits for better performance and accuracy.

3.3 Related Work

OPF problems can be categorized into three forms: economic dispatch (ED) [30], Direct Current (DC-OPF) [160], and Alternating Current (AC-OPF) problems [29]. The AC-OPF problem is the original OPF problem, which is non-convex and the most challenging one among the three. ED and DC-OPF problems are the relaxed version of the AC-OPF problem, which is obtained by removing or linearizing some constraints in the AC-OPF problem, respectively. Traditionally, numerical iteration algorithms are used to solve the OPF problem [156, 97, 67, 22, 100]. However, the time complexity of these algorithms might be significant, especially when the scale of the transmission power system becomes large. To deal with this limitation, researchers have explored learning-based approaches to accelerate solving OPF problems.

Vaccaro et al. [168] use the principal component analysis (PCA) to identify unknown relationships among OPF variables, which reduces the number of variables to be solved for a solution. Ng et al. [114] use a statistical learning-based approach to set up a mapping between input power requirement and output dispatch scheme. However, the approaches mentioned above consider only the prediction accuracy without taking into account the correlation among OPF problem variables, which leads to a solution that can not satisfy all of the problem constraints. Pan et al. [117] use the multilayer perceptron (MLP) to learn the mapping between input and decisions for DC-OPF and apply it to obtain optimized operating decisions upon arbitrary inputs. While this approach is effective for DC-OPF, it has low generalization capacity and cannot be applied to a non-convex problem such as AC-OPF. Previous works [57, 6, 183] have leveraged machine learning to accelerate the AC problem. Zamzam et al. [183] develop an online method based on machine learning to obtain feasible solutions to the AC problem by loading the optimal generator set-points and enforcing generation limits. However, the AC grid contains more voltage phase angles beyond magnitudes and reactive parts of power generation. Unlike these methods, the proposed approach includes all the inputs of the AC problem and guarantees that the predicted solution is optimal while providing significant performance improvement.

In this work, we use NN models to solve the AC-OPF problem. We follow a radically different approach compared to previous work in that we employ ML to estimate a high-quality initial solution for the solver, greatly speeding up the entire computation, and then leverage traditional AC-OPF solver to guarantee precision and robustness of the so-

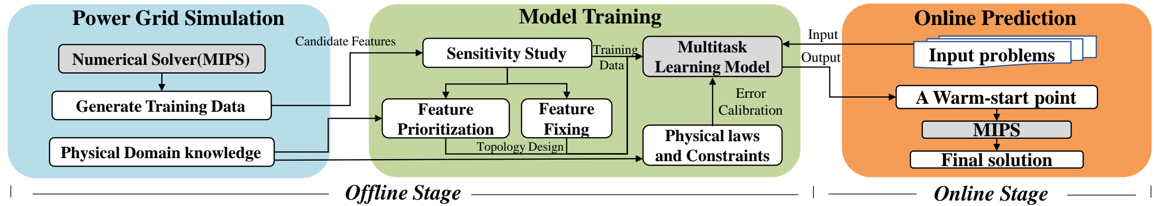

Figure 3.1: Workflow of the proposed Smart-PGsim

lution. We show in the next Sections that our approach can simultaneously provide large performance improvement *and* high-precision solutions.

3.4 Overview

This section overviews our proposed framework "Smart-PGsim". The Smart-PGsim framework includes two phases: offline and online phases. Figure 3.1 shows the workflow of Smart-PGsim.

The offline phase investigates the power grid simulation to find the most crucial features to construct an efficient NN model for online acceleration. In particular, our sensitivity study (Section 3.5) firstly identifies the most important features (in other words, determining variables in MIPS as the output of the prediction model) and quantifies the impact of the imprecise variables (i.e., variables with some accuracy loss) on the success rate of simulation and performance in terms of execution time. The results in sensitivity study are used to guide the NN topology design.

Then, Smart-PGsim constructs a multi-task learning (MTL) model (Section 3.6) guided by the sensitivity study. The model shares domain information between prediction tasks, while uses a customized topology design for each task. Smart-PGsim prioritizes features to distinguish main tasks and auxiliary tasks and applies a physics-dependent hierarchy for those features have domain specific dependency.

Next, Smart-PGsim incorporates physical domain knowledge during model training to improve prediction quality (Section 3.7). The domain knowledge presents physical constraints providing explicit and implicit error bounds. Using the domain knowledge improves prediction accuracy, interpretability, and defensibility of the MTL model, while simultaneously augmenting physical data as complementary.

After the above offline phase, the well-trained MTL model can be used to generate a

Table 3.1: **Ablation study on the input signals**

	X	λ	μ	Z	bus 5		bus 9		bus 14		bus 30		bus 39		bus 57		bus 118		bus 300		Observation
					SR	SU	SR	SU	SR	SU	SR	SU	SR	SU	SR	SU	SR	SU	SR	SU	
I	0	0	0	0	100	1	100	1	100	1	100	1	100	1	100	1	100	1	100	1	baseline
II	0	0	0	1	0	–	10	0.67	0	–	89	0.66	0	–	6	0.99	0	–	0	–	
III	0	0	1	0	100	1.04	100	0.73	100	0.92	100	0.93	95	0.95	88	0.60	99	1.03	100	1.24	
IV	0	0	1	1	100	1.09	100	0.96	100	0.85	3	0.22	75	1.02	99	0.72	0	–	68	1.24	
V	0	1	0	0	100	0.98	100	0.99	100	1.00	30	1.06	100	1.00	100	0.98	100	0.98	98	0.99	OBS 3
VI	0	1	0	1	0	–	8	0.73	0	–	79	0.70	0	0.61	9	0.90	0	–	0	–	
VII	0	1	1	0	100	0.99	0	–	80	0.61	98	0.90	100	1.09	100	0.89	100	0.93	100	1.08	
VIII	0	1	1	1	100	1.05	100	1.24	100	1.32	30	0.19	100	1.39	100	1.25	100	1.59	100	1.75	OBS 3
IX	1	0	0	0	100	1.17	100	0.99	100	0.99	100	0.95	100	1.06	100	1.01	100	0.99	100	1.08	OBS 1, 4
X	1	0	0	1	0	–	0	–	0	–	0	–	0	–	11	0.94	0	–	0	–	OBS 2, 4
XI	1	0	1	0	100	1.28	100	0.85	100	0.88	100	1.33	95	1.48	100	0.67	90	0.84	96	1.24	OBS 4
XII	1	0	1	1	100	1.45	100	1.03	100	0.80	95	1.26	80	1.22	100	0.65	0	–	53	0.87	OBS 2
XIII	1	1	0	0	100	1.18	100	1.00	100	0.98	100	0.94	100	1.06	100	1.00	100	0.99	100	1.07	OBS 3, 4
XIV	1	1	0	1	0	–	0	–	0	–	0	–	0	–	10	0.97	0	–	100	1.18	
XV	1	1	1	0	100	1.28	100	0.79	100	0.90	100	1.09	100	1.22	100	0.93	100	0.94	100	1.32	
XVI	1	1	1	1	100	5.21	100	4.58	100	3.74	100	6.15	100	6.60	100	4.58	100	7.63	100	14.6	OBS 1, 3

warm-start point for MIPS as online prediction. The MIPS (or other numerical solvers) can use these high-quality start points for quick convergence.

3.5 Sensitivity Study

The ability of NN to produce high-quality results is the key to improve simulation performance (making the simulation quickly converged). In this section, we discuss the opportunity available in NN with the assist of a sensitivity study tool that detects and analyzes those variables critical to simulation convergence and execution time.

We introduce two data types, i.e., *imprecise default data* and *precise simulation data*, to study the impact of noisy feature to simulation quality and execution time. By doing so, we can check the (lowest) highest performance brought by these (im-)precise data, which demystifies the contribution of each feature to success rate and speedup.

1. Imprecise default data: The default value at the initial point in MIPS.
2. Precise simulation data: The exact solution collected in the numerical solver, i.e., MIPS. We take the ground-truth value as the precise data.

Our sensitivity study first checks the convergence criteria in the MIPS code and collects those variables critical to the simulation converge, namely X, λ, μ, and Z. We use these (im-)precise data as initial points to test the importance of each variable in two aspects, i.e., the impact on success rate and speedup. Here, *success rate* refers to the ratio of those

initial solution can reach the convergence criteria to the total number of input problems. *Speedup* is time acceleration, namely the rate of actual solving time to the exact solving time in MIPS.

Then, we include eight test systems[1] and generate $10,000$ samples for each system by varying input loads to analysis the impact of using different initial points. Initializing the four variables with precise and imprecise types, we have 2^4 combinations to analyze the contribution of each variable. Table 3.1 shows the results of 16 combinations. For each combination, we use "0" and "1" to indicate the imprecise data and precise data respectively. To calculate the success rate and speedup, we take a baseline, the combination where all variables using imprecise default data.

Table 3.1 reveals that precision improvement on these initial variables does not always bring benefits to simulation performance in terms of success rate and speedup. A high improvement on precision might even decrease the success rate of the simulation. For example, the baseline case I (using all default parameters) has a success rate of 100%, while improving the precision on the feature Z (e.g., the case II) alone leads to failed convergence at most input problems. Hence, blindly building a NN model to numerically approach those precise values may reduce success rate and lose simulation performance.

We further analyze the performance results in Table 3.1 and summarize some interesting observations.

- **Observation 1**: Using precise X leads to a 100% success rate (see case IX), while the features X, λ, μ and Z jointly contribute to high speedup see case XVI).

- **Observation 2**: The contribution of Z to the success rate and speedup strongly depends on whether μ use precise data. For example, the success rate is dropped down when involve a precise Z without a precise μ (see case XII with respect to X).

- **Observation 3**: The contribution of λ to the success rate and speedup is independent of whether the other features use precise data or not. For example, the success rate of initialing with a precise λ does not change with/without a precise X or μ and Z (see case V, VIII, XIII and XVI).

- **Observation 4**: Features X, λ, μ and Z have implicit dependency. Improving accuracy of one feature cannot guarantee the overall performance improvement to the

[1]Refer to Table 3.2 for a more detailed illustration of test systems.

success rate and speedup. For example, improving the accuracy of λ, μ, or Z on a precise X can not guarantee the improvement of success rate and speedup (comparing case IX with cases X, XI and XIII.)

Observations 1 and 4 indicates that the analyzed features are highly inter-dependent, which can be targeted on multi-task prediction. We build a MTL model to enable the information sharing for the inter-dependency. We decide feature priority by *dependency between features*, namely, the contribution of a feature to success rate and speedup is changed with a variation of another feature. For example, since λ, μ and Z have dependency on X, we should make X very accurate. We give X the highest priority.

Observation 2 and 3 reveals features show differences on dependency. Some features (e.g., λ) are relatively independent while others (e.g., μ and Z) have dependency, which implies customized design for different feature prediction should be considered in modeling. Also, we observe that λ is an equality factor while μ and Z contribute to inequality together in Eqn 3.3. Such information from physical law validates feature dependency and maybe can be used to deal with the dependency in model training inversely.

Driven by these observations, we introduce an interactive learning model for multi-objective modeling (discussed in Section 3.6) and impose domain knowledge to strength physical understanding for prediction quality (discussed in Section 3.7).

3.6 An interactive learning model

In this section, we develop a MTL model to enable multitask prediction. Based on the observation from sensitivity study, we implement domain specific design through prioritizing features and enforcing a physics-dependent hierarchy in the MTL model. After that, we depict the details of MTL parameters.

3.6.1 Multitask Learning

Multitask learning is an inductive transfer learning method [17, 166]. A MTL model is typically composed of shared layers and task-specific layers. Unlike using multiple separate models for each task, the MTL model enables us to share information from common layers while customizing specific layers for corresponding tasks. The training signals of

different tasks can be learned as inductive biases to facilitate the learning of all tasks, which achieves a unification of the shared information and the task-specific information.

Information-sharing in shallow layers. Observation 1 reveals that there is correlation among these four features and we would like to leverage these relations in our model. To incorporate this correlation, we utilize information sharing in MTL by parameter sharing and loss sharing.

- **Parameter sharing.** The common layers share the same weights and topology between tasks. By sharing parameters, tasks share low-level semantic information to complement domain knowledge with each other. Meanwhile, parameter sharing can alleviate the risks of overfitting due to the noise brought by multiple tasks.

- **Loss sharing.** The tasks to predict X, λ, μ, Z share a common loss function to update training gradient in the MTL. The minimal loss of different tasks are usually in different positions. By sharing losses, the MTL passes information and avoids being trapped in a local optimal.

Task-specific learning in deeper layers. Besides the features X, λ, μ, Z being correlated, observations 2 and 3 shows that specific design for different feature prediction should be considered. In task-specific layers, we introduce specific model topologies (estimators) for each task based on the task demands. For example, a task requires a positive output. We apply a rectified linear unit (ReLU), a type of activation functions, at the last layer to bound the output always positive.

Figure 3.2 shows the topology of the proposed MTL. Given a power network topology, we use the power load (including both the active part P_d and reactive part Q_d) as model inputs and estimate seven tasks (four variables in X). In the MTL, the shared layers are extracting information from different tasks, while the task-specific layers (estimators) utilize customized designs to generate their own dedicated results.

3.6.2 Domain-Specific Design

Besides using shared layers and task-specific layers, we introduce a domain-specific design into MTL: this design is driven by our observations on (1) the contribution difference of the four features to success rate and speedup and (2) the dependency between features.

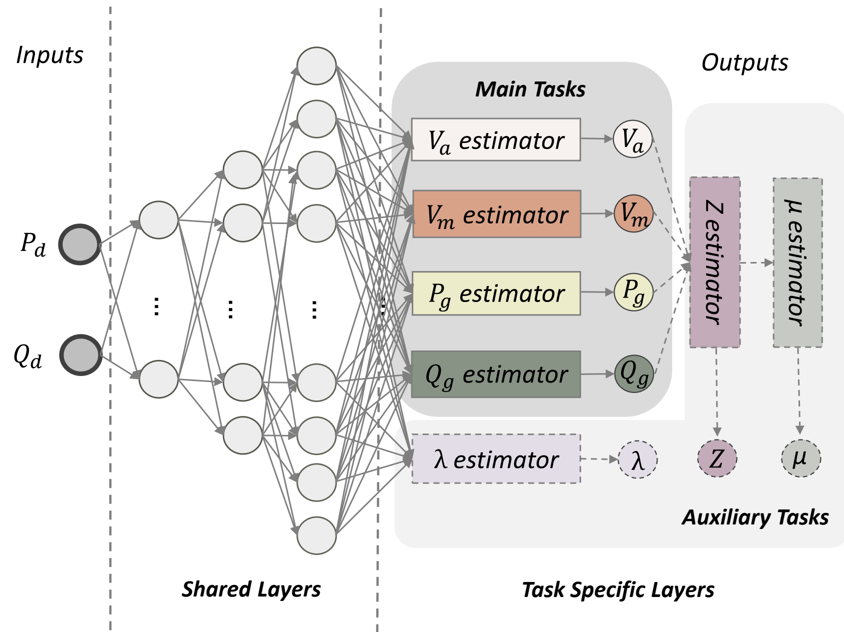

Figure 3.2: Topology of the MTL.

The domain-specific design includes two techniques, feature prioritization and a physics-dependent hierarchy, discussed as follows.

Feature prioritization. Observation 1 shows that precise X guarantees the success of simulation convergence, while precise λ, μ, and Z contribute to simulation acceleration. We prioritize the four features by specifying the prediction of X as the main task while the prediction of other three features as auxiliary tasks. The auxiliary tasks are used as an augmentation to provide additional information for the main task. Through "eavesdropping" the main tasks, the auxiliary tasks interact with the main task implicitly. More importantly, learning the direct solution X in the main task gives the user high simulation quality while estimating the Lagrangian factors (λ, μ, and Z) in the auxiliary tasks maximize performance speedup. Technically, we apply "detach()" operation [120] for these auxiliary tasks periodically. The detach operation blocks the gradient back-propagation to the shared layers (which are contribute to the main task X). In other word, we set a knob of detach operation to alternately train the main task or the entire model. In particular, the MTL focuses on improving main tasks when we activate the detach operation, while facilitates the interaction between main tasks and auxiliary tasks when the detach operation is disabled.

A physics-dependent hierarchy. Observations 2 and 3 reveal the dependence between Z and μ and the independence of λ, respectively. We find these observations are

consistent with the computation order in the solving process. In particular, the simulation process takes the order of (1) computing X and λ; (2) computing Z based on X; and (3) computing μ based on X and Z. This is consistent with the existing work [147, 148].

To fully exploit the benefit of information sharing, we enforce a physics-dependent hierarchy in MTL. As shown in Figure 3.2, we first infer the main task $X(V_a, V_m, P_g, Q_g)$ and an independent auxiliary task λ. Then, we predict task Z based on X. After that, we estimate μ based on the predicted Z.

3.6.3 Details on Multitask Learning Model

In this section, we present details about the topology parameters, the loss function and the pre-processing method of the MTL model. We use a power grid system of 300 buses as an example, but the MTL modeling method is general for any other power grid systems. Figure 3.2 generally depicts the model topology.

The shared layers take the power load P_d and Q_d at each bus as input, totaling 600 inputs. There are five fully-connected layers as the shared layers. We set the numbers of neurons in the five layers as 600, 720, 840, 960, and 1080 respectively. The five fully-connected layers extract shared features and feed them to seven specific estimators (four in X and λ, μ, Z), each of which is a fully-connected network customized for a task. We use ReLU as the activation function to increase the model nonlinearity. We use a variant of L_1 loss [69], the Charbonnier loss, as our loss function. This is a supervised loss function calculating the difference between each of the predicted output variables v and the corresponding ground-truth value v_{gt} collected in the MIPS solver. Our loss function is defined as follows.

$$L = \frac{1}{|\mathcal{V}|} \sum_{v \in \mathcal{V}} W_v \sqrt{(v - v_{gt})^2 + \epsilon^2} \tag{3.4}$$

where $\mathcal{V}$ is a set consisting of V_a, V_m, P_g, Q_g, Z, λ and μ; W_v is a weight for a task v and ϵ is a small constant for numerical stability. We set ϵ as $1e-9$ in our study.

3.7 Physics-Informed Learning

The solution in power grid simulation must respect several physical constraints, such as power generation, line flow, and bus voltage constraints. Incorporating these constraints

into the MTL model not only improves model accuracy but also increases the model interpretability.

In general, the constraints are classified into hard and soft constraints. The *hard constraint* includes some strict bounds on the variable ranges in applications; The *soft constraint* includes domain knowledge to improve model accuracy, such as physical principles, conservation laws, and others gained from theoretical or computational studies. We introduce four objective functions to incorporate domain knowledge and impose those constraints by minimizing the objective functions.

3.7.1 Embedding AC Nodal Power Balance Equations

The power grid simulation includes a power flow equality constraint shown in Eqn. 3.2 to make the simulation of the power grid system stable and make the simulated solution feasible. We integrate the AC nodal power balance equations (Eqn. 3.2) into the objective function f_{AC} to guide model training.

$$\begin{aligned} f_{AC} = &|P_d + P_{bus}(Y_{bus}, V_a, V_m) - P_i(C_g, P_g)| + \\ &|Q_d + Q_{bus}(Y_{bus}, V_a, V_m) - Q_i(C_g, Q_g)| \end{aligned} \tag{3.5}$$

The above objective function bridges model inputs (P_d, Q_d), outputs (X, λ, μ, Z) and the physics information (C_g, Y_{bus}) of power networks to yield quantitatively better physical connection. In particular, the generator connection matrix C_g and the bus admittance matrix Y_{bus} are critical information determined by the physical network of power system. Eqn. 3.2 shows the AC power system keeps stable only if the power generation equals to the power load. In the objective function f_{AC}, we calculate the differences between power load and generation and minimize the difference approaching to zero.

