January 2022

Deep Learning As Native Scientific Workflows In The Modern Swfms - Dataview

Junwen Liu
Wayne State University

DEEP LEARNING AS NATIVE SCIENTIFIC WORKFLOWS IN THE MODERN SWFMS - DATAVIEW

by

JUNWEN LIU

DISSERTATION

Submitted to the Graduate School

of Wayne State University

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2022

MAJOR: COMPUTER SCIENCE

Approved By:

____

Advisor Date

____

____

____

DEDICATION

Dedicated to my wife Ziyun Xiao,

my son Alan Zice Liu,

my father Enzhong Liu & mother Jiamin Liu.

my passed grandma Guangfeng Xiong and grandfather Chuanli Liu

ACKNOWLEDGEMENTS

I would first like to thank my first and second supervisors in Henry ford hopsital - Dr. Jinkoo Kim and Dr. Winston Wen, for consistently encouraging me to chase after perfection and keep pursuing self-improvement as a lifetime mission, for fully supporting my decision to pursue a part time Ph.D. program in Computer Science while working full-time in the Radiation Oncology, Henry Ford Hospital, and for keeping me in their prayers to persevere and keep growing.

I would also like to express my sincere and profound gratitude to my Ph.D. advisor Dr. Shiyong Lu in Wayne State Unviersity, for accepting me as a part-time Ph.D. student in the Big data Lab since 2018 Fall, and for his constant help, encouragement, and support throughout my Ph.D. studies in the past 4 years. Tremendous thanks go to Dr. Lu, for his vision, instruction, kindness and empathy that have consistently kept me in FOCUS on my studies and research succeeds.

I would also like to sincerely thank all my committee members: Dr. Robert Reynolds, Dr. Ming Dong and Dr. Dunren Che, for being in my dissertation committee and for providing their constructive feedback, valuable comments and helpful suggestions on my research. Especially, I would like to profoundly thank my co-advisor Dr. Dunren Che, for instructing, helping and supporting me as his own Ph.D. student in the past 4 year. If not for his help, I would hardly submit all ready manuscripts to good conference or journal. I would also like to genuinely thank Dr. Rober Reyonolds, for instructing me in the course of artificial intelligence, patiently advising and explaining the Culture and iCAT algorithms. I would also like to sincerely thank Dr. Ming Dong, for instructing me in the course of pattern recognition and deep learning in my master and Ph.D. programs, which set a solid foundation in my Ph.D. research.

I would also like to thank all my excellent academic colleagues from the WSU Big Data Laboratory: Dr. Ishtiaq Ahmed, Changxin Bai and Saeid Mofrad for their enjoyable cooperation, close relationship and supportive help.

I want especially thank my wonderful wife, Ziyun Xiao, for supporting my decision to quit my job and switch to a full-time Ph.D. student in Wayne State university. Throughout my Ph.D. program, she was always fully supportive, considerate and helpful. We shared the same faith and prayer in bad times during the past 4 years. I am also grateful for my son Alan Zice Liu, he has become one of my greatest source of joy and motivation, and even brightened the world of mine.

My sincere gratitude goes to my father Enzhong Liu and mother Jiamin Liu, for their endless love, support and encouragement to me and my family. I am deeply thankful to my passed grandfather Chuanli Liu and grandma Guangfeng Xiong, for their unconditional love and encouragement since my childhood.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

In this chapter, first, the notions of workflow and SWFMS are introduced; then a brief overview of the requirement of deep learning (DL) functionalities in SWFMSs is provided; next, the major research problems of DL integration in a SWFMS are stated; and finally, an organization of the rest chapters is outlined for the dissertation.

1.1 Scientific workflow and artificial intelligence

Scientific workflow, which allows scientists to conveniently model the complex data processing steps and data dependencies, has demonstrated great potential as a key accelerator for various scientific discovery processes across numerous scientific domains [19, 25, 26, 96]. Behind the scene are numerous Scientific Workflow Management Systems (SWFMSs) that have been developed to manage the modeling and execution of scientific workflows, including Pegasus [23], Kepler [57], Taverna [71], Swift [99], and DATAVIEW [44]. These SWFMSs have been increasingly exploited by various research communities [18,26, 62,77] including astronomy, bioinformatics, ecology, computational engineering, etc. Traditionally, scientific workflows are formulated as directed-acyclic-graphs (DAGs) [55], in which nodes represent computational tasks and edges represent data dependencies among tasks, to organize complex computations and data analysis.

On the other hand, advanced artificial Intelligence (AI) techniques [28] has been thriving so fast since early this century, ML, especially DL, has become increasingly popular and been utilized in broad scientific processes and projects across nearly all scientific domains [20, 78, 100]. Thanks to the continued advance in new GPU micro-architectures, DL models can now be trained on very large datasets in accelerated speed, and deliver extraordinary prediction accuracy across broad application disciplines [103]. However, current machine learning (ML) or DL models are constrained to be running within their specified platforms(Tensorflow [2], PyTorch [74], Theano [89],etc), which places burden on scientists to adapt their existing work on various of platforms with non-trivial learning curve.

Thanks to the emerging of new HPC and memory technologies, many data-intensive scientific workflows [55] have been developed and new requirements have been introduced, such as convenient deployment on multi-clouds or HPC platforms, and superior data reproducibility. To meet above needs, new techniques [9, 104] such as Container have been introduced to ensure better reproducibility, and various APIs have been adopted to enable users to programmatically define the workflows and tasks, which enhances the flexibility of customization. Such emerging techniques holds great potentiality to benefit ML/DL in a SWFMS.

Some works [4, 66, 79] have been done to implement simple CPU-based ML applications on existing SWFMSs such as unsupervised k-mean and other clustering algorithms, however these traditional AI techniques come with huge limitations due to their simplicity, which holds scientists back from fully taking advantages of the legacies on the burst of modern AI techniques (e.g. Neuron Networks, Capsule Networks, etc.) as well as emerging High Performance Computing(HPC) technologies (e.g. GPU, TPU, cloud computing, etc.), to further accelerate their scientific discoveries.

In order to integrate ML/DL to the existing scientific management systems, some works [85, 92, 93] have been done for the construction of unified frameworks/systems for developing AI applications. However, as these frameworks/systems may involve multiple

ML/DL libraries at the same time, which makes it even harder to trace any root cause of low/intermediate-level API errors, bringing more uncertainties on performance especially when outsourcing ML/DL execution to various third-party providers.

Overall, though the idea of incorporating GPU-enabled ML/DL in existing SWFMSs is quite fascinating, identifying the research issues standing in the way and solving them are the major research tasks of this dissertation that I will elaborate them in the next section.

1.2 Statement of the problems

Although there are many GPU-enabled DL libraries available, such as PyTorch [75], Tensorflow [2] Keras [37], Caffe [41], Theano [8] and Mxnet [14], they are not readily usable in a SWFMS environment. As a consequence, tremendous work such as architectural design, model training, and optimization has to be first carried out outside of a SWFMS and then integrated into a workflow in an inefficient, ad-hoc manner [80], which is neither trivial nor optimal, for the reasons that: 1) it requires expertise with one or more DL libraries and the underlying SWFMS; 2) transferring data between DL models and data-intensive scientific workflows [19] in SWFMS tend to be time-consuming and less efficient; 3) the separate development of DL models and computation-intensive scientific workflows [19] based on completely different platforms tend to be complicated and error-prone.

One simple idea would be to simply adopt the existing DL APIs (e.g. PyTorch/Keras Java APIs [36, 87]) in a SWFMS with the hope to quickly apply these production-ready and user-friendly APIs on the development of the DL functionalities (as components) for scientific workflows. However, as these DL APIs are not designed for scientific workflow environments, simply outsourcing DL tasks to third-party APIs ineluctably bears limitations, of which I shed more light below.

First, it is very hard to customize any real-time output from intermediate nodes/layers (e.g. feature maps) of developed DL models based on external frameworks/libraries, and pipeline them to other workflow tasks, or integrate them as a part of a larger workflow and stream live data through such workflow in real-time, since the output format and data types of those DL APIs are fixed/locked by third-party providers, which inevitably introduce various type-I or type-II shimming problems [51] (i.e. data format and type incompatibilities) while chaining ordinary tasks and DL tasks together in a workflow.

Second, it is hard to trace the root cause of any bugs, errors or performance issues in a sophisticated scientific workflow with DL models involving intermediate level DL APIs, as most of the existing DL libraries are not open-source in their intermediate level APIs. For example, cuDNN [15] and cuBLAS [69] are NVIDIA's intermediate level DL APIs (closed source), which are built upon CUDA [82] (an open-source low-level API) and are profoundly utilized by popular DL libraries/APIs such as PyTorch, TensorFlow and Keras. This may result in a situation where errors are untraceable and resultant performance becomes unpredictable to both SWFMS developers and workflow users.

Therefore, it is necessary to provide a SWFMS with infrastructure-level support for GPU-enabled DL capability that is natively implemented and seamlessly integrated into the SWFMS.

1.3 Organization of this dissertation

The remaining part of this dissertation is organized as follow: Chapter 2 reviews the research on ML in scientific workflow, GPU-based DL and modern SWFMSs that are closely related to this research; Chapter 3 presents an innovative approach for supporting DL in SWFMS (DATAVIEW) on single GPU; Chapter 4 describes the extended support of this approach on heterogeneous GPU cluster and the reuse of trained DL models in DATAVIEW; Chapter 5 outlines the usability of DLaaW in DATAVIEW to SWFMS community from JAVA API and web interface; Chapter 6 concludes the dissertation and list some of the potential interesting research problems.

CHAPTER 2 RELATED WORK

Considerable research and industrial advancement have been done in the area of SWFMSs and ML/DL. In this Chapter, I focus on reviewing the work that is most closely related to the research in this dissertation: Section 2.1 recaps the goals, requirements and challenges of SWFMSs and summarize the modern SWFMSs ; Section 2.2 presents the cutting-edge GPU-based DL techniques and frameworks; and Section 2.3 reviews research of ML in SWFMSs from multiple perspectives .

2.1 Modern SWFMSs

A SWFMS provides a platform for domain scientists to compose and execute scientific workflows, which are pipelined series of computational and/or data processing tasks designed to solve complex computation-intensive and/or data-intensive scientific problems. Scientists can remotely collaborate on complex scientific projects based on scientific workflow platforms through GUI or command line (CMD) tools. Since this research is heavily correlated with SWFMSs, which enables native GPU-enabled DL capabilities in one of modern SWFMSs, this section briefly recaps the goal, requirements and challenges of SWFMSs and reviews on existing representative SWFMSs.

2.1.1 SWFMS: goals, requirements and challenges

One of essential functionality of SWFMS is for Workflow scheduling, which is the procedure of mapping workflow tasks to compute resources (e.g. virtual machines (VMs)) which are needed for tasks' execution. The goal of scheduling is to get an efficient scheduling plan (SP) that optimizes certain objectives, such as minimizing the makespan and/or the monetary cost of a workflow's execution. In general, workflow scheduling algorithms can be classified into three classes: *static*, *dynamic* and *hybrid*, of which 1) *static* algorithms generate schedules statically, i.e., before the workflow execution starts; 2) *dynamic* algorithms flexibly intermingle the scheduling and execution steps; and 3) *hybrid* algorithms typically construct a preliminary schedule first, then start its execution and dynamically adapt/optimize (parts of) the SP during the execution based on newly available dynamics of the actual execution of the workflow.

A SWFMS provides a platform for domain scientists to compose and execute scientific workflows, which are pipelined series of computational and/or data processing tasks designed to solve complex computation-intensive and/or data-intensive scientific problems. Scientists can remotely collaborate on complex scientific projects based on scientific workflow platforms through GUI or command line (CMD) tools. This section briefly describes the goal, requirements and challenges of SWFMSs.

Below, I will describe the goals of modern SWFMS from three perspectives:

1) **Automate and distribute workflow executions:** SWFMS assembles scientific data processing tasks with data dependencies among them, and automates the scheduling and execution of the tasks in order to accomplish the overall workflow under user-specified Quality of Service requirements. Nowadays, scientific workflows are becoming more and more data- and computation-intensive, running scientific workflows in a single workstation becomes less practical and may take significant amount of time, which urges SWFMSs to exploit large amounts of distributed resources and increase the degree of parallelism at either task level or workflow level or both. As cloud computing offers on-demand, elastic, pay-as-you-go and multi-tenant resource models, more and more scientific discoveries are embracing advantages offered by cloud computing, such as the virtually unlimited com-

puting resources and the fine-grained control on the trade-off between the computation power and the cost, in order to be able to flexibly configure the execution of their scientific workflows in an ideal balance between the execution cost and the completion time.

2) **Trade-off between Service Level Agreement (SLA) and Quality of Service (QoS) requirements:** In cloud computing, cloud service providers typically furnish a SLA that consists of full performance metrics for on-demand services and defines the quality of services promised to users at different levels of cost [97]. On the other hand, users can conveniently select and lease services at different quality levels with different costs suitable for their QoS requirements, which generally include 1) functional requirements such as $makespan$, $budget$, or their combinations, and 2) nonfunctional requirements such as energy consumption, security and reliability. A SWFMS must have the capability to facilitate workflows to realize optimal or near optimal QoS possible per the provider's SLA, based on the workflow's workload characteristics (size of datasets and complexity of workflows) and at the cost level of workflow user's choice. More specifically, a SWFMS need to automatically determine appropriate (or best) hyper-parameters such as the types and number of the VMs to be leased from the cloud provider, which is typically accomplished via workflow scheduling such as the various scheduling algorithms, for the execution of scientific workflows on behalf of the workflow users.

3) **Real-time monitoring and failure handling:** As scientific workflows can be very computation-intensive and may take days or even weeks for execution, real-time monitoring of workflow execution becomes more and more crucial for users to be instantly aware of the status of a workflow's execution, to check the intermediate products (stored in the form of provenance data), and to steer the execution path whenever considered neces-

sary. Furthermore, to ensure the availability and reliability of SWFMS during workflow execution, the capability to detect or even predict potential failures during the workflow execution is becoming a necessity. Reactive failure handling include, 1) handling failure caused by data, e.g., due to mismatching data types or the shimming problem [45, 51] in general, which in most of case requires SWFMS to reveal enough information to users for a manual fix, re-execute some tasks, and resume workflow execution, 2) handling failures caused by the system, e.g., due to memory overflow or system crash, which usually requires re-trying the failed tasks manually or automatically. On the other hand, proactive failure handling tries to avoid potential failures by predicting failures and proactively replacing the suspected tasks/sub-workflows and/or data artifacts, which, as a promising new feature of SWFMS, requires to integrate advanced statistical analysis and ML techniques into SWFMSs.

Moreover, in the following, I will summarize the three specific requirements of SWFMSs:

1) **User-friendly and customizable UI:** A user-friendly user interface(UI) is critical to improve the usability of a SWFMS. A user friendly UI is often required to support the entire life-cycle of scientific workflows, including workflow design, workflow execution, data presentation and provenance analysis, etc. A good SWFMS UI can greatly accelerate the process of scientific discovery by facilitating scientists to design, construct and execute their application workflows all at great ease, e.g. allowing scientists to conveniently design workflows by dragging and dropping graphic workflow elements. A good UI in a SWFMS should allow scientists to easily run or re-run the whole or some parts of a complex workflow, check and visualize intermediate data products when needed. There is currently an increasing need for SWFMSs to provide highly customizable UIs so that the domain-specific

requirements or preference can be maximally satisfied via flexible UI customization, and such customization shall not cause any unintended effect on the functional components of the SWFMS. Cloud computing particularly calls for SWFMS functionalities to be provided with resort to the mechanisms of web servers or websocket servers which provide the basis of supporting UI/GUI and command line interactions between a SWFMS and the users.

2) **Reproducibility and provenance support:** Reproducibility is one of the fundamental requirements of scientific experiments, which requires scientific results produced by scientific workflows to be reproducible. SWFMSs greatly facilitate reproducibility of scientific results via their convenient means of collecting and managing provenance data. In the context of a SWFMS, provenance data includes the metadata that captures the history of data product derivation, data transactions among tasks/workflows, and workflow execution paths, etc. Provenance data are collected and stored during workflow execution. In order to support provenance analysis and reproducibility of scientific results, SWFMS must be able to query and present provenance data, and automatically rebuild the same execution environment based on the stored provenance data when it is required to rerun the workflow for reproducing previously produced results. In general, the provenance data management module in a SWFMS needs be able to answer a series of questions regarding provenance: what were the specifications of an implemented infrastructure? (e.g. operation system, hardware specifications, etc.) Which steps were taken to produce a specific intermediate result? What input data had contributed to a specific result? Were there any human interactions involved in producing a specific result? In clouds, various types of storage platforms are available, such as relational SQL database, NoSQL database, and file servers, which provide flexible and scalable storage support for provenance data; the

high availability and distributed nature of cloud computing can assure the availability and reliability of stored provenance data.

3) **Interoperability between multi-service providers:** As more and more scientific research projects are collaborated among multiple parties in nature and involving multiple geographically distributed institutions [83], the availability of SWFMSs may vary by different regions. Moreover, different parties may have varied focus on their scientific problems and different preference with the SWFMSs. Complex scientific workflows tend to be decomposable and distributed into subworkflows so that the subworkflows can be managed by different involved parties using different SWFMSs. Therefore, a key architectural requirement for modern SWFMSs is to promote and facilitate the interoperability between different SWFMSs so that multiple parties can conveniently collaborate on some common or related parts. The interoperability for SWFMSs lies in three levels: 1) task-level interoperability, which requires that various tasks and data products from different SWFMSs can interoperate one with another; 2) workflow-level interoperability, which requires that a scientific workflow in one SWFMS can be executed in or invoked by another SWFMS; and 3) subsystem-level interoperability, which requires that a subsystem in one SWFMS can be called, reused or shared by different SWFMSs.

Last but not least, I will highlight three challenges of modern SWFMSs in the mainstream of workflow execution environment:

1) **Hardware performance fluctuation:** Service computing utilizes gigantic shared resource pools (backed by grid, cluster, huge datacenters, etc.) [12]. Workflow users thus shall expect high dynamics and sometimes noticeable performance fluctuation, much like one shall expect much stronger waves while in ocean than in a small pond. For example,

in the cloud, multiple VMs may share the same CPUs, main memory, network and disk I/O of the same physical machine, the performance of the VMs may fluctuate due to varied status of hardware utilization and network traffic caused by other users, especially when facing seasonal or other periodical variations of demands. Studies [6] showed 4%-16% deviation from the mean I/O performance due to network and I/O interference between VMs. In addition, Hardware and software failure happens, provisioned instances may crash, network re-partition may be on the fly due to a failed network device, which can all lead to additional performance fluctuation and uncertainty in the execution of scheduled workflows, especially long-running complex scientific workflows in the cloud.

The influence on scientific workflows caused by the performance fluctuation can be much mitigated by: 1) dynamic scheduling algorithms, which is capable to dynamically adjust scheduling strategy based on real-time execution status, such as WRPS [81] and DPDS [60], and 2) performance prediction techniques, which can potentially improve the decision making in dynamic scheduling in a proactive fashion — scheduling decisions are made not only based on the current status of workflow execution, but also on predicted performance variation/offset according to provenance data regarding past performance variation. For the latter, there are already interesting techniques being proposed, for example, F.Moradi et al. [67] proposed a performance prediction model in dynamic clouds using Transfer Learning. The goal of their model is to predict the service quality at the client during execution time based on available observations of the infrastructure without heavily relying on extensive measurements and data collection for training their prediction model, by means of transfer learning on deep neural networks. More specifically, it is noted that the mapping between infrastructure metrics X and service-level metrics Y may change in a cloud environment due to resource scaling, service migration, switching of hardware platforms, or other structural dynamics. Also, this work adopted transfer learning that transfers (part of) the knowledge embedded in a neural network learnt from one source environment to another neural network for solving the problems in another source environment; the knowledge transfer is selective — retraining only partial layers of the learnt performance model after some changes made with the load pattern, infrastructure configuration, service configuration and performance metric. The transfer model configuration determines which weights of original model will be retained intact and which weights will be retrained; if the output type of of the model in the target domain is different from the original, then only the output layer will be replaced by a new layer with weights randomly initialized and retrained.

2) **Data security and integrity:** Although the advantages of service computing are appealing, their data storage usually requires users to relinquish physical possession of data, thus yields their data to potential security risks in regard to numerous aspects of confidentiality, data quality, correctness, consistency, completeness, and loss of data, etc.

In order to secure sensitive data, G.S. Mahmood *et al.* in [59] proposed a novel approach to enhance confidentiality and integrity of data while uploading image data to cloud. Steganography is a data hiding technique that makes a confidential image embed into a cover image based on a shared key, thereby producing a stego image to achieve security. Steganography methods include two types: 1) Spatial domain, in which the original image is modified to encode secret information, 2) transform domain, in which an image is first changed from spatial domain to frequency domain, then image coefficients are altered to hide secret data. The first type has higher payload but is weak to attacks, while the second type type has low payload but is robust against statistical attacks. Steganography methods have great potential for being used as a security means to secure or to add extra layer of protection to confidential cloud data in the cloud.

On the other hand, in architecture level for example, S. Mofrad et al. proposed sec-Dataview [64], which enhances confidentiality and integrity of code and data for workflows executed on public clouds by adopting Hardware-assisted trusted execution environments: 1) Intel Software Guard extension (SGX) [61], 2) AMD secure encrypted virtualization (SEV) [43]. SGX protects workflow execution and workflow data by means of a shielding approach and SGX-LKL library OS, and AMD SEV protects sensitive worker nodes during the workflow runtime. SGX enclave page cache (EPC) memory paging may significantly increase execution time ($>$1000x overhead) of a workflow when tasks require a large amount of secure memory (due to heavy memory paging between EPC and outside of EPC); to mitigate, SEV worker nodes are called for workloads that require larger amount of secure memory but is less security-sensitive, leaving SGX worker nodes mainly to high security-sensitive confidential tasks. Also, WCPAC protocol is proposed for securing execution of workflow tasks in remote worker nodes. This protocol, including provisioning and attesting secure worker nodes, is used to establish secure communication among master nodes and worker nodes for secure code provisioning for both the *TaskExecutor* and workflow tasks on secure worker nodes. The code provisioning attestation runs on the master node to verify the integrity of the *CodeProvisioner* module executed at a remote worker node; upon integrity is verified, it sends the decryption key, workflow's input data, and SSL certificate to the *CodeProvisioner* module and then returns the control to the *workflowExecutor*.

3) **Real-time enormous dataset processing:** With IoT and Edge computing [27, 52] start to saturate every aspects of life, more and more large blocks of real-time data collected from scientific experiment need to be processed in SWfMSs in real-time fashion, in order to return immediate analysis results or feedback.

For example, in order to address the enormous real-time data processing challenge in clouds, E. Lyons et al. proposed in [58] a novel network-centric platform that enables high-performance adaptive data flows across distributed cloud providers and data repositories for atmospheric scientists. In their work, a system called DyNamo is developed, to enable high performance data-flows through layer2 global dynamic-circuit network across multiple distributed cloud providers and data repositories. Compare with common layer3 network that accomplishes segmented routing over an internet protocol (IP) network, layer2 is a broadcast Media Access Control(MAC) level network, and capable to forward all traffic including ARP and DHCP broadcasts in a fast speed.

Another example is Li, F., & Song, F. proposed in [49] to couple scientific simulations with in-situ(or in-memory) data visualizations. The in-situ data analysis analyze simulation data while data still reside in memory instead of outputting to secondary storage, to enable user to monitor and get real-time notifications of special patterns or anomalies from ongoing extreme-scale turbulent data flows. They also designed a computational fluid dynamic (CFD) specific ML method, in which the data communication is realized through Remote Direct Memory Access (RDMA), to connect different applications dynamically at runtime with high-productivity in distributed/parallel computing, in order to automatically detect anomaly flows, automate data analysis and expedite the process of online simulation analysis.

Table 1: A summary of SWFMSs.

Systems	**Pegasus [23]**,**DATAVIEW [44]**, **Kepler [57]**, **Taverna [71]**, **Swift [99]**, Galaxy [34], VisTrails [30], TimeStudio [70], KNIME [10], Pipeline Pilot [94], ClowdFlows [47], TextFlows [76], VIEW [50], U-Compare [42], SecDATAVIEW [64]

Table 2: Comparison on representative workflow systems.

Systems	Domains	Execution Envs	SaaS	UI	API	Language support	Open Source
Pegasus	Scientific computing	local, cluster, grid, clouds	No	Cmd	Yes	Java, Python, Perl	`https://pegasus.isi.edu/downloads/`
DATAVIEW	Big data analytics	local, clouds	Yes	Web	Yes	Java, Python	`https://github.com/shiyonglu/DATAVIEW`
Kepler	Big data analytics	local, clusters, web services	No	Desktop	Yes	Java, R	`https://kepler-project.org/`
Taverna	Bioinformatics	local, web services	No	Desktop, Cmd	Yes	Scufl2	`https://taverna.incubator.apache.org/`
Swift	Scientific computing	local, cluster, grid, clouds	No	Cmd	Yes	Swift	`https://github.com/swift-lang`

More innovative works are expected for addressing the enormous data processing with DL capabilities in native workflows as an integral part of future SWFMSs. I will present five modern SWFMSs and their key features in following section.

2.1.2 Representative SWFMSs

A SWFMS provides a platform for domain scientists to compose and execute scientific workflows, which are pipelined series of computational and/or data processing tasks designed to solve complex computation-intensive scientific problems. Scientists can remotely collaborate on complex scientific projects based on scientific workflow platforms through GUI or command line (CMD) tools.