Figure 3.3 shows how the objective function f_{AC} works in MTL training. In the power grid simulation, we utilize domain information C_g and Y_{bus}, which provide power-grid bus topology and resistance information respectively. Power loads (P_d and Q_d) are the input fed to the MTL to produce solutions V_a, V_m, P_g, Q_g. We integrate the AC power balance law (Eqn. 3.5) to calibrate the training loss in f_{MTL}. In particular, we calculate the power generation based on the prediction solution X and domain information C_g and Y_{bus}, and subtract power generation from the power loads. We then calculate the difference

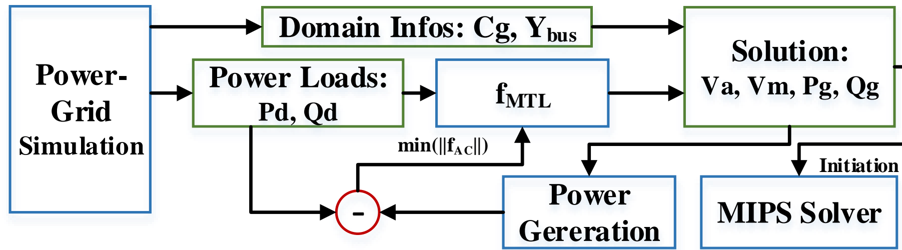

Figure 3.3: Embedding AC physical laws in MTL training

between the power loads and power generation, and try to minimize the difference within the objective function f_{AC}. The only block with training parameters is the f_{MTL} and all blocks are differentiable.

The above training process is driven by the predicted data and facilitates the prediction inversely. Such a data-driven architecture f_{AC} mitigates the risk of obtaining infeasible solutions, such as those predicted solution misled by the noise of training data and violating the basic AC power balance law. Embedding the objective function f_{AC} has two significant benefits. First, since the information of model inputs is limited to predict its outputs, we incorporate non-trivial data-augmentation as a complementary to increase prediction accuracy. Second, we can efficiently perform transfer learning with fewer training data even if the typology of power network is modified, e.g., a transmission line in the power-grid bus is suddenly broken. With this, we can improve our MTL prediction and facilitate the solution robustness.

3.7.2 Guarding Inequality Constraints

The AC-OPF formulation includes two inequality constraints: one is explicit, quantitatively bounding X by $X_{min} < X < X_{max}$ (Eqn. 3.1d), and the other is implicit, limiting branch flow by $H(X) > 0$ (Eqn.3.1c). For the implicit inequality, we impose physics information C_g and Y_{bus} to calculate the branch flow state $H(X)$ and check if the $H(X)$ violates the bounds. We utilize exponential functions to punish the overflow error in these inequality constraints and force the prediction to be bounded by the expected, normalized range. Eqn. 3.6 shows how we use the exponential functions. In the equation, we build an objective function f_{ieq} to incorporate the inequality equations as a penalty loss.

$$f_{ieq} = e^{-H(X)} + e^{(X-X_{max})} + e^{(X_{min}-X)} \tag{3.6}$$

Where X is a predicted feature in MTL. Once the predicted X violates the inequality constraints, for example, $H(X_k) < 0$, the overflow error will be visibly shown up in the objective function f_{ieq} and calibrated through backpropagation in the training phase. Hence, the main task X is restricted in a quantitative way to improve simulation quality.

Guarding inequality constraints in our model mitigates the overflow error in inequality constraints while facilitates the feasibility of model prediction.

3.7.3 Optimization of Cost Function

The ultimate goal of the AC-OPF is to minimize the cost function $f(X)$ (Eqn. 3.1a). We explore the physics information in $f(X)$ to construct an objective function $f_{f(X)}$ and minimize the loss between the predicted cost and ground truth cost.

$$f_{f(X)} = |f(X) - f_0| \tag{3.7}$$

where f_0 is the ground-truth value of the cost collected by the numerical solver, MIPS. Feature X is our model prediction. In $f(X)$, we utilize the characteristics of energy consumption on generators to calculate the predicted cost $f(X)$. Then, the objective function $f_{f(X)}$ calibrates the predicted cost $f(X)$ with the ground-truth cost f_0 to reach the optimal solution.

3.7.4 Implying Lagrangian Conservation

In Eqn. 3.3, the AC-OPF problem can be solved as the equality constraints $G(x) = 0$ and slacked inequality constraints $H(x) + Z = 0$ approach zero. Here, we apply two ways to imply the Lagrangian formulation into MTL training. First, we reconstruct the inequality and equality constraints as soft constraints, which is imposed in the loss function to guide the training. Then, we refer the variable bounds $Z > 0$ and $\mu > 0$ as hard constraints and apply an activation function to strictly bound model prediction. We construct an objective function f_{Lag} to guide the training subject to the soft constraints.

$$f_{Lag} = |\lambda^{\intercal} G(X)| + |\mu^{\intercal}(H(X) + Z)| \tag{3.8}$$

We incorporate the hard constraints during the training phrase by projecting predic-

tions onto a region induced by the constraints. In particular, we first pre-process the raw data of ground truth into the normalized range $[0, 1]$. Then, we apply a "sigmoid" activation function at the last layer to bound the output range of Z and μ to be positive and into the same range $[0, 1]$. The above techniques provide hard upper and lower bounds on prediction and guarantee its feasibility.

Incorporating f_{AC} and f_{ieq} improves the feasibility and robustness of the simulation solution X; Incorporating $f_f(X)$ improves the accuracy of X; Incorporating f_{Lag} can optimize auxiliary tasks λ, μ and Z. Hence, we arithmetically compose these objective functions into the loss function L(Eqn. 3.4).

The L_{total} efficiently combines supervised learning (with ground-truth labels) and unsupervised learning (without ground-truth labels) to guide the MTL training. By doing so, we maintain the prediction accuracy while increase the model feasibility and interpretability.

3.8 Evaluation

We evaluate our framework by examining its impacts on performance and simulation quality of power grid simulation.

Platform.

We conduct all experiments on an NVIDIA DGX-1 cluster with 16 nodes, and each node is equipped with two Intel Xeon E5-2698 v4 CPUs (40 cores running at 2.20GHz) and 8 NVIDIA TESLA V100 (Volta) GPUs. We use CUDA 10.1/cuDNN 7.0 [21] to run NNs on NVIDIA GPUs. We use Pytorch for model training and inference.

Matpower. Matpower 6.0 is an open-source Matlab power system simulation package [185], which is used widely in research and education for AC- and DC- power flow simulations. The default OPF solver, i.e., Matlab Interior Point Solver (MIPS), is a high-performance primal-dual interior-point solver.

Load Sampling.

We sample the loads within $[(1-t) \times P_{di}, (1+t) \times P_{di}]$ uniformly at random, where P_{di} is the default power load at the i-th bus, and t is the variation percentage, i.e., 10% in this paper, consistent with state-of-the-art [117, 57, 183].

Input Datasets. To comprehensively evaluate the performance, we use five power net-

Table 3.2: Configurations in IEEE bus systems.

Problem size	14-bus	30-bus	57-bus	118-bus	300-bus
Buses	14	30	57	118	300
Generators	5	6	7	54	69
Branches	20	41	80	185	411
#λ	29	61	115	237	601
#$\mu(Z)$	48	166	142	452	876

works in Table 3.2 as test systems. We generate 10,000 input problems for each test system, in which 8,000 of them are for training and 2,000 for validation. These samples are fed into Matpower to produce the optimal solutions as the supervision ground truth signal. Uniform sampling is applied to avoid over-fitting issues common in generic DNN approaches [54].

3.8.1 Smart-PGsim Performance Evaluation

In our approach, we use Smart-PGsim to generate a high-quality initial condition for the MIPS solver, thereby drastically reducing the overall time-to-solution. We introduce the following performance metric to calculate the achieved speedups by Smart-PGsim:

$$SU = \frac{T_{MIPS}}{T_{MTL} + T'_{MIPS} + T_{MIPS} \times (1 - SR)} \tag{3.9}$$

where T_{MIPS} represents the solving time when using the traditional approach with MIPS, T_{MTL} represents the inference time of the MTL model, and T'_{MIPS} represents the convergence time in MIPS initializing with the output of Smart-PGsim. $T_{MIPS} \times (1 - SR)$ calculates the restart execution time with the default initial point in MIPS if the simulation fails. SR represents the overall success rate, $SR = N_{suc}/N_{total}$, where N_{suc} represents the number of problems successfully solved by MTL and N_{total} represents the total number of input problems. Whenever the initial condition provided by Smart-PGsim does not lead to the simulation converge successfully, we fall back to the traditional MIPS solver to guarantee the final convergence. Hence our method always provides 100% guarantee on simulation convergence, though it might come at an additional cost of re-executing overhead in the workflow.

Figure 3.4(a) compares the execution time of the traditional numerical simulation performed with MIPS and that of our framework in terms of the SU metric described above.

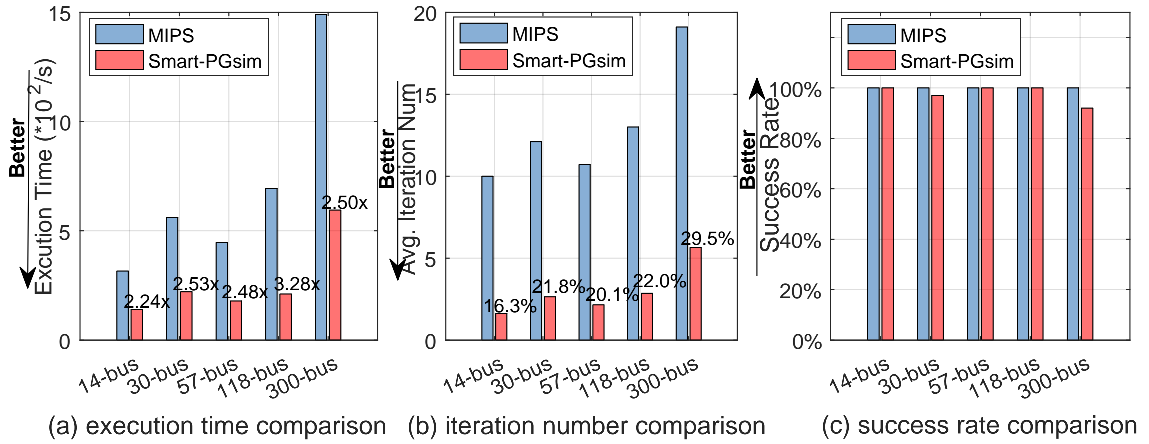

Figure 3.4: Comparison of three aspects between MIPS and Smart-PGsim.

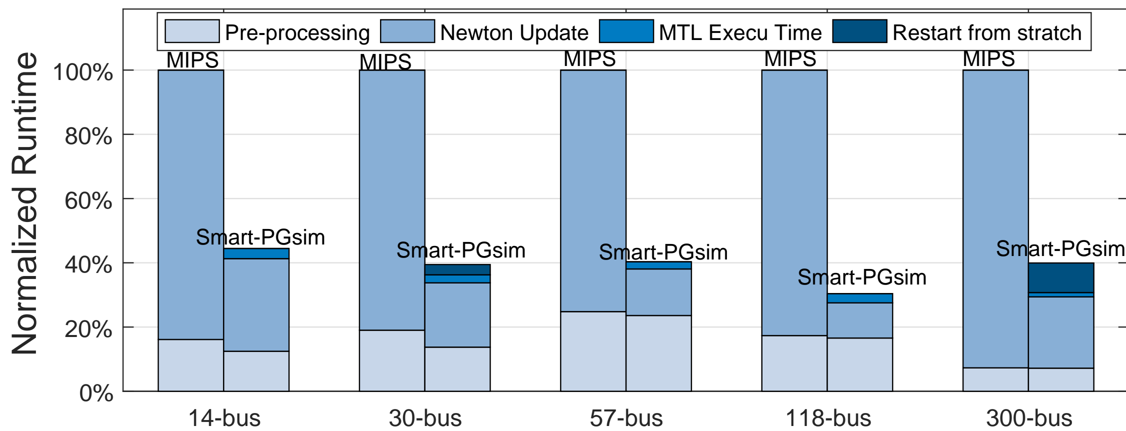

Figure 3.5: Execution time breakdown

Each test system is run on $2,000$ input problems. Performance measurements of Smart-PGsim comprise the end-to-end runtime, including the time to produce the warm-start points in MTL, the convergence time in MIPS with the warm-start points, and the restart execution time in MIPS if the simulation fails. In the Figure 3.4(a), we also label the speedup at the top of Smart-PGsim bar. The Smart-PGsim speedups over MIPS observed in the plot are considerable, ranging from nearly a $2.24\times$ speedup up to over a $3.28\times$ speedup. Furthermore, the performance benefit of Smart-PGsim is more evident as the size of power networks increases, which indicates a notable potential in accelerating large-scale power grid systems. It is important to note that using Smart-PGsim as warm-start for MIPS generates the same solution as produced by MIPS directly. Figure 3.4(b) presents the average iteration number of MIPS and Smart-PGsim across different test systems, in which we only consider the iteration acceleration produced by Smart-PGsim. We measured the

average iteration number during the convergence process until the terminate criteria are reached. The iterative process is the most computationally intensive part of the power grid simulation. We also label the ratio of the Smart-PGsim iteration number to the MIPS iteration number on Smart-PGsim bar. The results in Figure 3.4(b) show that Smart-PGsim dramatically reduces the number of iterations required to converge, taking only 16.3% to 29.5% iterations of previous conduction (MIPS). The accelerated convergence drives the overall performance improvement of Smart-PGsim.

3.8.2 Performance Breakdown

To further explore the performance improvement provided by Smart-PGsim, Figure 3.5 shows the runtime breakdown of MIPS and Smart-PGsim, normalized to the overall runtime where the problems run with MIPS. The pre-processing refers the execution time of problem construction and data preparing for power grid simulation, in which MIPS and Smart-PGsim show almost the same processing time. The Newton update represent the execution time spent in Newton iteration. Smart-PGsim have extra overheads about the inference time of the MTL model for generating warm-start solution and the restart time with failure cases. Note that we restart the failed cases with the default setting in the numerical solver MIPS to guarantee the final convergence. As Figure 3.5 depicts, Smart-PGsim is effective at reducing the time spent on the convergence, i.e., Newton Update. Smart-PGsim demonstrates significant performance improvement for the tested input problems despite the extra overhead introduced by the MTL model.

3.8.3 Prediction Accuracy

Figure 3.4(c) compares MIPS and Smart-PGsim in terms of the success rate, in the case in which we do not restart Smart-PGsim after a failed execution. As we discuss above, Smart-PGsim guarantees *100% success rate in practice* by re-executing those computations that do not provide high-enough accuracy, while still considerably outperforming MIPS execution. The success rate is how many input problems can converge successfully in the simulation. Figure 3.4(c) reveals that Smart-PGsim leads to a high percentage of success rate in all case. Smart-PGsim provides 100% success rate on 14-bus, 57-bus, 118-bus while maintains a relatively high success rate as 97% and 92% on 30-bus, 300-bus respectively.

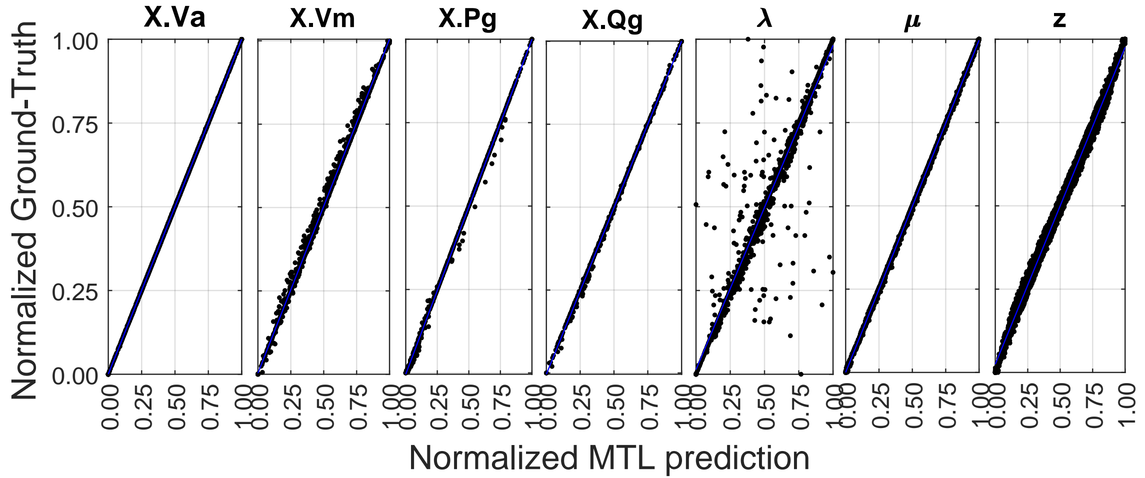

Figure 3.6: Prediction accuracy of each feature used in the proposed MTL model.

Figure 3.6 presents the prediction accuracy of each feature used in Smart-PGsim. We compare the accuracy of warm-start points predicted by Smart-PGsim with the exact solution in MIPS (Ground-truth). The prediction and ground-truth are normalized to the range [0,1]. The x-axis is the predicted value of Smart-PGsim and the y-axis is the ground-truth value. If the prediction of Smart-PGsim is perfect, all points should be lie on the $y = x$ line. There is negligible accuracy lost in the prediction of $X.Va$, $X.Vm$, $X.Pg$, $X.Qg$, μ and z. For λ, there is a larger variation in the predicted values representing over-prediction and under-prediction. Such variation in λ is acceptable because λ is the equality constraints factor in Eqn. 3.3, which will not affect the final convergence if the equality constraints are satisfied.

3.8.4 Efficiency of Multitask Learning and Physical Constraints

In this section, we analyze the effectiveness of multitask learning and imposing physics constraints. First, we develop a model of multiple separate NNs without information sharing. For peer comparison, we use the same number of layers and neurons as MTL model in the multiple separated networks. Then, to show the efficiency of physical constraints, we remove physics constraints in MTL model as a comparison.

Figure 3.7(a) shows the speedup comparison with the multiple separated NNs, MTL model and Smart-PGsim. Here, "MTL" refers to the multitask learning model without physical constraints whereas "Smart-PGsim" refers the multitask learning model with phys-

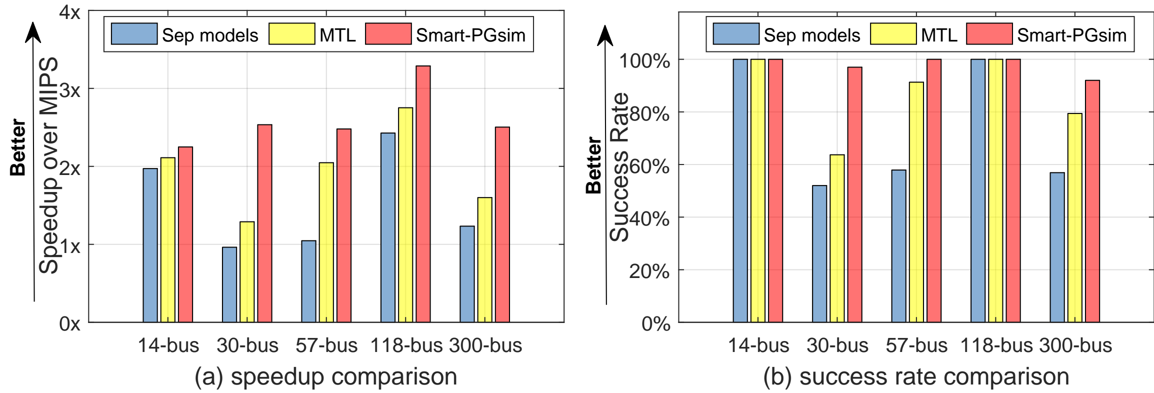

Figure 3.7: Performance comparison

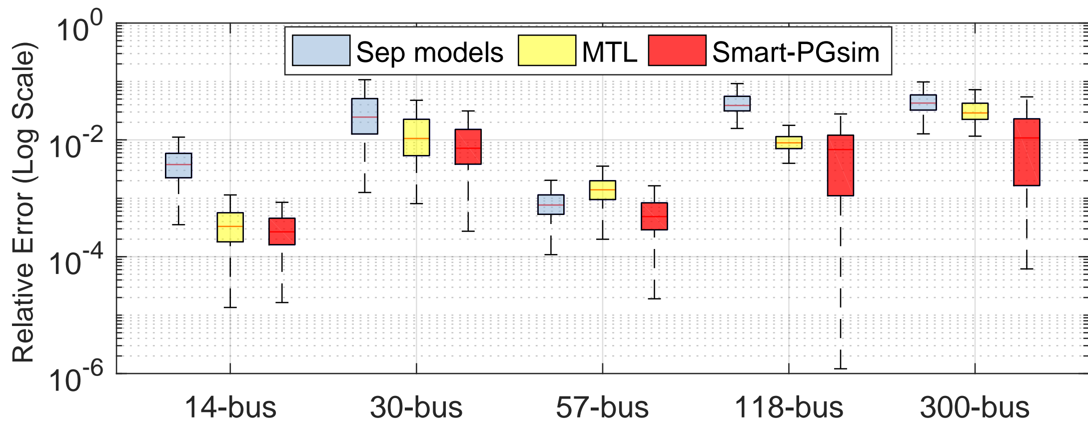

Figure 3.8: Accuracy comparison.

ical constraints. Note that all speedup are measured with our performance metric SU in Eqn. 3.9. Figure 3.7 shows that the performance of the speedup SU and the success rate SR are significantly improved by the multitask learning and incorporation of the physical constraints. MTL provides notable speedup and success rate improvement over the multiple separated models, average speedup of $1.36\times$ and 22.5% success rate improvement. In particular, the multiple separated NNs show inefficiency on the test system 30-bus with a $0.96\times$ speedup, in which the 52.0% success rate produce a soaring overhead on restart. Adding the physical constraints further improve the speedup and success rate by 40% and 18.3% over MTL. In summary, our proposed framework Smart-PGsim, a multitask model with physical constraints offers the highest average speedup and solution feasibility.