In this section, I will present five representative workflow management systems. They were selected due to their respective outstanding features. For example, Pegasus contributed to LIGO (Laser Interferometer Gravitational wave Observatory) [3] that successfully helped detect the gravitational wave - a discovery that won the Noble prize; DATAVIEW manifested the notion of Workflow-as-a-Service (a special kind of SaaS) that allows users to utilize the system through the DATAVIEW website without the need to download and install the system; Kepler and Taverna both provide highly intuitive client-side UIs that ease workflow construction and execution, while Swift comes with a scripting based language tool that allows users to use C-like syntax to enact rapid applications of workflows involving big data. All these five selected systems are open-source and can be freely downloaded from their respective project websites (URLs are provided in Table 2 for readers' convenience). This work particularly addresses them in terms of their targeted application domains, execution environments, and other features such as third-party API support, programmatic language support, etc.

Pegasus [23], is the workflow management system encompasses a set of technologies that facilitate scientific workflow application execution. Pegasus was designed to manage workflow execution on potentially distributed data and compute resources, in close collaboration with domain scientists. Pegasus workflows are based on Directed Acyclic Graphs (DAG), a model that has been commonly assumed by various SWFMSs. Pegasus allows a node in a workflow DAG be a sub-DAG, which facilitates composition of very large workflows in the scale of millions of task nodes. In Pegasus, tasks exchange data between machines in the form of files, and workflow execution can be arranged to take place in a local or remote cluster, or in a grid or cloud. User interaction with Pegasus is through either command line commands or API interfaces. Pegasus provides programmatic API in python, Java and Perl for workflow generation in the form of DAX (or DAG in XML). The system also keeps variety of catalogs in order to support workflow optimization.

Pegasus is open-source. It has contributed to the LIGO software infrastructure and executes the main analysis pipelines of LIGO to detect the gravitational wave. Pegasus uses HTCondor as its workflow engine and scheduler, and can be setup on distributed or cloud environments. In Pegasus, graph transformations and optimizations are performed during *mapping* when a workflow is mapped onto a distributed environment before its *execution*. Optimizations is also performed during run-time by interleaving *mapping* and just-in-time planning. In order to improve reliability of workflow execution, during run-time, Pegasus performs actions such as job retry and failed workflow recovery .

DATAVIEW [44], is a generic and comprehensive SWFMS. The applications of DATAVIEW range from ML, medical image analysis, bioinformatics, to automotive data analysis, etc. DATAVIEW is also based on DAG and adopts a layered architecture design that includes a presentation layer, a workflow management layer, a task management layer, and an infrastructure layer. DATAVIEW features a user-friendly Web portal for workflow creation and execution, and workflow execution can be flexibly arranged to run locally or on a cloud platform such as AWS. DATAVIEW adopts a master-slave deployment architecture and supports fast provisioning of virtual machines through VM images created and saved on AWS. The DATAVIEW VM images include the DATAVIEW kernel that schedules and executes workflows. With a developer-friendly Java API, DATAVIEW supports programmatic workflow development through Java and Python. DATAVIEW seamlessly integrates Dropbox as optional storage capacity for feeding workflow input and storing workflow output products.

DATAVIEW is open-source. In addition to local installation, it can be used as a SaaS (Software-as-a-Service) from www.dataview.org without download and installation of the system. In DATAVIEW, web-based GUI allows users to compose and edit workflows in an appealing visual style, e.g., by dragging and dropping task components and data elements onto the design panel and connecting them through edges as executable workflows. Its workflow engine manages the workflow schedulers, workflow specification mappers, dataflow storage, provenance data, compute resources, run-time monitor and analysis tools, etc. Workflow specifications are written in JSON-based SWL (Scientific Workflow Language). Its elastic *Cloud Resource Management* module dynamically provisions and de-provisions virtual machines throughout workflow execution, based on user specified preferences. DATAVIEW features an open extensible architecture for its workflow engine, which consists of a set of workflow planners and a set of workflow executors. A developer can easily choose any existing or to develop their own custom workflow planners and executors.

Kepler [57], is a community-driven workflow system that supports scientific workflow applications, and help scientists, analysts and programmers to create and analyze scientific data such as sensor data, medical images and simulations, etc. Kepler provides a Java-based component assembly framework with a graphical user interface to support the assembly of concurrent task components. The key underlying principle of Kepler is to utilize well-defined models of computation to govern the interactions between task components in a workflow during execution. Using Kepler's graphical desktop GUI, scientists can create executable scientific workflows by simply dragging and dropping task components. Kepler supports workflow execution on a local machine, a cluster or through web services. Kepler is capable to invoke remote Restful web APIs and broadcasts the response through its output port. Java and R are supported in Kepler for programmatic workflow application development.

Kepler is open-source. It can perform type checking at both design-time (static) and run-time (dynamic) on workflows and data. Kepler adopted the “one thread for each task” strategy, in which tasks are run as local Java threads by default, while distributed execution threads are provided via Grids and web services. The Web service support in Kepler allows users to take a WSDL (Web Service Description Language) description and the name of a web service to customize a scientific workflow. The Grid support in Kepler consists of certificate-based authentication, job submission, third-party data transfer, and SRB (Storage Resource Broker), etc. Kepler also supports execution of MapReduce tasks on the Hadoop Master-slave architecture, and the tasks can be executed in batch mode using Kepler’s background execution.

Taverna [71], is a tool suite written in Java, and can help scientists in diverse domains, including biology, chemistry, medicine, etc., to create and execute scientific workflows. Taverna supports workflow execution locally or remotely via WSDL-style web services or RESTful APIs. Taverna system includes a workbench application that provides a GUI interface for composition of workflows, and a Taverna Server that executes remote workflows. Besides desktop GUI support, Taverna also provides a command-line tool for executing workflows from a terminal. Workflows in Taverna are written in an XML-based language called *Scufl2* (Simple conceptual unified flow language). Taverna supports user-interaction with a running workflow within a web browser and has built-in support for myExperiment so that users can browse the myExperiment website within the Taverna Workbench. Users can access the full myExperiment search options and publish their workflows on myExperiment for others to use.

Taverna is open-source. In Taverna, a workflow consists of three main types of entities: *processors*, *data links*, and *coordination constraints*. *processors* take input data and produce a set of output data; *data links* mediate the data flow between a data source and a data sink; *coordination constraints* bind two processors and control their execution to ensure their executions are in a certain order. Workflows can be executed in the *Scufl workbench* using its enactor panel, which allows users to specify their input data for a workflow and launch a local instance of the *Freefluo* enactment engine. The *Freefluo* engine is not tied to any workflow language nor to any execution architecture, thus in effect is decoupled from both the textual form of a workflow specification and the details of a service invocation.

Swift [99], which represents an interesting and distinct category of workflow management and is reviewed below in comparison with other systems presented above. By its nature, Swift is both a general-purpose programming language and a scripting language for distributed parallel scripting. It is used for composing integrated parallel applications/workflows that can be executed on multicore processors, clusters, grids, or clouds. In Swift, the scripts express the execution of constituent programs that consume and produce file-resident datasets. Swift is a compiled language that uses C-like syntax and supports local clusters, grids, HPCs, and clouds. It explicitly declares files and other command-line arguments as the inputs to each program invocation. A focal point in Swift's design is that it provides a simple set of language constructs that regularize and abstract the notions of processes and external data for distributed parallel execution of large application programs.

Swift is open-source. Workflow execution is implicitly parallel and location-independent in Swift. As the number of processing units available on the shared resources varies with time, Swift can exploit the maximal concurrency permitted by data dependencies within the script and the resources available. Swift can use whatever resources available or economical at that moment when the user needs to run a swift application, without the need to continuously reprogram the execution scripts. The implicit parallelism achieved through Swift functions is no necessarily executed in the source-code order but rather based on their input data's availability. Applications should not assume that they will be executed on a particular host, or in any particular order with respect to other application invocations in a script, or whether their working directories will be cleaned up after execution.

Though above modern SWFMSs provides strong support (from different aspects) on composing and executing scientific workflow to solve complex computation-intensive and/or data-intensive scientific problems, none of them genuinely support GPU-enabled DL functionalities in a native workflow from infrastructure level. Such pressing need urges us to develop a SWFMS that can satisfactorily address two major needs of SWFMS community at the same time: 1) fully supports GPU-enabled DL capabilities/tasks which can be integrated with ordinary tasks in a comprehensive workflow, in which some tasks are executed by GPU and other tasks are executed by CPU; 2) fully aligns with the goals, requirements of modern SWFMSs and can adequately address SWFMS challenge in the latest trend of workflow execution infrastructure - clouds. Besides, workflow scheduling algorithms preserve the great potentiality of being leveraged by DL workflows in a SWFMS, for further optimizing the distributed DL workflow executions on variety types of GPU infrastructure (e.g. GPU cluster, GPU cloud).

2.2 ML in SWFMSs

As the technology of ML, especially DL, is fast advancing, the need for ML/DL across all application areas increases even faster [13]. ML/DL has greatly contributed scientific discoveries [80] in numerous disciplines. Along the same line, the need and desire for integrating ML/DL capabilities into SWFMSs raise higher than ever. In this section, I will review major recent related works and make comparison with ours as relevant.

CPU-based ML has been exploited by workflow researchers on improving workflow scheduling algorithms. For example, T.Miu & P. Missier [63] used the C4.5 Decision Tree algorithm on historical data inputs and makespans to train the model for estimating the makespans of scientific workflows on new input data. A. Nascimento et al. [68] promoted application of Reinforcement Learning (RL) and Q-Learning [95] on workflow scheduling, aiming at discovering/learning the best scheduling plan based on the historical executions in absence of a mathematical model. Z.Tong et al. [90] proposed a task scheduling algorithm called QL-HEFT which combines Q-Learning with the HEFT algorithm [91] to reduce the makespan of workflow execution.

Our current work wraps NNWorkflow execution (scheduling) plans in respective NNTrainers/NNExecutors modules to support our DLaaW approach. Though this work have yet to reach the stage of optimizing NNWorkflow scheduling, this could be one of research directions in our future work – i.e., to leverage ML/DL and exploit existing workflow scheduling/optimization algorithms [54] (designed for ordinary workflows) on optimizing NNWorkflows scheduling in the SWFMS.

On the other hand, much work has been done to incorporate CPU-based ML applications in SWFMSs and workflows. A comprehensive survey [24] on CPU-based ML applications in SWFMSs has been made by E Deelman et al. The survey covers most of the representative works published before 2017. Below, I will comment on more recent representative works published after 2018. I Ahmed et al. [4] implemented a semi-supervised clustering-based diagnosis recommendation model in DATAVIEW [44] SWFMS for improving the diagnosing accuracy via self-training and co-training of the model. ML Mondelli et al. [66] proposed the BioWorkbench to manage and analyze bioinformatics experiments in Swift [99] SWFMS by leveraging CPU-based ML using the free software Weka [39]. N Radosevic et al. [79] utilized CPU-based Decision Tree in solar radiation modeling in the KNIME [94] SWFMS to increase reproducibility and warrantability of environmental models. On the other hand, J Herbst and D karagiannis [40] leveraged ML techniques that combines two ML algorithms to induce the structure of sequential workflows and transition conditions, and aim at enabling an inductive approach to workflow acquisition and adaption.

Compare with our approach, while all these works dwell in the realm of leveraging CPU-based implementations, our effort embraces GPU-based ML approach which bears a great advantage – the superior parallelization backed by thousands of GPU cores.

Moreover, as a powerful platform for data- and computation-intensive scientific applications, SWFMSs can benefit the learning and tuning of AI/ML models in multiple ways. For example, Li and Song [49] proposed to couple scientific simulations with unsupervised ML and in-situ (or in-memory) data visualizations, which enables user to monitor and get real-time notifications of special patterns or anomalies from ongoing extreme-scale turbulent flow simulations. On the other hand, J Ozik et al. [73] presented a extreme-scale ML model exploration with Swift/T [101], via parallel scripting and running workflows on variety of computing resources.

As a foreseeable future, SWFMS can be leveraged on optimizing/expediting AI/ML model training in multiple directions. For example, SWFMSs can be used to automate the long and complex training process on any user-specified triggers or conditions. Moreover, the provenance data [84] of a complex training process collected by underlying SWFMS platform can be analyzed and utilized to optimize the ML models, i.e. the knowledge can be mined from the provenance data, such as which inputs/hyperperameters have higher leverage on certain type of output change, may suggest more/less weight updates for certain neurons, resulting in a more enhanced "supervised" learning than training merely based on ground truth values/labels. As provenance data is collected and stored in a SWFMS, the provenance data may potentially enable scientists to tap into the "black boxes" of deep neuron networks, acquire better insights, or even be able to "debug" a particular neuron network in a similar way as we debug a regular programming language project.

2.3 GPU-based DL

General-purpose computing on graphics processing units (GPGPU) is the use of graphics processing units (GPU), which typically handles computation only for computer graphics, in contrast, the computations in traditional applications are performed in central processing unit (CPU) [16, 31, 32]. Due to the privilege of utilizing large numbers of GPU cores at the same, higher level of parallelization can be easily achieved by the ready parallel nature of graphics processing units.

Basically, GPGPU pipeline is a kind of parallel processing between one or more GPUs and CPUs, which analyzes data in the same way as for image or other graphic form. For the sake of GPUs operate at lower frequencies with many times the number of cores, GPUs can process far more pictures and graphical data per second than a traditional CPU. Thus, processing data as a graphical form in GPUs can create a large speedup.

As GPUs have specialized processors with dedicated memory that conventionally perform floating point operations required for rendering graphics, they are optimized for training DL models (especially dealing with large scale/size of datasets) as they can process thousands of computations simultaneously.

Here I will introduce two main GPGPU languages as follow: OpenCL [88] is one of dominant open-source GPGPU languages, which provides a cross-platform GPGPU platform that additionally supports data parallel compute on CPUs. OpenCL is actively supported on Intel, AMD, Nvidia, and ARM platforms. Another dominant proprietary language is Nvidia CUDA [82]. Nvidia launched CUDA in 2006, which provides a software development kit (SDK) and application programming interface (API) that allows using CUDA C/C++ to code and execute on Nvidia GPUs. Programming standards for parallel computing include OpenCL, CUDA, OpenACC [98].

Although above libraries are fully support DL functionalities, these DL capabilities are constrained to be running within their specified platform, and they are not immediately read for use in a SWFMS. Due to above facts, such DL usuability in ad-hoc manner places burden on scientists to adapt their existing work on various of platforms with non-trivial learning curve.

Next, I will introduce several popular DL libraries/frameworks as follow:

TensorFlow [2] is an open source library for numerical computation using data flow graphs, in which nodes represent mathematical operations, while the edges represent the multidimensional data arrays (tensors) that flow between them. It enables computation on one or more CPUs or GPUs in a desktop, server, or mobile device without rewriting code. Besides, TensorFlow offers TensorBoard visualizing TensorFlow results.

PyTorch [75], which is an open source ML library based on the Torch library that used for applications such as computer vision and natural language processing. It was primarily developed by Facebook's AI Research lab (FAIR), and provide two high-level features: 1) Tensor computation (like numpy) with strong GPU acceleration; 2) Deep neural networks built on a type-based automatic differentiation system.

MATLAB [46], which provides DL option for engineers, scientists and domain experts. With tools and functions for neural networks [1], computer vision, and automated driving. MATLAB also enables users to generate high-performance CUDA code for DL and vision applications automatically from MATLAB code.

Though above frameworks provide programmatic interface for higher languages, they mainly narrowed down to Python or Matlab locales (C, Fortran), which does not align with most of Java based SWFMSs [54]. Moreover, they are not readily available to traditional SWFMS users. As a results, scientific processes associate with DL are developed and executed under an external environment in an ad-hoc manner.

On the other hand, interesting works [11, 38, 102] have been done to exscale distributed DL based on existing DL libraries/APIs (as mentioned in above subsection). One particularly interesting project related to workflows is the CANDLE [102] project carried out at Argonne National laboratory. This project aims to develop exascaled DL networks (trained on massive datasets) for accelerating cancer research. CANDLE is built upon

Swift/T [101] which is a well-known SWFMS centered on the Swift language, involving various HPC schedulers to leverage DOE supercomputing resources for exascaling computing tasks. Under the hood, CANDLE distributes time-consuming DL tasks to HPC nodes via Massage Passing Interface (MPI) and leverages Keras (API) to carry out the actual DL executions. CANDLE particularly aims at hyperparameter optimization to identify the most effective DL model implementations and scalable parallel learning where very large data are required. By directly utilizing these Keras API and libraries, developers can immediately gain user-friendly APIs and deployment agility.

However, as the workflow users inevitably need to face the inherited limitations/issues from third-party intermediate level DL APIs in SWFMSs (as pointed out in Chapter 1). To this end, introducing a genuinely GPU-enable support from infrastructure level becomes a great demand for SWFMS community.

Also, there are works [85, 86, 93] dedicated to the construction of unified frameworks/systems for developing AI applications. For example, Bazaar [85], is a such a ML framework for developing ML models and automated ML applications; it introduces its own ML primitives to uniformly leverage different ML/DL libraries (e.g. scikit-learn, Keras, OpenCV) via a unified API and specification for data processing, through which it allows data scientists to efficiently construct and automate a variety of ML applications. However, as its applications may involve multiple ML libraries at the same time, it could be even harder to trace any root cause of low/intermediate-level API errors, which may bring more uncertainties on any performance issue while outsourcing ML/DL execution to various third-party providers. On the other hand, Agora [93], which is a data management system, aims to provide a unified asset ecosystem that goes beyond marketplace and cloud services and provides infrastructure-level support for ML/DL applications. Considering the limitations of surrendering infrastructure-level control to third-party providers, Agora's architectural design [92] includes a unified data management system with infrastructure-level support for AI applications and optimization behind its descriptive syntax. Although Agora is still under construction, the effort is encouraging.

CHAPTER 3 DEEP-LEARNING-AS-A-WORKFLOW (DLAAW) ON SINGLE GPU IN DATAVIEW

Scientific workflow has become a popular cyberinfrastructure paradigm to accelerate scientific discoveries by enabling scientists to formalize and structure complex scientific processes. With the recent success of DL models in many scientific applications, there is a rising need for infrastructure-level support for DL technologies in scientific workflow cyberinfrastructures. However, current scientific workflow cyberinfrastructures and GPU-enabled DL frameworks are developed separately, neither alone can be a satisfactory choice. This work proposes the Deep-Learning-as-a-Workflow approach in DATAVIEW, which for the first time incorporates native infrastructure level support for GPU-enabled DL in a SWFMS and enables the fast training and execution of neural networks as workflows (NNWorkflows) leveraging various types of GPU resource configurations. The experiments demonstrate the salient usability feature of DATAVIEW in providing seamless infrastructure-level support to both scientific and DL workflows in one system, while delivering competitive (better in most cases) learning efficiency compared to the conventional implementations based on Keras.

3.1 Introduction

Scientific workflow modeling and execution has become a common practice for scientists to accelerate scientific discoveries in numerous research fields. The Montage workflow, for example, is used by thousands of astronomers for constructing image mosaics of the sky [26]. In the CyberShake project, more than 230 scientific workflows were used by seismologists to generate seismic hazards maps in one year alone [60]. In Bioinformatics, the myExperiment website currently contains 3935 public scientific workflows shared by

11161 members from 429 groups [33]. The Pegasus workflow system [22] aided the LIGO (Laser Interferometer Gravitational wave Observatory) to successfully detect gravitational wave – a discovery that won the Noble prize!

Meanwhile, in the past few years, ML, especially DL, has become increasingly popular and been utilized in broad scientific processes and projects across nearly all disciplines. Although there are many ML/DL libraries available, such as Keras/TensorFlow [37] and PyTorch [75], they are not immediately ready (not designed) for scientific workflow environments. As a consequence, many ML/DL functionalities, such as architectural design, hyperparameter tuning, and optimizations have to be conducted outside of a scientific workflow system and then integrated into a workflow in an ad-hoc manner [5], which is neither trivial nor optimal as it requires expertise with the ML/DL libraries and the underlying, sophisticated scientific workflow system. Besides, separate handling of DL for data- and/or computation-intensive projects [19] from a SWFMS tends to be time consuming and inefficient in data transfers, which makes the integrated, direct DL support by SWFMS a necessity. Some recent projects like CANDLE [102] leverage the HPC/GPU infrastructure facilitated by workflow platforms to accelerate the model design and training of exascale neural networks, however they simply utilize third-party ML/DL libraries (e.g. Keras in CANDLE) in a loosely coupled way, but not through a deeply integrated (native) approach. Furthermore, such obtained neural networks are very hard to be pipelined into a larger, enclosing scientific workflow. This section address the above limitations and makes the following contributions:

- We propose and implement a novel DLaaW (Deep-Learning-as-a-Workflow) approach in DATAVIEW, more specifically, by extending its workflow and task classes to two new subclasses: $NNWorkflow$ and $NNTask$. This approach is the first (to our best knowledge) that attempts to implement a DL neural network as a *native* workflow in a workflow management system.

- We introduce an NNWorkflow Engine that wraps multi-type of $NNTrainers$, which are responsible for executing NNWorkflows according to specific execution plans (e.g. regular train and test, K-fold cross validation) using various types of underlying GPU resources.
- We implement a generic *GPU Resource Management* module, to leverage various GPU resource configurations. Currently this work provides three options: the local NVIDIA GPU of a host PC, a single NVIDIA Xavier SoM (System-on-Module), and a single NVIDIA Nano SoM, for executing DL workflows in DATAVIEW.

3.2 Challenges of integrating GPU-enabled DL in SWFMSs

Seamless integration of DL capability into SWFMSs brings numerous benefits: 1) the coherent usability of SWFMSs gets extended to DL applications, e.g., the convenient programmatic and graphical design interfaces enjoyed by the scientific workflow community can be made readily available to the design, training, and execution of NNWorkflows; 2) neural network can leverage the same support as offered to ordinary scientific workflows in a typical SWFMS, i.e., supporting neural networks to be constructed, executed and reused in the same manner as ordinary workflows; 3) the rich optimization strategies and scheduling algorithms [54] designed for workflows can be utilized to boost the neural network execution performance. To achieve the above benefits, several major challenges need to be addressed.

3.2.1 NNWorkflows construction Challenge

To construct a neural network as a native workflow in a SWFMS, firstly we need corresponding, well-defined neural network tasks (NNTasks). A scientific workflow is constructed by pipelining various workflow tasks through their input/output ports and executed on available hardware resources by a workflow executor in a SWFMS. Traditionally, scientific workflows are formulated as directed acyclic graphs (DAGs) [55], in which data always flow from entry nodes to exit nodes and each task will be visited (and executed) exactly once. However, a neural network is typically trained through a certain number of epochs, which means each task is revisited multiple times and the weights trained from prior epochs must be retained and updated by subsequent epochs throughout the whole training process. The construction challenge affects how an NNWorkflow is going to be structured and executed. Generally, there are at least two granularity levels for structuring a neural network as a workflow: 1) A-Layer-as-a-Task, and 2) A-Neuron-as-a-Task. The decision can greatly affect the complexity of NNWorkflow construction and implementation.

3.2.2 CPU/GPU communication Challenge

GPUs were originally designed to accelerate graphics rendering, but since the early 2010's, GPUs has been increasingly used to parallelly accelerate computation involving massive amounts of data. As the representation of data in neural networks are tensors and the computation on tensors basically consists of massive repetitive operations on tensor elements, thus modern GPUs are optimized for training DL neural networks to leverage their superior capability in simultaneously running thousands of cores. However, conducting General-Purpose GPU (GPGPU) computation is still rather abstruse due to the substantial difference between CPU and GPU computing in hardware architecture, computing mechanisms, and programming languages [72]. Efficient communication between CPU and GPU becomes a big challenge, which involves 1) bridging a SWFMS with GPU computing since modern SWFMSs are all built upon the CPU infrastructure (i.e. CPU based hardware and operating systems) [54], 2) smoothing the collaboration between CPU and GPU since GPU computing is initiated and coordinated by and finally reduced to CPU.

3.2.3 Challenge of neural network implementation in GPU

Currently, there exist several popular computing platforms and models for GPGPU computing: 1) NVIDIA's CUDA, 2) OpenCL, or 3) OpenACC. They are all focused on providing a unified language and platform to bridge&bind CPU and GPU together for GPGPU computing. Although such parallel computing platforms/models provide higher-level languages than the native hardware languages of CPU&GPU, such programming languages are still considered as low-level APIs for GPU computing. Consequently, the construction and execution of neural networks on any one of above GPU computing platforms remains a great challenge, which includes implementing various neural network layers, constructing the architecture of a neural network, conducting forward&backward propagations across layers, etc. These are all non-trivial issues that need to be carefully addressed in a SWFMS in order to make DL as a readily available functionality for scientific workflows.

3.2.4 CPU and GPU I/O overhead Challenge

In GPU-enabled implementation of NNWorkflows, the input&output ports of NNWorkflow tasks reside on the CPU side, and the input&output of each task (in either neuron or layer granularity) are pipelined into/from the GPU, which inevitably aggregate excessive I/O overhead that is multiplied by the massive number of neurons/layers of a large neural network. The accumulated I/O cost between CPU and GPU can be enormous and overwhelming, which remains as a big stumbling block preventing traditional SWFMSs from leveraging the computing power of GPUs at the infrastructure level. Designing an efficient data transportation mechanism restraining the I/O communication cost to its minimum is another major challenge for implementing GPU-enabled support for NNWorkflows in traditional SWFMSs.

3.2.5 NNWorkflow dynamic mapping Challenge

In order to execute an NNWorkflow as a native workflow on GPU, a mapping mechanism is needed to map the NNWorkflow from CPU-recognizable specification to GPU-recognizable specification. Since a neural network can be composed of arbitrary types, arbitrary numbers and in arbitrary order of neural network neurons and layers, such a mapping mechanism needs to be generic and dynamic so that any native NNWorkflow can be mapped into a corresponding GPU-recognizable specification. Given a native NNWorkflow specification as input, the mapping mechanism must be able to uniformly and consistently output a legitimate GPU execution specification for the NNWorkflow to be executed on corresponding GPU resources. Designing such a generic and dynamic CPU-to-GPU mapping mechanism is yet another major challenge for incorporating GPU enabled DL in traditional SWFMSs.

3.2.6 Challenge of uniformly supporting diverse GPU types

Recognizing the fact that different GPU resources require different computing platforms and execution mechanisms, in order to uniformly execute any NNWorkflow across various types of GPU resources in a SWFMS, all backend GPU APIs should be developed under the same standard protocol. Regardless of the variation of interfaces (e.g. message passing, procedural calls) that bridge CPU with a particular GPU, on receiving the same NNWorkflow specification from an upstream component in SWFMS, all GPU resources should uniformly construct the same neural network and conduct the same execution. Implementing a standardized protocol for various heterogeneous in-house GPU Services is one additional challenge in our way.

3.3 Our Approach and Implementation

In order to implement DL as a native functionality in SWFMSs and to address the challenges outlined in Section 2, based on our prior work on DATAVIEW – an established SWFMS – this work extends DATAVIEW's prior architecture with novel dedicated components. Based on this extended and new architectural design, it particularly emphasizes the following characteristics of inherent implementation of NNWorkflows in DATAVIEW: 1) make the design, execution and reuse of any native NNWorkflow as easy and in the same manner as any ordinary workflow in DATAVIEW; 2) retain the convenience of the current user interfaces (both programmatic and graphical) of DATAVIEW and extend them to facilitate efficient incorporation of NNWorkflows into more complex scientific workflows in DATAVIEW; 3) provide a generic and extensible GPU Resource Management mechanism that allows users to conveniently choose suitable GPU infrastructures (e.g. local GPU, GPU SoMs, GPU cluster, cloud GPU) for their NNWorkflows.

Figure 1 shows the new architecture of DATAVIEW which provides inherent support for DL through the DLaaW approach, which we believe is extendable to other SWFMSs. The original architecture of DATAVIEW consists of the following main components: 1) the *Workflow Design and Configuration* component, which provides intuitive programmatic and

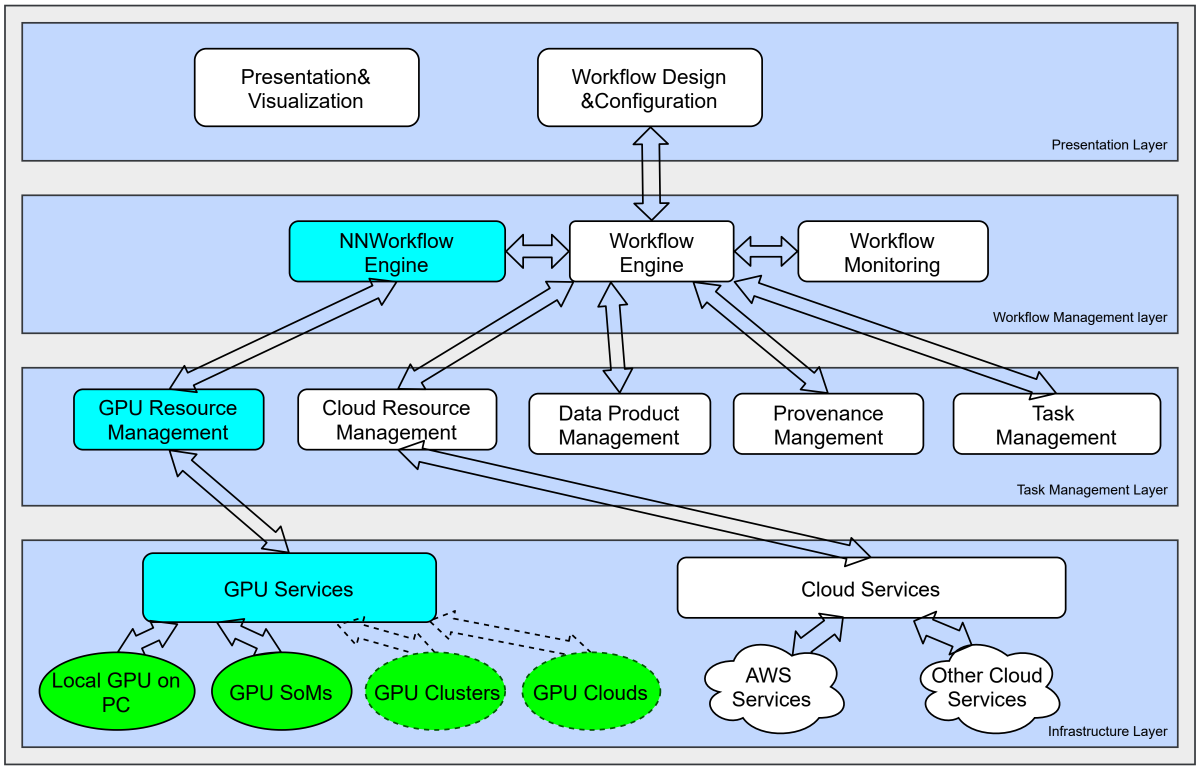


Figure 1: DATAVIEW's new architecture with inherent support for Deep-Learning-as-a-Workflow.

graphical UIs for users to design, execute and reuse workflows; 2) the $Workflow\ Engine$ component, which serves as a central component that controls the execution of workflows; 3) the $Workflow\ Monitoring$ component, which keeps track of the status of workflow execution (e.g. "initialized", "executing", "finished", and "error"); 4) the *Data Product Management* component, which stores all data products that are used/produced by workflows; 5) the *Provenance Management* component, which is responsible for storing, browsing, and querying workflow provenance; 6) the *Task Management* component, which enables the execution of heterogeneous atomic tasks such as calling web services and running scripts; 7) the *Cloud Resource management* component, which plays a key role in provisioning, cataloging, configuring, and terminating the computation resources in clouds.

Built upon the original architecture of DATAVIEW, the new architectural design adds the following new DL-specific components: 1) the $NNWorkflowEngine$ component, which, if the input workflow is an NNWorkflow, takes over the control, parses the NNWorkflow and outputs a pack of GPU recognizable specification to the downstream component; 2) the $GPU\ Resource\ Management$ Component, which, upon receiving the NNWorkflow specification, acts as a unified interface to route the specification to the target GPU services; 3) The *GPU Services* component, which provisions the local GPU of a host PC or GPU SoM resources for actually carrying out the execution of an NNWorkflow.

Listing 1: Construct a sample NNWorkflow with 4 neural network layers:

```
public void design() {
  NNTask[] layers = new NNTask[4];
  layers[0] = new Linear(5,3);
  layers[1] = new ReLU();
  layers[2] = new Linear(3,1);
  layers[3] = new Sigmoid();

  Sequential(layers);
}
```

Initially, an NNWorkflow is designed and constructed through programmatic (Java) or graphical UI as a native workflow in DATAVIEW. More specifically, an NNWorkflow is constructed as an NNTask array that specifies the type and order of each neural network layer in the NNWorkflow. Then, the NNTask array is fed into the Sequential() function. Listing 1 shows a sample of Java code of the design() method in the SampleNNWorkflow class (a subclass of NNWorkflow) to construct a simple NNWorkflow that contains 4 neural network layers: layers[0] is a Linear layer with 5 input neurons and 3 output neurons; layers[1] is a ReLU layer; layers[2] is a Linear layer with 3 input neurons and 1 output neurons; layers[3] is a Sigmoid layer. This sample NNWorkflow shows that the established usability of DATAVIEW [44] is preserved and extended to NNWorkflows, i.e., NNWorkflows are constructed and managed in the same way as traditional workflows.

The constructed NNWorkflow is then fed into an NNWorkflow JSON Mapper module (Written in Java) that maps all constructs and primitives of the NNWorkflow to their neural network counterparts that are recognizable by the backend GPU services. Upon receiving the NNWorkflow specification from the NNWorkflow JSON Mapper module, a specific NNTrainer (written in Java) which is selected by the user (programmatically or graphically), encodes the target GPU resource infrastructure information to the NNWorkflow specification. The aggregated NNWorkflow specification is then fed as input to the *GPU Resource Management* component by calling the train() method of the NNTrainer. Listing 2 is the sample code showing a sample NNWorkflow, w, being fed into two NNTrainers which respectively trigger their train() methods. The NNTrainer_LocalGPU and NNTrainer-crossValOnSingleNano are the two NNTrainers that respectively wrap up two different execution plans and target at two GPU services. For an input NNWorkflow, w, NNTrainer_LocalGPU generates a regular train&test plan for execution on the local GPU of a host PC; on the other hand, NNTrainer_crossValOnSingleNano generates a k-fold cross validation plan for execution on a single NVIDIA Nano SoM. The input dataset is split into 6 batches and the model will be trained through 1000 epochs (see Listing 2). Our implementation satisfactorily addresses Challenge B.1 and enables GPU computing for DL in the SWFMS.

Listing 2: Select and run NNTrainers for the NNWorkflow:

```
NNTrainer_LocalGPU trainer1 = new NNTrainer_LocalGPU(w, 6, 1000);
NNTrainer_crossValOnSingleNano trainer2 = new
    NNTrainer_crossValOnSingleNano(w, 6, 1000);


String result1 = trainer1.train();
String result2 = trainer2.train();
```

Next, the $GPU\,Resource\,Management$ component, which acts as a universal interface/-gateway at the back door on Java side, to route those unified and aggregated specification to the CUDA side. The routing is implemented via necessary interface calls (e.g. JNI, MPI) on GPU API services which are compiled together with our in-house core CUDA implementation to be executed on a targeted GPU resource. All the backend GPU services are designed to accept the uniform execution specification (conforming to an internal standard protocol).

Lastly, the neural network execution plan is initiated on the target GPU resource, and the corresponding neural network object (comprising NNLayer subobjects) is automatically constructed in the GPU's global memory. Our in-house developed CUDA kernels (written in CUDA C++), which are functions executed by GPU, are then triggered in turn (matching the procedural arrangement and order of layers) to finally carry out preprocessing, training and testing of the neural network according to its execution plan.

In our implementation, NNWorkflow specification (in JSON) are routed to a local GPU of a host PC or a GPU SoM via dynamic .dll or static .a API services. A CUDA C++ parser is called to parse the input JSON specification into C++ key-value pair specification, and the API service (by its NNConstructor) will correspondingly construct the neural network in the CUDA environment. Proper memories is then allocated on both CPU (host) and GPU (device) through Memory Allocator according to the size of input dataset and the architectural design of the neural network (e.g., number of layers, weights and bias dimensionalities). In addition, a universal data preprocessing scheme is automatically applied to each input dataset that does the following: 1) eliminates rows with empty values; 2) normalizes the data across all batches per column-wise normalization defined as follows:

$$X_{new} = (X - X_{min})/(X_{max} - X_{min}) \tag{3.1}$$

The actual training process is finally kicked off, going through a number of epochs (defined by user) performing forward and backward propagations, during which respective CUDA kernels are called for each layer (object) to carry out tensor computations on thousands of cores available in the GPU. Once the training process completes, the trained model and the prediction scores are copied from GPU memory to CPU memory, and finally all the results are returned as a JSON object to the NNTrainer (caller) via interface calls (e.g. JNI, MPI). Through the above processing scheme Challenge B and C are successfully addressed. The analytical derivatives of Linear layer, ReLU layer, Sigmoid layer and binary cross entropy in our CUDA GPGPU implementation can be found in Appendix B.

3.4 Experiments

In order to evaluate the proposed approach and implementation, experiments have been conducted to validate its correctness and compare its performance with counterpart python implementations based on Keras in the GPU environment. All Keras-based implementations in our experiments adopt the same structure of python code with variations on neural network architectural designs and input datasets, and all DLaaW implementations are based on the same CUDA code.

This work adopted 4 different settings of CUDA infrastructure, of which some use JNI and some use MPI instead, to train and test 5 neural networks respectively designed for 5 popular binary classification datasets (characterized in Table 3 in Appendix A). These infrastructure settings include i) one Keras-based python implementation on a local GPU of a host PC, and ii) three DLaaW implementations under three different infrastructural settings: local GPU on a host PC, single Xavier GPU SoM, and single Nano GPU SoM, of which the first setting utilizes JNI calls and the last two utilize MPI calls to pass information between JAVA and CUDA.

3.4.1 Hardware and Datasets

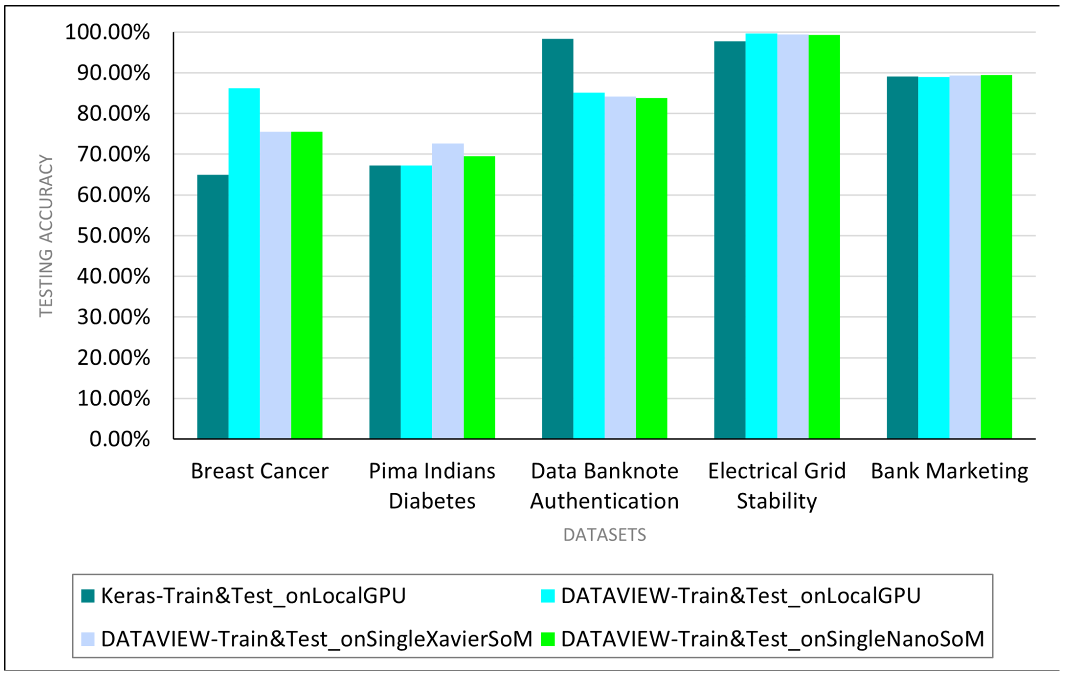
In the preliminary implementation of DLaaW, hardware have been adopted as follow: 1) An x64-based Windows Desktop with AMD Ryzen 5 3600 6-core CPU, 16GB DDR4 RAM, 500GB SSD, one NVIDIA GeForce RTX 2080 Super GPU with 3072 CUDA cores and 8GB GDDRR6 memory; 2) an NVIDIA Jetson Xavier SoM , which contains a GPU with 384 CUDA cores and 48 Tensor cores (Tensor cores are more recent release and more capable on matrix computation compared to CUDA cores), a 6-core NVIDIA Carmel ARM CPU and 8GB LPDDR4 share memory for GPU and CPU; 3) an NVIDIA Jetson Nano SoM , which contains a GPU with 128 CUDA cores, a Quad-core ARM CPU, 4GB LPDDR4 share memory for both CPU and GPU.

5 datasets have been adopted (as showed in Table 3) by considering 1) their popularity,

all the datasets are with high popularity among ML users and scientific researchers, e.g. the Pima Indians Diabetes Database dataset has more than 1 million views and 0.2 million downloads on Kaggle , the banknote authentication Dataset gained more than 0.32 million web hits on UCI ML repository, which is one of the most popular ML repositories with more than 3400 citations [29]; and 2) their diversity, the selected datasets also show good diversity in i) application domains (e.g. bank, medical, electrical), which is important to alleviate potential bias towards any specific application domain; ii) data size, which is important to test scalablity. In the collection of our selected datasets, the Breast Cancer dataset, the Pima Indians Diabetes dataset and the Banknote Authentication dataset are small datasets with less than 1500 instances, while the other two datatsets are relatively bigger, containing more than 10000 instances.

3.4.2 Experiment Results

The results of experiments are showed in Figure 9. The bar charts on top in this figure shows the testing accuracy on each trained model based on Keras and our DLaaW (with various GPU settings). The charts clearly demonstrate the superb prediction accuracy of the 5 neural networks (described in Table 3) implemented (as NNWorkflows) through DLaaW as compared to Keras-based implementations. These NNWorkflows consistently outperform their Keras-based counterparts for 4 of the 5 datasets. The sole exception is with the Data Banknote Authentication that Keras-based implementation delivers higher accuracy. One explanation for this exception could be the use of different data shuffle and partition mechanisms in DATAVIEW and Keras, leading to high data occasional bias [21] on small datasets that may in turn affect the model training. Overall, the accuracy result of the experiment convincingly support the validity of our DLaaW approach and its



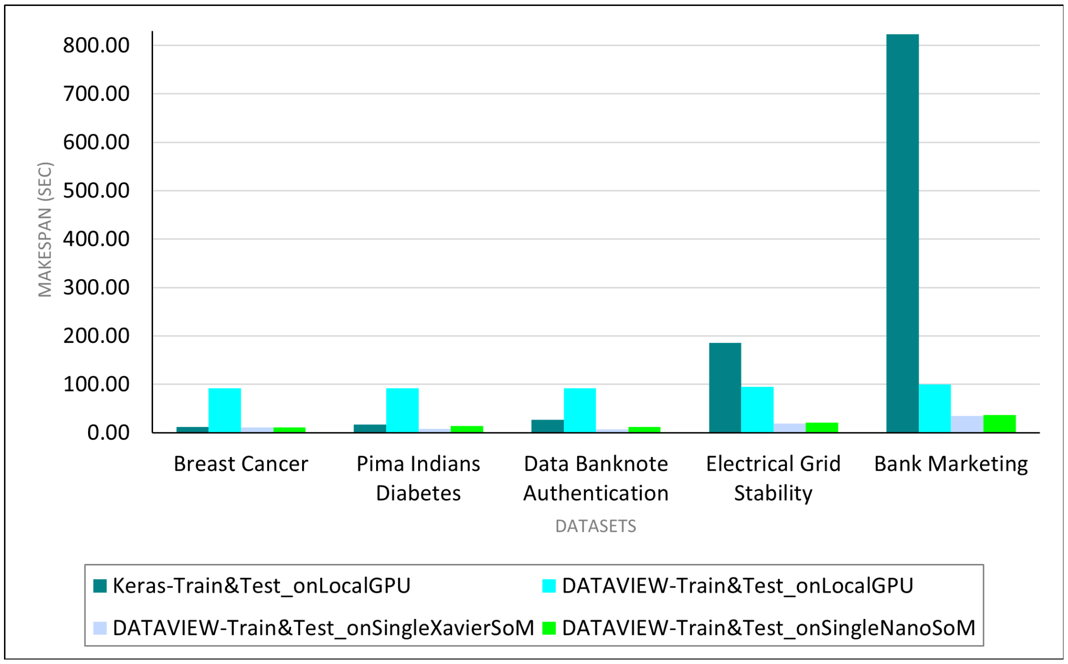


Figure 2: Training models: i) trained models testing accuracies and ii) DLaaW Timespans (in seconds).

competitiveness in comparison to the conventional implementations of neural networks. This result is very exciting as it will function as a cornerstone for our ongoing research that tries to leverage GPU-enabled DL to benefit broad scientific workflows in DATAVIEW.

The bar charts in the bottom of Figure 9 demonstrated the timespans of training and testing on each neural network. The Keras-based implementation on local GPU delivers very swift execution on the first three relatively small datasets. However, with the much bigger 4th and 5th datasets, the execution timespans increase dramatically. This result suggests that Keras-based implementation of neural networks may severely suffer from bad scalability as reported by other developers[1]. The scalability issue of Keras may be due to the inefficient handling of data loading and synchronization between GPU and CPU in its low level CUDA implementation, which aggregate I/O overhead exponentially as the data size increases. In contrast, the NNWorkflows implemented per our DLaaW approach enjoys great scalability. All the NNWorkflows implemented based on our DLaaW show minimum increase in their execution timepsans – almost unnoticeable – across the datasets of varied (increasing) sizes. This is exciting since scability is one of the greatest changes brought up by bigdata to the research community.

Thanks to the Jetson zero-copy mechanism adopted in NVIDIA Jetson SoMs, where CPU and GPU physically share the same system memory so that synchronization overhead can be greatly alleviated, which is adequately exploited in our implementation of the DLaaW approach.

[1]fit_generator slows down when dealing with large dataset,`https://github.com/keras-team/keras/issues/5390`

3.5 Conclusions and future work

In this chapter, it proposes the DLaaW approach which creates, executes and reuses any neural networks as native workflows in a general SWFMS – DATAVIEW. This work makes DATAVIEW the first SWFMS that supports GPU-enabled DL on various GPU resources at the infrastructure level. Through carefully designed comparative experiments with the Keras-based counterpart implementations, it validated our proposed DLaaW approach and the correctness of our various implementations on different GPU resource settings, and demonstrated the effectiveness of our proposed approach and implementation in terms of prediction accuracy and training scalability. As future work, we plan to investigate and incorporate more GPU services, enrich CUDA APIs implementations, and provide DLaaW as an open service for use beyond our own SWFMS – DATAVIEW.

CHAPTER 4 DEEP-LEARNING-AS-A-WORKFLOW (DLAAW) EXTENDED ON HETEROGENEOUS GPU CLUSTER IN DATAVIEW

Scientific workflow has become a common practice for scientists to effectively formalize and structure complex scientific processes, which in turn has accelerated scientific discoveries in numerous research fields. With the recent thriving of DL in broad scientific projects, there is a rising need for DL support in scientific workflow infrastructures - SWFMSs. However, current GPU-enabled DL frameworks are developed separately, not suitable for direct exploitation in SWFMSs, which forces scientists to handle DL outside of SWFMSs and then integrate in workflows in an ad-hoc manner. What workflow users pressingly need today is a user-friendly and well-integrated SWFMS to facilitate GPU-enabled DL as native workflows so that they can conveniently design, train, reuse, and share DL models. In this section, it reports the latest research progress in supporting GPU-enabled DL at infrastructure-level in a popular SWFMS - DATAVIEW, which facilitates: 1) fast design, train and reuse neural networks as native workflows per Deep-Learning-as-a-Workflow (DLaaW) via JAVA API or WebBench GUI; 2) flexibly leverage various types of GPU resources for executing DL workflows. This approach and its implementations are thoroughly evaluated through experiments that demonstrate the efficacy and efficiency as compared to conventional PyTorch-based implementations.

4.1 Introduction

Scientific workflow modeling and execution using a SWFMS has become a common practice for scientists to accelerate scientific discoveries across numerous scientific domains. For example, the national Ecological Observatory Network (NEON) [35] relies on a sensor based data-driven workflow to collect ecological data from sensors across US for studying the ecological processes and changes; the 1000 Genomes project [17] utilizes a bioinformatics workflow to fetch and parse data and to analyze mutation overlaps in humans for the statistical evaluation of potential disease-related mutations. In addition, the Montage workflow [26] has been used by thousands of astronomers for constructing image mosaics of the sky. In the Bioinformatics field, the myExperiment repository currently contains 3935 public scientific workflows shared by 11161 members from 429 groups [33]. The Pegasus workflow system [22] aided the LIGO (Laser Interferometer Gravitational wave Observatory) project to successfully detect gravitational waves – a discovery that won the Noble prize!

Since the past decade, ML, especially DL, has become increasingly popular and been utilized in broad scientific processes and projects across nearly all scientific domains [20] [78]. Thanks to the continued advance in new GPU micro-architectures, DL models can now be trained on very large datasets in accelerated speed, and deliver extraordinary prediction accuracy across broad application disciplines [103]. Although there are many GPU-enabled DL libraries available, such as PyTorch [75], Keras/TensorFlow [37], Theano [8] and Mxnet [14], they are not readily usable in a SWFMS environment. As a consequence, tremendous work such as architectural design, model training, and optimization has to be first carried out outside of a SWFMS and then integrated into a workflow in an inefficient, ad-hoc manner [80], which is neither trivial nor optimal, for the reasons that: 1) it requires expertise with one or more DL libraries and the underlying SWFMS; 2) transferring data between DL models and data-intensive scientific workflows [19] in SWFMS tend to be time-consuming and less efficient; 3) the separate development of DL models and computation-intensive scientific workflows [19] based on completely different platforms tend to be complicated and error-prone.