Moreover, Figure 3.8 presents box-plots[2] to show the results of the prediction accu-

[2]In the box-plots, the boxes are bounded by 25-th and 75-th percentiles of the variables; The central marks of the boxes indicate the median [33].

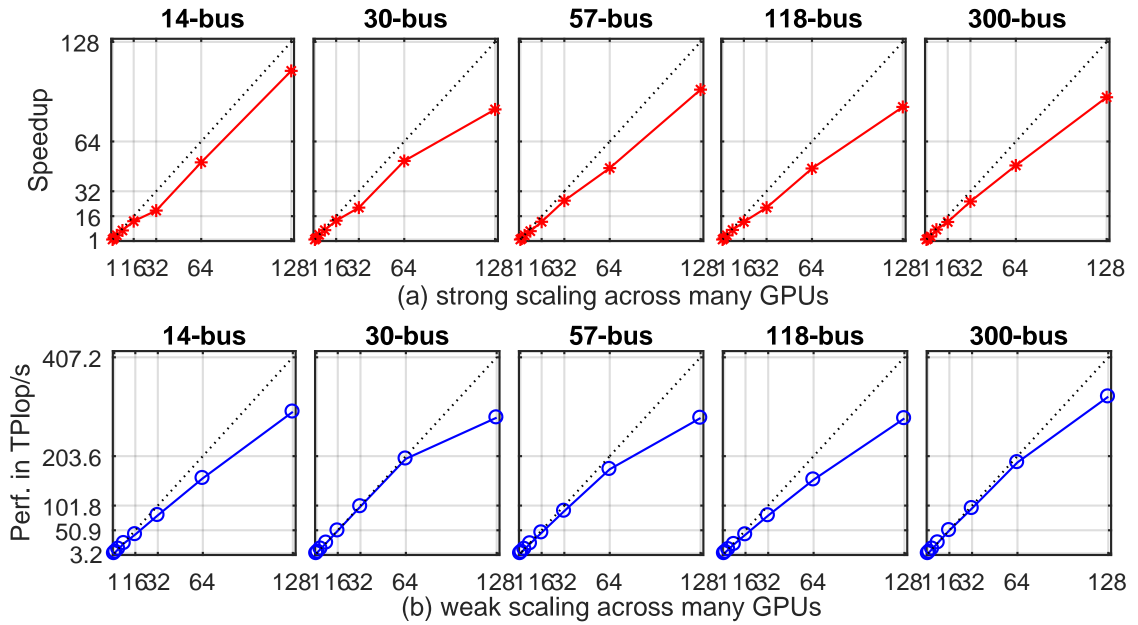

Figure 3.9: Scaling across many GPUs

racy in different models. We use relative error $RE = |V_{predict} - V_{gt}|/V_{gt}$ to measure the prediction accuracy. $V_{predict}$ refers the prediction values of MTL and V_{gt} refers the exact solution in MIPS (i.e., ground-truth). The lower relative error means the prediction is more accurate and closer to the groud-truth. We draw two observations from Figure 3.8: (1) The prediction provided by Smart-PGsim has the lowest average error with all five test systems; (2) Most of the predictions in Smart-PGsim is under the error line of 10^{-2}, which shows Smart-PGsim consistently produces prediction within 1% relative error. These two observations reveal that Smart-PGsim can provide more *consistent acceleration* than multiple separate models and MTL, which is crucial for dealing with widely diversified input problems in real-time.

3.8.5 Scalability Analysis on Multi-Node Systems

As we stated earlier, the AC-OPF problem is solved many times per day by power operators throughout the life of the power grid. Additionally, the real-life problem is further complicated by the uncertainty involved by equipment security, the reliability of alternative power sources (solar, eolic, and hydro power), and the stability of power generators and power transmission lines. Considering all those factors together is generally referred to solving Security-Constrained ACOPF (SC-ACOPF) [24, 144] and it originates very large

and complex uncertain scenario trees that need to be analyzed to maintain the robustness of the global solution, i.e, an optimal solution that survives under all uncertain scenarios. From a computational perspective, these scenarios are largely independent (although some similarities can be exploited to reduce computational requirements) and result in a computational problem that is largely embarrassingly parallel and, thus, inherently scalable on parallel computers (e.g., assigning a batch of scenarios to each compute node, and then assigning a set of scenarios from the batch to each GPU).

While the focus of this work is mainly on accelerating each AC-OPF instance of a larger SC-ACOPF problem by providing high-quality initial conditions for the numerical solver, one can easily imagine that speedup similar to the ones reported in Section 3.8.1 can be expected for the SC-ACOPF problem. To verify such assertion, we conducted experiments on a 16-node compute cluster, where each node is an NVIDIA DGX-1 equipped with eight NVIDIA V100 GPUs (128 GPUs in total). We study both *strong scalability* and *weak scalability*. Strong scaling is measured with a fixed number of scenarios, while weak scaling linearly increases the number of scenarios with respect to the number of processors. Smart-PGsim is expected to generate an initial solution for each scenario. In these experiments, we use data parallelism for scaling out the Smart-PGsim workload, in which each GPU has an identical copy of the entire network and each computes results for a separate set (the local batch) of input scenarios. We focus on scaling Smart-PGSim, which emulates the use case where there are needs to generate initial solutions for a large number of scenarios for the SC-ACOPF problem.

Figure 3.9(a) shows strong scaling behavior for each of the five test systems with a fixed problem size (10k scenarios) from 1 to 128 GPUs. The black dotted lines in the plots represent ideal scaling for data parallelism. As expected, increasing the number of GPUs naturally leads to a higher speedup and shows an almost linear tendency. However, the speedup is not linear.

Such a non-linear speedup is caused by our work distribution strategy: While our distribution algorithm has been designed to equally distribute scenarios between GPUs, communication effects can skew this balance. Specifically, when running in a node with 8 GPUs, we first copy the MTL model and data to the first GPU device and then copy it to the other GPUs leveraging GPUDirect and NVLINK, which generates some load imbalance that translates into efficiency loss.

Table 3.3: Prediction Performance Comparison.

	Test system	–	39-bus	57-bus	118-bus	–
Zamzam's [183]	**SF**	–	$15.38\times$	$9.49\times$	$7.97\times$	–
	L_{cost}	–	0.326%	0.457%	0.821%	–
	Test system	**14-bus**	**30-bus**	**57-bus**	**118-bus**	**300-bus**
Smart-PGsim	**SF**	$21.17\times$	$40.19\times$	$21.72\times$	$36.15\times$	$105.64\times$
	L_{cost}	0.007%	0.074%	0.040%	0.003%	0.008%

Figure 3.9(b) shows similar results for the weak scaling experiments, where the number of scenarios increases from 10k to 1,280k when increasing the number of GPUs from 1 to 128 (10k scenarios per GPU). The scalability shown in the plots for weak scaling is better, compared to that for strong scaling. This is because the weak scaling uses larger problems which amortizes the load imbalance problem, but we still notice similar issues as the strong scaling experiments.

Overall, Smart-PGsim scales up to 128 GPUs: for the test system of "300-bus" (the largest system we evaluated), we achieve a peak performance of 604.7 TFLOPS and a sustained performance of 326.1 TFLOPS, reaching 43% of the peak performance of Volta V100 (double precision).

3.8.6 Comparison with Prior Work

Previous work [183, 57] use ML to directly replace the exact solver to achieve a high speedup. For a fair comparison with the state-of-the-art method, i.e., Zamzam et al.'s model [183] that leverages DNN for prediction, we assume that the prediction of Smart-PGsim is the final solution, effectively replacing the entire solving computation. In Table 3.3, we compare performance and optimality loss to what has been used in Zamzam's model. Cost deviation measures simulation quality. We donate a speedup factor (SF) to measure the computational improvements: $SF = \frac{1}{n}\sum_{i=1}^{n}(T_i^{MTL}/T_i^{MIPS})$, where T_i^{MTL} refers the execution time of the MTL and T_i^{MIPS} represents the execution time of the numerical solver (MIPS) for each input problem i. We measure the loss using the average fractional difference between the predicted cost C' and the true cost C: $L_{cost} = \frac{100\%}{n}\sum_{i=1}^{n}|1 - C'_i/C_i|$. The results presented in Table 3.3 show that our framework outperforms the state-of-the-art even in the case in which we directly use Smart-PGsim output as final solution of the computation. Smart-PGsim achieves an average

$44.97\times$ speedup which provides 310.7% improvement comparing the average speedup of Zamzam's model ($10.95\times$). Moreover, Smart-PGsim decreases the cost loss $12.16\times$ by average comparing with Zamzam's model. Although Smart-PGsim show significant improvement over the state-of-the-art, we remark that the solutions produced by both models might not satisfy the strict requirements of power-grid simulations, hence our approach further refines Smart-PGsim output in the traditional solver MIPS at the back end and achieving high-quality solutions, although with reducing the speedup.

3.9 Discussions

In this section, we discuss the generality of our techniques and analyze solving processes with and without convergence.

3.9.1 Generality of Proposed Approach

The three major techniques, including sensitivity study (Section 3.5), multi-tasking learning (Section 3.6), and incorporating domain knowledge (Section 3.7), can be broadly applied to many scientific HPC applications, and are not limited to the optimization problem in power grid simulations. In this section, we highlight some potential application of our techniques to other scientific applications.

Fluid dynamic simulation aims to study the flow of fluid materials. Smart-fluidnet model [41] is a convolutional NN model to accelerate fluid simulation. We can build a multitask learning model by predicting the output velocity filed $\vec{u}$ as a main task and pressure field p as an auxiliary task since p impacts the fluid movement ($\vec{u}$). Moreover, we can incorporate the incompressibility condition, $\bigtriangledown \cdot \vec{u} = 0$, as a physical constraint regularized as a soft constraint $L_{loss} = \bigtriangledown \cdot \vec{u}$.

Molecular Dynamics (MD) simulation. The DPMD model [98] is an NN model to accelerate MD simulation. This model can be enhanced by using the techniques presented in this work by developing a multi-task model where the main task predicts the potential energy and an auxiliary task predicts the symmetry-preserving descriptor. Also, the potential energy should be positive, which can be enforced as a hard constraint.

Cosmology modeling. CosmoFlow [103] is an NN model to predict three cosmological parameters that can be directly implemented as multi-task learning. The Cosmic Microwave

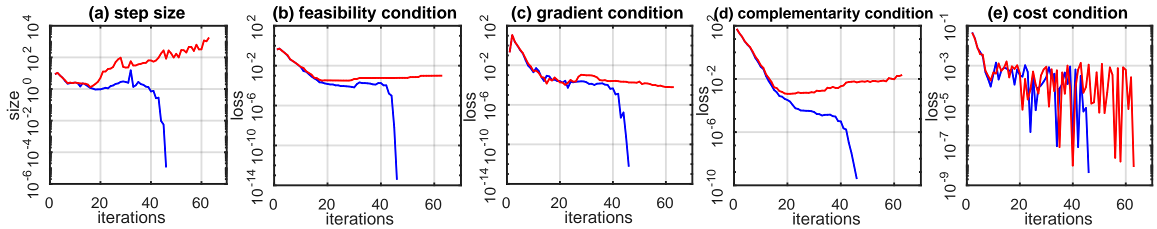

Figure 3.10: The asymptotic convergence of the tracking loss along the iterations

Background [2] can be enforced as a hard constraint to bound the projection range of modeling.

3.9.2 Analysis of Diverging Cases

The solving process for the AC-OPF problem can fail to converge. Figure 3.10 shows the inconvergence process given a bad initial solution and compares it with the convergence process given a good initial solution. Figure 3.10 shows the variance of step size and four convergence conditions across iterations. The step size $|\Delta x|$ refers to the length of the updating step during the simulation; The four conditions are used to determine if the simulation is converged in each iteration.

Figure 3.10 shows that, for the case with bad initial solution, the step size rapidly increases. Accordingly, the four convergence conditions remain relatively stable without being able to converge. For the case with good initial solution, the step size and three conditions (feasibility, gradient and complementary) decrease quickly. We notice that the cost condition goes through great variance in both cases, which makes it difficult to correlate to convergence.

The step size is critical to determine the direction to explore to find the optimal solution. If the initial solution is bad, the solving process aims to use a larger step size to find a promising direction. However, using a large step size could lead a failure of convergence (Figure 3.10.a).

As our results show, it is difficult to guess whether the numerical solver will converge based on the first iterations: both good and bad initial conditions behave similarly during the initial iterations of the power grid simulation and there is no clear indication that some computation will later fail. Given this complexity, we resort to re-initialize and re-execute the numerical solver from the beginning without employing the initial conditions generated

by the MTL model. Overall, as our results demonstrate, even considering restart time, Smart-PGsim still significantly outperforms state-of-the-art solutions.

3.10 Conclusions

Using NN to approximate and/or accelerate high performance computing applications has shown promising results. However, how to effectively apply a NN to those applications is still an open question. The approximations introduced by the NN models need to be carefully analyzed, so that the simulation quality in the application is not lost and even improved; at the same time, the execution time of the application should be reduced after applying NN. In this paper, we apply a NN to accelerate a specific power grid simulation problem, AC-OPF. As a simulation to solve complex nonlinear optimization problems based on iterative numerical methods, AC-OPF raises challenges on simulation robustness (i.e., ensuring the optimality of the simulation solution for various input problems) and respecting the physical constraints imposed by the power flow. We introduce a framework, Smart-PGsim, that facilitates the construction of a NN model by studying the impact of the output accuracy on simulation convergence and execution time and automatically imposing the physical constraints. Using a novel multitask-learning NN model generated by Smart-PGsim, we produce high-quality initial solutions for 10,000 input problems. Based on those solutions, the AC-OPF simulation reduces simulation time by an average of $2.60\times$ (up to $3.28\times$) without losing the optimality of the solution.

Chapter 4

Auto-HPCnet: An Automatic Framework to Build Neural Network-based Surrogate Model for HPC applications

4.1 Introduction

Solving many scientific computing problems involves complex computer simulations to obtain accurate and high-fidelity information about physical systems. However, the simulations often involve physics-based algebraic equations that need to be explicitly solved by closed-form optimization solvers. Those numerical solvers can be time-consuming and difficult to be ported to emerging hardware (e.g., GPU-like accelerators).

The neural network-based surrogate model can address the above problem and recently shows its power in a wide range of HPC applications, such as analyzing turbulent flow [11], subsurface flow modeling, solid mechanics modeling of diverse materials [180], and combustion modeling [40]. The neural network-based surrogate model replaces a numerical solver or an execution phase (e.g., PCG [41] and FFT [88]) in the application with a neural network (NN) model. The NN model uses the same input/output as the solver or the execution phase, and but brings large performance improvement to the application without violating numerical simulation correctness and stableness [99, 42, 88, 41]. Using NN-

based surrogate models, domain scientists are able to port the application to GPU (or other accelerators) that can run NN workloads efficiently, even though the original application does not have any accelerator-based implementation. Even better, the NN-based surrogate models can be optimized by deterministic parameters of choices, such as the number of layers in NN and neuron size, to balance prediction accuracy and cost.

Although using NN-based surrogate models to accelerate HPC applications is promising, there is a lack of tools that can automate the process of applying this method to the HPC application. In practice, once the domain scientist selects a numerical solver in an HPC application to be accelerated by NN techniques, he/she has to manually find the type of NN-based surrogate model, determine its topology, train and test it with the application. This process is labor-intensive, and could be repeated multiple times before the NN-based surrogate model is finalized. Even worse, the domain scientist may not have sufficient knowledge to efficiently deploy the surrogate model on different hardware. As a result, there is a large gap between domain scientists and NN usage.

In this paper, we introduce a framework, named AutoHPCnet, to democratize the usage of NN-based surrogate models in HPC applications. AutoHPCnet is the first end-to-end framework that makes past proposals for building NN-based surrogate models practical and disciplined. AutoHPCnet is based on our observations on multiple challenges of applying NN-based surrogate models in practice.

First, identifying the inputs/outputs of the NN-based surrogate model is difficult. In a surrogate model, we must keep the same inputs/outputs as those of the replaced code region. The input variables are read inside the code region to update other variables; the output variables are updated in the code region and used after the code region. Manually examining the code region to identify those variables is labor-intensive, because the number of variables is greatly large.

Second, removing redundancy elements (e.g., zero elements) in sparse input variables of the NN model to construct an efficient but accurate NN model is a challenge. In HPC applications, we observe that input variables are usually sparse matrices stored in Coordinate list (COO), Compressed Sparse Row (CSR), or Compressed Row Storage (CRS) formats. However, the state-of-the-art DNN frameworks, such as PyTorch and TensorFlow, cannot deal with these input variables very well. The existing DNN frameworks do not provide gradient descent functions to process the sparse matrices in the common formats

(e.g., CSR). Because of the lack of support for sparse matrices, it is inevitable to decompress sparse matrices and re-construct input variables using a format (i.e., the dense format) recognizable by the DNN framework. The new format significantly increases memory consumption because of the exponential volume of the sparse matrices after decompression.

Third, the feature reduction and selection of NN model topology are tightly coupled, and how to coordinate the two processes to minimize execution time and maximize the accuracy of the NN model is a challenge. The NN model topology refers to the number of network layers, the type of each layer (e.g., fully connected, convolution, deconvolution, or recurrent), and the number of neurons in each layer. Both the number of features and NN model topology impact model execution time and accuracy. The number of features determines the first layer in the NN model and impacts the design of the following layers; on the other hand, the topology selection of the NN model reflects feature eligibility. The existing Neural Architecture Search (NAS) methods [72, 10, 116] do not consider such interaction between feature reduction and NN topology. Also, most of them consume text, audio, and images as model input, and have difficulty consuming data structures in HPC applications as input.

Fourth, how to enable the automatic construction of NN-based surrogate models and allow users to efficiently explore the usage of surrogate models in a given HPC application is a challenge. This challenge includes how to build the whole workflow of finding NN models and making the workflow easy to use; this challenge also includes how to maximize the performance benefit while minimizing the user efforts, and how to integrate the automation into the user's decision-making process of using the NN-based surrogate models.

To address the above problems, we propose, AutoHPCnet, an automatic construction framework to build NN-based surrogate models for HPC applications. Figure 4.1 depicts AutoHPCnet. To address the first problem on the identification of input/output variables of the code region to use the NN model, AutoHPCnet introduces a set of LLVM-based tools (labeled as "Compiler-based Extractor" in Figure 4.1). Those tools instrument load and store instructions to trace memory read/write operations and enable the generation of a tree-based data dependency graph based on dynamic profiling; the tools also automatically analyze the graph to identify input/output and generate training samples based on the identified inputs/outputs.

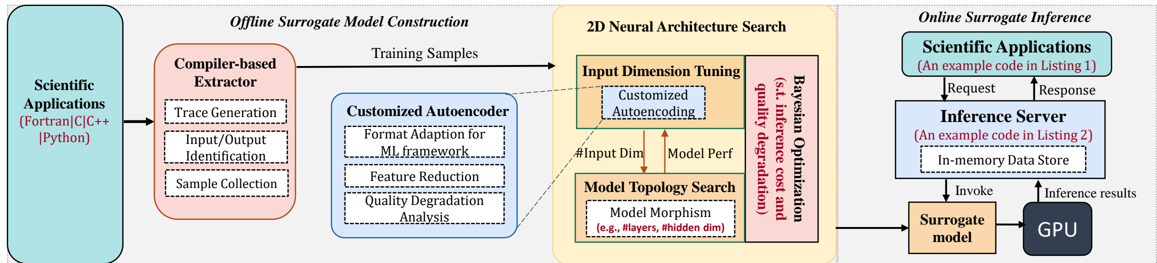

Figure 4.1: AutoHPCnet's workflow

To address the second problem on the large sparse matrix of the NN model, AutoHPCnet introduces an autoencoder-based mechanism, i.e., "Customized Autoencoder" in Figure 4.1. This mechanism applies customized designs to provide painless support for the sparse matrix to reduce features. During the offline training, the autoencoder adapts a gradient checkpoint technique to address the GPU memory limitation, which stores snapshots of the autoencoder parameters at the forward time to save memory space. At online usage, the autoencoder uses a "TensorFlow embedding API" to directly take sparse matrices as input without decompression effort.