Therefore, it is very necessary to provide a SWFMS with infrastructure-level support for GPU-enabled DL capability that is natively implemented and seamlessly integrated into the SWFMS. In our previous work [56], it proposed the DLaaW (DL as a Workflow) approach and conducted a feasibility study in the DATAVIEW SWFMS. To our best knowledge, this is the first effort for implementing a DL neural networks as native workflows in an integral SWFMS. More specifically, it introduced an NNWorkflow Engine that wraps up multiple types of $NNTrainers$ for executing any specified NNWorkflow training/execution plans (e.g. regular train and test, K-fold cross validation) on any chosen, particular type of GPU Resources (e.g. Local GPU of a host PC, a NVIDIA SoM (System-on-Module)). Accordingly, it implemented a generic *GPU Resource Management* module to leverage diverse GPU resource configurations and maintain great extensibilty for incorporating any new GPU resource types as they become available in the future. In the preliminary work [56], it focused on the design, construction and execution of NNWorkflows in DATAVIEW via programmatic JAVA API, supporting two types of GPU infrastructures, including the local NVIDIA GPU of a host PC and a single NVIDIA SoM (Xavier or Nano), for executing NNWorkflows in DATAVIEW, in which all of DLaaW implementations delivered very competitive performance compared with Keras-based (counterpart) implementations.

Based on our preliminary exploration [56], this work has made tremendous progress, fully implemented the novel DLaaW approach that was introduced in [56] and thoroughly tested it. Our new implementations allow workflow users/developers to leverage a full life-cycle DL utility to not only design, construct and execute deep neural networks in the form of NNWorkflows, but also reuse previously trained NNWorkflow models on new datasets for prediction in an ordinary workflow, and all of them take place in one integral SWFMS environment - DATAVIEW. Moreover, a heterogeneous GPU cluster as a new type of GPU infrastructure has recently been implemented and incorporated in DATAVIEW. Our newly conducted, more extensive experiments demonstrate not only the efficacy but great advantages of our DLaaW approach. In particular, DLaaW (as implemented in DATAVIEW) allows more adequate exploitation of the high-degree parallelism enabled by the host SWFMS, which in turn significantly boosts DL performance. The DATAVIEW project, supported by multiple NSF grants, is open-source and freely downloadable from Github. The current version - *DATAVIEW Release 3.0* [53], is released at github.com.

Based on the preliminary work [56], the most recent progress as reported in this article makes following additional main contributions:

- We introduce and implement a new $Neural\ Network\ Executor$ module in the $NNWorkflow$ $Engine$ that supports the reuse of any trained NNWorkflow model on new datasets, accomplishing a full life-cycle DL utility – from design to reuse of a native neural network workflow in DATAVIEW.
- We introduce and implement the graphical WebBench GUI to facilitate NNWorkflow design, construction, run, and reuse in DATAVIEW. The appealing intuitiveness of the GUI adds to the usability of DLaaW and DATAVIEW as a whole.
- We introduce heterogeneous GPU clusters as a new type of GPU infrastructure for accelerated training and execution of NNWorkflows, on which we evaluate how well NNWorkflows can leverage the high-degree parallelism offered by a SWFMS in our experiments.

- We conduct the performance comparison on DLaaW implementations and PyTorch-based counterparts (alternative to the Keras-based in our previous work), to assure the validation of this work not only holds for one particular DL library's counterpart implementations.

4.2 Architecture

In order to bring DL as a native functionality into modern SWFMSs and to address the challenges outlined in Section 2 based on the SWFMS – DATAVIEW, this work proposes a new architecture, which is extended from DATAVIEW's prior architecture, with DL-specific components added to the archetecture to give inherent support for NNWorkflows (models) in DATAVIEW. Through this new architectural design, it particularly addresses the following requirements pertaining to NNWorkflows in DATAVIEW: 1) supporting easy design, execution and reuse of any native NNWorkflow in the same manner as any ordinary workflow; 2) extending DATAVIEW's current user interfaces (including both programmatic and graphical interfaces) so that users can conveniently build NNWorkflows as native scientific workflows via user interfaces; 3) implementing a generic and extensible GPU Resource Management mechanism that enables users to conveniently choose a suitable GPU infrastructure (e.g. local GPU, GPU SoM, GPU cluster) for accelerated building, training and reusing of target NNWorkflows and their trained models.

Figure 3 shows the new architecture of DATAVIEW which provides inherent support for DL through the DLaaW approach, which we believe is extendable to other SWFMSs. The original architecture of DATAVIEW consists of the following main components (shown in white rectangles in Figure 1): 1) the *Workflow Design and Configuration* component, which provides intuitive programmatic and graphical UIs for users to design, execute and reuse

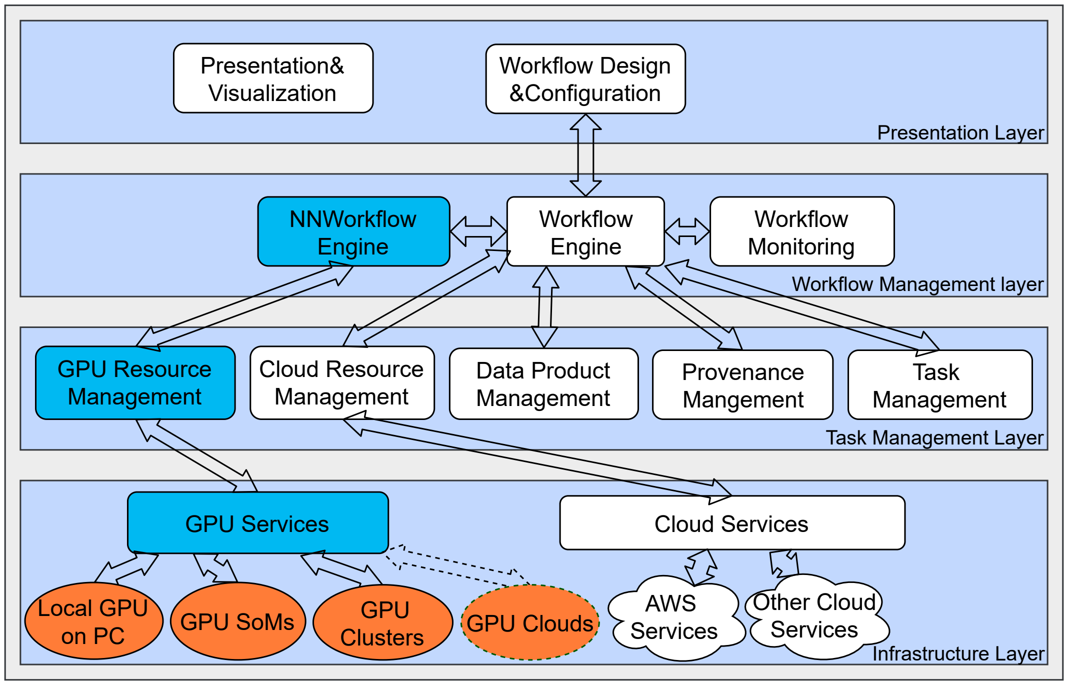


Figure 3: DATAVIEW's new architecture supports GPU clusters in deep-learning-as-a-workflow.

workflows; 2) the $Workflow\ Engine$ component, which serves as a central component that controls the execution of workflows; 3) the $Workflow\ Monitoring$ component, which keeps track of the status of workflow execution (e.g. “initialized”, “executing”, “finished”, and “error”); 4) the *Data Product Management* component, which stores all data products that are used/produced by workflows; 5) the *Provenance Management* component, which is responsible for storing, browsing, and querying workflow provenance; 6) the *Task Management* component, which enables the execution of heterogeneous atomic tasks such as calling web services and running scripts; 7) the *Cloud Resource management* component, which plays a key role in provisioning, cataloging, configuring, and terminating the computation resources in clouds.

Built upon the original architecture of DATAVIEW, the new architectural design adds the following new DL-specific components (shown in light blue rectangles in Figure 1): 1) the

$NNWorkflow\ Engine$ component, which, if the input workflow is an NNWorkflow, takes over the control, parses the NNWorkflow and outputs a corresponding GPU-recognizable specification to the downstream component; 2) the $GPU\ Resource\ Management$ Component, which, upon receiving the NNWorkflow specification, acts as a unified gateway to route the specification to the target GPU services; 3) The *GPU Services* component, which provisions a pool of various GPU resources that actually carry out the execution of an NNWorkflow on an associated GPU infrastructure.

Below, I will elaborate each newly introduced DL-specific component for NNWorkflows and how they help solving the 4 major research challenges (3.1-3.4) in our system.

4.2.1 NNWorkflow Engine Component

In contrast to the traditional workflow engine of DATAVIEW, which consists of two layers accommodating alternative workflow planners and alternative executors for planing and executing workflow schedules, the new $NNWorkflow\ Engine$ component consists of two DL-specific modules: $Neural\ Network\ Trainers$ and $Neural\ Network\ Executors$. The former is responsible for training (and testing) a newly constructed NNWorkflow, the latter reuses the trained NNWorkflow models and applies to new datasets for prediction.

The $Neural\ Network\ Trainers$ module consists of two layers - NNWorkflow Mapper and NNWorkflow NNTrainers, which are dedicated to map (from newly constructed NNWorkflows) and encode GPU execution specification. The NNWorkflow Mapper maps native NNWorkflows to GPU recognizable specification in a specific textual data format. In DATAVIEW, this work adopts the so-called *a-Layer-as-a-Task* construction strategy so that neural network layers are implemented as NNTasks and the connections between the layers naturally make up the data flow. Pragmatically, we believe our choice is the best compromise between the two extremes: a neuron as a task and a whole neural network as a task (from the perspective of incorporating a neural net as a component into an ordinary enclosing scientific workflow). At the lower layer, a corresponding NNworkflow NNTrainer will take the NNWorkflow specification generated by the upper layer and encode the trainer-specific GPU infrastructure information (e.g. type of GPU resources, number of GPU nodes to be used, etc.), and then forward them to the downstream $GPU\ Resource\ Management$ component (detailed in the following subsection).

On the other hand, the $Neural\ Network\ Executors$ module also consists of two layers - NNworkflow Specification Parser and NNWorkflow NNExecutors. The former parses the trained NNWorkflow model (saved as a text file) and generates a GPU-recognizable specification in a textual data format. At the lower layer, a corresponding NNworkflow NNExecutor will take the NNWorkflow specification (including the new dataset to be used) generated by the upper layer and encode the executor-specific GPU infrastructure information into the specification, and then forward them to the downstream $GPUResourceManagement$ component (Detailed in the following subsection).

By constructing neural networks in the granularity of a-Layer-as-a-Task in native NNWorkflows and mapping NNWorkflows to GPU-recognizable specifications, it can satisfactorily address Challenge 3.1 and 3.4 as discussed earlier Section 3.

4.2.2 GPU Resource Management Component

Upon receiving the aggregated NNWorkflow specification from the $NNWorkflowEngine$ component, the $GPUResourceManagement$ component will parse the NNWorkflow per its specification, call a corresponding in-house GPU service and route the NNWorkflow specification to that target GPU service through a unified gateway. Each GPU service stands behind the gateway require a unified specification (including accessible input datasets) to kick off an end-to-end neural network GPU computation. All real-time GPU execution logs (e.g. the GPU execution status, printouts, errors) will be automatically forwarded back to the $NNWorkflow\ Engine$ component in real-time. Once the GPU execution is done, the target GPU service returns the output (e.g. the saved model and testing accuracy) to the $NNWorkflow\ Engine$ component.

As the I/O communication between CPU and GPU is not required at the inter-task/layer level. This arrangement in our implementation significantly reduces the overall I/O cost between CPU and GPU, which in turn helps boost the overall system performance. Challenge 3.3 as mentioned in Section 3 thus is satisfactorily solved.

4.2.3 GPU Services Component

The $GPU\ Services$ component maintains a pool of diverse GPU resources. After a NNWorkflow specification ”fanned out” from the $GPU\ Resource\ Management$ component to a specific GPU resource in the $GPU\ Services$ component, the target GPU service will 1) setup the execution environment, which includes loading and preprocessing of the input datasets, allocating memories on both CPU and GPU sides (Memory synchronization is needed between GPU and CPU in GPGPU computing); 2) construct the neural network (according to the received specification) in GPU's global memory and finally ignite the neural network' execution on the target GPU device.

In our approach, $GPU\ Resource\ Management$ is introduced as a gateway (an intermediate interface) between the $NNWorkflowEngine$ and $GPUServices$ components to gain implementation independence and better future extensibility. The higher layer is built on an abstraction, i.e., a standard interfacing protocol, and all concrete implementations in the lower layer are accordingly aligned to that interface. As long as the abstraction does not change, any change or adding GPU services in the future will not affect the higher level components. As a result, Challenge 3.4 and Challenge 3.5 (discussed in Section 2) are solved.

4.3 Implementation

In DATAVIEW, native DL capability is implemented according to our DLaaW approach (i.e., a DL network as a workflow) and the design choice of a-layer-as-a-task. A number of new architectural components (as shown in Figure 1) are accordingly introduced to support the implementation of our approach. The implementation of these components involves Java, C++, CUDA C++, Java Native Interface (JNI) and Message Passing Interface (MPI). Our system manifests the following important features: 1) NNTasks takes user-definable hyperparameters as input and allows users to either pragmatically or graphically construct NNWorkflows based on their provided hyperparameters; 2) the $NNWorkflow\ Engine$ component encapsulates execution plans (including training, testing, k-fold cross validation, etc.) and infrastructure configuration/settings (e.g. types and number of GPU resources) in NNWorkflow specification; 3) any NNWorkflow (whether a yet to be trained or already trained model) can be uniformly fed into any chosen GPU resource (wrapped in the form of a GPU service) and result in the same execution (training, validation, or prediction) result. Our proposed approach and implementations of DL not only supports scientists to conveniently build and orchestrate any NNWorkflow at any scale, but also aid them to configure suitable execution plans on available GPU resources for particular DLaaWs.

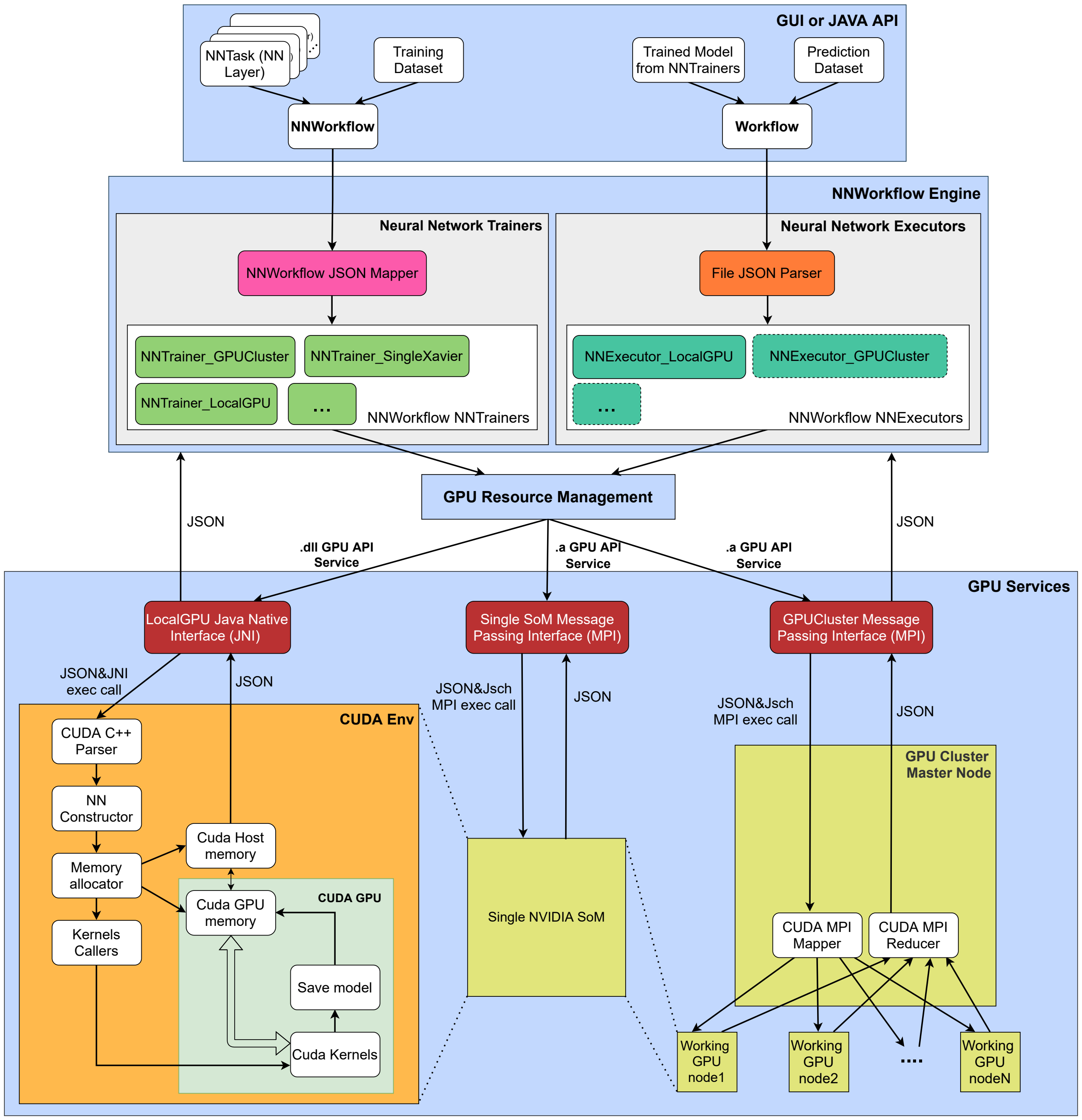


Figure 4: Procedures to construct, execute and reuse a deep-learning-as-a-workflow in DATAVIEW.

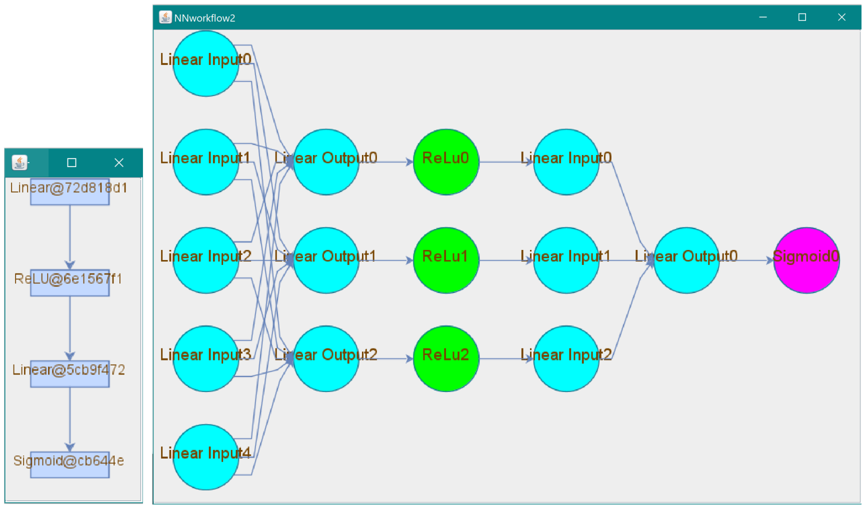


Figure 5: NNWorkflow Visualization by DATAVIEW: i) The sample NNWorkflow and ii) its Neuron-level architecture.

Figure 4 shows the procedural layout of the latest implementation of the DLaaW in DATAVIEW, which lays out the details of the three new components. This section will discuss the implementation details of the DLaaW approach and explain how the 2 major engineering challenges (3.5- 3.6) mentioned in Chapter 4 are addressed in the new implementation.

4.3.1 User Interfaces: GUI and JAVA API for Workflow/NNWorkflow design, construct, run and reuse

In DATAVIEW, all workflows, including NNWorkflows, can be conveniently designed, constructed, run, reloaded/reused through a uniform JAVA API or an intuitive GUI.

Regarding an NNWorkflow, it is constructed layer-by-layer and saved in an NNTask array that specifies the type and order of each neural network layer in the NNWorkflow. The NNTask array is then fed into the Sequential() function along with other training parameters such as how many batches the input data is to be split and how many epochs to be used to train the model. Figure 5 shows the visualization of this sample NNWorkflow

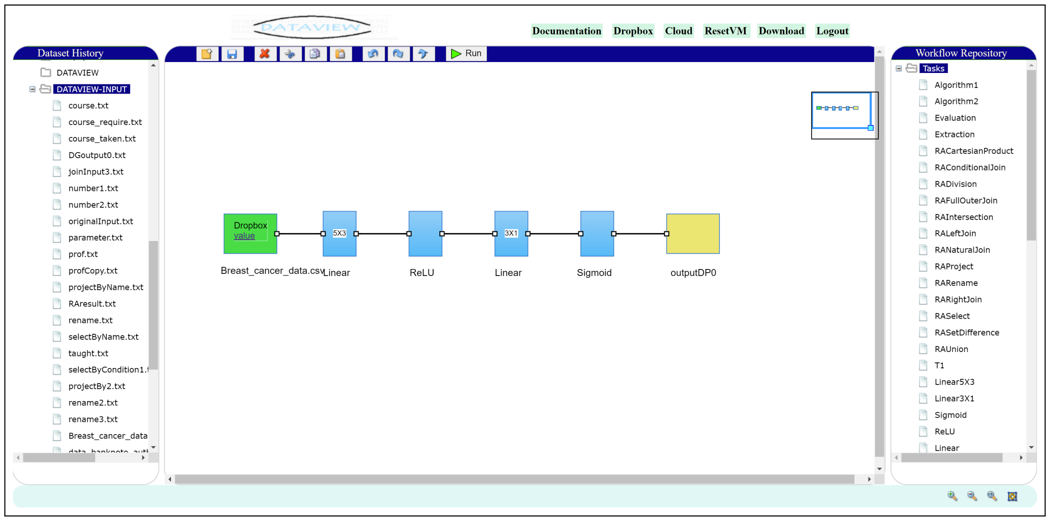


Figure 6: Design and construct an NNWorkflow in DATAVIEW webbench.

produced by the $Presentation\&Visualization$ component in DATAVIEW.

Alternatively, the same SampleNNWorkflow can be designed, constructed and run using DATAVIEW's webbench GUI in a drag-and-drop manner, as showed in Figure 6. The input datasets and Workflow Tasks are pulled from user's dropbox account via dropbox API v2, and they can be dragged into the webbench and can be chained by drawing edges from prior tasks' output ports to subsequent tasks' input ports.

Listing 3: Construct a sample Workflow that reuse the NNWorkflow trained model:

```
wins[0] = new DATAVIEW_BigFile("NNWorkflow@749702");
wins[1] = new DATAVIEW_BigFile("New_dataset.csv");
wouts[0] = new DATAVIEW_BigFile("output.txt");
public void design()
{
  Task stage1 = addTask("NNExecutor");
  addEdge(0, stage1, 0);
  addEdge(1, stage1, 1);
  addEdge(stage1, 0, 0);
```

}

In addition, a DL enabled general scientific workflow (for reusing NNWorkflow trained models) can be constructed in the usual way, except that trained NNWorkflow models are incorporated through the predefined generic $NNExecutor$ task, concatenated with other tasks in the workflow by $Edges$ between the input/output ports of the tasks. Listing 3 shows sample Java Code of the design() method in the NNExecutorWorkflow class, a subclass of *Workflow* in the DATAVIEW system to quickly wrap up trained NNWorkflow model as an ordinary native workflow (or sub-workflow, from the perspective of a larger, enclosing workflow). The sample code takes the input trained NNWorflow model, NNWorkflow@749702 (saved as a text file), and performs intended prediction on New_dataset.csv and saves the result (in JSON format) in the output.txt file as specified. Alternatively, this workflow can be designed, constructed and run using DATAVIEW's graphical webbench UI as showed in Figure. 7. As trained NNWorkflow models are designed, constructed and run as an ordinary workflow in DATAVIEW, they can leverage all existing ordinary workflow executors (for ordinary workflows) in DATAVIEW, including WorkflowExecutor_Beta [65], WorkflowExecutor_Local [7], etc.

The above two samples show that the established usability of DATAVIEW [44] (either programmatically or graphically) via JAVA API or webbench GUI are naturally preserved and extended to NNWorkflows and any workflows subsuming trained NNWorkflow models as components.

4.3.2 NNWorkflow Engine: from native NNWorkflow to GPU recognizable specification

In this subsection, I will elaborate on detailed implementation of $NeuralNetworkTrainers$

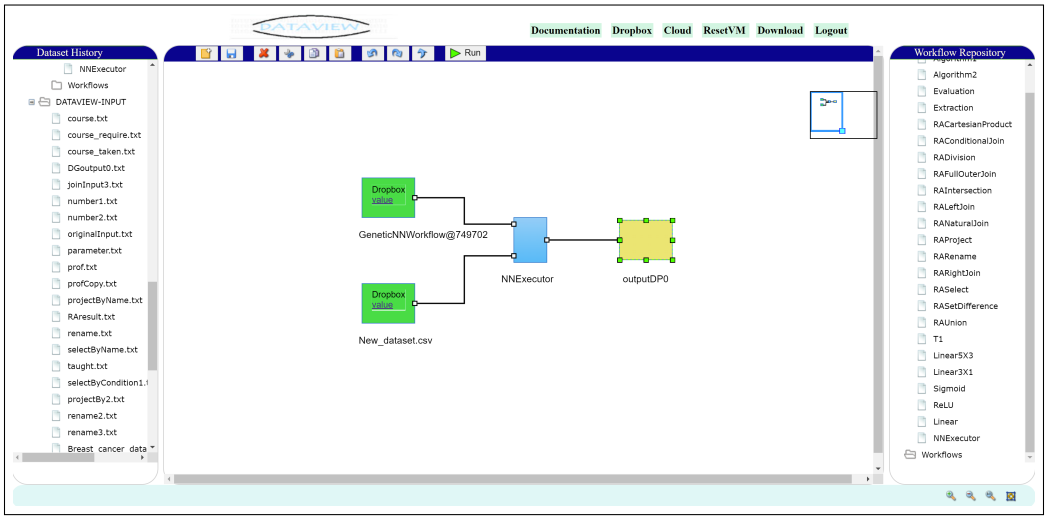


Figure 7: Reuse a trained NNWorkflow model on new dataset for prediction in DATAVIEW webbench.

module and *Neural Network Executors* module.