To address the third problem of coordinating feature reduction and selection of NN model topology, AutoHPCnet introduces a "2D neural architecture search". This strategy is automated. At the high-level (input dimension tuning), this strategy uses Bayesian optimization to decide the number of features; at the low-level (model topology search), this strategy uses another Bayesian optimization to decide the NN topology using an existing AutoML framework (particularly Autokeras [72]). The low-level is based on the decision of the high-level. The two levels work iteratively and coordinately to consider the impact of both feature reduction and NN topology. Furthermore, we consider both execution time and correctness of using the NN-based surrogate model during the 2D neural architecture search. We formulate a user-given threshold as an application-specific metric and incorporate the metric into the search of the NN model, which guarantees the correctness of the application final output.

Putting together a set of tools customized for HPC applications, AutoHPCnet builds a workflow that relieves the domain scientist from labor-intensive work to apply NN-based surrogate models to HPC applications. The paper makes the following contributions.

- We introduce a framework that enables an automatic construction and use of NN-based surrogate models in HPC applications.

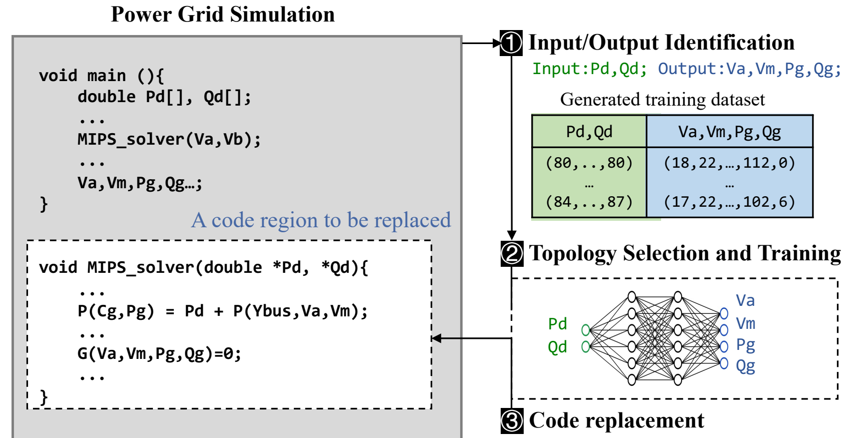

Figure 4.2: An example of applying the surrogate model.

- We introduce a workflow and a set of techniques in AutoHPCnet to address unique challenges when applying the NN-based surrogate method to HPC applications.

- We demonstrate the effectiveness of AutoHPCnet by applying it to a set of HPC applications. Our experiments show that with AutoHPCnet, the applications can achieve $5.50\times$ speedup on average (up to $16.8\times$ and with data preparation cost included) without loss in the final computation quality by replacing execution phases with an NN-based model. We show that in terms of speedup, AutoHPCnet can generate NN models outperform those models generated by the state-of-the-art methodologies including a competitive AutoML framework (Autokeras), a manual NN construction tool (ACCEPT), and an alternative approximation strategy (i.e., the loop perforation).

4.2 Background and Motivation

The NN-based surrogate model has shown tremendous performance benefits in HPC [43, 7, 104, 180, 40, 88]. Such a model introduce approximation to the original code in the application. Many HPC applications can tolerate some level of approximation due to their iterative nature [7, 104, 180, 99, 42, 115, 41, 88, 150, 75, 146, 102, 90, 15, 93, 89, 151]. For example, some HPC applications have a threshold to determine when the solution is acceptable or when the simulation should be terminated. Such a threshold-based approach allows applications to tolerate approximate computation. We use the terms "surrogate model" and "NN-based surrogate model" interchangeably in this paper.

Figure 4.2 shows three steps to generate a surrogate model for an application. The cur-

rent state-of-the-art approaches for HPC applications performs these steps manually [99, 42, 115, 41, 165, 88, 71], which is labor-intensive and motivates us to create a framework to automate this process. In more detail, the user first chooses a code region for replacement and manually examines it to identify input and output variables of the code region. The candidate code regions are generally identified by domain scientists through profiling. There could be a number of code regions amenable to surrogate models in an application. Ideally, one would accelerate through surrogate models where the application spends a large percentage of time so that the final performance acceleration is maximized. The manual identification of input and output variables is error-prone, especially when the number of input/output variables is large. After identifying inputs/outputs, the user manually generates a training dataset by varying the input problems and collecting the results.

As a next step, the user manually constructs a surrogate model to replace the code region. The surrogate construction is a process to select a network topology that balances between accuracy and execution time. A larger, more complex surrogate model has the potential to offer better model accuracy but is likely to be slower during inference than a smaller model. Given a large number of parameters and their combinatorial combinations, this step is, by far, the most complicated and time-consuming in the entire process and specific to the application. There are existing tools to automate the ML model construction process such as Autokeras [72], Google automl services [10], and Keras Tunner [116] but they cannot meet the requirements of the scientific application in terms of application outcome quality and performance improvement (of execution time).

Finally, once the surrogate model is built and trained, it is plugged into the HPC application to improve application performance. The approximate nature of the surrogate model might impact the application outcome (i.e., the final computation quality of the application). Many HPC applications have a mechanism to determine the validness of application outcomes. For example, LAMMPS (an atomic and molecular simulator) [82] examines a set of invariants to determine the validness of LAMMPS simulation outcomes. Hence, the impact of the surrogate model on application outcomes can be detected. Note that it is not enough to verify the accuracy of the surrogate model that replaces the original code region. Rather, the correctness of the entire application needs to be verified when employing the surrogate model, as the application may consist of various components, some of which are accelerated through the surrogate models. This is fundamentally different from traditional

NN approaches, where the NN process generally represents the entire application.

We take Smart-PGSim [42] as an example to depict the above steps (see Figure 4.2). Smart-PGsim applies a surrogate model to a power-grid simulation, an important HPC application. Smart-PGSim uses the above three steps manually. In the power-grid simulation, the "MIPS solver" is identified as the most time-consuming part and replaced with a surrogate model. Once the surrogate model is added into the power grid simulation (Line 4 in Figure 4.2), the simulation runs to completion and uses a quality threshold to determine the simulation validness.

The use of surrogate models for computation, like existing efforts in approximate computing [48, 112, 161, 139, 83], does not guarantee that the application outcome is valid for all input problems. If the application outcome is not valid, the application may restart using the original code region [42, 161]. Restarting the application loses performance but provides an automated mechanism to produce correct results. AutoHPCnet makes best efforts to improve the accuracy of the surrogate model while guaranteeing valid application final output, which is discussed in Section 4.6 and evaluated in Section 4.7. With AutoHPCnet, the human effort to ensure the validness of application outcome during the practice of NN-based surrogates is reduced.

Even though introducing surrogate models for computation comes with the cost of identifying features, building, and training NN models, such cost can be amortized during the frequent execution of the HPC applications. The existing work has demonstrated the performance benefit of using surrogate models [48, 112, 139]. AutoHPCnet enables the use of surrogate models for HPC applications while significantly reducing human efforts during the practice of augmenting NN technologies in HPC applications.

4.3 Data Acquisition

Given a code region selected by the user, AutoHPCnet classifies variables within the code region as input variables, output variables, and internal variables. *Input variables* are declared outside of the code region and referenced in the code region. *Output variables* are written in the code region and read after the code region. Other variables that the code region writes to or reads from are internal variables. Given a target code region, AutoHPCnet uses the following steps to acquire input and output variables.

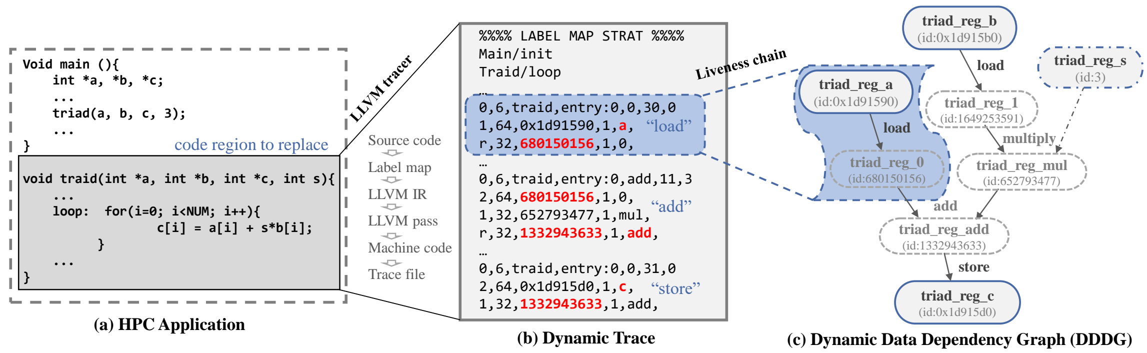

Figure 4.3: An example of acquiring input and output variables.

Trace generation. AutoHPCnet integrates an LLVM tool, LLVM-Tracer [13] which is an LLVM instrumentation pass to generate a dynamic LLVM instruction trace. This trace stores metadata for each instruction, such as the instruction type, names of registers, and operand values. Figure 4.3 shows an example to depict trace generation. Figure 4.3(b) shows the trace for an example code. AutoHPCnet extends LLVM-Tracer to reduce the trace size to simplify the identification process of input/output variables. In particular, during the trace generation, AutoHPCnet recognizes loop structures in the code region. If a loop has no control flow divergence across iterations of the loop and accessed (array) variables across iterations remain the same, then AutoHPCnet does not generate the whole trace for the loop. Instead, only the trace for one iteration is generated.

Identification of input and output variables. We build a tool to construct a dynamic data dependency graph (DDDG) from the instruction trace based on the existing method [58]. In DDDG, vertices are the values of variables obtained from registers or memory; Edges are LLVM instructions (or operations) transforming input values into output values of variables. With DDDG, the root nodes represent inputs, and leaf nodes represent outputs.

We extend the construction of DDDG in [58] to fit the needs of the surrogate model from two perspectives. *First*, we group variables for effective feature reduction. In particular, the number of input variables recognized by DDDG can be large. Some of those variables can come from the same array; The large number of input variables can come from multiple arrays. During the feature reduction phase, some individual variables used as individual input features can be selected together for reduction, even though they come from different arrays. Using those variables as individual input features loses the array

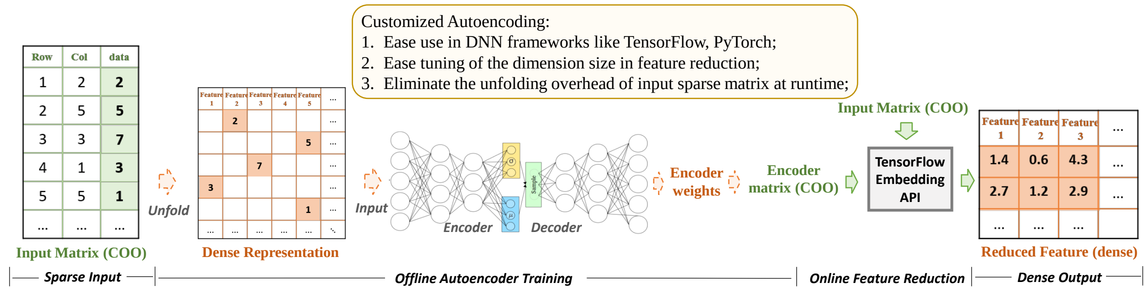

Figure 4.4: The workflow of applying Autoencoder in AutoHPCnet

semantics, which leads to either ineffective feature reduction or lower accuracy in the surrogate model. Hence, during the identification of input variables, if some variables come from the same array, then the array (not individual variables) is used as the input feature of the surrogate model. *Second*, we parallelize the construction of DDDG to shorten the construction time and make the tool more user-friendly. In particular, instead of processing instructions one by one, we process a group of instructions by multiple threads at the same time, which allows us to explore thread-level parallelism to accelerate instruction analysis when there is less dependence between instructions within the group.

Generating Training Samples. Training a surrogate model needs many training samples to ensure the model is sufficiently trained. A training sample is a pair of input features and output features, where the input and output features come from the values of the input and output variables of the target code region respectively. In cases that the user cannot find enough input problems to generate training samples, AutoHPCnet introduces a tool allowing the user to introduce perturbation into the values of input variables. The perturbation follows a specific distribution, such as the Gaussian distribution, i.e, $X' \sim \mathcal{N}(\mu, \sigma^2)$, where X' is the randomized new sample given a predefined mean μ and variance σ. The distribution can be chosen by the user based on the application domain knowledge.

4.4 Input analysis

We depict how AutoHPCnet uses autoencoder with customized designs to reduce the dimensionality of sparse input matrix in this section.

4.4.1 Autoencoders for Feature Reduction

Autoencoders aims to use a deep learning architecture to capture key representational information by mapping the high-dimensional data into a low-dimensional space [56]. Unlike traditional feature extraction technologies, Autoencoder provides an unsupervised method to do feature reduction without requiring priori knowledge on application domains.

An autoencoder consists of an encoder and a decoder. ❶ **Encoder.** The encoder typically has an hourglass-shaped architecture in which the high-dimensional data is compressed into a low-dimensional latent space that preserves semantic relationships. The encoder takes a whole matrix as input for feature reduction. ❷ **Decoder.** The decoder uses a horn-shaped network to reconstruct the reduced features back to the original representation (raw inputs). The weights of the autoencoder (both encoder and decoder) are tuned together to minimize a loss function, which typically penalizes deviations between the input of encoder and the output of decoder. Once the autoencoder is trained, the encoder is used to reduce features discussed in Section 4.5.

4.4.2 Customized Design for Sparse Input

Figure 4.4 depicts the workflow of applying the autoencoder. In Autoencoder, we adapt the following designs to address problems when dealing with sparse inputs.

First, during offline training, we adapt a gradient checkpoint technique [18] to address the GPU memory limitation. In particular, since the existing DNN frameworks do not support DNN training on sparse matrix formats, if we unfold those sparse inputs to dense representation during the autoencoder training, the memory consumption to store the dense representation becomes a bottleneck. To address this problem, we adapt the gradient checkpoint technique [18], which stores snapshots of Autoencoder parameters at a forward time and recomputes those parameters at a backward time. The gradient checkpoint technique trades the computational cost (recalculation time) for GPU memory usage (parameter storage).

Second, at online feature reduction, we provide an API for sparse matrix formats without any decompression effort. After Autoencoder is trained and no optimization is applied at online usage, the user still has to take the input matrix with dense format (which requires decompression). Here, we apply a "TensorFlow embedding API". This API con-

ducts matrix-multiplication in the format of sparse representation (e.g., CSR) and saves the multiplication result into the dense format. Autoencoder takes this API to implement the matrix-multiplication function at the first layer of Autoencoder, which helps the Autoencoder directly take the sparse matrix as an input without decompression. By doing so, we provide painless support for sparse matrices in the HPC application by eliminating both the temporal cost (format transformation of decompression) and spatial cost (memory usage of storing the dense representation) in the feature reduction process.

Third, we develop a metric to quantify the quality degradation in real-time and the user can define a lower bound of quality constraint to guide the encoding process. In traditional methods like K-means and PCA, it is hard to quantitatively measure the quality degradation before and after the feature reduction, because the size of the output matrix is not the same as the size of the input matrix. Here, we take the advantage that the autoencoder can reconstruct an output matrix (which has the same size as the input matrix) to do an element-by-element comparison. We propose a metric to evaluate the difference between the original input and the reconstructed matrix (i.e., the decoder output).

$$\sigma_y = \frac{1}{N}\sum_{i=1}^{N}\begin{cases} 0 & \text{if } |y_i - x_i| \leq \mu|x_i| \\ 1 & \text{otherwise} \end{cases} \tag{4.1}$$

We donate the metrix σ_y, defined by comparing each element y_i in the reconstructed matrix to its corresponding value x_i in the original sparse matrix and calculate the proportion of those elements that are at least in a feasible range away from x. The user determines a scaling factor μ based on the application domain requirement. Computing the similarity σ_y is lightweight and can be done on-the-fly during the autoencoder training. Based on the metric, the user can configure the lower bound (shown in Table 4.1) for different tasks.

4.4.3 Workflow of Applying Autoencoding

The workflow of autoencoder (shown in Figure 4.4) has two parts, (1) offline training happened during the Bayesian optimization and (2) online usage when the HPC application is running. The offline training happens in high-level Bayesian optimization. The high-level Bayesian optimization is a loop structure (discussed in detail in Section 4.5.2).

In each iteration of the loop, AutoHPCnet trains a new autoencoder. Across iterations, different autoencoders generate different numbers of features (the number of features is de-

termined by the low-level Bayesian optimization and the Gaussian process in the high-level Bayesian optimization). After an autoencoder is trained in an iteration of the high-level Bayesian optimization, the encoder is then used in the low-level Bayesian optimization for encoder-model inference to generate reduced features for the NN architecture search. After the hierarchical Bayesian optimization is done, the autoencoder trained in the last iteration of the high-level Bayesian optimization is used for online usage.

4.5 2D Neural Architecture Search

AutoHPCnet uses a two-dimensional neural architecture search to jointly decide NN architecture (i.e., NN topology) and the number of features. In this section, we first formulate our optimization problem. Then, we discuss algorithm details for the Bayesian optimization.

4.5.1 Problem Definition

We describe the formulation of the 2D neural architecture search as follows: given an input dataset and bound on the output error, find the best surrogate model that (i) meet the error bound and (ii) minimize the cost. This can be formulated as the following constrained optimization problem.

Problem Formulation. *Given:*

- K: a set of tunable input dimension, and
- θ: a set of tunable surrogate topology parameters.

Find the best K', θ' *such that*

- $f_c(K', \theta')$ is minimized
- $f_e(K', \theta') \leq \epsilon$.

where θ' includes #kernel sizes, #channel, #pooling size, #unpooling size, and #residual connection of each layer. We involve three functions here.

- Output: Let $f(K', \theta')$ represent the function of the NN-based surrogate model to generate output.

- Quality degradation: Let $f_e(K', \theta')$ be the magnitude of the final computational quality degradation of the application.
- Cost: Let $f_c(K', \theta')$ be the cost of computing the output at runtime. This can be the running time, energy or other execution metric to be optimized.

A solution to the optimization problem is a point that lies within the feasible region $f_e(K', \theta') \leq \epsilon$ and minimizes the objective function $f_c(K', \theta')$.

4.5.2 Hierarchical Bayesian Optimization

The Bayesian optimization [122] has been used in architecture parameter tuning for machine learning models [37], in which Bayesian optimization searches among different combinations of architecture parameters. Many Bayesian optimization processes use a Gaussian process model to model the objective function f and an acquisition function to decide where to do the next evaluation. The traditional Bayesian optimization is an iterative process consisting of three steps: update, generation, and evaluation; (1) Update: train a Gaussian process model with a combination of an optimization vector and performance (in the case of NN architecture search, the performance is the NN model cost f_c and error f_e); (2) Generation: generate a new combination to observe by optimizing an acquisition function; and (3) Evaluation: apply the new combination to the optimization target (in the case of NN architecture search, this means we use the optimization vector to train an NN model) and measure performance.

Algorithm. The optimization vector includes two types of parameters: (1) the feature reduction knobs K and (2) the NN architecture parameters θ. In the Gaussian process used in the Bayesian optimization, the optimization vector has to be vectorized in a Euclidean space because the Bayesian optimization must measure the distance between different optimization vectors. Because of the difference in physical properties, arithmetically adding the two types of parameters loses the parameter semantics, which leads to a suboptimal selection of parameters. To address the above problem, we introduce a two-level optimization strategy, which separates the optimization processes for the two types of parameters, but coordinates the two separate processes for optimal selection of those parameters.

Algorithm 3 depicts the strategy in hierarchical Bayesian optimization. There is a two-level loop in the algorithm. The *high-level loop* (Lines 2-13) searches the optimal setting

Algorithm 3 Hierarchical Bayesian optimization

Require: Input dataset D; Acceptable quality degradation f_e;
1: Given search space: $\theta = \{\theta_1, .., \theta_n\} \cup K = \{k_1, k_2, ..., k_m\}$;
2: **do** *// Outer Loop*
3: k_i = initRandom();
4: $D(k_i)$ = FeatureReduction(k_i);
5: **do** *// Inner Loop:*
6: θ_i = initModel($D(k_i)$);
7: f_e, f_c = problem.evaluate(θ_i);
8: model = GaussianProcess();
9: model.update();
10: **while** run out of searching time.
11: **Return** the performance of best NN model f' in terms of f'_c and f'_e).
12: model = GaussianProcess($_n$, f');
13: model.update();
14: **while** f_c* is minimized and $f_e* < \epsilon$.
15: **return** the optimal input size $i*$ and topology parameters $k*$.

K_i of input dimension. The *low-level loop* (Lines 5-10) searches the best architecture parameters θ_i of surrogate model. These two loops can interact with each other: the high-level loop generates an input sample with the dimension of K_n, which is then applied in the low-level loop during the architecture parameter search (Line 6); the low-level loop returns the performance (f_c and f_e) of the best model to the high-level loop (Line 11). Then the high-level loop makes a response based on an acquisition function to determine the next promising search point (i.e., k_{n+1}) and trains a corresponding autoencoder.

In the high-level loop, we apply the customized autoencoder (in Section 4.4.2) to conduct the feature reduction. In the low-level loop, we apply Autokeras [116] for model architecture search. In each iteration of the search (in either low-level loop or high-level loop), we apply the regular Bayesian optimization, which iteratively searches the optimal by the three steps: update, generation, and evaluation. The whole search process is terminated, if the required performance f_e and f_c is achieved or a continuing search does not lead to enough improvement of the performance.

4.6 Implementation

We discuss the implementation details in this section.

4.6.1 Interaction with Users

To allow the user to annotate a code region for approximation with a surrogate model without hassle, AutoHPCnet introduces two directives to mark the boundary of the code region. The annotation controls the LLVM instrumentation for trace generation, which in turns controls the usage of the surrogate model. After the annotation and building the application with LLVM, the user is expected to run a script to trigger the following workflow: (1) running the application to generate the LLVM instruction trace, (2) analyzing the trace to identify input and output variables using a tool in AutoHPCnet, and (3) uses a script to run the application N times to collect training samples (at each time, the script triggers a perturbation of input variables to collect the output of the code region).

After collecting training samples, AutoHPCnet employs the 2D neural architecture search (incorporating the customized Autoencoder). As shown in Table 4.1, AutoHPCnet gives two sets of configurations to accommodate the needs of different users. The first set, named as search-level, allows the user to easily configure the parameters of Bayesian optimization algorithm. The initial model architecture (i.e., “searchType”) can be specified based on the user’s knowledge to accelerate the search process. Also, the user is able to specify the initial sample size and the objectives of the Bayesian optimization (i.e., the acceptable encoding loss and acceptable quality loss). The initial samples are used to generate a Gaussian process model for exploring the best solution in Bayesian optimization. The second level, named model-level, is used to tune the hyperparameters (e.g., batch size and learning rate) for the surrogate model training. Also, the user can search for a specific type of neural network architectures (e.g., multi-layer perceptron or CNN).

Besides the above, AutoHPCnet has a checkpoint mechanism that allows the user to stop and restore the model architecture search. AutoHPCnet also allows the user to easily save and share the Autoencoder and the surrogate model across applications.

4.6.2 Quality-Oriented Optimizations

We summarize the implementation details in AutoHPCnet with the awareness of final computational outcome quality in this section. Such awareness separates us from the existing AutoML tools like Google AutoML [10] and Autokeras [72].

Final Computation Quality-Aware Surrogate Model Construction. During the

Table 4.1: Configurations in AutoHPCnet.

Level	Option	Description
Search-level	–searchType	(1) "autokeras" (start with the Autokera's default topology) (2) "userModel" (start with a user-given topology) (3) "fullInput" (no feature reduction applied)
	–bayesianInit	Initial samples for bayesian algorithm
	–encodingLoss	Acceptable encoding loss
	–qualityLoss	Acceptable quality loss
Model-level	–initModel	Surrogate model type (default=MLP)
	–preprocessing	Training data preprocessing
	–numEpoch	Number of epochs to train
	–trainRatio	Split dataset into training and validation
	–BATCHSIZE	Batch size
	–lr	Learning rate

model construction, AutoHPCnet considers not only the model prediction accuracy itself but also the final outcome quality of the HPC application. This is achieved by integrating a customized outcome quality metric during the surrogate model construction. Those users who have clear knowledge on how to evaluate the final computational outcome quality of the application can develop a quality metric and use it in the loss function to guide the training of the surrogate model. For example, the surrogate model in fluid simulation predicts the fluid velocity, while the condition of simulation convergence is the relative error of the velocity divergence (REVD), which must be under a certain quality constraint (e.g., 10%). Here, AutoHPCnet will take the REVD as the final outcome quality metric.

AutoHPCnet calculates REVD in the loss function of the surrogate model, using the surrogate model prediction and ground truth (i.e., the exact solution collected using the numerical solver). Inversely, the REVD is back propagated to the surrogate model to calibrate the loss function and then the REVD is gradually minimized to zero in model training. With this, AutoHPCnet incorporates the final outcome quality constraint in the surrogate model and guarantees the solution of the surrogate model does not violate the simulation stability.

AutoHPCnet implements the API "$Loss(\#components, \#metric)$", where the components defines the model inputs/outputs those should be involved to calculate the simulation metric (e.g., velocity in fluid simulation), and the metric defines the final outcome quality metric. With our 'Loss' created, the user can add their final outcome quality metric to guide the surrogate model training.

Final Computation Quality-Aware Feature Reduction. To meet the constraint on quality degradation in the HPC application, the feature reduction, which is used to optimize the surrogate model in our work (Section 4.4.2), must consider the impact of

the feature reduction on the final computation quality. AutoHPCnet implements the API "$Autoencoder.evl(\#inputs, \#compaction)$". This API measure the quality degradation before ($\#inputs$) and after (#outputs) the feature reduction, using the metric defined in Eqn 4.1. AutoHPCnet also has the innovation of coupling the feature reduction and surrogate model construction through the 2D neural architecture search (Section 4.5).

4.6.3 Online Inference Invocation

To allow the user to easily integrate the surrogate model into an application, AutoHPCnet provides two libraries: (1) a lightweight library working as a request client and compiled into the application to request the surrogate inference, and (2) a server client library to conduct NN inferences on GPU. AutoHPCnet use SmartSim Orchestrator [119] to set up a in-memory storage to enable data sharing between the HPC application written in Fortran, C, C++, or Python, and NN models written with TensorFlow, Keras, or Pytorch.

AutoHPCnet and SmartSim Orchestrator are coupled to work together. When launched through Orchestrator, applications using the AutoHPCnet clients are directly connected to any Orchestrator launched in the same experiment.

This is because AutoHPCnet client adapts a Redis module (i.e., RedisAI [121]), provides the NN runtimes, creating a library agnostic middleware between the HPC application and NN libraries. Because of this middleware, the user of AutoHPCnet can smoothly switch between the NN framework and HPC application. This method greatly reduces the deployment complexity and overhead of adding the surrogate model in the HPC application.

Making Inference Call in AutoHPCnet. Listing 1 shows an example of requesting NN inferences based on AutoHPCnet client in a C-based application. The client first sends the input tensors to the inference server (Line 5) and then makes an inference call to the server. Using the above approach simplifies implementations: the user only needs to change a few lines of code in the application.

SmartSim Implementation of Inference Call. Listing 2 shows how to spin up a database with SmartSim Orchestrator and invoke a CNN model using the AutoHPCnet client. In Listing 2, the "exp.start(orc, block=Flase)" (Line 7) uses the SmartSim library to launch a in-memory data storage. The server receives the inference request and fetches the input data from the storage (Line 11). Then, the server loads the pre-trained autoencoder (Line

Listing 1: Example of HPC simulation for surrogate request

```
1  #include “autoHPCnet_client.h”
2
3  // Initialize a Client object
4  autoHPCnet::Client client(false);
5  // Put the input features on the database
6  client.put_tensor(in_key, autoHPCnet);
7  // Run model already in the database
8  client.run_model(“AI-CFD-net”, {in_key}, {out_key});
9
10 // Get the result of the model
11 client.unpack_tensor(out_key, autoHPCnet);
```

Listing 2: Example of invoking surrogate model

```
1  from autoHPCnet import Client
2  from smartsim.database import Orchestrator
3  # import other packages …
4
5  # create and start a database
6  orc = Orchestrator(port=REDIS_PORT)
7  exp.generate(orc)
8  exp.start(orc, block=False)
9
10 # get input from database
11 sparse_tensor = client.get_tensor(input_feature)
12
13 # feature reduction and format transformation
14 compact_tensor = client.autoencoder(sparse_tensor)
15
16 # load a pretrained model from file
17 client.set_model_from_file("AI-CFD-net", "./saved_net.pt",
18 "TORCH", "GPU")
19
20 # Run model and retrieve outputs
21 client.run_model("AI-CFD-net", inputs=compact_tensor,
22 outputs=output_tensor)
```

14) and surrogate model (Line 17) to make an inference on GPU (Line 21). Notably, despite being written in Python, all surrogate models are executed in a C runtime.

4.7 Evaluation

Platform. We conduct all experiments on an NVIDIA DGX-1 cluster with 8 nodes, and each node is equipped with two Intel Xeon E5-2698 v4 CPUs (40 cores in total running at 2.20GHz) and eight NVIDIA TESLA V100 (Volta) GPUs.

Applications. Table 4.2 lists applications we evaluate. We comprehensively cover three types of applications, which have been widely studied in HPC. Type-I includes numerical solvers that are often the most time-consuming in HPC applications. Type-II includes a set of general applications from the PARSEC parallel benchmark suite [9]. Those applications are evaluated in previous efforts [139, 48, 101] for approximate computing. Type-III

Table 4.2: Applications for Evaluation.

Type	**Application:** replaced function	**Description**	**Quality of Interest (QoI)**
I	**CG**:CG_solver	Conjugate Gradient	Solution of linear equations
	FFT:FFT_solver	Fast Fourier Transform	Output sequence of FFT
	MG:MG_solver	Multi-Grid method	The final residual of the solver
II	**Blackscholes**:$BlkSchlsEqEuroNoDiv$	Investment pricing	The computed price
	Canneal:$Annealing$	VLSI routing	Routing cost
	fluidanimation:$NS_equation$	Fluid dynamics	Particle distance
	streamcluster:$Dimension_reduction$	Online clustering	Cluster center distance
	X264:$Encoding$	Video encoding	Structure similarity
III	**miniQMC:**$Determinant$	Quantum Monte Carlo	Particle energy
	AMG:PCG_solver	Solver of linear systems	Solution of linear systems
	Laghos: $SolveVelocity$	Compressible gas dynamics	Velocity Divergence

is the representative of large-scale HPC applications. Type-III comes from the Exascale Computing Project (ECP) Proxy Applications Suite 4.0 [108].

Quality of Interest (QoI). To evaluate the final computation quality of the application, we assume that the user provides application-specific QoI that can be used to quantify the difference between the solution of the surrogate model and the exact solution. Table 4.2 lists the QoI of each application. The QoI differs among applications.

4.7.1 AutoHPCnet Effectiveness

We use two metrics to evaluate AutoHPCnet effectiveness: speedup and prediction hit rate. The speedup is used to evaluate the performance of AutoHPCnet, and the prediction hit rate is used to evaluate the quality of the surrogate models generated by AutoHPCnet.

Equation 4.2 defines the speedup. We report the speedup of the whole application (instead of only the NN-replaced code region).

$$Speedup = \frac{T_{Numerical_solver}}{T'_{NN_infer} + T'_{Data_load} + T_{Other_part}} \tag{4.2}$$

where $T_{Numerical_solver}$ represents the execution time of the application using the original code (e.g., a traditional numerical solver). T'_{NN_infer} is the inference time of the surrogate model generated by AutoHPCnet and T'_{Data_load} is the data communication overhead for loading the NN model input to GPU. T_{Other_part} refers to the execution time of the rest part (the code regions without applying the NN surrogate model).

Equation 4.3 defines the prediction hit rate (i.e., $HitRate$), which refers to the ratio of the number of input problems that can reach the quality requirement with the NN surrogate

Table 4.3: Compare the performance of AutoHPCnet on GPU with the performance of the original code on GPU. The results are for AMG.

Methods	CPU-only	Original code on GPU	AutoHPCnet on GPU
Floating-Point Operations	30.66G	72.82G	21.97G
L2 level cache-miss rate	37.47%	26.31%	17.81%
Mem Bandwidth (MB/s)	3523.15	7518.85	6735.54
Wall clock time (seconds)	2.47	2.11	0.51

models, to the total number of input problems (N):

$$HitRate = \frac{1}{N}\sum_{i=1}^{N}(1,\ \text{if } |V_i' - V_i| \leq \mu|V_i|) \tag{4.3}$$

Where V is the user-specified QoI, V_i' is the calculated QoI after the surrogate model is applied to the application with the ith input problem, and V_i is the calculated QoI without applying the surrogate model to the application with the ith input problem. The difference between V_i' and V_i should be smaller than $\mu|V_i|$ in order to claim that applying the surrogate model to the application with the ith input problem generates a high-quality application outcome that meets the user's quality requirement.

$\sum_{i=1}^{N}(1,\ \text{if } |V_i' - V_i| \leq \mu|V_i|)$ in Equation 4.3 counts the total number of input problems that can meet the user's quality requirement after applying the surrogate model. μ is a parameter set by the user (see Section 4.5). In our evaluation, μ is set as 10%, which is aligned with the existing efforts [139, 48, 112] for neural network-based computation approximation.

Using the two metrics, we evaluate AutoHPCnet with 11 applications. Each application use 2,000 input problems for evaluation. Figure 4.5 shows the results.

Performance. There is $1.89\times$ - $16.8\times$ speedup with a harmonic mean of $5.50\times$ across all three types of application, compared with the application performance on CPU (using all 40 cores). Among all applications, Blackscholes has the largest speedup. The large speedup comes from the fact that the surrogate model removes all control flows in the original code, and AutoHPCnet is able to offload BlkSchlsEqEuroNoDiv (the most computation-intensive part) to GPU.

To further study the performance, we compare the performance of using the original

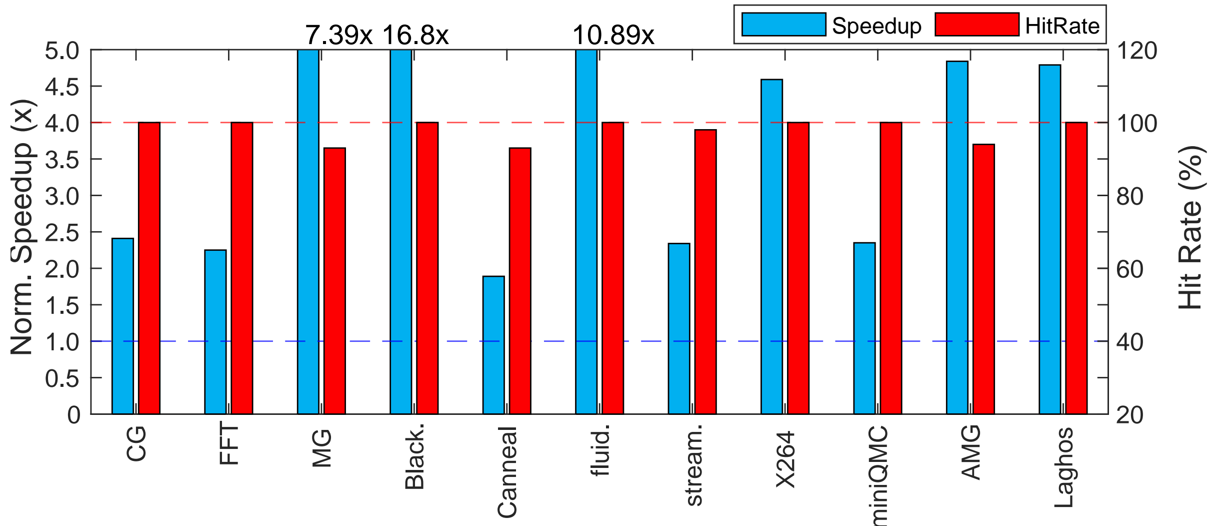

Figure 4.5: Speedup and prediction hit rate in AutoHPCnet.

code on GPU and using the surrogate models generated by AutoHPCnet on GPU. Table 4.3 shows the results for AMG, a production code that can run on either CPU or GPU. To run AMG on GPU, we use AMGX [145]. We observe that AutoHPCnet leads to 15.6% better performance than AMGX. To look into why the surrogate model on GPU performs better, we measure the number of Floating-Point (FP) operations, last-level cache miss rate, and (global) memory bandwidth consumption, shown in Table 4.3. With the surrogate model, the number of FP operations and last-level cache miss rate are reduced by 69.83% and 52.47% respectively. Such reductions come from the fact that the surrogate model on GPU, as an NN model, is highly optimized by the GPU vendor and able to run highly efficiently on GPU. For other applications, the number of FP operations and L2 level cache-miss rate are reduced by an average of 42.7% and 35.1% respectively. The main reasons for this observation are the reduction of model size and excellent data locality of matrix multiplication in neural network inference.

Quality. Figure 4.5 also reports $HitRate$. We observe that AutoHPCnet leads to high $HitRate$: $HitRate$ for MG, Canneal, streamcluster and AMG is 93%, 93%, 98% and 94% respectively; for the other seven applications, $HitRate$ is 100%. Note that for an application where $HitRate$ is not 100%, when running a specific input problem using the surrogate model leads to the final output failing to meet the quality requirement, the application has to restart and use the original code.

4.7.2 Comparison with Existing Work

In essence, AutoHPCnet is a tool to apply NN-based approximation to accelerate applications. We compare AutoHPCnet with the following tool or research effort for approximate computing.

- **ACCEPT** [139] is a tool to apply NN-based approximation to applications. ACCEPT relies on the user to manually identify the replaced code region and generate NN models without considering the impact of NN models on the final computation outcome quality.
- **Loop perforation** is a technique that selectively skips loop iterations to accelerate applications without causing significant quality degradation. Recently, loop perforation has been applied to HPC applications successfully [118]. We apply loop perforation to the 11 applications according to the recent work HPAC [118]. In particular, we use HPAC to decide how frequently the loop iterations can be skipped without causing significant quality degradation.
- **Autokeras** [116] is a tool that automatically generates NN models given training datasets. It has been reported [167] that Autokeras shows similar performance as other commercial AutoML frameworks such as Google's AutoML, H2O-AutoML, and Auto-sklearn. Autokeras cannot be used for NN-based approximation. We compare AutoHPCnet with Autokeras in terms of the NN model effectiveness of accelerating applications.

Note that we only apply ACCEPT to Type-II applications, but not Type-I and Type-III applications, because ACCEPT heavily relies on the user to specify the NN model topology. For those applications in Type-II, ACCEPT defines their NN model topology, but not for other types of applications. To enable fair comparison, ACCEPT, loop perforation, and Autokeras, and AutoHPCnet are used to accelerate the same code regions depicted in Table 4.2. During the evaluation, we ensure that the final computation quality meets the pre-determined requirement (i.e., 10%)

Figure 4.6 shows the application performance speedup after applying the above work and AutoHPCnet. The speedup is calculated with respect to the execution time of the exact execution (i.e., the original execution), using Equation 4.2. Figure 4.6 shows that AutoHPCnet consistently performs better in all applications than the other work. AutoHPCnet is able to find simple but effective NN architectures for small applications (Type-II) and also find more complicated NN architectures for larger applications (Type-III).