Neural Network Trainers The NNWorkflow JSON Mapper module (Written in Java) maps all constructs and primitives of the NNWorkflow to their neural network counterparts that are recognizable by the backend GPU services. Listing 4 is the NNWorkflow specification mapped from the sample NNWorkflow in subsection 5.1, which includes: 1) input data repository; 2) output data repository; 3) number of batches to split the input data; 4) number of training epochs; 5) neural network architectural design. At the time of this writing, the NNWorkflow Mapper contains only one specific mapper - NNWorkflow JSON mapper, alternative mappers will be investigated and included in the future.

Listing 4: Sample GPU recognizable specification mapped from native NNWorkflow:

```
"{\"wIpt\":\"C:\\\\Users\\\\DATAVIEW\\\\Downloads\\\\Breast_cancer_data.csv\",
\"wOpt\":\"finalprediction.txt\",
\"numOfBatches\":\"6\",
\"numOfEpochs\":\"1000\",
\"LayerArc\":{\"0\":\"0,5,3\",\"1\":
\"0,3,1\",\"2\":\"2\"}}"
```

One of the NNWorkflow NNWTrainers (written in Java), which is selected by user (programmatically or graphically), will encode the target GPU resource information into the NNWorkflow specification. The aggregated NNWorkflow specification is then sent to the *GPU Resource Management* component by calling the *train*() method in the NNTrainer. Listing 2 is the sample code showing a sample NNWorkflow, w, being fed into two NNTrainers which respectively trigger their *train*() method. Our system currently supports 7 different NNTrainers for choice. The NNTrainer_LocalGPU and NNTrainer_crossValOnGPUCluster are the two NNTrainers that respectively wrap up two different execution plans targeted on two GPU services based on two corresponding GPU resources (configurations). For the same NNWorkflow, w, NNTrainer_LocalGPU creates a train&test plan running on the local GPU of a PC, while NNTrainer_crossValOnGPUCluster creates a k-fold cross training&validation plan running on a GPU Cluster. Our implementation satisfactorily solves Challenge B.1, letting the SWFMS, DATAVIEW, inherently leveraging GPU computing.

Neural Network Executors The File JSON Parser module (Written in Java) reads and parses primitives of trained NNWorkflow models into the counterparts that are recognizable by the backend GPU services. Listing 5 shows the sample Workflow specification mapped from the trained NNWorkflow model - NWorkflow@749702 in subsection 5.1, which includes: 1) input data repository; 2) neural network architectural design; 3) saved model on weight and bias in each layer. At present, our system supports only one specific parser - File JSON Parser. we are considering to support more alternative parsers in future releases of DATAVIEW.

Listing 5: Sample GPU-recognizable neural network specification:

```
"{\"wIpt\":\"C:\\\\Users\\\\DATAVIEW\\\\Downloads\\\\Breast_cancer_data.csv
    \",\"LayerArc\":{\"0\":\"0,5,3\",\"1\":\"1\",\"2\":\"0,3,1\"
,\"3\":\"2\"},\"SavedModel\":{\"0\":{\"bias
    \":\"[44.12920380,−70.366096501,1.17895031]\",
\"weight\":\"[[2.44433546,−2.16660404,4.90515614],
[−2.16514421,−0.49593592,0.22421788],
[1.38725889,−0.33631948,1.02884686],
[0.71579784,−0.51459634,0.55676436],
[0.01803080,−2.92734289,1.12091708]]\"},\"2\":{\"bias\":\"[23.86134911]\",\"
    weight\":\"[3.34796143,−5.95831585,2.21814156]\"}}}}"
```

Currently, DATAVIEW only support one type of NNExecutor - NNExecutor_LocalGPU, which is capable to apply any trained NNWorkflow model to a new dataset to make predictions on local GPU of the host PC. Alternative options (such as single NVIDIA SoM, NVIDIA GPU cluster) will be investigated and integrated in the future. By now, this work has provided a complete life-cycle DL use case (from design to reuse on new datasets) in DATAVIEW via both JAVA API and WebBench GUI.

4.3.3 GPU Resource Management: Universal gateway to route specification to target GPU services

The *GPU Resource Management* component acts as a universal gateway at the back door on the Java side to route consolidated neural network specification to the CUDA side. Such routing is implemented via necessary API calls (e.g. JNI calls, Jsch MPI calls) for target GPU services (e.g. Local GPU Service, GPU Cluster Service). Each API call is compiled together with our in-house core CUDA implementation as a whole (e.g. .dll, .a) to be executed on a targeted GPU resource. All the backend GPU services are designed to accept a uniform execution specification (per an internal standard protocol).

4.3.4 GPU Services: Execute neural networks

Finally, an neural network execution plan is initiated on the target GPU resource where a the corresponding neural network object (comprising NNLayer subobjects) is automatically constructed in the GPU's global memory for execution. Our in-house developed CUDA kernals (written in cuda C++), which are functions executed by GPU, are then triggered in turn (matching the procedural actions and order of layers) to carry out the preprocessing, training and testing of the neural network according to the execution plan. Currently, DATAVIEW supports three types of GPU infrastructure for the execution of NNWorkflows as follow:

A local NVIDIA GPU of a host PC: NNWorkflow specification (in JSON) can be routed to a local NVIDIA GPU of a PC via direct .dll service call. More specifically, first, a CUDA C++ parser needs to parse the input JSON specification into a C++ key-value pair specification, which is then used by NNConstructor to construct the neural network in the CUDA Environment; second, proper memories need to be allocated on both CPU (host) and GPU (device) according to the size of input dataset and architectural design of the neural network (e.g., number of layers and each layer's weights and bias). In addition, a universal data preprocessing scheme is automatically applied to each input dataset that does the following: 1) eliminates rows with empty values; 2) normalizes the data across all batches based on the column-wise normalizor as:

$$X_{new} = (X - X_{min})/(X_{max} - X_{min}) \tag{4.1}$$

The actual training process is finally kicked to start for a user-defined number of epochs through forward and backward propagation, during which respective CUDA kernels are called within layer objects to carry out tensor computations on thousands of cores available in the GPU. Once the training process completes, the trained model and the prediction scores are copied from GPU (device) memory to CPU (host) memory, and all the results, including the training cost for every 100 epochs (for the sake of catching potential overfitting or underfitting), are returned as a JSON object to the NNTrainer_localGPU caller via JNI.

On the other hand, to reuse a trained NNWorkflow model on a given new dataset, the trained model (with its saved model architecture, weights and bias in each layer) gets reloaded to a target GPU resource and the execution of neural network is carried out just like an ordinary workflow except for the leveraging of GPU computing. Once the execution is done, following information will be return to the NNExecutor: 1) device info (e.g. the GPU's available memory, registers); 2) the prediction accuracy.

A single NVIDIA SoM: An NNWorkflow specification can also be routed to a single NVIDIA SoM via Jsch MPI call to .a service. The neural network construction and execution would be in a very similar manner as in a local NVIDIA GPU (as described in 5.4.1). In contrast to the scenario of using a local GPU, message passing between DATAVIEW Java-end and the single NVIDIA SoM leverages MPI instead of JNI. Finally, the single SoM will return the execution results back to the caller (NNTrainer or NNExecuter) on Java side.

A heterogeneous GPU cluster: As the third option, an NNWorkflow specification can be routed to the master node of a heterogeneous GPU cluster via Jsch MPI calls on another static link (.a) API service. The GPU cluster consists of multiple NVIDIA Jetson SoMs. According to the specific execution plan (e.g. distributed k-fold cross validation), the master GPU node accordingly maps and distributes the execution tasks to other (working) nodes via MPI calls. Afterwards, each working node independently handles the rest of the execution of its allocated task in a very similar manner as with a local NVIDIA GPU (as described in 5.4.1). The message passing between cluster nodes also utilizes MPI. The master node must wait until all working nodes complete their work and reduce their results via a CUDA MPI Reducer. Finally, the total result will be passed back to the caller (NNTrainer or NNExecutor) on the Java side.

Through careful design and implementation of CUDA-based GPGPU computing, and integration of diverse message passing mechanisms (through a uniform GPU service interface), the two engineering challenges Challenge 3.5 and 3.6 are satisfactorily addressed.

4.4 Experiments

In order to validate our proposed approach, this work conducted two groups of experiments to compare the performance of implementations on DLaaW with the counterpart python implementations on PyTorch in GPU environment. All PyTorch-based implementations adopt the same structure of python code with variations on neural network architectural designs and input datasets, and all DLaaW implementations are based on the same CUDA C++ code except for the variations of native interfaces in different GPU infrastructures. These two groups of experiments are:

4 infrastructures have been adopted, to train and test 5 neural networks respectively designed for 5 popular binary classification datasets (characterized in Table 3). These infrastructures include i) one PyTorch-based python implementation on a local GPU of a host PC, and ii) three DLaaW implementations under three different infrastructural settings: a local GPU of a host PC, single Xavier GPU SoM, and single Nano GPU SoM, of which the last two settings utilize MPI calls (instead of JNI) to pass information between CPU and GPU.

5 infrastructures have been adopted to conduct 5-fold cross validation on the same neural networks, using the same datasets as showed in Table 3. These 5 infrastructures are: i) one PyTorch-based python implementation on a local GPU of a host PC and ii) four DLaaW implementations on four different infrastructural settings: a local GPU of a host PC, a heterogeneous GPU cluster, single Xavier GPU SoM, and single Nano GPU SoM, of which the infrastructure of GPU cluster utilizes 5 GPU nodes (2 Xavier SoMs and 3 Nano SoMs).

Next, I will provide the details about the hardware and the datasets used for above experiments.

4.4.1 Hardware

In our preliminary implementation of DLaaW in DATAVIEW, the adopted hardware include: 1) A x64-based Windows Desktop as showed on the left side in Figure 8, with AMD Ryzen 5 3600 6-core CPU, 16GB DDR4 RAM, 500GB SSD, one NVIDIA GeForce RTX 2080 Super GPU with 3072 CUDA cores and 8GB GDDRR6 memory; 2) A heterogeneous GPU cluster as showed on the right side in Figure 8, which consists of i) 2 NVIDIA Jetson Xavier

Module and Developer Kits , each contains a GPU with 384 CUDA cores and 48 Tensor cores (Tensor cores are more recent release and more suitable for matrix computation compared to CUDA cores), a 6-core NVIDIA Carmel ARM CPU and 8GB LPDDR4 share memory for GPU and CPU; ii) 4 NVIDIA Jetson Nano Developer Kits , each contains a GPU with 128 CUDA cores, a Quad-core ARM CPU, 4GB LPDDR4 share memory for both CPU and GPU. The specification of each GPU node in the heterogeneous GPU cluster can be found in Figure 25 in Appendix C.

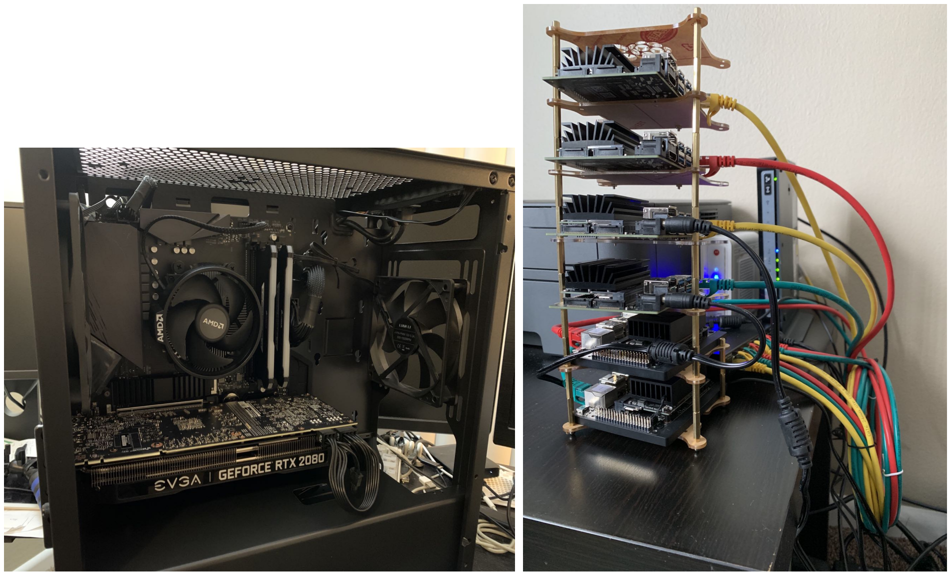


Figure 8: Hardware setup for DLaaW: i) the desktop with a local GPU and ii) a heterogeneous GPU Cluster (6 nodes).

4.4.2 Datasets

5 representative datasets have been adopted (as showed in Table 3) to validate our proposed approach and implementations, including: 1) Breast Cancer Predicton Dataset, which is obtained from the University of Wisconsin Hospitals, Madison, aimed to correlate the abnormal lump (radius, texture, smoothness, etc.) with actual cancerous diagnosis; 2) Pima Indians Diabetes Database, originated from the National Institute of Diabetes and

Digestive and Kidney Diseases for diagnostically predicting whether a patient has diabetes based on certain diagnostic measurements; 3) a banknote authentication data set, in which data are extracted from images via Wavelet Transform tool to evaluate an authentication procedure for banknotes; 4) Electrical Grid Stability Simulated Data Set, which focus on the stability analysis (i.e. stable/unstable) of the 4-node star system (electricity producer is in the center) for implementing Decentral Smart Grid Control concept; 5) Bank Marketing Data Set, in which the data is related with direct marketing campaigns (phone calls) of a Portuguese banking institution and the classification goal is to predict if the client will subscribe a term deposit.

Above datasets were selected by considering their 1) popularity, all the datasets are with high popularity among ML users and scientific researches, e.g. the Pima Indians Diabetes Database dataset has more than 1 million views and 0.2 million downloads on Kaggle, the banknote authentication Data gained more than 0.32 million web hits on UCI ML repository, which is one of the most popular ML repositories with more than 3400 citations [29]; and 2) diversity, the collection of datasets shows good diversity in i) application domains (e.g. bank, medical, electrical), which is important to alleviate the potential bias towards any specific application domain; ii) data size, which is important to test scalablity. In the collection of our selected datasets, the Breast Cancer dataset, the Pima Indians Diabetes dataset and the Banknote Authentication dataset are small datasets with less than 1500 instances, while the other two datatsets are relatively larger, containing more than 10000 instances.

4.4.3 Experiment Results

Experiment Group I: The purpose of this group of experiments is to evaluate our

DLaaW approach and make direct performance comparison of the various implementations of DLaaW with the conventional implementations of neural networks on PyTorch.

The results of this group of experiments are showed in Figure 9. The bar charts on top in this figure shows the testing accuracies on each trained model in DATAVIEW and PyTorch, which demonstrate the superb prediction accuracies of the 5 neural networks (as described in Table 3) implemented through DLaaW as NNWorkflows. Compared to PyTorch-based implementations, NNWorkflows consistently outperform for 4 of the 5 datasets. Within the exception of one dataset, Data Banknote Authentication, PyTorch-based implementation delivers higher accuracies. Since PyTorch's intermediate level APIs (e.g cuDNN, cuBLAS) are closed source, and I implemented our in-house GPU services from scratch based on the pure CUDA, there can be many potential reasons that lead to the different testing accuracies by two approaches on the same neuron network modeling, here I will list out 3 most inclined reasons: 1) our in-house CUDA services embedded its own universal data preprocessing scheme (as described in 5.4.1), which can potentially improve the input data quality and deliver better trained models; 2) though both approaches adopts Xavier weights initialization [48], the weights are randomly initialized by mean 0 and uniform/standard deviation, which can result in the same neural network being trained to different local optimals (in a non-convex optimization problem); 3) I was introducing a small constant (e.g. exp(-10)) in analytical derivative of binary cross entropy, to avoid the illegitimate case of $log(0)$, the absence/difference of such small constant may lead to slightly different backward propagations. The results in terms of the accuracy convincingly support the validity of our DLaaW approach through its competitive prediction accuracies in comparison to conventional implementation of neural networks, which serve as a cornerstone for our ongoing research that tries to leverage GPU enabled DL and provide the functionality to broad scientific workflows in DATAVIEW.

The bar charts in the bottom of Figure 9 demonstrated the timespans of training and testing on each neural networks. The PyTorch-based implementation on local GPU delivers very swift execution on the first three relatively small datasets. However, with the larger 4th and 5th datasets, the execution timespans increase dramatically. This phenomenon suggests that the PyTorch-based implementation of neural networks may suffer from bad scalability (in terms of data size), which is also caught in our Keras-based implementation and by other Keras developers in [56]. This scalability issue is also reported by other PyTorch developers[2,3]. Sharing this same issue in Keras and PyTorch may suggest the root cause can be in their commonly used intermediate level APIs (e.g. cuDNN, cuBLAS). In contrast, NNWorkflows implemented per our DLaaW approach in DATAVIEW enjoy great scalability. The scalability issue of PyTorch may due to the inefficient handling of data loading and synchronization between GPU and CPU in its low level CUDA implementation, which aggregate I/O overhead in exponential rate as the data size grows larger. Across the different datasets of varied sizes, the DLaaW implementations on local GPU shows pretty consistent but higher than timespans of other DLaaW implementations. Based on our best knowledge, this may likely suggest that communication through JNI is less efficient than communication through MPI. On the other hand, the timespans obtained from DLaaW implementations on single Xavier and single Nano GPU SoMs are the shortest, with only very moderate increase as the data size increases. This result likely indicates that our in-house

[2]PyTorch official discussion #95247,`https://discuss.pytorch.org/t/training-fast-with-small-dataset-slow-with-large-dataset/95247`

[3]PyTorch issue #2829 in Github.com,`https://github.com/pytorch/fairseq/issues/2829`

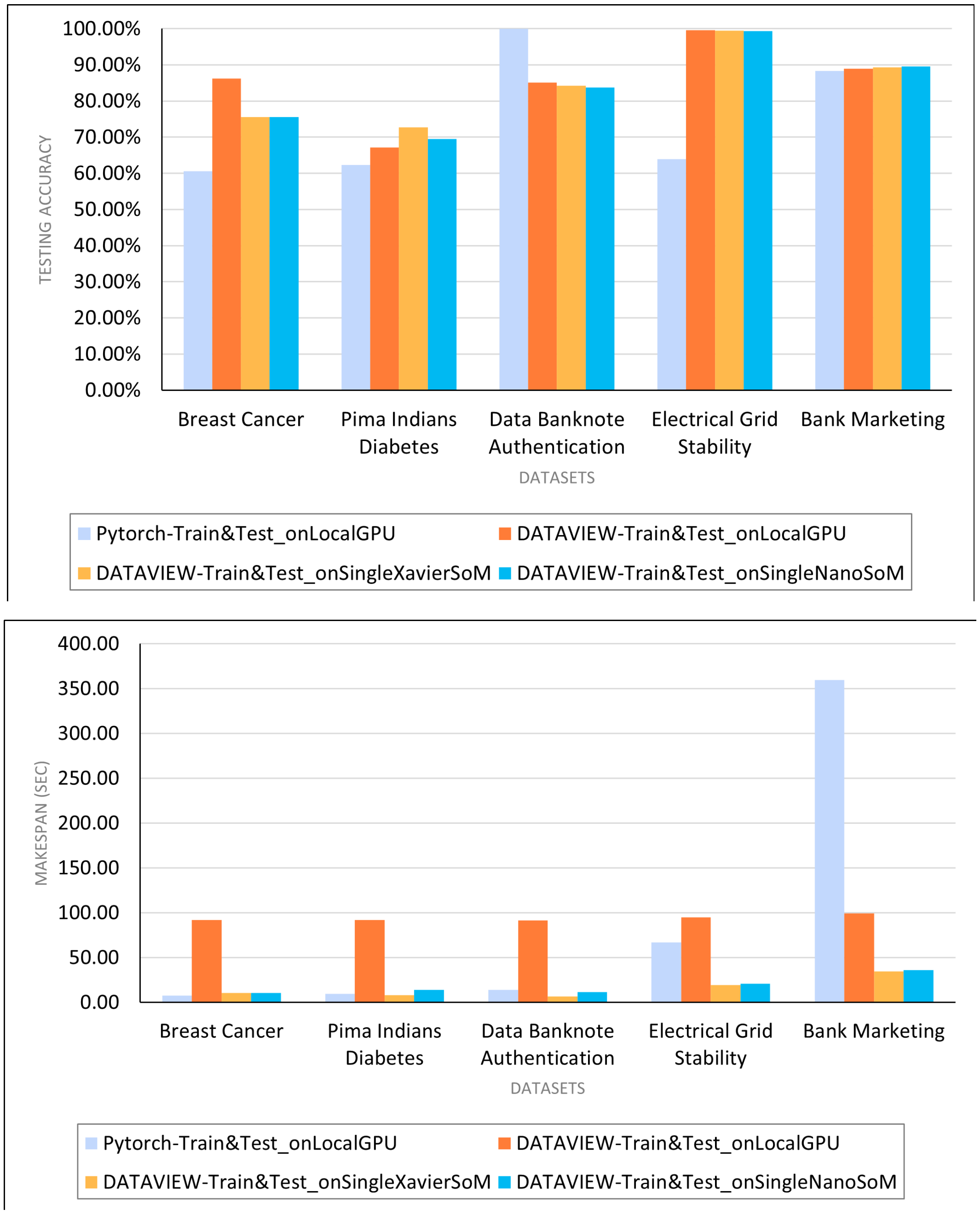


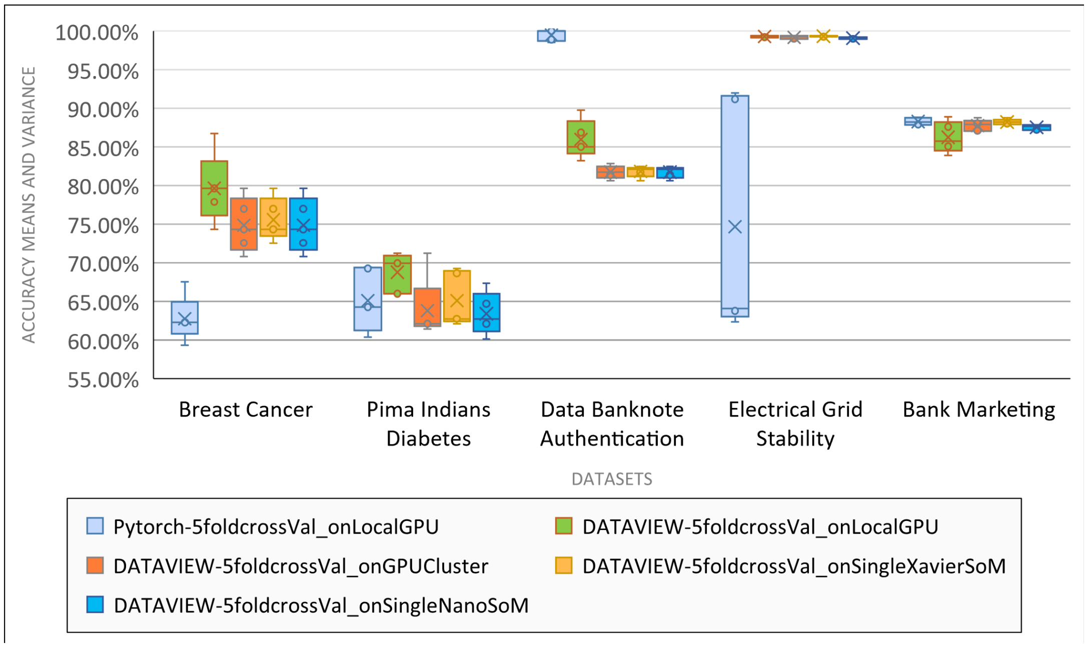
Figure 9: Regular train and test: i) Testing accuracies and ii) Timespans (in seconds)

CUDA implementation does a much better job on low level data loading/synchronisations, meanwhile thanks to the Jetson zero-copy mechanism[4] adopted in NVIDIA Jetson SoMs, where CPU and GPU physically share the same system memory so that synchronization overhead between CPU and GPU can be greatly alleviated. The obtained timespans also indicate that NNWorkflow run on a single GPU SoM leads much better scalability with regard to increased dataset sizes.