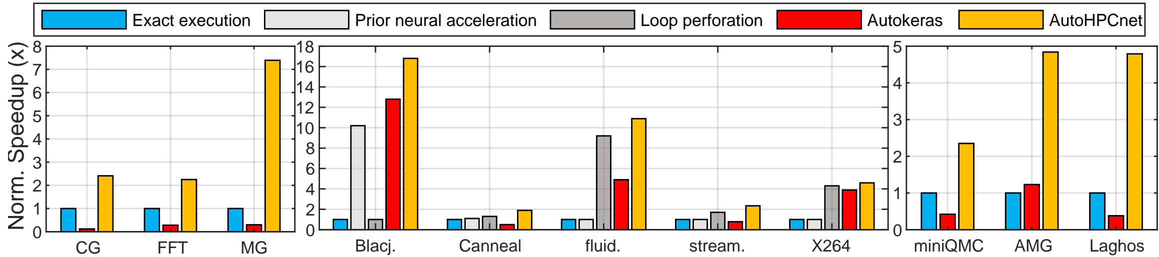

Figure 4.6: Performance comparison of other representative methods

ACCEPT and the loop perforation method perform well on a few applications (i.e., Blackschole with ACCEPT, and fluidanimation and X264 with the loop perforation) with more than 2x speedup. ACCEPT and the loop perforation perform poorly on other applications with less than 2x speedup, because of the following reason: (1) ACCEPT heavily relies on the user to specify NN models, which limits its feasibility to explore a wide range of NN models. (2) The loop perforation limits its performance improvement because its approximation granularity is only at the loop's iteration level.

Autokeras achieves 12.8x and 10.89x speedup on Blackschole and fluidsimulation respectively, which is impressive. However, Autokeras cannot lead to better performance than AutoHPCnet because of the following reasons. (1) Autokeras does not use feature reduction and does not consider model inference time, hence the NN model produced by Autokeras can have long inference time; and (2) Autokeras has problems handling sparse matrices with many zero elements, because those zero elements will cause a gradient overflow problem during the NN model training. In fact, Figure 4.6 shows that using models generated by Autokeras, there is even dramatically slowdown in those applications whose inputs are high-dimensional sparse matrices (e.g., CG, FFT, MG, miniQMC and AMG). AutoHPCnet does not have the above problems.

4.7.3 Effectiveness of AutoHPCnet Components

Effectiveness of Autoencoder. We measure the compression ratio after using a set of feature reduction techniques, including Principal Component Analysis (PCA) [86], Latent semantic analysis (LSA) [149], and autoencoder in AutoHPCnet. The compression ratio is the ratio of the matrix size after applying feature reduction to the matrix size before applying it. Table 4.4 shows the results. In our evaluation, we ensure that the final computation quality meets the pre-determined requirement.

Compared with PCA and LSA, AutoHPCnet is better by 37.6% and 26.1% for Type-I

Table 4.4: Comparison of input compression ratio and model size.

Application	Compression ratio			Model size (#parameters)	
	PCA	LSA	Autoencoder	Autokeras	AutoHPCnet
CG	25%	22%	16%	$3.6e^8$	$1.8e^7$
FFT	47%	41%	30%	$3.6e^7$	$4.5e^6$
MG	67%	58%	42%	$8.4e^7$	$3.4e^6$
Blackscholes	76%	83%	52%	$1.0e^6$	$7.6e^5$
Canneal	38%	41%	26%	$9.8e^6$	$2.6e^6$
fluidanimation	72%	79%	49%	$1.8e^6$	$8.4e^5$
streamcluster	49%	53%	33%	$2.9e^6$	$9.7e^5$
X264	51%	59%	34%	$7.4e^5$	$6.3e^5$
miniQMC	37%	34%	17%	$3.0e^{12}$	$5.5e^{11}$
AMG	56%	52%	25%	$8.6e^{10}$	$2.2e^{10}$
Laghos	29%	27%	14%	$1.2e^{12}$	$1.0e^{11}$

applications, 31.4% and 37.2% for Type-II applications, and 52.7% and 50.4% for Type-III applications, respectively. Autoencoder in AutoHPCnet reduces the input matrix size by at least 25%, which in turn reduces the size of surrogate models and inference time. We observe that Type-III applications have larger compression ratio, because the sparsity in their input matrices is higher.

Effectiveness of Model Topology Search. AutoHPCnet is able to effectively construct NN models smaller than those found by Autokeras, because of effective feature reduction. Table 4.4 shows the results. AutoHPCnet reduces the model parameters by 22.5% (Blackcholes) - 89.5% (CG), compared with those constructed by Autokeras.

Effectiveness of Bayesian Optimization. AutoHPCnet uses the Bayesian optimization to choose an NN model to replace the original code. The Bayesian optimization in AutoHPCnet has three steps, i.e., update, generation, and evaluation. We compare the Bayesian Optimization with a traditional approach, grid search [116], which simply makes a complete search over a given subset of the topologies space of neural network search. We use the default setting of grid search in AutoKeras.

We count the number of search steps per time unit (i.e., one hour) to reach the same model quality, as an indicator of search efficiency. For Type-I, II, and III applications, the average number of search steps per hour using the Bayesian optimization is 3.3, 6.5, and 2.1 respectively, while using the grid search, it is 1.6, 3.2, and 1.9 respectively. The Bayesian optimization has higher search efficiency, especially for Type-I and Type-II applications, because the quality-aware algorithm adopted by Bayesian optimization can effectively guide the search in the right direction compared with the grid search.

4.7.4 Feasibility Analysis

AutoHPCnet includes offline and online phases. We quantify the time spent on the two phases to analyze the tool feasibility.

Offline time. The offline phase of AutoHPCnet includes the trace generation, the Bayesian optimization, and Autoencoder training. The execution time of the offline phase differs from one application to another. In our evaluation, the trace generation, Bayesian optimization, and Autoencoder training take 24-59 minutes, 6-13 hours, and 1.4-2.2 hours respectively. Note that the overhead of the offline phase in AutoHPCnet, like other NN-based approximation, can be amortized, because the offline phase happens only once, and the NN-based approximation is expected to happen many times with performance benefit.

Online time. The online time includes (1) fetching input data to GPU memory, (2) encoding input data to low-dimensional features, (3) loading a pre-trained surrogate model from a file, and (4) running the surrogate model and retrieving the model output for the application. (1), (2), (3) and (4) take 21.2%, 10.1%, 1.6% and 67.1% of the whole online time on average. The online time is reported in Figure 4.5 and Figure 4.6.

4.8 Related Work

Scientific Machine Learning. Scientific machine learning [150, 75, 146, 102, 90, 15, 93, 89, 151] aims at using machine learning methods to solve scientific and engineering problems. There are many successful cases in scientific machine learning, such as using machine learning to reproduce molecular energy surfaces [154] and simulate infrared spectra for molecular dynamics [51]. In [154], researchers use a DNN to approximate Discrete Fourier Transform (DFT). By doing so, the Quantum chemistry (QC) simulation achieves $10^4\times$ speedup with a high accuracy. After that, developing new drugs can be accomplished in minutes that would have taken more than 10 years before. NN potentials can achieve the same simulation accuracy as the traditional quantum chemical method, but is several-orders-of-magnitude faster than using several hundreds of electronic structure points for simulation. In our work, we introduce a framework to automatically construct surrogate models to approximate computation in HPC applications. Different from the existing efforts, the tool development does not assume any prior knowledge on application domains.

Approximate Computing. Our work, in essence, belongs to approximate computing. Ap-

proximate computing can be leveraged to shorten execution or save energy by trading computation accuracy. Approximation computing methods use machine learning-based approximation to approximate computation The computation approximation usually happens at a coarse granularity (e.g., the whole application or multiple functions). Approximate computing has been explored in many fields, including hardware accelerators [47, 60, 140], compiler optimization [5, 109, 138, 152], programming language designs [25, 134, 137, 141], and runtime system designs [14, 55, 63]. Approximate computing has also been applied to many applications, such as streaming applications [76, 80, 136].

4.9 Conclusions

Using the surrogate models to replace computation in HPC applications is promising, but is difficult to be applied in practice, because of a series of challenges on feature acquisition, feature reduction, and NN model construction. Relying on the domain scientist to manually use those steps to apply the surrogate models is time-consuming, and fundamentally prevents the popularity of using this promising method to accelerate the performance of HPC applications. This paper aims to address the above problem and introduces an end-to-end framework (named AutoHPCnet) that democratizes the usage of NN-based approximation in HPC applications. The design of AutoHPCnet is driven by the observations on the major challenges of applying the surrogate models in practice. Built upon a novel hierarchical Bayesian optimization, customized autoencoder for sparse matrix and NN model construction, and compiler-assisted feature acquisition, AutoHPCnet can effectively ease and accelerate the exploring process of applying the surrogate models to HPC applications.

Chapter 5

Other work

Besides the research on NN-based approximation to HPC applications, I study other interesting research topics related to machine learning by collaborating with my colleagues. These topics include system optimization for decision tree on GPU [177], ML-assisted query optimization on database [92], and ML-assisted reliability prediction for large-scale programs [173].

- **System Optimization for Decision Tree.** Decision trees are widely used as ML algorithm and often assembled as a forest to boost prediction accuracy. However, using decision trees for inference on GPU is challenging, because of irregular memory access patterns and imbalance workloads across threads. In work [177], we propose Tahoe, a tree structure-aware high performance inference engine for decision tree ensemble. Tahoe rearranges tree nodes to enable efficient and coalesced memory accesses; Tahoe also rearranges trees, such that trees with similar structures are grouped together in memory and assigned to threads in a balanced way. Besides memory access efficiency, we introduce a set of inference strategies, each of which uses shared memory differently and has different implications on reduction overhead. We introduce performance models to guide the selection of the inference strategies for arbitrary forests and data set. As a result, Tahoe consistently outperforms the state-of-the-art industry-quality library FIL by 3.82x, 2.59x, and 2.75x on three generations of NVIDIA GPUs (Kepler, Pascal, and Volta), respectively.

- **NN-based application in Database.** Cardinality estimation is a fundamental and critical problem in databases. Recently, many estimators based on deep learning

have been proposed to solve this problem and they have achieved promising results. However, these estimators struggle to provide accurate results for complex queries, due to not capturing real inter-column and inter-table correlations. Furthermore, none of these estimators contain the uncertainty information about their estimations. Our work [92] present a join cardinality estimator called Fauce, which uses NN to learn the correlations across all columns and all tables in the database. Fauce is the first estimator that incorporates uncertainty information for cardinality estimation into a deep learning model. By leveraging NN model, Fauce is light-weighted and robust: it has 10× faster inference speed and it provides 1.3×-6.7× smaller estimation errors for complex queries compared with the state of the art estimator.

- **Large-scale Parallel System Resilience.** In the work [173], we introduce a new methodology to evaluate the resilience of the application running in large scales. Instead of injecting errors into the application in large-scale execution, we inject errors into the application in small-scale execution and serial execution to model and predict the fault injection result for the application running in large scales. Evaluating with four NAS parallel benchmarks and two proxy scientific applications, our results are promising: we demonstrate that using the fault injection result to predict for 64 MPI processes, the average prediction error is 8%. While using the fault injection result to make the same prediction for eight MPI processes, the average prediction error decreases to 7%.

- **Applying Persistent Memory-based Optimization for HPC applications (LAMMPS).** Molecular dynamics (MD) simulation is a fundamental method for modeling ensembles of particles. In this direction [176], we introduce a new method to improve the performance of MD by leveraging the emerging TB-scale big memory system. In particular, we trade memory capacity for computation capability to improve MD performance by the lookup table-based memoization technique. The traditional memoization technique for the MD simulation uses relatively small DRAM, bases on a suboptimal data structure, and replaces pair-wise computation, which leads to limited performance benefit in the big memory system. We introduce MD-HM, a memoization-based MD simulation framework customized for the big memory system. MD-HM partitions the simulation field into subgrids, and replaces computation in each subgrid as a whole based on a lightweight pattern-match algorithm to recog-

nize computation in the subgrid. MD-HM uses a new two-phase LSM-tree to optimize read/write performance. Evaluating with nine MD simulations, our approach shows that MD-HM outperforms the state-of-the-art LAMMPS simulation framework with an average speedup of 7.6x based on the Intel Optane-based big memory system.

In these projects, I contribute to GPU profiling (from the perspective of memory and thread synchronization) and system evaluation (including identifying performance bottleneck and comparing with existing work).

Chapter 6

Conclusions

6.1 Summary of Contributions

In **Chapter 2**, we studied an adaptive neural network-based approximation method to accelerate the Eulerian fluid simulation. The Eulerian fluid simulation is a highly computation-intensive HPC application. The NN model has been applied to accelerate it in the existing work [165]. However, the current methods lack flexibility and generalization. We tackle the above limitations and aim to enhance the applicability of NN in the Eulerian fluid simulation. We introduce Smart-fluidnet, a framework that automates model generation and application. Given an existing neural network as input, Smart-fluidnet generates multiple neural networks before the simulation to meet the execution time and simulation quality requirement. During the simulation, Smart-fluidnet dynamically switches the neural networks to make the best efforts to reach the user's requirement on simulation quality. Evaluating with 20,480 input problems, we show that Smart-fluidnet achieves 1.46x and 590x speedup comparing with a state-of-the-art neural network model and the original fluid simulation respectively on an NVIDIA Titan X Pascal GPU, while providing better simulation quality than the state-of-the-art model.

In **Chapter 3**, we studied another HPC application, the powergird simulation. The traditional powergrid simulation that optimizes power flow suffers from high computational cost of numerical optimization. For the powergrid simulation, we introduce a multitask learning model to reduce the computation cost of the powergrid simulation. In particular, we analyze intermediate results in the original numerical optimization procedure (i.e., the MIPS solver) and discover several critical variables that are most influential to the final

simulation quality. Then, we study the sensitivity of the correctness of modeling output to the final simulation quality. Third, we propose a multi-task learning model to generate a warm startpoint solution for the powergrid simulation. The multitask-based model allows the prediction of the startpoint solution and simulation parameters to share knowledge, hence improves prediction accuracy. Extensive experimental results demonstrate that our method achieves comparable accuracy with the original numerical optimization algorithms while achieving $2.60\times$ speedup (in terms of execution time).

In **Chapter 4**, we developed an automatic tool to easy the use of NN-based surrogate model for domain scientists. NN has been widely used to approximate computation in HPC applications. By replacing an execution phase in the application, NN can bring significant performance improvement to the application. However, there is a lack of tools that can help the domain scientist automatically apply NN-based approximation to HPC applications. We introduce a framework, named AutoHPC-net, to democratize the usage of NN-based approximation. AutoHPC-net is the first end-to-end framework that makes past proposals for NN-based approximation practical and disciplined. AutoHPC-net introduces a workflow to address unique challenges when applying the approximation, such as feature acquisition and meeting the application-specific constraint on computation quality. Evaluating with a set of HPC applications that previously cannot run on GPU, we show that using AutoHPC-net, those applications can leverage NN and GPU to achieve $4.34\times$ speedup on average (up to $7.39\times$ speedup and with data preparation cost included) while meeting the application-specific constraint on computation quality.

The key takeaway from this dissertation can be summarized as follows:

AI4Science is becoming the future of HPC, and we have seen in the above instances where these mission-critical HPC application can be dramatically improved by deep learning methods. AI4Science helps scientist to improve the program portability on GPU accelerators, even for the next generation AI accelerators. However, ensure the final computation quality is the key for the success of AI4science. The NN model should be fair and acceptable to scientific applications. Meanwhile, we need to take care of the uniqueness of HPC applications, for instance, how to absorb efficient information from high-dimensional scientific dataset and how to guarantee our solution satisfy simulation physical constraints.

6.2 Future Work

In future study, my overarching goal is to democratize the usage of ML-based approximation in HPC applications, and make ML models more approachable. Specific topics of interest currently include (but are not limited to) (1) constructing an robust surrogate modeling workflow based on runtime feedback; and (2) enabling energy-efficient ML training on heterogeneous accelerators. Needless to say, I always seek new collaborations, challenges and ideas from related fields.

Robust Surrogate Modeling Based on Runtime Feedback. Expanding on my prior works [41, 42], I want to further improve the scientific surrogate modeling in terms of robustness, explainability and feasibility. A significant challenge that these systems encounter lies in the drift in real-world input problem. In many of the most successful ML examples, such as image recognition, system developers assume that all the input data stem from a static distribution. Almost by definition, the most interesting scientific applications of SciML are those, such as ocean modeling or climate pattern recognition, where the input data drifts over time because it comes from a dynamic, time-evolving distribution, often even the answers ("ground-truth") are unknown beforehand. My future work will exploit an end-to-end workflow to construct trustworthy surrogate models for the drift detection and recovery with three steps:

- Drift detection and sample collection. Traditional drift detection algorithms focus on detecting input data distribution shifts. They detect the domain drift, but do not help in selecting the most important samples for labeling and retraining. In HPC domain, retrain ML model is essential, especially for those time-evolving simulations where labeling samples by domain experts is labor-intensive. Therefore, developing methods for selecting the most important samples before labeling may significantly help the scientific domain community. In the future, I am going to propose using uncertainty quantification-based drift and outlier detection method (a unsupervised algorithm) that helps selecting key samples for retraining based on inference output uncertainty (possibly combined with other quantification metrics).

- A specialized ensemble tailored towards the unseen input problem. Numerical solvers in HPC applications can be time-consuming and an order of magnitude slower than an ML-based solver. Before directly using the numerical solver to solve an unseen

input problem, it is promising to first try ML models. Here, we construct a specialized ensemble model through clustering appropriate ML models, to recover from the detected drift problem. Specifically, we have a family of models in hand for diversified input problems. At runtime, based on the attributes of the input problem, we pick the appropriate models for online prediction. For instance, in case of a ocean modeling dataset, we can uses an ensemble of expensive, more accurate models for object detection under high resolution problem and an ensemble of slower, less accurate models otherwise. The policy of model selection will be interesting to exploit.

- Drift recovery through active learning. A specialized ensemble improves the prediction accuracy on a novel input problem (outlier), but maybe still failed in the runtime simulation (not accurate enough). If the ensemble model fails, we need to collect more representative samples (measured with distance metric, and those representatives should fall into the same category as the input outlier) to retrain the model family. However, for HPC applications, it is unpractical to manually search those representative samples, because the number of samples too huge to take a exhaust exploration. For this problem, active learning can collect efficient representatives while minimizing labeling overhead. Furthermore, active learning does not require the ground truth when selecting samples (no labeling required before sample selection). This dramatically alleviates the overhead of sample labeling for domain experts.

Enabling Energy-Efficient DNN Training on Heterogeneous Hardware. DNN training is quickly emerging as a common yet heavy workload in HPC data centers. However, training DNNs often involves large data sets, high computation power and long training time. As a result, DNN training can be extremely energy-consuming. For example, training BERT (a large natural language processing model) with eight NVIDIA V100 GPUs for 12 days takes nearly 2.5 billion Joules [36]. Unfortunately, the situation only gets worse as more complex DNN models are being developed. Therefore, it is imperative to explore novel approaches to effectively reduce energy consumption during DNN model training. For this research topic, we have the following two research questions.

- How to reduce energy consumption and production cost without impacting the DNN training throughput?

- How to schedule training operations (on different components of heterogeneous hardware) to minimize energy consumption or meet an enforced power cap?

Indeed, the main reason for the large energy consumption of DNN training is the use of GPUs. Leveraging the massive thread-level parallelism, GPUs have become the predominant processing platform for DNN training. However, GPUs are very power hungry. NVIDIA GPUs, such as V100 and P100 with thermal design power (TDP) of 300 Watts and 250 Watts, respectively, can easily account for more than 90% of the total system power during DNN training [79]. Nevertheless, it is challenging to reduce GPU power and energy without impacting training throughput. The prevalent method of scaling down GPU core frequency to save power may degrade training throughput [172].

Meanwhile, heterogeneous architectures are recently deployed to boost promising performance with lower energy consumption. For example, Field-Programmable Gate Arrays (FPGAs) are highly customizable and especially good at handling streaming workloads in a pipeline fashion. Compared with GPUs, FPGAs have much lower power consumption (e.g., 90W in Intel S10 FPGA vs. 300W in Nvidia V100 GPU). This hinders the potential power of heterogeneous architectures in terms of the energy-friendly DNN training.

In my preliminary work, I take some first steps to explore new opportunities and viable solutions to schedule efficient DNN training on hybrid GPU-FPGA accelerators. A surprising corollary of our work is that, using memory bandwidth utilization and IPC of operations on GPUs as indicators, we identify those operations that can perform comparably or even better on FPGAs than on GPUs (FPGA is more flexible and energy-efficient).