Experiment Group II: The purpose of this group of experiments is to evaluate how well NNWorkflows can leverage the high-degree parallelism offered by a SWFMS and boost the execution performance of neural networks compared to the serialized implementations. Thus, this work conducted 5-fold cross validation through various implementations on five neural networks. These implementations can be further put into two categories: 1) serialized, which includes PyTorch-based implementation on local GPU, and three DLaaW implementations respectively on the local GPU, single Xavier GPU SoM and single Nano GPU SoM. These four implementations are sequentially conducting the 5-fold cross validation; 2) parallelized, which includes the DLaaW implementation on the GPU cluster. With this implementation, the five folds of validation are distributed to five working GPU nodes, and the final results are aggregated at the the master node. The memory tuilization on each GPU node in the GPU cluster during the 5-fold cross validation can be found in Figure 26 in Appendix D.

The results of this group of experiments are showed in Figure 10. The top chart shows the means and variances of validation accuracies on each neural network across 5-fold cross validation. All DLaaW implementations in DATAVIEW consistently deliver competi-

[4]Jetson Zero-Copy, `https://www.fastcompression.com/blog/jetson-zero-copy.htm`



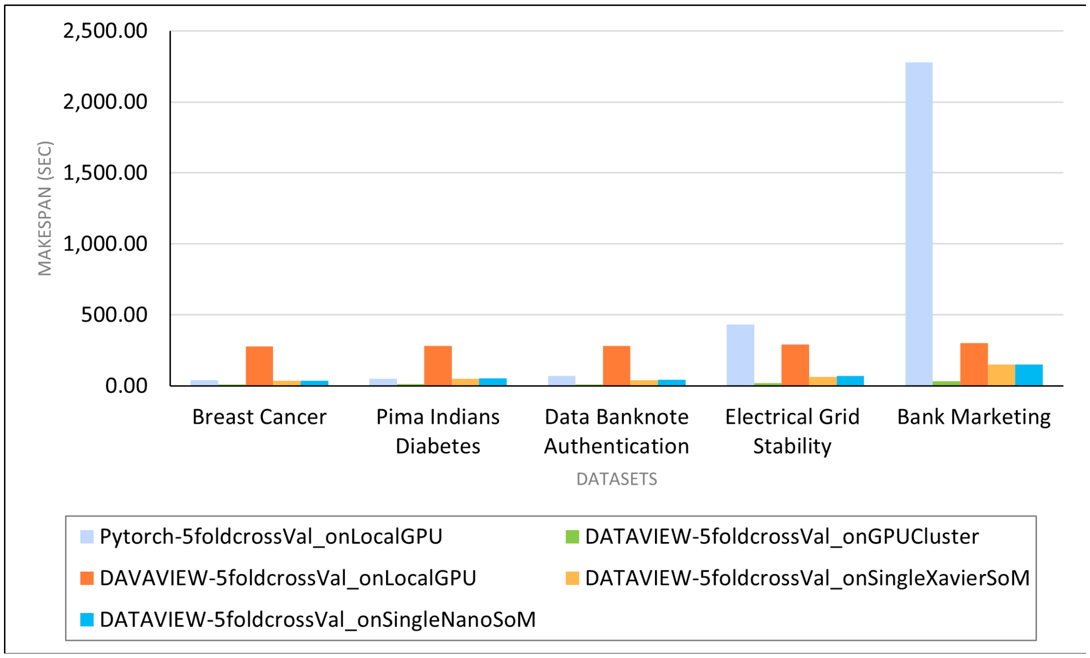


Figure 10: 5-fold cross validations: i) Means and Variances of testing accuracies on 5 folds; ii) Timespans (in seconds)

tive or higher average accuracies in comparison to the PyTorch-based Python implementation, which echos the conclusion from the first group of experiments from the accuracy perspective. On the other hand, all DLaaW implementations preserves moderate or low variance of accuracy across different folds of validation, which demonstrate our approach delivers relatively more robust models in contrast to PyTorch-based implementation (which outputs noticeable large variance across different folds' accuracies in the 4th dataset). Besides, we observe a larger variance of 5 folds' accuracies in small datasets (1st and 2st) than larger datasets (4th and 4th) across all infrastructural settings, which is likely to suggest the data bias [21] exerts higher impact on smaller datasets while conducting train and test on different portion of data.

The bottom chart in Figure 10 shows the timespans of 5-fold cross validation on all these neural networks. The experiment results concur on the conclusions in the first group of experiments: 1) poor scalability (regarding data size) in PyTorch-based implementation, which runs very fast in the first 3 datasets but slows down dramatically on the 4th and 5th larger datasets; 2) eminent JNI overhead in the DLaaW implementation on local GPU compare with other implementations through MPI; 3) high efficiency and superb scalability in DLaaW implementations on single SoM, even though the five folds of validation are carried out sequentially. Lastly, coming to the key purpose of this group of experiments, the timespans obtained from the DLaaW implementation on a GPU cluster are remarkably smaller than all the sequential implementations. This result clearly demonstrates that our DLaaW approach, besides all the virtues mentioned above, can gracefully leverage the high-degree parallelism offered by a traditional SWFMS without any extra effort.

4.5 Conclusions and future work

This section proposes an advanced version of DLaaW approach which constructs, executes and reuses any neural networks as native workflows via either Java API or graphical webBench interface in a general SWFMS – DATAVIEW. This work makes DATAVIEW the first SWFMS that supports infrastructure-level GPU-enabled DL. Currently, DATAVIEW support 4 types of GPU resources for DLaaW, including the local NVIDIA GPU on a PC, a single NVIDIA Xavier SoM, a single NVIDIA Nano SoM and a heterogeneous GPU cluster consisting of multiple NVIDIA SoMs. Through two groups of carefully designed experiments, it validated our proposed DLaaW approach and the correctness of its implementations on different GPU infrastructures, and demonstrate the effectiveness and efficiency (in terms of prediction accuracy and timespan) by comparing with the counterpart Python implementations on the cutting-edge DL library - PyTorch. It also shows DLaaW can gracefully leverage the superior parallelism offered by SWFMS on boosting the DL performance. Moreover, it demonstrates a clean and user-friendly interfaces (via either Java API or WebBench interface) to SWFMS community through a complete life-cycle use case demo. As future work, we plan to investigate and incorporate more GPU services, enrich CUDA APIs implementations, and provide DLaaW as an open service for use beyond our own SWFMS – DATAVIEW.

CHAPTER 5 THE USABILITY OF DEEP-LEARNING-AS-A-WORKFLOW (DLAAW) TO THE SWFMS COMMUNITY

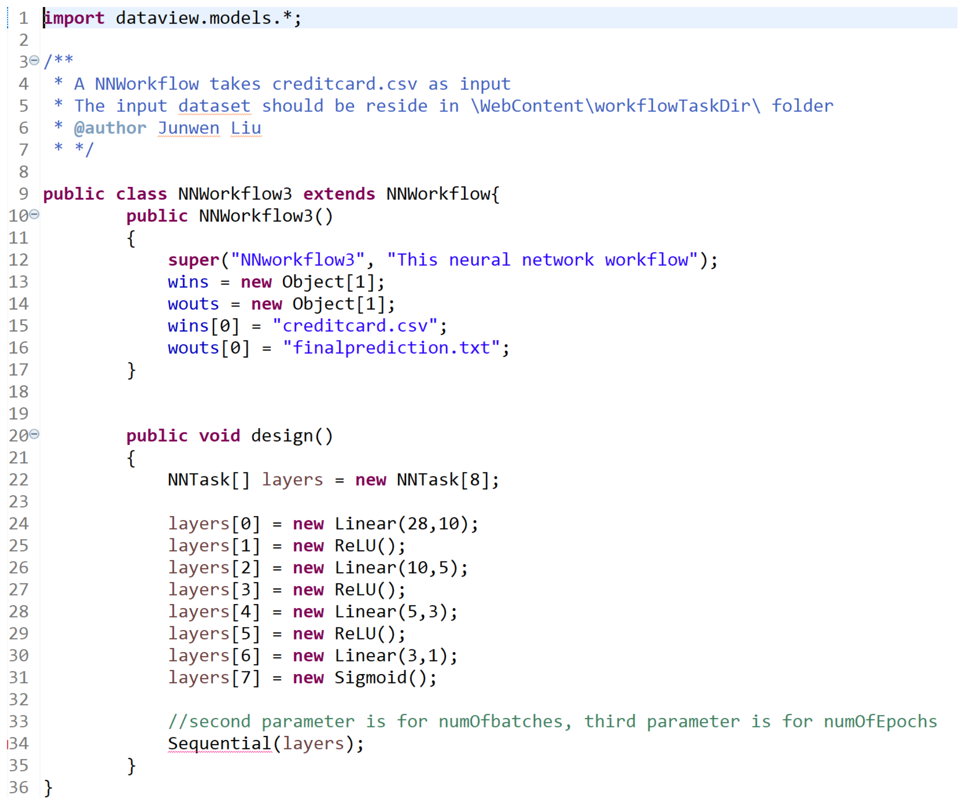
In this Chapter, I will demonstrate the usability of DLaaW in DATAVIEW to the SWFMS community, which will cover following aspects: 1) the process of designing, constructing and executing an NNWorkflow through JAVA API or graphical web interface, and 2) the process of reusing any trained NNWorkflow model on new datasets in JAVA API or graphical web interface; 3) Also, I will discuss the extensibility of DLaaW approach in DATAVIEW by providing and analyzing its up-to-date core JAVA class diagrams of the NNWorkflow Engine and GPU Resource Management components, as well as the core CUDA class diagram in our GPU services.

5.1 Design, construct and execute an NNWorkflow in DATAVIEW

DATAVIEW provides two options (programmtically&graphically) for workflow users to design, construct and execute their NNWorkflows, either approach will deliver the equivalent operation on any particular combination of NNWorkflow, execution plan (e.g. regular train and test, K-fold cross validation) and GPU infrastructure (e.g. local GPU, GPU SoMs) in the backend CUDA services.

5.1.1 Design, construct and execute an NNWorkflow through JAVA API

Figure 11 is an example of designing and constructing an NNWorkflow through JAVA API in DATAVIEW. This sample NNWorkflow consists of 8 layers as an NNTask array, in which layers[0] is a Linear layer with 28 input neurons and 10 output neurons; layers[1] is a ReLU layer; layer[2] is a Linear layer with 10 input neurons and 5 output neurons; layers[3] is a ReLu layer; layer[4] is a Linear layer with 5 input neurons and 3 output neurons; layers[5] is a ReLu layer; layers[6] is a Linear layer with 3 input neurons and 1



```
import dataview.models.*;

/**
 * A NNWorkflow takes creditcard.csv as input
 * The input dataset should be reside in \WebContent\workflowTaskDir\ folder
 * @author Junwen Liu
 * */

public class NNWorkflow3 extends NNWorkflow{
        public NNWorkflow3()
        {
            super("NNworkflow3", "This neural network workflow");
            wins = new Object[1];
            wouts = new Object[1];
            wins[0] = "creditcard.csv";
            wouts[0] = "finalprediction.txt";
        }


        public void design()
        {
            NNTask[] layers = new NNTask[8];

            layers[0] = new Linear(28,10);
            layers[1] = new ReLU();
            layers[2] = new Linear(10,5);
            layers[3] = new ReLU();
            layers[4] = new Linear(5,3);
            layers[5] = new ReLU();
            layers[6] = new Linear(3,1);
            layers[7] = new Sigmoid();

            //second parameter is for numOfbatches, third parameter is for numOfEpochs
            Sequential(layers);
        }
}
```

Figure 11: Design and construct an NNWorkflow through JAVA API

output neuron; layers[7] is a Sigmoid layer.

Figure 12 is the micro-observation of the corresponding neural networks constructed in figure 11. On the top of this figure, it demonstrates each line of JAVA code correspondingly and ultimately construct the corresponding part of neural network.

Figure 13 is the example of executing the sample NNWorkflow through JAVA API, in which NNWorkflow6 is the target NNWorkflow to be executed. The NNWorkflow-Trainer_LocalGPU is the NNTrainer that selected for executing the target NNWorkflow in 1000 epochs and the input dataset will be split into 6 batches. Ultimately, by triggering the train() method, it will provision the target GPU resource and carry out the actual neural network execution.

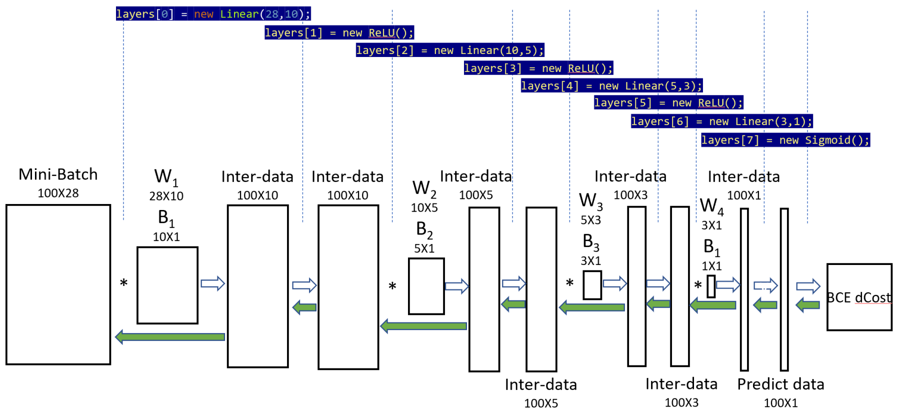


Figure 12: Micro observation of the sample NNWorkflow which designed and constructed through JAVA API

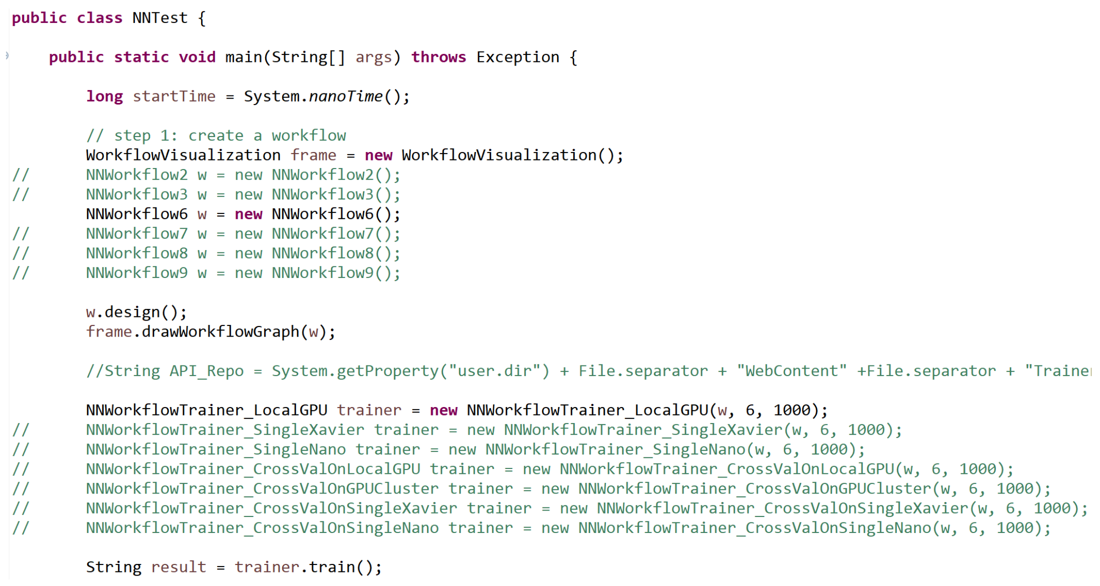

```
public class NNTest {

    public static void main(String[] args) throws Exception {

        long startTime = System.nanoTime();

        // step 1: create a workflow
        WorkflowVisualization frame = new WorkflowVisualization();
//      NNWorkflow2 w = new NNWorkflow2();
//      NNWorkflow3 w = new NNWorkflow3();
        NNWorkflow6 w = new NNWorkflow6();
//      NNWorkflow7 w = new NNWorkflow7();
//      NNWorkflow8 w = new NNWorkflow8();
//      NNWorkflow9 w = new NNWorkflow9();

        w.design();
        frame.drawWorkflowGraph(w);

        //String API_Repo = System.getProperty("user.dir") + File.separator + "WebContent" +File.separator + "Traine

        NNWorkflowTrainer_LocalGPU trainer = new NNWorkflowTrainer_LocalGPU(w, 6, 1000);
//      NNWorkflowTrainer_SingleXavier trainer = new NNWorkflowTrainer_SingleXavier(w, 6, 1000);
//      NNWorkflowTrainer_SingleNano trainer = new NNWorkflowTrainer_SingleNano(w, 6, 1000);
//      NNWorkflowTrainer_CrossValOnLocalGPU trainer = new NNWorkflowTrainer_CrossValOnLocalGPU(w, 6, 1000);
//      NNWorkflowTrainer_CrossValOnGPUCluster trainer = new NNWorkflowTrainer_CrossValOnGPUCluster(w, 6, 1000);
//      NNWorkflowTrainer_CrossValOnSingleXavier trainer = new NNWorkflowTrainer_CrossValOnSingleXavier(w, 6, 1000);
//      NNWorkflowTrainer_CrossValOnSingleNano trainer = new NNWorkflowTrainer_CrossValOnSingleNano(w, 6, 1000);

        String result = trainer.train();
```

Figure 13: Execute the sample NNWorkflow through JAVA API

5.1.2 Design, construct and execute an NNWorkflow in Web Interface

Figure 14 is an example of designing and constructing an NNWorkflow in web interface in DATAVIEW. This sample NNWorkflow is constructed in 4 layers by dragging and dropping the modules (e.g. dataset, NNTasks, output file) from the right or left panels, these modules are stored and retrieved from user's dropbox account. In this neural network, layers[0] is a Linear layer with 5 input neurons and 3 output neurons; layers[1] is a ReLU layer; layer[2] is a Linear layer with 3 input neurons and 1 output neurons; layers[3] is a ReLu layer; layer[4] is a Sigmoid layer.

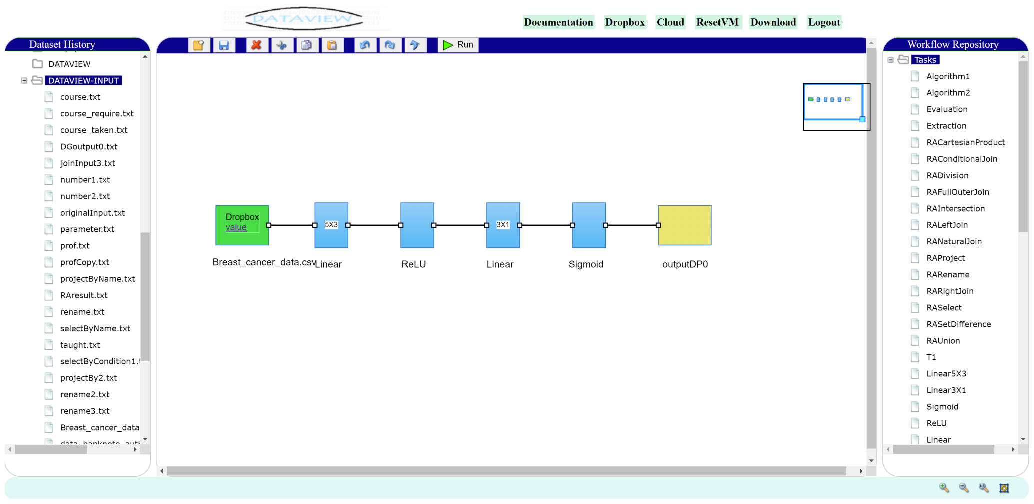
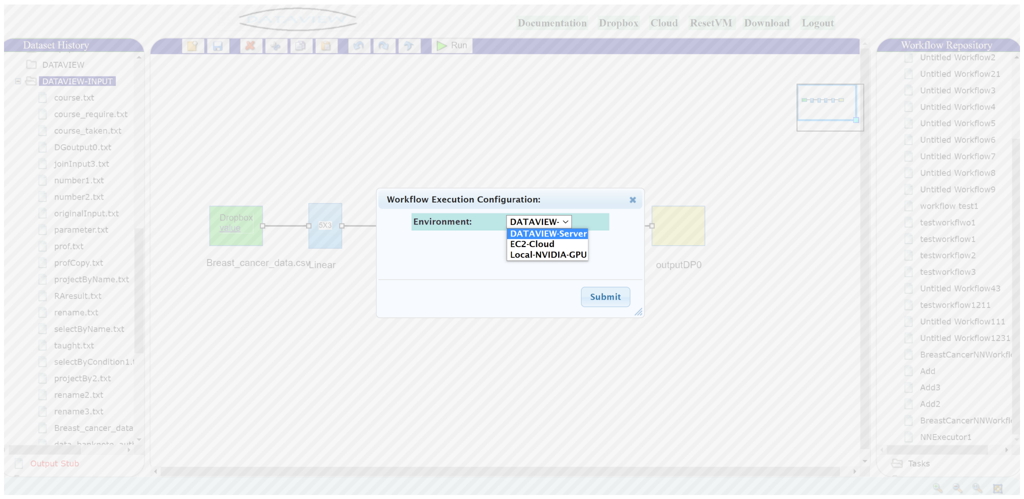


Figure 14: Design, construct and exectuion an NNWorkflow in Web interface

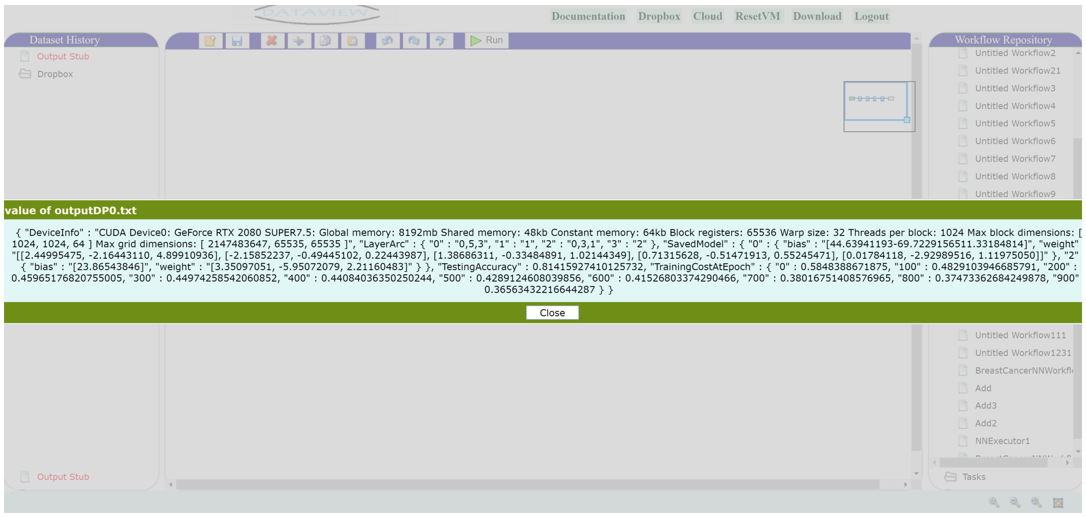
By clicking the "Run" button on the top of middle panel in the web interface, a popup window will show up for workflow users to select their proper execution environment (as showed in figure 15). Currently we provide 3 options of execution environment in the web interface: 1) The DATAVIEW-Server provides the local workflow execution environment, which will execute the workflow in the CPU multi-threading environment of the host PC; 2) The EC2-Cloud provides the AWS EC2-Cloud workflow execution environment to execute the target workflow; 3) The Local-NVIDIA-GPU provides the Local GPU execution enviroment, which will execute the NNWorkflow in the local NVIDIA GPU of the host PC.

Figure 16 shows the sample NNWorkflow execution result (in JSON fomat) returned from the backend GPU service (i.e. Local GPU of a host PC). The results includes: 1) DeviceInfo, which shows the details of the targeted GPU service; 2) LayerArc, which stores details of the trained model, including architectural design, weights and bias, etc; 3) TestingAccuracy, which shows the testing accuracy of the trained model on the last batch of input


Workflow Execution Configuration:

Environment: DATAVIEW-Server

Figure 15: Execute the sample NNWorkflow in Web interface


value of outputDP0.txt

```
{ "DeviceInfo" : "CUDA Device0: GeForce RTX 2080 SUPER7.5: Global memory: 8192mb Shared memory: 48kb Constant memory: 64kb Block registers: 65536 Warp size: 32 Threads per block: 1024 Max block dimensions: [
1024, 1024, 64 ] Max grid dimensions: [ 2147483647, 65535, 65535 ]", "LayerArc" : { "0" : "0,5,3", "1" : "1", "2" : "0,3,1", "3" : "2" }, "SavedModel" : { "0" : { "bias" : "[44.63941193-69.7229156511.33184814]", "weight"
"[[2.44995475, -2.16443110, 4.89910936], [-2.15852237, -0.49445102, 0.22443987], [1.38686311, -0.33484891, 1.02144349], [0.71315628, -0.51471913, 0.55245471], [0.01784118, -2.92989516, 1.11975050]]" }, "2"
{ "bias" : "[23.86543846]", "weight" : "[3.35097051, -5.95072079, 2.21160483]" } }, "TestingAccuracy" : 0.81415927410125732, "TrainingCostAtEpoch" : { "0" : 0.5848388671875, "100" : 0.4829103946685791, "200" :
0.45965176820755005, "300" : 0.44974258542060852, "400" : 0.44084036350250244, "500" : 0.42891246080398560, "600" : 0.41526803374290466, "700" : 0.38016751408576965, "800" : 0.37473362684249878, "900"
0.36563432216644287 } }
```

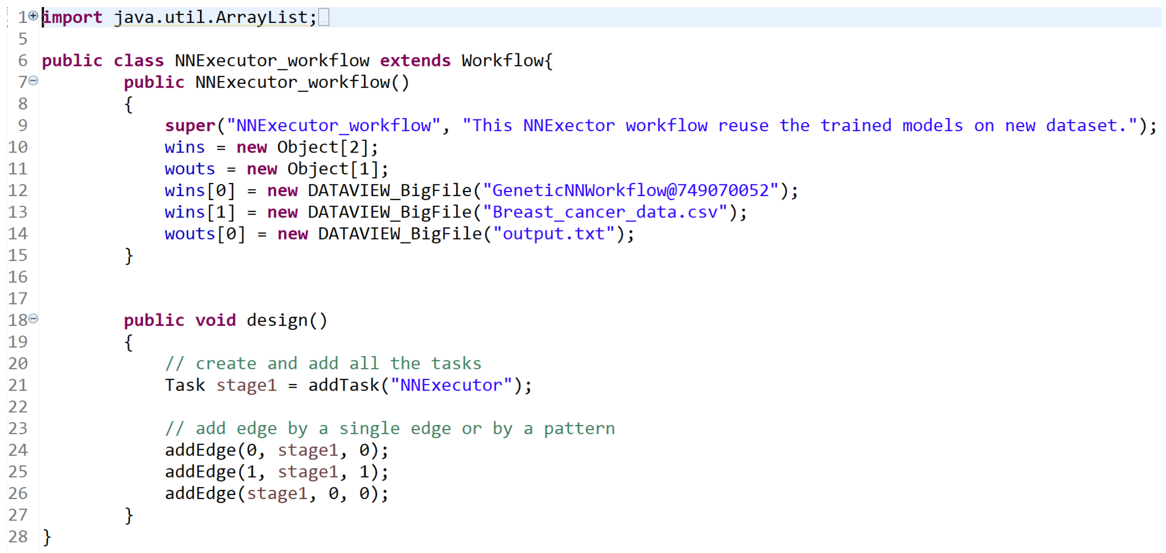
Figure 16: The result of the sample NNWorkflow execution in Web interface

dataset; 4) TrainingCostAtEpoch, which shows the training cost at every 100 epochs.

The execution result (in JSON format) will be automatically saved as a file, in which the saved model (neural network architecture, weights and bias, etc.)can be directly reused by the NNWorkflow executor (constructed as an ordinary Task, which will be elaborated in the following subsection) in an ordinary workflow in DATAVIEW.

5.2 Reuse any trained NNWorkflow models on new datasets in DATAVIEW

Similarly, DATAVIEW provides two options (programmtically&graphically) for workflow users to reuse any NNWorkflow trained model on new datasets in DATAVIEW, either approach will deliver the equivalent operation in DATAVIEW and carry out the same execution for a trained model in the target CUDA service.



```java
import java.util.ArrayList;

public class NNExecutor_workflow extends Workflow{
        public NNExecutor_workflow()
        {
            super("NNExecutor_workflow", "This NNExector workflow reuse the trained models on new dataset.");
            wins = new Object[2];
            wouts = new Object[1];
            wins[0] = new DATAVIEW_BigFile("GeneticNNWorkflow@749070052");
            wins[1] = new DATAVIEW_BigFile("Breast_cancer_data.csv");
            wouts[0] = new DATAVIEW_BigFile("output.txt");
        }


        public void design()
        {
            // create and add all the tasks
            Task stage1 = addTask("NNExecutor");

            // add edge by a single edge or by a pattern
            addEdge(0, stage1, 0);
            addEdge(1, stage1, 1);
            addEdge(stage1, 0, 0);
        }
}
```

Figure 17: Reuse a trained NNWorkflow model on new dataset through JAVA API

5.2.1 Reuse any trained NNWorkflow on new datasets through JAVA API

Figure 17 shows a sample ordinary workflow constructed for reusing a trained NNWorkflow model through JAVA API. The trained NNWorkflow model (in JSON format) is saved in the direct output from subsection 5.1, which is the auto generated output file from NNWorkflow execution and named as GeneticNNWorkflow@749070052. This workflow includes a single ordinary task – NNExecutor, which is a regular Task that forward the trained neural network model (in JSON format) to NNWorkflowExecutor service, which will execute the trained model with new dataset on the target GPU resources (specified in the config.json file in DATAVIEW project).

5.2.2 Reuse any trained NNWorkflow on new datasets in Web Interface

Similar to JAVA API, figure 18 shows a sample ordinary workflow constructed for reusing a trained NNWorkflow model through dragging and dropping manner, user can drag and drop modules (e.g. dataset, input file that stores trained model) from left-/right side panels and construct the target workflow in web interface. In this figure, the trained NNWorkflow model (in JSON format) is one of input files named GeneticN-NWorkflow@749070052, which was the direct output from the subsection 5.1 (i.e. the output file of NNWorkflow execution). The NNExecutor is a regular Task that forward the trained neural network model (in JSON format) to NNWorkflowExecutor service, which will execute the trained model with new dataset on the Local NVIDIA GPU of a PC.

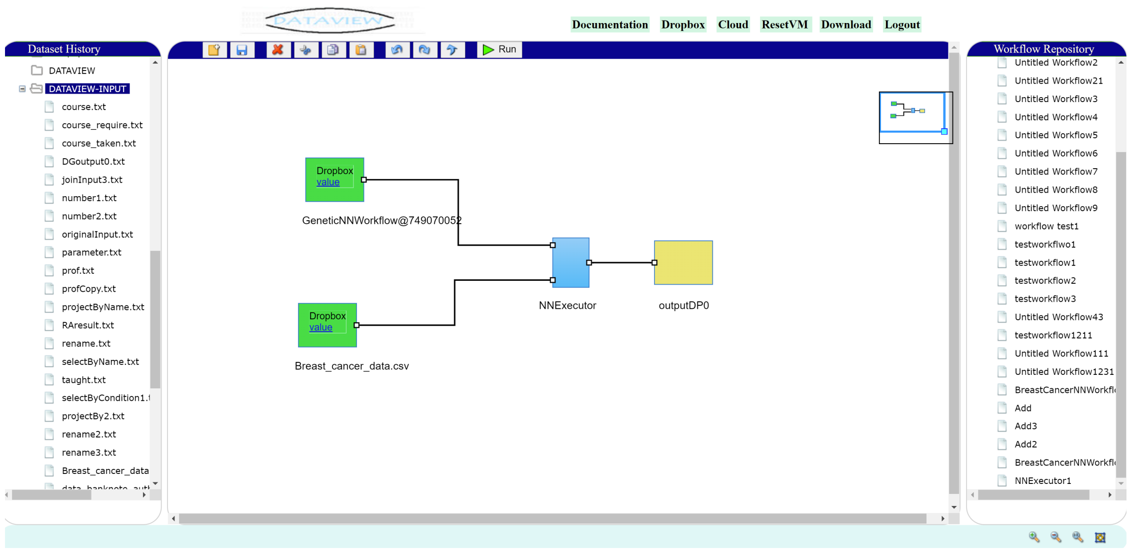
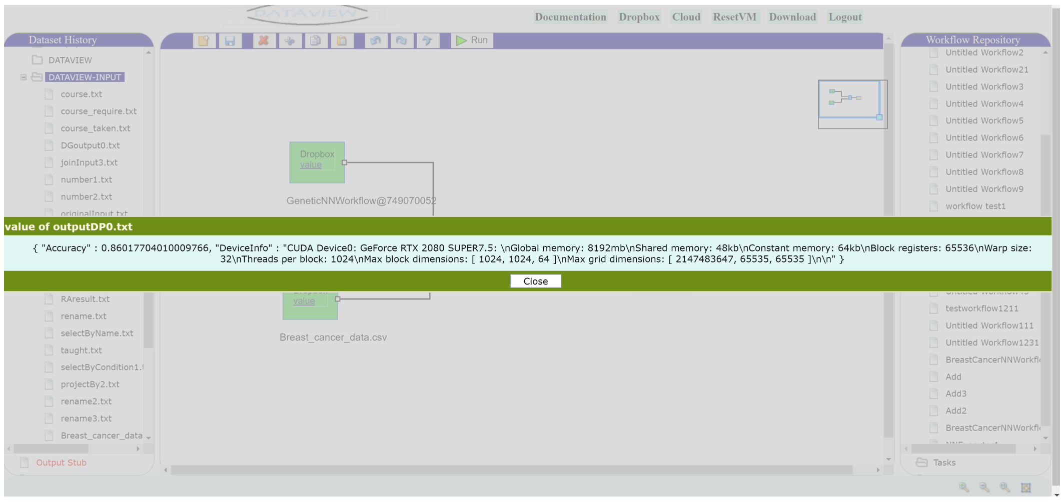


Figure 18: Reuse a trained NNWorkflow model on new dataset in web interface

Thanks to the NNExecutor was constructed as an ordinary Task in DATAVIEW and its constructed workflow is an ordinary workflow (instead of NNWorkflow), the workflow can be executed by an ordinary workflow execution environment (e.g. the first and second type of environments as described in figure 15), as long as the environment (e.g. local PC, cloud) equiped with an NVIDIA local GPU.



value of outputDP0.txt

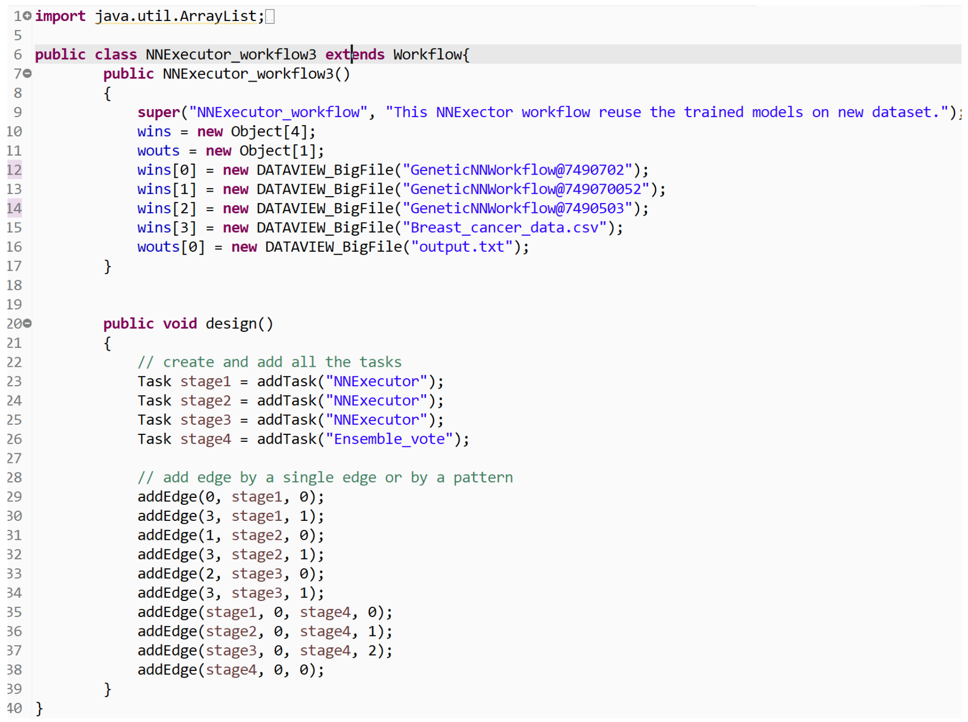
```
{ "Accuracy" : 0.86017704010009766, "DeviceInfo" : "CUDA Device0: GeForce RTX 2080 SUPER7.5: \nGlobal memory: 8192mb\nShared memory: 48kb\nConstant memory: 64kb\nBlock registers: 65536\nWarp size:
32\nThreads per block: 1024\nMax block dimensions: [ 1024, 1024, 64 ]\nMax grid dimensions: [ 2147483647, 65535, 65535 ]\n\n" }
```

Figure 19: Execution result of reusing a trained NNWorkflow model on new dataset in web interface

Figure 19 shows the execution result by reusing the above mentioned NNWorkflow trained model (GeneticNNWorkflow@749070052) on the new dataset. The result (in JSON format) will be saved in a text file under the DATAVIEW project's working directory, and include the information as follow: 1) prediction accuracy, which is predicted by the trained model on the new dataset; 2) DeviceInfo, which shows the information of device (e.g. a PC) running the target workflow.

5.3 Integrate NNTasks and ordinary Tasks in one comprehensive workflow

To further exploit NNTasks as a part of one comprehensive workflow, which consists of both NNTasks and ordinary Tasks, DATAVIEW provides two options (programmtically&graphically) for workflow users to integrate them in one comprehensive workflow. In this section, I will present a simple use case of **Ensemble Learning** on neural networks, which includes 3



```
import java.util.ArrayList;

public class NNExecutor_workflow3 extends Workflow{
        public NNExecutor_workflow3()
        {
            super("NNExecutor_workflow", "This NNExector workflow reuse the trained models on new dataset.");
            wins = new Object[4];
            wouts = new Object[1];
            wins[0] = new DATAVIEW_BigFile("GeneticNNWorkflow@7490702");
            wins[1] = new DATAVIEW_BigFile("GeneticNNWorkflow@749070052");
            wins[2] = new DATAVIEW_BigFile("GeneticNNWorkflow@7490503");
            wins[3] = new DATAVIEW_BigFile("Breast_cancer_data.csv");
            wouts[0] = new DATAVIEW_BigFile("output.txt");
        }


        public void design()
        {
            // create and add all the tasks
            Task stage1 = addTask("NNExecutor");
            Task stage2 = addTask("NNExecutor");
            Task stage3 = addTask("NNExecutor");
            Task stage4 = addTask("Ensemble_vote");

            // add edge by a single edge or by a pattern
            addEdge(0, stage1, 0);
            addEdge(3, stage1, 1);
            addEdge(1, stage2, 0);
            addEdge(3, stage2, 1);
            addEdge(2, stage3, 0);
            addEdge(3, stage3, 1);
            addEdge(stage1, 0, stage4, 0);
            addEdge(stage2, 0, stage4, 1);
            addEdge(stage3, 0, stage4, 2);
            addEdge(stage4, 0, 0);
        }
}
```

Figure 20: Design, construct and run a neural network ensemble learning workflow through JAVA API

weak NNWorkflow classifiers, these classifiers are pre-trained NNWorkflow models on different portion of the breast cancer dataset. By leveraging these weak classifiers at the same time and independently outputting their own predictions, the final prediction can be derived through a voting process by choosing the majority votes from 3 classifiers.

5.3.1 Design, construct and run neural network ensemble learning workflow through JAVA API

Such Neural Network ensemble learning workflow can be designed and constructed through JAVA API (as showed in Figure 20). This comprehensive workflow has 1) 4 inputs, including three pre-trained NNWorkflow models (GeneticNNWorkflow@7490702, @749070052, @7490503) and one dataset - Breast_cancer_data.csv; 2) 4 Tasks, including three NNExecutor Tasks that are going to be executed in the GPU environment, and one ordinary Ensemble_Vote Task, which is executed in the CPU environment; 3) and one

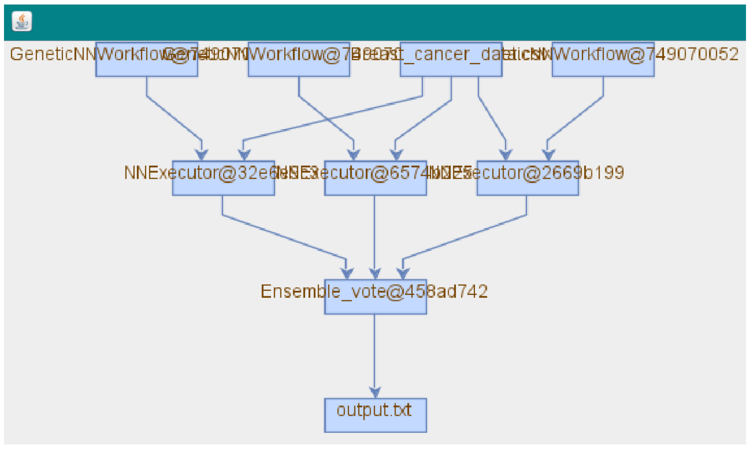


Figure 21: JAVA visualization of the neural network ensemble learning workflow

output file (output.txt). Figure 21 shows the visualization of this comprehensive workflow when run it on a proper workflow executor (e.g. workflowExecutor_Beta, worklfowExecutor_Local) in DATAVIEW. The intermediate outputs from NNExecutors will be pipelined as inputs for the Ensemble_Vote Task (as showed in Table 27).

5.3.2 Design, construct and run neural network ensemble learning workflows in web Interface

Similar to JAVA API, figure 22 shows how this comprehensive workflow can be constructed through a dragging and dropping manner, user can drag and drop either NNTasks or ordinary Tasks to the working panel and connect their input/output ports via edges. The constructed Neural Network Ensemble Learning workflow will be executed as a regular workflow, such that all regular workflow executors (WorkflowExecutor_local, WorkflowExecutor_Beta) can be utilized. The execution result of this NN ensemble learning workflow is showed in the figure 23, in which each 1/0 represents the voted outcome (based on the 3 weak classifiers' independent predictions) for a particular row/record in

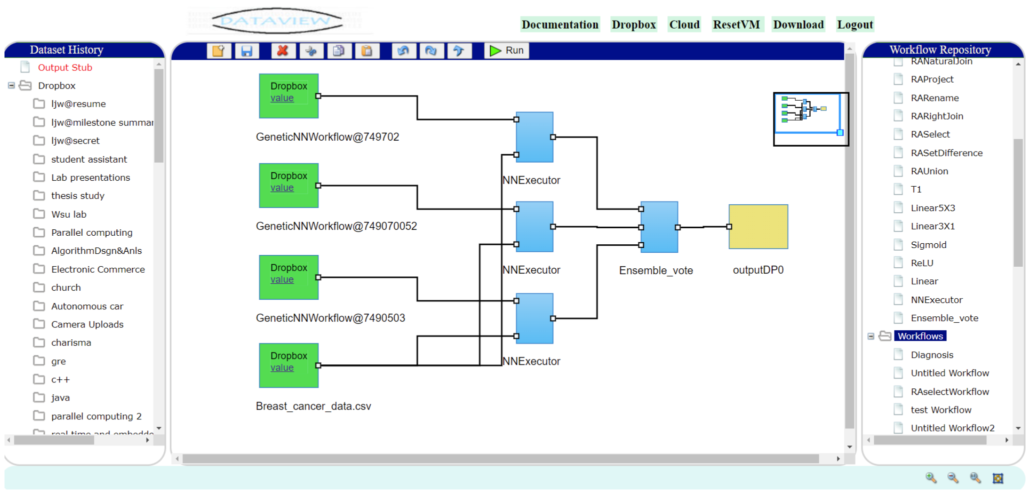
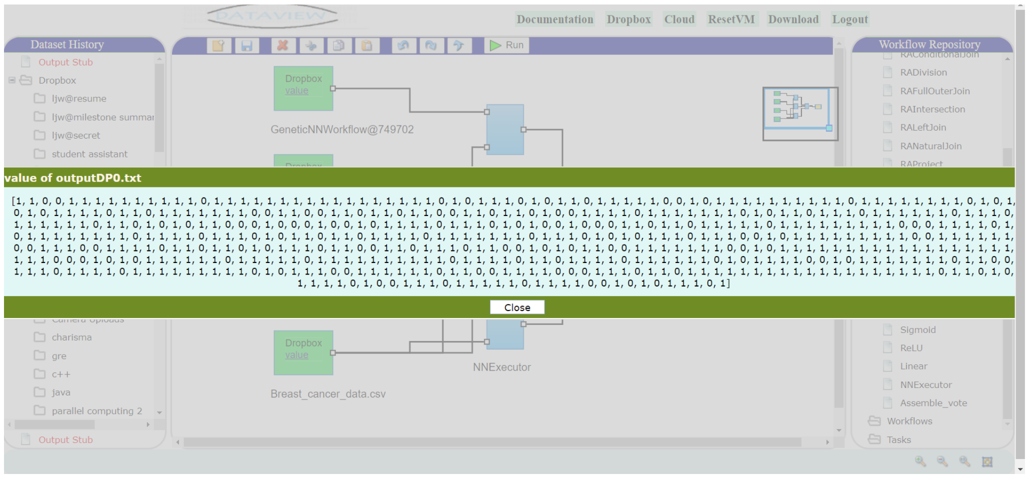


Figure 22: Design, construct and run the neural network ensemble learning workflow in web interface



```
[1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1,
 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0,
 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1,
 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0,
 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0,
                                                                                    1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1]
```

Figure 23: Execution result of the neural network ensemble learning workflow in web interface

the input dataset.

5.4 Extensibility of DLaaW in DATAVIEW

In this subsection, I will briefly walk through the up-to-date core DLaaW implementation in DATAVIEW, and analyze the extensibility of DLaaW within/beyond DATAVIEW in long run. Hopefully this can be set as a base work for future developers and DATAVIEW users who are likely to extend this work on more sophisticated DL workflows.

5.4.1 Current DLaaW implementation in DATAVIEW

Figure 24 shows the up-to-date class diagrams (include core classes in each component) and the relationship among three newly introduced DL-specific components for DLaaW in DATAVIEW (i.e. NNWorkflow Engine, GPU Resource Management and GPU services).

On the top tier of figure 24 shows the core JAVA class diagram of NNWorkflow Engine component, in which I outline a bounding box with red dash-line, outside which are the cores workflow classes (e.g. Workflow, Task, ports) that have been implemented in DATAVIEW prior this work, inside which are the newly implemented classes solely for DLaaW approach in DATAVIEW. These newly implemented classes can be classified into four categories: 1) NNTask class (which extends Task class) and its child classes (e.g. Linear, ReLU, Sigmoid), in which the NNTask class is the blueprint of any instantiated neural network layers (i.e. NNLayers) in an NNWorkflow; 2) NNWorkflow class (which extends Workflow class) and its child classes, in which the NNWorkflow class is the blueprint for any instantiated neural network workflows (i.e. NNWorkflows); 3) NNWorkflowTrainer class and its child classes, in which the NNWorkflowTrainer wraps the JSON Mapper module and serves as the blueprint for any NNTrainers; 4) NNExcutor class (which extends Task class) is a unified ordinary Task to reuse any trained NNWorkflow model on new dataset in a comprehensive workflow in specified GPU infrastructure.

At the bottom tier of figure 24 shows the core CUDA class diagram of GPU services, in which I listed out all major CUDA classes for constructing and executing an neural network (e.g. NNLayer, NeuralNetwork, Matrix). These classes can also be classified into three categories: 1) NNLayer and its child classes, in which the NNLayer class serves as a

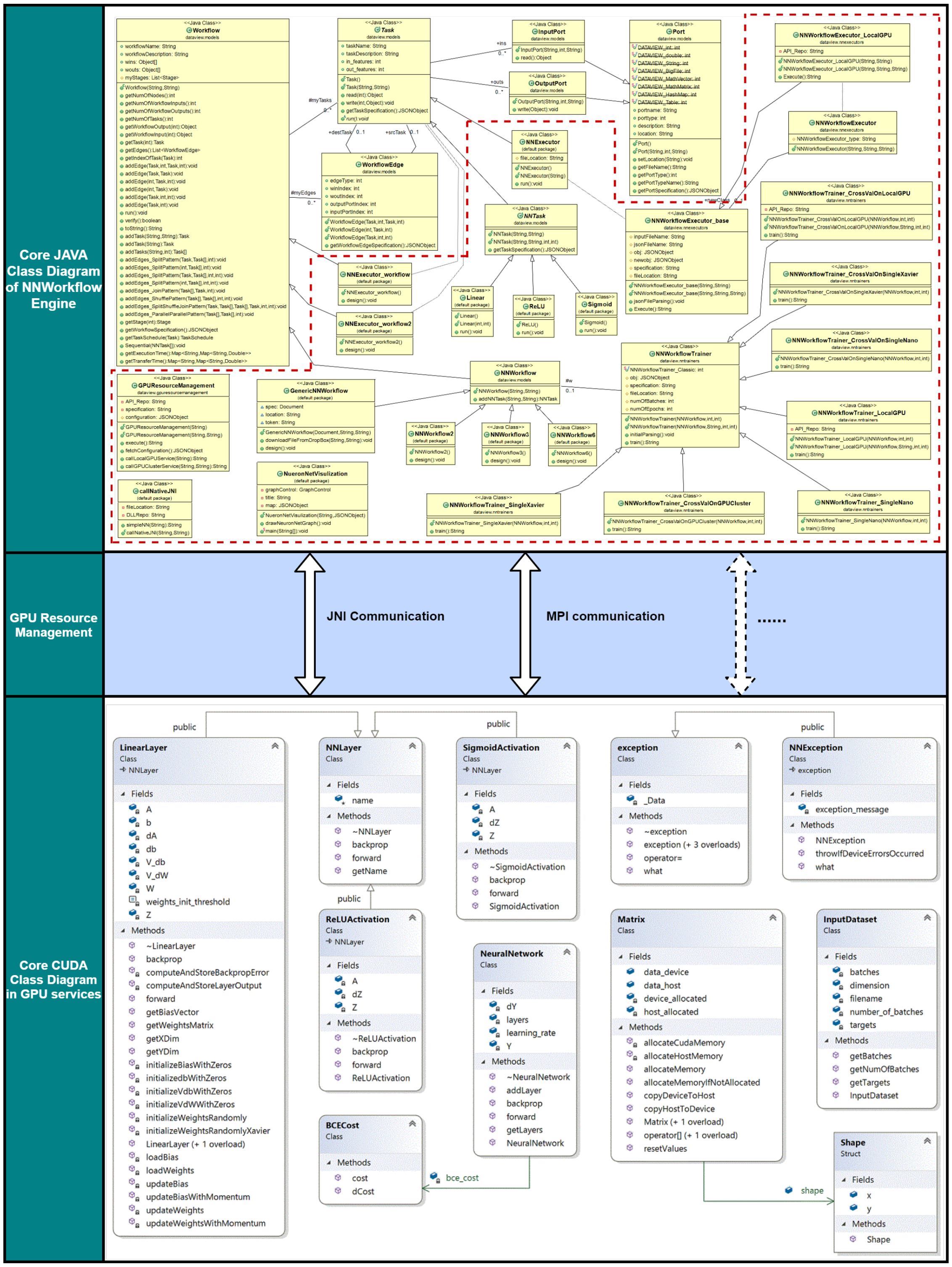


Figure 24: Overall class diagrams of DLaaW in DATAVIEW

blueprint for any instantiated neural network layer to be implemented; 2) NeuronNetwork class, in which NNLayers are concatenated as an array and all forward&backward propagations have been properly implemented across any arbitrary combination of NNLayers; 3) Supporting classes (Matrix, shape, inputDataset, etc.), which are the supporting classes to ensure calculation/data processing can be adequately and corectly taken care of.