Based upon the preliminary work, I will study more specialized architectures, like Sambanova [45], Xilinx Versal [169] and NVIDIA A100. My future work towards answering the questions of 1) which ML operation should run on which hardware component to maximize the production performance; 2) how to hide data movement overhead with task scheduling. Overall, this research direction will uncover new techniques at the intersection of DNN computation scheduling and hybrid accelerators.

Broader Vision

I have been working on how to make the ML model easier to use from both of the algorithm and system perspectives, including physics-informed machine learning for scientific applications, adaptive online approximation toward a rigorously quality guarantee,

and an end-to-end framework to democratize NN usage in HPC applications. These study makes past proposals for NN-based approximation practical and disciplined.

In conclusion, a general theme across much of my work and what has developed into my research philosophy has been to draw unexpected connections between seemingly distant areas or to look for challenges and ideas in fields adjacent to high performance computing. Examining the same problem from different angles has allowed me to give "fresh" perspectives on fundamental questions. For example, the observation in feature extraction led me to contributions in NN model construction via domain specific designs. In the future, I hope to continue on this path through my research pursuits and my active collaborations, especially with colleagues in scientific computing fields.

Chapter 7

Related Work

Machine learning has been used to enhance and augment scientific applications to analyze very large data sets to reveal properties that are too complex to be discovered by previous systems. From predicting Molecular energetics to tracking neutrinos, machine learning-driven enhancement dramatically advances the efficiency and accuracy of the solution of well-known scientific problems [1]. There are couples of simulation examples that successfully apply enhancement methods to their research field.

In weather prediction, Racah et al. [127] use a semi-supervised multichannel spatiotemporal CNN model to realize a better localization of extreme weather. In cancer treatment, U.S. DOE laboratories, as well as the National Institutes of Health (NIH), have recently launched a synthetic project, CANDLE [159], targeting the top challenges in cancer diagnosis and treatment. At the current stage, researchers are leveraging information of millions of cancer patient records to diagnose cancer and figure out the best treatment strategy using a scalable DNN for modeling.

Moreover, in particle physics, George and Huerta [53] use GPUs to accelerate training DNN for fast detection and processing gravitational wave data; the new machine learning-based approach is much efficient and resilient to noise than established gravitational wave detection algorithms. Seismologists and geophysicists recently reveal that machine learning techniques can help them identify earthquake patterns from three years of earthquake records at The Geysers in California, one of the world's oldest and largest geothermal reservoirs [64]. This is an unprecedented achievement. The subtle difference between patterns is unseen by traditional methods, which are less accurate. The patterns help researchers find the fluctuating amounts of water injected below ground during the energy-extraction

process.

Modulation Methods. Besides that, machine learning has also be used to create refined input data for the next iteration round in a scientific simulation to modulate the simulation process. There are several successful examples. B. Wigley et al. [4] propose a machine learning-based online optimization process for the production of Bose-Einstein condensates (BEC). With the repeated machine learning led learning, the optimization process finds the optimal evaporation ramp for BEC production shortly in fewer iterations. In Thermal–hydraulic modeling [16], an NN is trained using the output from simulations and then used to learn the dynamic behavior of the heated line. The NN is then added to the 4C circuit model as a new part. The NN model enables online control and fast assessment of the dynamic thermal-hydraulic system. Similarly, Richard et al. [133] apply an NN to ITER magnets aiming to predict when a disruption will occur in order to avoid damage to ITER and to adjust the reaction to keep generating power. The NN-based approach exceeds the best traditional methods in accuracy (95% v.s. 85%).

Approximation Methods. Approximation Methods becomes a favorite field in the last two years. Our work, in essence, belongs to approximation methods. Approximation methods can be leveraged to shorten execution or save energy by trading computation accuracy. Approximation Methods use machine learning approximation to replace scientific simulation Those code replacements happen at a coarse granularity. Typically the whole scientific simulation (instead of the fine-grained code regions) is replaced.

There are a couple of successful cases, such as using machine learning to reproduce molecular energy surfaces [154] and simulate infrared spectra for molecular dynamics [51]. In [154], researchers use a DNN to replace Discrete Fourier Transform (DFT). By doing so, the Quantum chemistry (QC) simulation achieves 10e4x speedup with a high accuracy. After that, developing new drugs can be accomplished in minutes that would have taken more than 10 years. Similarly, in [51], an NN is used to reproduce the potential energy surface (PES) of a chemical system using the data computed by quantum chemistry methods. NN potentials can realize the accuracy of the underlying quantum chemical method, but also can be several-orders-of-magnitude faster using only several hundreds of electronic structure points.

Machine learning approximation is also managed to be used to speed up quantum computing kernels [15], in which Carleo and Troyer apply machine learning approximation

on one of greatest challenges in quantum physics: the many-body problem, which describes the complex correlations within the many-body wave function. Carleo and Troyer use an NN to reproduce the quantum many-body wave function. This forces the neural network to learn properties of the ground state of the wave function. This machine learning-based approach outperforms the state-of-the-art numerical simulation methods in accuracy.

In Computer Science, Approximation methods has been explored in many sub-fields, including hardware [47, 60, 131, 140], compilers [5, 109, 138, 152], programming languages [25, 134, 137, 141], and runtime systems [14, 55, 63]. Approximate methods have been applied to many applications, such as streaming applications [76, 80, 136], However, there are only a few cases in HPC applications (e.g., molecular dynamics simulation [15], atmospheric modeling [44] and large-scale eigen decomposition [184]). We want to test more HPC applications to extend approximation methods in HPC.

Applying neural networks to HPC applications. Neural networks (especially deep neural networks (DNNs)) have been employed in HPC applications recently. In [133], a neural network is used to generate input data for modulating the simulation process. Wigley et al. [4] propose a neural network-based online optimization process for the Bose-Einstein condensates (BEC). In [16], neural networks are used to accelerate thermal–hydraulic modeling and enables faster assessment for the dynamic thermal-hydraulic system. Richard et al. [133] apply a neural network to ITER magnets to predict the occurrence of the interruption, which can be used to adjust the reaction to continue generating power and avoid ITER damage. Mathuriya et al. [103] build a CNN model to determine the physical model that describes our universe.

In our work, we use neural network models to approximate computation in an HPC application. Different from the existing efforts, we aim to address the limitation of model flexibility and generality in the existing work [165].

Acceleration of the fluid simulation. Since the fluid simulation is an important HPC application, many research efforts have been focusing on improving its performance. Popovic et al. use a multi-grid approach to pre-process data for the PCG method in the traditional fluid simulation [105].

Molemaker et al. use iterated orthogonal projections and Michael Lentine et al. apply a coarse-grid correction method [84] to approximate the Poisson's equation. These two approaches are effective, but both of them are inexact and only competitive in low-resolution

settings. Some recent efforts [34] address the above limitation by using a data-driven approach or leveraging some useful statistical characteristics in the data distribution. Our work is different from them, because we use neural network models (not traditional solvers) to improve performance of the fluid simulation.

Some recent efforts attempt to build a neural network model that makes prediction for stability, collisions, forces and velocities of data objects in images or videos [85]. Given pressure data from previous frames, voxel occupancy, and velocity divergence, Yang et al. use a patch-based neural network to predict the next pressure [65]. Jonathan et al. use a convolutional neural network to predict the pressure value in the Eulerian fluid simulation [165]. The existing NN-based approximation approaches lack flexibility and generalization, discussed in Section 2.1.

Model compression. During the process of model construction, we use some operations to generate simpler neural network models. The recent work using model compression aims to generate simpler models [20]. There are multiple techniques for model compression, including quantization parameters [68, 174], layer pruning [181, 110], binarized networks [132, 32], low rank approximation [162], and knowledge distillation [142, 96]. Different from the existing work that focuses on resource-constrained execution environment (e.g., mobile devices), we focus on simplifying model without resource constrains and study how to build the models to meet the simulation quality for an HPC application.

Clustering Algorithms for stochastic programming. Scenario clustering has been employed in stochastic programming recently [24, 123, 164] to reduce scenarios or warm-up an approximation prior to the true solution. Scenario reduction [170, 23] performs clustering on scenario data and generate a representative small-scale problem. These existed approaches to eliminate scenarios(either by sampling or by measuring strong/weak influence) will leave a gap between the true solution and the solution from the reduced KKT structure.

Warm-up strategies [31] solve the large-scale problems by warm up through getting approximate solutions from a smaller reduced one. While the warm-up strategies lack of knowledge about the importance of different scenarios. We can't measure the strong or weak influence on the solution. Some outliers are also not captured for the preconditioner, especially for one with strong influence.

Bibliography

[1] *A Converging Path for Simulation, Machine Learning, and Big Data*. https://computation.llnl.gov/newsroom/simulation-machine-learning-big-data-converging-path. Accessed APRIL 18, 2016.

[2] Peter AR Ade et al. "Planck 2015 results-xiii. cosmological parameters". In: *Astronomy & Astrophysics* 594 (2016), A13.

[3] Argonne National Lab. *CANDLE: Exascale Deep Learning and Simulation Enabled Precision Medicine for Cancer*. http://candle.cels.anl.gov.

[4] P B. Wigley et al. "Fast machine-learning online optimization of ultra-cold-atom experiments". In: *Scientific Reports* 6 (2015), p. 25890.

[5] Woongki Baek and Trishul M Chilimbi. "Green: a framework for supporting energy-conscious programming using controlled approximation". In: *ACM Sigplan Notices*. Vol. 45. 6. ACM. 2010, pp. 198–209.

[6] Kyri Baker. "Learning Warm-Start Points for AC Optimal Power Flow". In: *arXiv preprint arXiv:1905.08860* (2019).

[7] Prasanna Balaprakash et al. "Scalable reinforcement-learning-based neural architecture search for cancer deep learning research". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2019, pp. 1–33.

[8] DJ Best and DE Roberts. "Algorithm AS 89: the upper tail probabilities of Spearman's rho". In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 24.3 (1975), pp. 377–379.

[9] Christian Bienia. *Benchmarking modern multiprocessors*. Princeton University, 2011.

[10] Ekaba Bisong. "Google AutoML: Cloud Vision". In: *Building Machine Learning and Deep Learning Models on Google Cloud Platform*. Springer, 2019, pp. 581–598.

[11] Mathis Bode et al. "Deep learning at scale for subgrid modeling in turbulent flows: regression and reconstruction". In: *International Conference on High Performance Computing*. Springer. 2019, pp. 541–560.

[12] Anne H de Boer et al. "Computational fluid dynamics (CFD) assisted performance evaluation of the Twincer™ disposable high-dose dry powder inhaler". In: *Journal of Pharmacy and Pharmacology* 64.9 (2012), pp. 1316–1325.

[13] Simone Campanoni et al. "A highly flexible, parallel virtual machine: design and experience of ILDJIT". In: *Software: Practice and Experience* 40.2 (2010), pp. 177–207.

[14] Simone Campanoni et al. "HELIX-UP: Relaxing Program Semantics to Unleash Parallelization". In: *IEEE/ACM International Symposium on Code Generation and Optimization*. 2015.

[15] Giuseppe Carleo and Matthias Troyer. "Solving the quantum many-body problem with artificial neural networks". In: *Science* 355.6325 (2017), pp. 602–606.

[16] Stefano Carli et al. "Incorporating Artificial Neural Networks in the dynamic thermal–hydraulic model of a controlled cryogenic circuit". In: *Cryogenics* 70 (2015), pp. 9–20.

[17] Rich Caruana. "Multitask learning". In: *Machine learning* 28.1 (1997), pp. 41–75.

[18] Tianqi Chen et al. "Training deep nets with sublinear memory cost". In: *arXiv preprint arXiv:1604.06174* (2016).

[19] Tsung-Hao Chen and Charlie Chung-Ping Chen. "Efficient large-scale power grid analysis based on preconditioned Krylov-subspace iterative methods". In: *Proceedings of the 38th annual Design Automation Conference*. 2001, pp. 559–562.

[20] Yu Cheng et al. "A survey of model compression and acceleration for deep neural networks". In: *arXiv preprint arXiv:1710.09282* (2017).

[21] Sharan Chetlur et al. "cudnn: Efficient primitives for deep learning". In: *arXiv preprint arXiv:1410.0759* (2014).

[22] N. Chiang, C. G. Petra, and V. M. Zavala. "Structured nonconvex optimization of large-scale energy systems using PIPS-NLP". In: *2014 Power Systems Computation Conference*. Aug. 2014, pp. 1–7. DOI: 10.1109/PSCC.2014.7038374.

[23] Nai-Yuan Chiang and Victor M Zavala. "Large-scale optimal control of interconnected natural gas and electrical transmission systems". In: *Applied energy* 168 (2016), pp. 226–235.

[24] Naiyuan Chiang and Andreas Grothey. "Solving security constrained optimal power flow problems by a structure exploiting interior point method". In: *Optimization and Engineering* 16.1 (2015), pp. 49–71.

[25] V. K. Chippa et al. "Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency". In: *Design Automation Conference*. 2010.

[26] Sanghun Choi, Shinjiro Miyawaki, and Ching-Long Lin. "A Feasible Computational Fluid Dynamics Study for Relationships of Structural and Functional Alterations with Particle Depositions in Severe Asthmatic Lungs". In: *Computational and mathematical methods in medicine* (2018).

[27] Nian Shong Chok. "Pearson's versus Spearman's and Kendall's correlation coefficients for continuous data". PhD thesis. University of Pittsburgh, 2010.

[28] Alexandre Joel Chorin. "Numerical solution of the Navier-Stokes equations". In: *Mathematics of computation* 22.104 (1968), pp. 745–762.

[29] R. D. Christie, B. F. Wollenberg, and I. Wangensteen. "Transmission management in the deregulated environment". In: *Proceedings of the IEEE* 88.2 (Feb. 2000), pp. 170–195. ISSN: 1558-2256. DOI: 10.1109/5.823997.

[30] Jason M Cohen and Douglas B Page. *System and method for economic dispatching of electrical power*. US Patent 5,621,654. Apr. 1997.

[31] Marco Colombo, Jacek Gondzio, and Andreas Grothey. "A warm-start approach for large-scale stochastic linear programs". In: *Mathematical Programming* 127.2 (2011), pp. 371–397.

[32] Matthieu Courbariaux et al. "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1". In: *arXiv preprint arXiv:1602.02830* (2016).

[33] Robert Dawson. “How significant is a boxplot outlier?” In: *Journal of Statistics Education* 19.2 (2011).

[34] Tyler De Witt, Christian Lessig, and Eugene Fiume. “Fluid Simulation Using Laplacian Eigenfunctions”. In: *ACM Trans. Graph.* 31.1 (Feb. 2012), 10:1–10:11. ISSN: 0730-0301. DOI: 10.1145/2077341.2077351. URL: http://doi.acm.org/10.1145/2077341.2077351.

[35] Deepjyoti Deka and Sidhant Misra. “Learning for DC-OPF: Classifying active sets using neural nets”. In: *arXiv preprint arXiv:1902.05607* (2019).

[36] Jacob Devlin et al. “Bert: Pre-training of deep bidirectional transformers for language understanding”. In: *arXiv preprint arXiv:1810.04805* (2018).

[37] Ian Dewancker, Michael McCourt, and Scott Clark. “Bayesian optimization for machine learning: A practical guidebook”. In: *arXiv preprint arXiv:1612.04858* (2016).

[38] Ghulam Mohi Ud Din and Angelos K Marnerides. “Short term power load forecasting using deep neural networks”. In: *2017 International Conference on Computing, Networking and Communications (ICNC)*. IEEE. 2017, pp. 594–598.

[39] *DOE news wise*. https://www.newswise.com/doescience/pnnl-researchers-speed-power-grid-simulations-using-ai. Accessed: 2020-11-10.

[40] Pascale Domingo et al. “From Discrete and Iterative Deconvolution Operators to Machine Learning for Premixed Turbulent Combustion Modeling”. In: *Data Analysis for Direct Numerical Simulations of Turbulent Combustion*. Springer, 2020, pp. 215–232.

[41] Wenqian Dong et al. “Adaptive neural network-based approximation to accelerate eulerian fluid simulation”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2019, pp. 1–22.

[42] Wenqian Dong et al. “Smart-PGSim: Using Neural Network to Accelerate AC-OPF Power Grid Simulation”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’20. Atlanta, Georgia: IEEE Press, 2020. ISBN: 9781728199986.

[43] Jan Drgona, Aaron Tuor, and Draguna Vrabie. “Constrained physics-informed deep learning for stable system identification and control of unknown linear systems”. In: *arXiv e-prints* (2020), arXiv–2004.

[44] Peter D. Düben et al. “On the use of inexact, pruned hardware in atmospheric modelling”. In: *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 372.2018 (2014).

[45] Murali Emani et al. “Accelerating scientific applications with sambaNova reconfigurable dataflow architecture”. In: *Computing in Science & Engineering* 23.2 (2021), pp. 114–119.

[46] Hugo Jair Escalante et al. “AutoML@ NeurIPS 2018 challenge: Design and Results”. In: *arXiv preprint arXiv:1903.05263* (2019).

[47] Hadi Esmaeilzadeh et al. “Architecture Support for Disciplined Approximate Programming”. In: *ASPLOS*. 2012.

[48] Hadi Esmaeilzadeh et al. “Neural acceleration for general-purpose approximate programs”. In: *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2012, pp. 449–460.

[49] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. “Visual simulation of smoke”. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM. 2001, pp. 15–22.

[50] Steven J Fernandez et al. *Real-time simulation of power grid disruption*. US Patent App. 13/747,779. July 2013.

[51] Michael Gastegger, Jörg Behlerb, and Philipp Marquetand. “Machine Learning Molecular Dynamics for the Simulation of Infrared Spectra ”. In: *Chemical Science* (2017), pp. 6695–7270.

[52] Olivier Génevaux, Arash Habibi, and Jean-Michel Dischler. “Simulating Fluid-Solid Interaction.” In: *Graphics Interface*. Vol. 2003. 2003, pp. 31–38.

[53] Daniel George and EA Huerta. “Deep Learning for real-time gravitational wave detection and parameter estimation: Results with Advanced LIGO data”. In: *Physics Letters B* 778 (2018), pp. 64–70.

[54] Alex Gittens and Michael W Mahoney. “Revisiting the Nyström method for improved large-scale machine learning”. In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 3977–4041.

[55] Inigo Goiri et al. “ApproxHadoop: Bringing Approximations to MapReduce Frameworks”. In: *ASPLOS*. 2015.

[56] Ian Goodfellow et al. *Deep learning*. Vol. 1. 2. MIT press Cambridge, 2016.

[57] Neel Guha et al. “Machine learning for AC optimal power flow”. In: *arXiv preprint arXiv:1910.08842* (2019).

[58] L. Guo et al. “FlipTracker: Understanding Natural Error Resilience in HPC Applications”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. 2018.

[59] Yi Guo, David J Hill, and Youyi Wang. “Nonlinear decentralized control of large-scale power systems”. In: *Automatica* 36.9 (2000), pp. 1275–1289.

[60] J. Han and M. Orshansky. “Approximate computing: An emerging paradigm for energy-efficient design”. In: *ETS*. 2013.

[61] Francis H Harlow and J Eddie Welch. “Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface”. In: *The physics of fluids* 8.12 (1965), pp. 2182–2189.

[62] Sten Henrysson. “Gathering, analyzing, and using data on test items”. In: *Educational measurement* 2 (1971).

[63] Henry Hoffmann et al. “Dynamic Knobs for Responsive Power-aware Computing”. In: *ASPLOS*. 2011.

[64] Benjamin K Holtzman et al. “Machine learning reveals cyclic changes in seismic source spectra in Geysers geothermal field”. In: *Science advances* 4.5 (2018), eaao2929.

[65] Yang Hong et al. “Precipitation estimation from remotely sensed imagery using an artificial neural network cloud classification system”. In: *Journal of Applied Meteorology* 43.12 (2004), pp. 1834–1853.

[66] Y Huang, T Kashiwagi, and S Morozumi. “A parallel OPF approach for large-scale power systems”. In: *IEEE* (2002).

[67] R. A. Jabr. "Radial distribution load flow using conic programming". In: *IEEE Transactions on Power Systems* 21.3 (Aug. 2006), pp. 1458–1459. ISSN: 1558-0679. DOI: 10.1109/TPWRS.2006.879234.

[68] Benoit Jacob et al. "Quantization and training of neural networks for efficient integer-arithmetic-only inference". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 2704–2713.

[69] Katarzyna Janocha and Wojciech Marian Czarnecki. "On loss functions for deep neural networks in classification". In: *arXiv preprint arXiv:1702.05659* (2017).

[70] Alan Jeffrey. *Applied partial differential equations: an introduction*. Academic Press, 2003.

[71] Chiyu Max Jiang et al. "Meshfreeflownet: A physics-constrained deep continuous space-time super-resolution framework". In: *arXiv preprint arXiv:2005.01463* (2020).

[72] Haifeng Jin, Qingquan Song, and Xia Hu. "Auto-keras: An efficient neural architecture search system". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2019, pp. 1946–1956.

[73] Haifeng Jin, Qingquan Song, and Xia Hu. "Efficient neural architecture search with network morphism". In: *arXiv preprint arXiv:1806.10282* (2018).

[74] David E. Johnson et al. *Electric Circuit Analysis (3rd Ed.)* USA: Prentice-Hall, Inc., 1997. ISBN: 0132524791.

[75] Sungmoon Jung and Jamshid Ghaboussi. "Neural network constitutive model for rate-dependent materials". In: *Computers & Structures* 84.15-16 (2006), pp. 955–963.

[76] Daya S Khudia et al. "Rumba: An online quality management system for approximate computing". In: *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. 2015.

[77] Byungsoo Kim et al. "Deep fluids: A generative network for parameterized fluid simulations". In: *Computer Graphics Forum*. Vol. 38. Wiley Online Library. 2019, pp. 59–70.

[78] Theodore Kim et al. "Wavelet turbulence for fluid simulation". In: *ACM Transactions on Graphics (TOG)*. Vol. 27. ACM. 2008, p. 50.

[79] Toshiya Komoda et al. "Power capping of CPU-GPU heterogeneous systems through coordinating DVFS and task mapping". In: *2013 IEEE 31st International Conference on computer design (ICCD)*. IEEE. 2013, pp. 349–356.

[80] Dhanya R. Krishnan et al. "IncApprox: A Data Analytics System for Incremental Approximate Computing". In: *WWW*. 2016.

[81] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[82] *LAMMPS Molecular Dynamics Simulator*. `http://lammps.sandia.gov/bench.html`. Apr. 2014.

[83] Michael A Laurenzano et al. "Input responsiveness: using canary inputs to dynamically steer approximation". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2016, pp. 161–176.

[84] Michael Lentine, Wen Zheng, and Ronald Fedkiw. "A novel algorithm for incompressible flow using only a coarse grid projection". In: *ACM Transactions on Graphics (TOG)*. Vol. 29. ACM. 2010, p. 114.

[85] Adam Lerer, Sam Gross, and Rob Fergus. "Learning physical intuition of block towers by example". In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning-Volume 48*. JMLR. org. 2016, pp. 430–438.

[86] Martin D Levine. "Feature extraction: A survey". In: *Proceedings of the IEEE* 57.8 (1969), pp. 1391–1407.

[87] Tian Li et al. "Ease.Ml: Towards Multi-tenant Resource Sharing for Machine Learning Workloads". In: *Proc. VLDB Endow.* 11.5 (2018).

[88] Zongyi Li et al. "Fourier neural operator for parametric partial differential equations". In: *arXiv preprint arXiv:2010.08895* (2020).

[89] Guanghui Liang and K Chandrashekhara. "Neural network based constitutive model for elastomeric foams". In: *Engineering structures* 30.7 (2008), pp. 2002–2011.

[90] Julia Ling, Andrew Kurzawski, and Jeremy Templeton. “Reynolds averaged turbulence modelling using deep neural networks with embedded invariance”. In: *Journal of Fluid Mechanics* 807 (2016), pp. 155–166.

[91] Cong Liu, Jianhui Wang, and Jiaxin Ning. “Optimal Power Flow (OPF) in Large-scale Power Grid Simulation”. In: *FERC Conference*. Citeseer. 2010, pp. 23–24.

[92] Jie Liu et al. “Fauce: Fast and Accurate Deep Ensembles with Uncertainty for Cardinality Estimation”. In: ().

[93] Yunjie Liu et al. “Application of deep convolutional neural networks for detecting extreme weather in climate datasets”. In: *arXiv preprint arXiv:1605.01156* (2016).

[94] Zhengchun Liu et al. “Deep learning accelerated light source experiments”. In: *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*. IEEE. 2019, pp. 20–28.

[95] Zhengchun Liu et al. “Tomogan: Low-dose x-ray tomography with generative adversarial networks”. In: *arXiv preprint arXiv:1902.07582* (2019).

[96] Raphael Gontijo Lopes, Stefano Fenu, and Thad Starner. “Data-free knowledge distillation for deep neural networks”. In: *arXiv preprint arXiv:1710.07535* (2017).

[97] Steven H. Low. “Convex Relaxation of Optimal Power Flow—Part II: Exactness”. In: *IEEE Transactions on Control of Network Systems* 1 (2014), pp. 177–189.

[98] Denghui Lu et al. “86 PFLOPS Deep Potential Molecular Dynamics simulation of 100 million atoms with ab initio accuracy”. In: *arXiv preprint arXiv:2004.11658* (2020).

[99] Denghui Lu et al. “86 PFLOPS Deep Potential Molecular Dynamics simulation of 100 million atoms with ab initio accuracy”. In: *Computer Physics Communications* 259 (2021), p. 107624.

[100] Miles Lubin et al. “Parallel distributed-memory simplex for large-scale stochastic LP problems”. In: *Computational Optimization and Applications* 55.3 (2013), pp. 571–596.

[101] Divya Mahajan et al. “Towards statistical guarantees in controlling quality tradeoffs for approximate acceleration”. In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 66–77.

[102] Erik Marchi et al. “A novel approach for automatic acoustic novelty detection using a denoising autoencoder with bidirectional LSTM neural networks”. In: *Proceedings 40th IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2015*. 2015, 5–pages.

[103] Amrita Mathuriya et al. “CosmoFlow: using deep learning to learn the universe at scale”. In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 819–829.

[104] Romit Maulik et al. “Recurrent neural network architecture search for geophysical emulation”. In: *arXiv preprint arXiv:2004.10928* (2020).

[105] Aleka McAdams, Eftychios Sifakis, and Joseph Teran. “A Parallel Multigrid Poisson Solver for Fluids Simulation on Large Grids”. In: *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation*. Ed. by MZoran Popovic and Miguel Otaduy. The Eurographics Association, 2010. ISBN: 978-3-905674-27-9. DOI: 10.2312/SCA/SCA10/065-073.

[106] Aleka McAdams, Eftychios Sifakis, and Joseph Teran. “A parallel multigrid Poisson solver for fluids simulation on large grids”. In: *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. Eurographics Association. 2010, pp. 65–74.

[107] Sanjay Mehrotra. “On the implementation of a primal-dual interior point method”. In: *SIAM Journal on optimization* 2.4 (1992), pp. 575–601.

[108] Paul Messina. “The exascale computing project”. In: *Computing in Science & Engineering* 19.3 (2017), pp. 63–67.

[109] Sasa Misailovic et al. “Quality of Service Profiling”. In: *ICSE*. 2010.

[110] Pavlo Molchanov et al. “Pruning convolutional neural networks for resource efficient inference”. In: *arXiv preprint arXiv:1611.06440* (2016).

[111] James A Momoh and JZ Zhu. “Improved interior point method for OPF problems”. In: *IEEE Transactions on Power Systems* 14.3 (1999), pp. 1114–1120.

[112] Thierry Moreau et al. “SNNAP: Approximate computing on programmable socs via neural acceleration”. In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2015, pp. 603–614.

[113] MD Nefzger and James Drasgow. "The needless assumption of normality in Pearson's r." In: *American Psychologist* 12.10 (1957), p. 623.

[114] Yeesian Ng et al. "Statistical learning for DC optimal power flow". In: *2018 Power Systems Computation Conference (PSCC)*. IEEE. 2018, pp. 1–7.

[115] Frank Noé, Gianni De Fabritiis, and Cecilia Clementi. "Machine learning for protein folding and dynamics". In: *Current opinion in structural biology* 60 (2020), pp. 77–84.

[116] Tom O'Malley et al. "Keras Tuner". In: *Retrieved May* 21 (2019), p. 2020.

[117] Xiang Pan, Tianyu Zhao, and Minghua Chen. "DeepOPF: A Deep Neural Network Approach for Security-Constrained DC Optimal Power Flow". In: *arXiv preprint arXiv:1910.14448* (2019).

[118] Konstantinos Parasyris et al. "HPAC: evaluating approximate computing techniques on HPC OpenMP applications". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–14.

[119] Sam Partee et al. "Using Machine Learning at Scale in HPC Simulations with SmartSim: An Application to Ocean Climate Modeling". In: *arXiv preprint arXiv:2104.09355* (2021).

[120] Adam Paszke et al. "PyTorch: An imperative style, high-performance deep learning library". In: *Advances in Neural Information Processing Systems*. 2019, pp. 8024–8035.

[121] Raj Patel. *Data+ Education. Redis Is a Cache or More?* Tech. rep. EasyChair, 2021.

[122] Martin Pelikan, David E Goldberg, Erick Cantú-Paz, et al. "BOA: The Bayesian optimization algorithm". In: *Proceedings of the genetic and evolutionary computation conference GECCO-99*. Vol. 1. Citeseer. 1999, pp. 525–532.

[123] Cosmin G Petra and Mihai Anitescu. "A preconditioning technique for Schur complement systems arising in stochastic optimization". In: *Computational Optimization and Applications* 52.2 (2012), pp. 315–344.

[124] Cosmin G Petra, Olaf Schenk, and Mihai Anitescu. "Real-time stochastic optimization of complex energy systems on high-performance computers". In: *Computing in Science & Engineering* 16.5 (2014), pp. 32–42.

[125] *PNNL NEWS&MEDIA*. https://www.pnnl.gov/news-media/pnnl-researchers-speed-power-grid-simulations-using-ai. Accessed: 2020-10-09.

[126] Jiantao Pu and Karthik Ramani. "On visual similarity based 2D drawing retrieval". In: *Computer-Aided Design* 38.3 (2006), pp. 249–259.

[127] Evan Racah et al. "ExtremeWeather: A Large-scale Climate Dataset for Semi-supervised Detection, Localization, and Understanding of Extreme Weather Events". In: *NIPS*. 2017.

[128] Evan Racah et al. "ExtremeWeather: A large-scale climate dataset for semi-supervised detection, localization, and understanding of extreme weather events". In: *arXiv preprint arXiv:1612.02095* (2016).

[129] Alexander Radovic. "Neutrino Identification with a Convolutional Neural Network in the NOvA Detectors". In: *International Conference on High Energy Physics*. 2016.

[130] Alexander Radovic. "Neutrino Identification with a Convolutional Neural Network in the NOvA Detectors". In: *International Conference on High Energy Physics*. 2016.

[131] Ashish Ranjan et al. "ASLAN: Synthesis of Approximate Sequential Circuits". In: *DATE*. 2014.

[132] Mohammad Rastegari et al. "Xnor-net: Imagenet classification using binary convolutional neural networks". In: *European Conference on Computer Vision*. Springer. 2016, pp. 525–542.

[133] L Savoldi Richard et al. "Artificial Neural Network (ANN) modeling of the pulsed heat load during ITER CS magnet operation". In: *Cryogenics* 63 (2014), pp. 231–240.

[134] Martin Rinard. "Probabilistic Accuracy Bounds for Fault-tolerant Computations That Discard Tasks". In: *PInternational Conference on Supercomputing*. 2006.

[135] David R Rutkowski et al. "Surgical planning for living donor liver transplant using 4D flow MRI, computational fluid dynamics and in vitro experiments". In: *Computer Methods in Biomechanics and Biomedical Engineering: Imaging & Visualization* 6.5 (2018), pp. 545–555.

[136] Mehrzad Samadi et al. "Paraprox: Pattern-Based Approximation for Data Parallel Applications". In: *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2014.

[137] Mehrzad Samadi et al. "Sage: Self-tuning approximation for graphics engines". In: *Microarchitecture (MICRO), 2013 46th Annual IEEE/ACM International Symposium on*. 2013.

[138] Adrian Sampson et al. "ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing". In: *University of Washington Technical Report UW-CSE-15-01* (2015).

[139] Adrian Sampson et al. "Accept: A programmer-guided compiler framework for practical approximate computing". In: *University of Washington Technical Report UW-CSE-15-01* 1.2 (2015).

[140] Adrian Sampson et al. "Approximate Storage in Solid-state Memories". In: *ACM TOCS*. 2014.

[141] Adrian Sampson et al. "EnerJ: Approximate Data Types for Safe and General Low-power Computation". In: *PLDI*. 2011.

[142] Bharat Bhusan Sau and Vineeth N Balasubramanian. "Deep model compression: Distilling knowledge from noisy teachers". In: *arXiv preprint arXiv:1610.09650* (2016).

[143] Savu C Savulescu. *Real-time stability in power systems: techniques for early detection of the risk of blackout*. Springer, 2014.

[144] M. Schanen et al. "Toward Multiperiod AC-Based Contingency Constrained Optimal Power Flow at Large Scale". In: *2018 Power Systems Computation Conference (PSCC)*. 2018, pp. 1–7.

[145] SIAM Journal on Scientific Computing. "AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods". In: *Notices of the AMS* 37.5 (2015), S602–S626.

[146] Baris Sen and Suresh Menon. "Representation of chemical kinetics by artificial neural networks for large eddy simulations". In: *43rd Aiaa/Asme/Sae/Asee Joint Propulsion Conference & Exhibit*. 2007, p. 5635.

[147] Baljinnyam Sereeter, Cornelis Vuik, and Cees Witteveen. "On a comparison of Newton–Raphson solvers for power flow problems". In: *Journal of Computational and Applied Mathematics* 360 (2019), pp. 157–169.

[148] Baljinnyam Sereeter et al. "Linear power flow method improved with numerical analysis techniques applied to a very large network". In: *Energies* 12.21 (2019), p. 4078.

[149] Foram P Shah and Vibha Patel. "A review on feature selection and feature extraction for text classification". In: *2016 international conference on wireless communications, signal processing and networking (WiSPNET)*. IEEE. 2016, pp. 2264–2268.

[150] Uri Shaham, Alexander Cloninger, and Ronald R Coifman. "Provable approximation properties for deep neural networks". In: *Applied and Computational Harmonic Analysis* 44.3 (2018), pp. 537–557.

[151] Yuelin Shen et al. "Finite element analysis of V-ribbed belts using neural network based hyperelastic material model". In: *International Journal of Non-Linear Mechanics* 40.6 (2005), pp. 875–890.

[152] Stelios Sidiroglou-Douskos et al. "Managing performance vs. accuracy trade-offs with loop perforation". In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011.

[153] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[154] J. S. Smith, O. Isayev, and A. E. Roitberg. "ANI-1: an extensible neural network potential with DFT accuracy at force field computational cost". In: (4 2017).

[155] Troy Snyder and Minel Braun. “A CFD-Based Frequency Response Method Applied in the Determination of Dynamic Coefficients of Hydrodynamic Bearings. Part 1: Theory”. In: *Lubricants* 7.3 (2019), p. 23.

[156] A. A. Sousa, G. L. Torres, and C. A. Cañizares. “Robust Optimal Power Flow Solution Using Trust Region and Interior-Point Methods”. In: *IEEE Transactions on Power Systems* 26.2 (May 2011), pp. 487–499. ISSN: 1558-0679. DOI: 10.1109/TPWRS.2010.2068568.

[157] Evan R. Sparks et al. “Automating Model Search for Large Scale Machine Learning”. In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 2015.

[158] Joseph L Steger and RF Warming. “Flux vector splitting of the inviscid gasdynamic equations with application to finite-difference methods”. In: *Journal of computational physics* 40.2 (1981), pp. 263–293.

[159] Rick Stevens. *Exascale Deep Learning and Simulation Enabled Precision Medicine for Cancer*. http://candle.cels.anl.gov/. Argonne National Laboratory.

[160] B. Stott, J. Jardim, and O. Alsac. “DC Power Flow Revisited”. In: *IEEE Transactions on Power Systems* 24.3 (Aug. 2009), pp. 1290–1300. ISSN: 1558-0679. DOI: 10.1109/TPWRS.2009.2021235.

[161] Xin Sui et al. “Proactive control of approximate programs”. In: *ACM SIGPLAN Notices* 51.4 (2016), pp. 607–621.

[162] Cheng Tai et al. “Convolutional neural networks with low-rank regularization”. In: *arXiv preprint arXiv:1511.06067* (2015).

[163] Nils Thuerey and Tobias Pfaff. *MantaFlow*. http://mantaflow.com. 2016.

[164] André L Tits, Pierre-Antoine Absil, and William P Woessner. “Constraint reduction for linear programs with many inequality constraints”. In: *SIAM Journal on Optimization* 17.1 (2006), pp. 119–146.

[165] Jonathan Tompson et al. “Accelerating eulerian fluid simulation with convolutional networks”. In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 3424–3433.

[166] Lisa Torrey and Jude Shavlik. "Transfer learning". In: *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI Global, 2010, pp. 242–264.

[167] Anh Truong et al. "Towards automated machine learning: Evaluation and comparison of AutoML approaches and tools". In: *2019 IEEE 31st international conference on tools with artificial intelligence (ICTAI)*. IEEE. 2019, pp. 1471–1479.

[168] Alfredo Vaccaro and Claudio Canizares. "A Knowledge-Based Framework for Power Flow and Optimal Power Flow Analyses". In: *IEEE Transactions on Smart Grid* PP (Apr. 2016), pp. 1–1. DOI: 10.1109/TSG.2016.2549560.

[169] Kees Vissers. "Versal: The xilinx adaptive compute acceleration platform (acap)". In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2019, pp. 83–83.

[170] Andreas Wächter and Lorenz T Biegler. "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". In: *Mathematical programming* 106.1 (2006), pp. 25–57.

[171] Hongye Wang. "On the computation and application of multi-period security-constrained optimal power flow for real-time electricity market operations". In: *Cornell University* (2007).

[172] Qiang Wang and Xiaowen Chu. "GPGPU performance estimation with core and memory frequency scaling". In: *IEEE Transactions on Parallel and Distributed Systems* 31.12 (2020), pp. 2865–2881.

[173] Kai Wu et al. "Modeling Application Resilience in Large-scale Parallel Execution". In: *Proceedings of the 47th International Conference on Parallel Processing*. 2018, pp. 1–10.

[174] Shuang Wu et al. "Training and inference with integers in deep neural networks". In: *arXiv preprint arXiv:1802.04680* (2018).

[175] Yidong Xia et al. "A GPU-accelerated package for simulation of flow in nanoporous source rocks with many-body dissipative particle dynamics". In: *arXiv preprint arXiv:1903.10134* (2019).

[176] Zhen Xie et al. “MD-HM: memoization-based molecular dynamics simulations on big memory system”. In: *Proceedings of the ACM International Conference on Supercomputing*. 2021, pp. 215–226.

[177] Zhen Xie et al. “Tahoe: tree structure-aware high performance inference engine for decision tree ensemble on GPU”. In: *Proceedings of the Sixteenth European Conference on Computer Systems*. 2021, pp. 426–440.

[178] NN Yanenko. “Simple schemes in fractional steps for the integration of parabolic equations”. In: *The Method of Fractional Steps*. Springer, 1971, pp. 17–41.

[179] Cheng Yang, Xubo Yang, and Xiangyun Xiao. “Data-driven projection method in fluid simulation”. In: *Computer Animation and Virtual Worlds* 27.3-4 (2016), pp. 415–424.

[180] Liu Yang et al. “Highly-Ccalable, Physics-Informed GANs for Learning Solutions of Stochastic PDEs”. In: *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*. IEEE. 2019, pp. 1–11.

[181] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. “Designing energy-efficient convolutional neural networks using energy-aware pruning”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 5687–5695.

[182] Zhaosheng Yu. “A DLM/FD method for fluid/flexible-body interactions”. In: *Journal of computational physics* 207.1 (2005), pp. 1–27.

[183] Ahmed Zamzam and Kyri Baker. “Learning optimal solutions for extremely fast ac optimal power flow”. In: *arXiv preprint arXiv:1910.01213* (2019).

[184] Qian Zhang et al. “ApproxEigen: An Approximate Computing Technique for Large-Scale Eigen-Decomposition”. In: *ICCAD*. 2015.

[185] Ray D Zimmerman and Carlos E Murillo-Sánchez. “MATPOWER 6.0 user’s manual”. In: *PSERC: Tempe, AZ, USA* (2016).