The middle tier of figure 24 shows the GPU Resource Management module which bridge the NNWorkflow Engine and GPU Services components together, which also implemented by a base GPUResourceManagement class and its child classes, currently the child classes are supporting JNI or MPI communication between JAVA and CUDA.

5.4.2 Extensibility analysis

In our approach, GPU Resource Management is introduced as an intermediate interface between the NNWorkflow Engine and GPU Services components to gain implementation independence and better future extendability, i.e., the higher layer will be built on abstractions and all concrete implementations in the lower layer be aligned with a standardized interfacing protocol. As long as the intermediate abstractions do not change, changing or adding any GPU service in the future will not require any modifications on the higher level components. In NNWorkflow Engine component, I have provided blueprints on core JAVA implementation NNTask (which will be extended by all JAVA NNLayer classes) and NNWorkflow (which will be extended by all JAVA NNWorkflow classes). In GPU Services component, I have provided blueprints on core CUDA implementation NNLayer (which will be extended by all CUDA NNLayer classes) and NeuralNetwork classes (which will be extended by all CUDA NeuralNetwork classes). In this way, I have carefully decoupled the higher layer and lower layer implementations and provide all basic blueprints for core classes (new features can be added or overridden in its deriving classes), the extensibility should be carefully concerned and properly taken care of for a long run development in or beyond DATAVIEW.

5.5 Conclusions and future work

By walking through the processes of designing, constructing, executing and reusing any NNWorkflow in DATAVIEW through JAVA API and graphical web interface, and carefully analyzing the extensibility of newly added DL-specific components with their core JAVA/CUDA classes. We can safely draw the conclusion that DLaaW provides very clean and user-friendly interfaces (both in programmatically and graphically) to SWFMS community. Which meanwhile preserves the great extensibility for the future SWFMS developers and users, to further expanding their research territories to wider area base on this work - DLaaW in DATAVIEW.

CHAPTER 6 CONCLUSIONS AND FUTURE WORK

Our research focuses on enabling GPU-based DL workflows in DATAVIEW, one of the leading big data workflow system in the community. Our goal is to bring DL as native functionalities into modern SWFMSs. Our main contributions are listing as follow:

1. First, we propose and implement a novel DLaaW (Deep-Learning-as-a-Workflow) approach in single GPU in DATAVIEW, more specifically, by extending its workflow and task classes to two new subclasses: $NNWorkflow$ and $NNTask$. This approach is the first (to our best knowledge) that attempts to implement a DL neural network as a *native* workflow in a workflow management system. Specifically, I introduce an NNWorkflow Engine that wraps multi-type of $NNTrainers$, which are responsible for executing NNWorkflows according to specific execution plans (e.g. regular train and test, K-fold cross validation) using various types of underlying GPU resources. I also implement a generic *GPU Resource Management* module, to leverage various GPU resource configurations. The supported single GPU includes three options: the local NVIDIA GPU of a host PC, a single NVIDIA Xavier SoM (System-on-Module), and a single NVIDIA Nano SoM, for executing DL workflows in DATAVIEW. Our approach is validated through performance comparison with Keras-based counterpart implementations.

2. Next, we extend the DLaaW on heterogeneous GPU cluster in DATAVIEW, by introducing heterogeneous GPU clusters as a new type of GPU infrastructure for accelerated training and execution of NNWorkflows, on which it evaluates how well NNWorkflows can leverage the high-degree parallelism offered by a SWFMS in our experiments. I also introduce and implement the graphical WebBench GUI to fa-

cilitate NNWorkflow design, construction, run, and reuse in DATAVIEW. To validate our extended work, it conducts the performance comparison on DLaaW implementations and PyTorch-based counterparts (alternative to the Keras-based in our previous work), to assure the validation of this work not only holds for one particular DL library's counterpart implementations.

3. Finally, we demonstrate the usability of DLaaW in DATAVIEW to SWFMS community. The appealing intuitiveness of the JAVA API/web interface and superior extensibilty of DLaaW add to the usability of DLaaW and DATAVIEW as a whole. Furthermore, for incorporating DL as a part of comprehensive workflow, I extend DLaaW (especially the NNExecutor's JAVA and CUDA services) to integrate NNTasks (which will be executed in the GPU environment) and ordinary/traditional Tasks (which will be executed in the CPU environment) in one comprehensive workflow. A use case of the neural network ensemble learning, which derives the final prediction through voting by multiple weak classifiers, has been successfully implemented and executed in JAVA API and in web interface in DATAVIEW.

There are still many open problems to be solved, for example:

1. **Scheduling DL workflows to the cloud:** Currently, our DLaaW supports GPU infrastructures including a local GPU of host PC, a single SoM and a heterogeneous GPU cluster. With the thriving of cloud computing in past decade, the latest trend of workflow execution also shifted to cloud computing, for the sake of its superior scalability and virtually infinite computing resources. Potential research problems can be: how can we incorporate the DLaaW approach in the cloud? And how much performance gain can DL workflows obtain by introducing so?

2. **Optimizing DL workflows scheduling:** For this moment, DL workflows can be statically scheduled via DLaaW to a single GPU or on multiple GPU nodes of a heterogeneous GPU cluster. With the growth of GPU infrastructure's diversity and quantity, a dynamic scheduling algorithm can be needed to schedule a DL workflow on various infrastructure in order to gain the optimal QoS. And how much better can it be compare with the static scheduling?

3. **Let one DL workflows learn another:** The provenance data of a complex training process collected by underlying SWFMS platform can be analyzed and utilized to optimize the DL models, i.e. the knowledge can be mined from the provenance data, such as which inputs/hyperperameters have higher leverage on certain type of output change, may suggest more/less weight updates for certain neurons, resulting in a more enhanced "supervised" learning than training merely based on ground truth values/labels. If so, the "black boxes" of deep neuron networks may potentially be capable for looking into.

APPENDIX A

Appendix A Specifications of neural networks and their target datasets used in the experiments can be found in Table 3. For each neural network and its target dataset, this table provides 1) the neural network's architecture design; 2) the specification of dataset; 3) the weights initialization mechanism; 4) the training hyperparameters; 5) the number of splitting batches; 6) the number of training epochs.

Table 3: Specifications of neural networks and their target datasets.

	Neural network 1	Neural network 2	Neural network 3	Neural network 4	Neural network 5
Model Arch Design	4 layers: Linear(5,3), ReLU, Linear(3,1), Sigmoid	6 layers: Linear(8,5), ReLU, Linear(5,3), ReLU, Linear(3,1), Sigmoid	4 layers: Linear(4,2), ReLU, Linear(2,1), Sigmoid	8 layers: Linear(13,8), ReLU, Linear(8,5), ReLU, Linear(5,3), ReLU, Linear(3,1), Sigmoid	8 layers: Linear(16,8), ReLU, Linear(8,5), ReLU, Linear(5,3), ReLU, Linear(3,1), Sigmoid
Target Dataset	Breast Cancer Dataset with 569 instances	Pima Indians Diabetes Dataset with 768 instances	Data Banknote Authentication Dataset with 1372 instances	Electrical Grid Stability Dataset with 10000 instances	Bank Marketing Dataset with 45211 instances
Initialization	Xavier weight init [48], shuffled input data	Xavier weight init, shuffled input data	Xavier weight init, shuffled input data	Xavier weight init, shuffled input data	Xavier weight init, shuffled input data
Training Hyperparameters	learningRate=0.1, momentumSGD =0.9	learningRate=0.1, momentumSGD =0.9	learningRate=0.1, momentumSGD =0.9	learningRate=0.1, momentumSGD =0.9	learningRate=0.1, momentumSGD =0.9
# of batches	6	6	6	6	6
# of Epochs	1000	1000	1000	1000	1000

APPENDIX B

The analytical derivatives of Linear layer, ReLU layer, Sigmoid layer and binary cross entropy in our CUDA GPGPU implementation can be found as follow:

Linear Layer:

$$Z = WA + b$$

$$dA = \frac{dJ}{dZ}\frac{dZ}{dA} = W^T \cdot dZ$$

$$dW = \frac{dJ}{dZ}\frac{dZ}{dW} = \frac{1}{m} \cdot dZ \cdot A^T$$

$$db = \frac{dJ}{dZ}\frac{dZ}{db} = \frac{1}{m}\sum_{i=0}^{m} dZ^{(i)}$$

ReLU Layer:

$$ReLU(x) = \begin{cases} x & \text{for } x > 0 \\ 0 & \text{otherwise} \end{cases} = max(0, x)$$

$$dZ = \frac{dJ}{dReLU(x)}\frac{dReLU(x)}{dx} = \begin{cases} dA & \text{for } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Sigmoid Layer:

$$\sigma(x) = \frac{e^x}{1 + e^x}$$

$$dZ = \frac{dJ}{d\sigma(x)}\frac{d\sigma(x)}{dx} = dA \cdot \sigma(x) \cdot \big(1 - \sigma(x)\big)$$

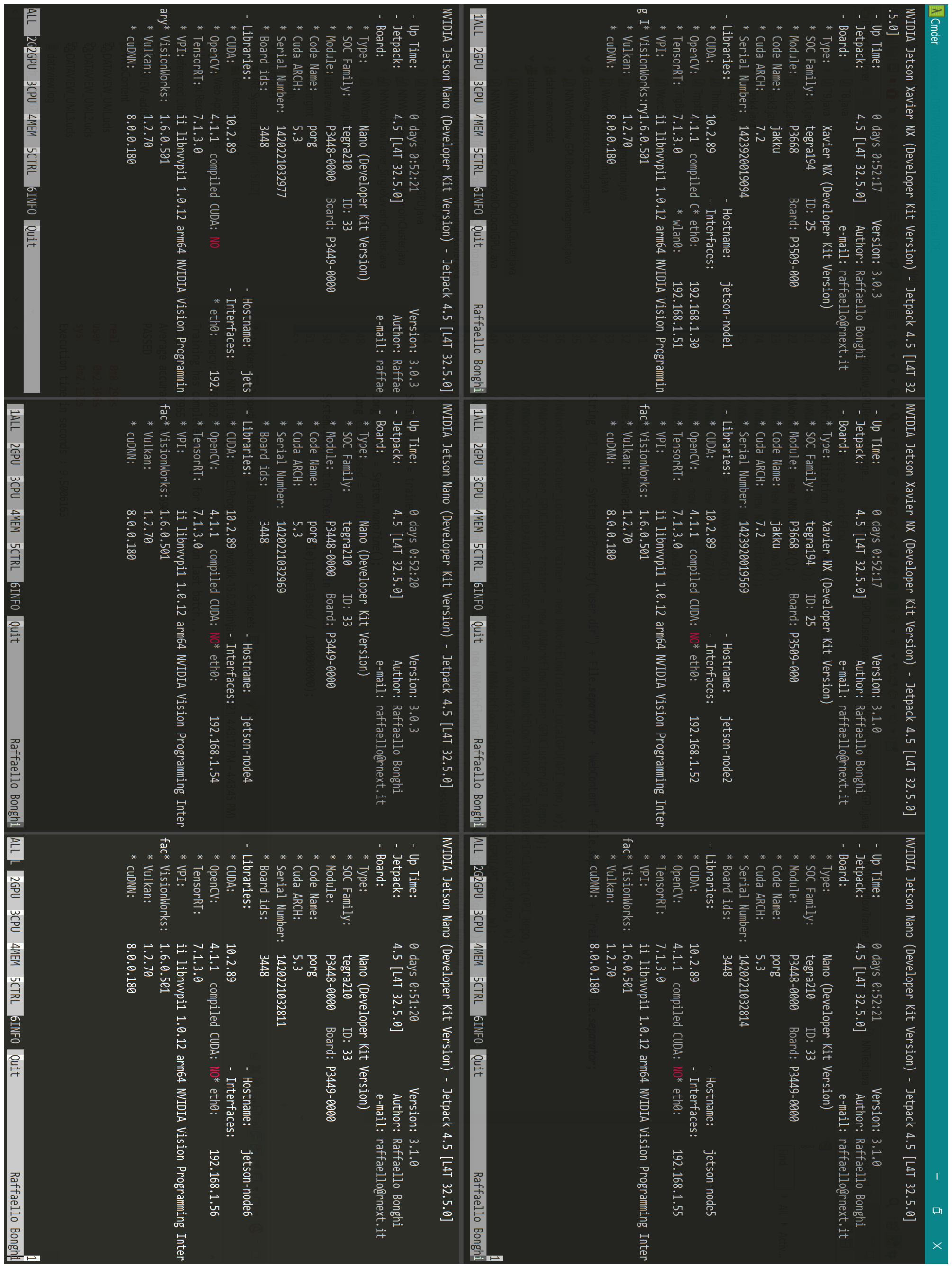
Binary cross entropy:

$$BCE = -\frac{1}{m}\sum_{i=0}^{m}(y\log(\hat{y}) + (1-y)\log(1-\hat{y}))$$

$$\bigtriangledown BCE = -(\frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}})$$

APPENDIX C

The specification of each GPU node (6 in total) in the heterogeneous GPU cluster are printed out in Figure 25, including SoM type, SOC family, Cuda ARCH, etc.



```
Cmder

NVIDIA Jetson Xavier NX (Developer Kit Version) - Jetpack 4.5 [L4T 32
.5.0]
 - Up Time:        0 days 0:52:17     Version: 3.0.3
 - Jetpack:        4.5 [L4T 32.5.0]    Author: Raffaello Bonghi
 - Board:                             e-mail: raffaello@rnext.it
   * Type:             Xavier NX (Developer Kit Version)
   * SOC Family:       tegra194     ID: 25
   * Module:           P3668        Board: P3509-000
   * Code Name:        jakku
   * Cuda ARCH:        7.2
   * Serial Number:  1423920019094
 - Libraries:                          - Hostname:     jetson-node1
   * CUDA:         10.2.89           - Interfaces:
   * OpenCV:       4.1.1  compiled C* eth0:      192.168.1.30
   * TensorRT:     7.1.3.0            * wlan0:     192.168.1.51
   * VPI:          ii libnvvpi1 1.0.12 arm64 NVIDIA Vision Programmin
g I* VisionWorks:ry1.6.0.501
   * Vulkan:       1.2.70
   * cuDNN:        8.0.0.180

 1ALL  2GPU  3CPU  4MEM  5CTRL  6INFO  Quit              Raffaello Bonghi
```

```
NVIDIA Jetson Xavier NX (Developer Kit Version) - Jetpack 4.5 [L4T 32.5.0]
 - Up Time:        0 days 0:52:17            Version: 3.1.0
 - Jetpack:        4.5 [L4T 32.5.0]           Author: Raffaello Bonghi
 - Board:                                    e-mail: raffaello@rnext.it
   * Type:             Xavier NX (Developer Kit Version)
   * SOC Family:       tegra194     ID: 25
   * Module:           P3668        Board: P3509-000
   * Code Name:        jakku
   * Cuda ARCH:        7.2
   * Serial Number:  1423920019569
 - Libraries:                            - Hostname:     jetson-node2
   * CUDA:         10.2.89               - Interfaces:
   * OpenCV:       4.1.1  compiled CUDA: NO* eth0:      192.168.1.52
   * TensorRT:     7.1.3.0
   * VPI:          ii libnvvpi1 1.0.12 arm64 NVIDIA Vision Programming Inter
fac* VisionWorks:  1.6.0.501
   * Vulkan:       1.2.70
   * cuDNN:        8.0.0.180

 1ALL  2GPU  3CPU  4MEM  5CTRL  6INFO  Quit                Raffaello Bonghi
```

```
NVIDIA Jetson Nano (Developer Kit Version) - Jetpack 4.5 [L4T 32.5.0]
 - Up Time:        0 days 0:52:21            Version: 3.1.0
 - Jetpack:        4.5 [L4T 32.5.0]           Author: Raffaello Bonghi
 - Board:                                    e-mail: raffaello@rnext.it
   * Type:             Nano (Developer Kit Version)
   * SOC Family:       tegra210     ID: 33
   * Module:           P3448-0000   Board: P3449-0000
   * Code Name:        porg
   * Cuda ARCH:        5.3
   * Serial Number:  1420221032814
   * Board ids:        3448
 - Libraries:                            - Hostname:     jetson-node5
   * CUDA:         10.2.89               - Interfaces:
   * OpenCV:       4.1.1  compiled CUDA: NO* eth0:      192.168.1.55
   * TensorRT:     7.1.3.0
   * VPI:          ii libnvvpi1 1.0.12 arm64 NVIDIA Vision Programming Inter
fac* VisionWorks:  1.6.0.501
   * Vulkan:       1.2.70
   * cuDNN:        8.0.0.180

 ALL  2GPU  3CPU  4MEM  5CTRL  6INFO  Quit                Raffaello Bonghi
```

```
NVIDIA Jetson Nano (Developer Kit Version) - Jetpack 4.5 [L4T 32.5.0]

 - Up Time:        0 days 0:52:21               Version: 3.0.3
 - Jetpack:        4.5 [L4T 32.5.0]              Author: Raffae
 - Board:                                       e-mail: raffae
   * Type:             Nano (Developer Kit Version)
   * SOC Family:       tegra210     ID: 33
   * Module:           P3448-0000   Board: P3449-0000
   * Code Name:        porg
   * Cuda ARCH:        5.3
   * Serial Number:  1420221032977
   * Board ids:        3448
 - Libraries:                                 - Hostname:     jets
   * CUDA:         10.2.89                    - Interfaces:
   * OpenCV:       4.1.1  compiled CUDA: NO     * eth0:      192.
   * TensorRT:     7.1.3.0
   * VPI:          ii libnvvpi1 1.0.12 arm64 NVIDIA Vision Programmin
ary* VisionWorks:  1.6.0.501
   * Vulkan:       1.2.70
   * cuDNN:        8.0.0.180

 ALL  2GPU  3CPU  4MEM  5CTRL  6INFO  Quit
```

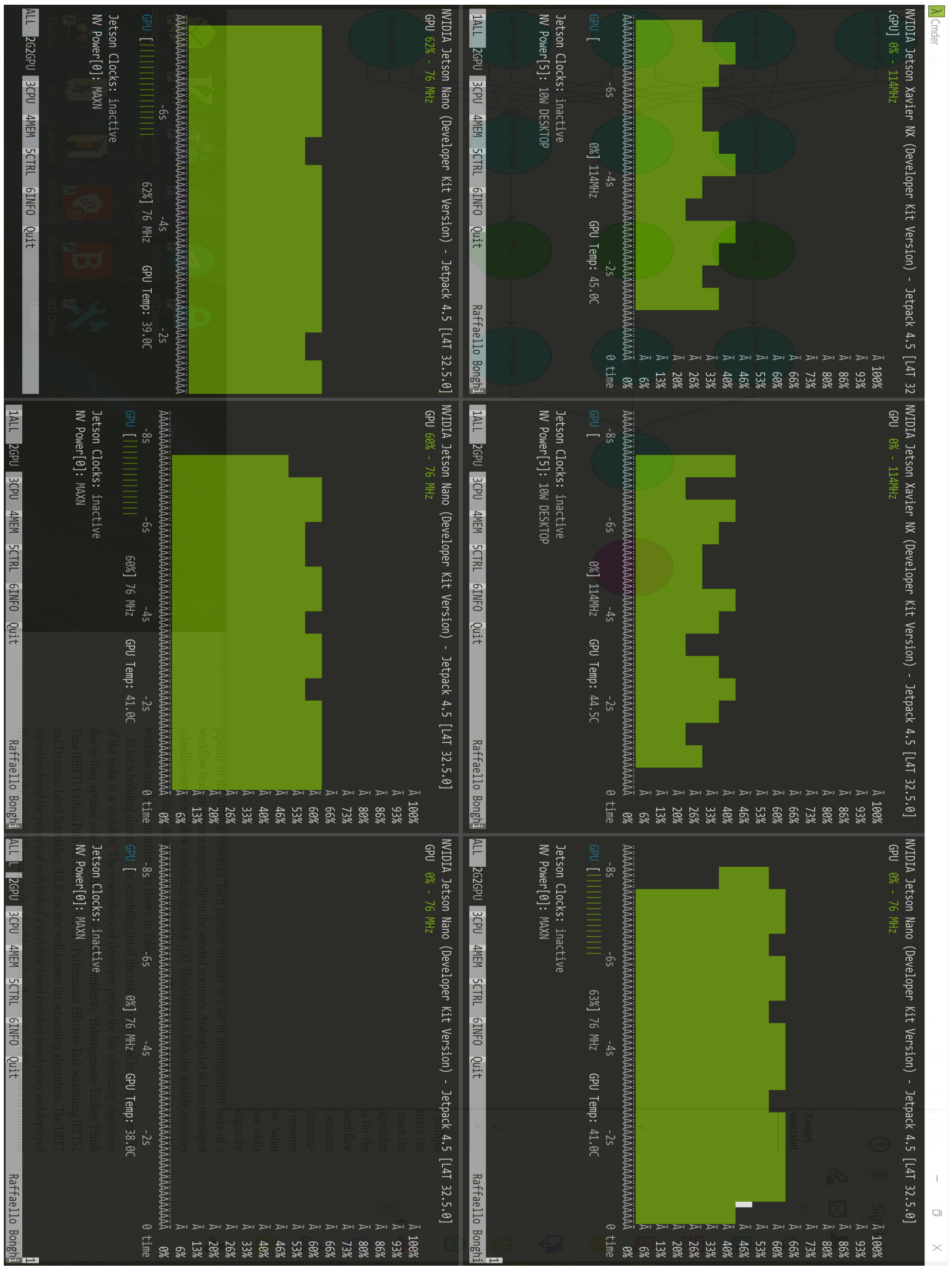
```
NVIDIA Jetson Nano (Developer Kit Version) - Jetpack 4.5 [L4T 32.5.0]

 - Up Time:        0 days 0:52:20            Version: 3.0.3
 - Jetpack:        4.5 [L4T 32.5.0]           Author: Raffaello Bonghi
 - Board:                                    e-mail: raffaello@rnext.it
   * Type:             Nano (Developer Kit Version)
   * SOC Family:       tegra210     ID: 33
   * Module:           P3448-0000   Board: P3449-0000
   * Code Name:        porg
   * Cuda ARCH:        5.3
   * Serial Number:  1420221032969
   * Board ids:        3448
 - Libraries:                            - Hostname:     jetson-node4
   * CUDA:         10.2.89               - Interfaces:
   * OpenCV:       4.1.1  compiled CUDA: NO* eth0:      192.168.1.54
   * TensorRT:     7.1.3.0
   * VPI:          ii libnvvpi1 1.0.12 arm64 NVIDIA Vision Programming Inter
fac* VisionWorks:  1.6.0.501
   * Vulkan:       1.2.70
   * cuDNN:        8.0.0.180

 1ALL  2GPU  3CPU  4MEM  5CTRL  6INFO  Quit                Raffaello Bonghi
```

```
NVIDIA Jetson Nano (Developer Kit Version) - Jetpack 4.5 [L4T 32.5.0]

 - Up Time:        0 days 0:51:20            Version: 3.1.0
 - Jetpack:        4.5 [L4T 32.5.0]           Author: Raffaello Bonghi
 - Board:                                    e-mail: raffaello@rnext.it
   * Type:             Nano (Developer Kit Version)
   * SOC Family:       tegra210     ID: 33
   * Module:           P3448-0000   Board: P3449-0000
   * Code Name:        porg
   * Cuda ARCH:        5.3
   * Serial Number:  1420221032811
   * Board ids:        3448
 - Libraries:                            - Hostname:     jetson-node6
   * CUDA:         10.2.89               - Interfaces:
   * OpenCV:       4.1.1  compiled CUDA: NO* eth0:      192.168.1.56
   * TensorRT:     7.1.3.0
   * VPI:          ii libnvvpi1 1.0.12 arm64 NVIDIA Vision Programming Inter
fac* VisionWorks:  1.6.0.501
   * Vulkan:       1.2.70
   * cuDNN:        8.0.0.180

 ALL  2GPU  3CPU  4MEM  5CTRL  6INFO  Quit                Raffaello Bonghi
```

Figure 25: Specification of each GPU node in the DATAVIEW GPU cluster

APPENDIX D

The memory utilization on each GPU node in the GPU cluster during the distributed 5-fold cross validation in DATAVIEW are printed out in Figure 26.

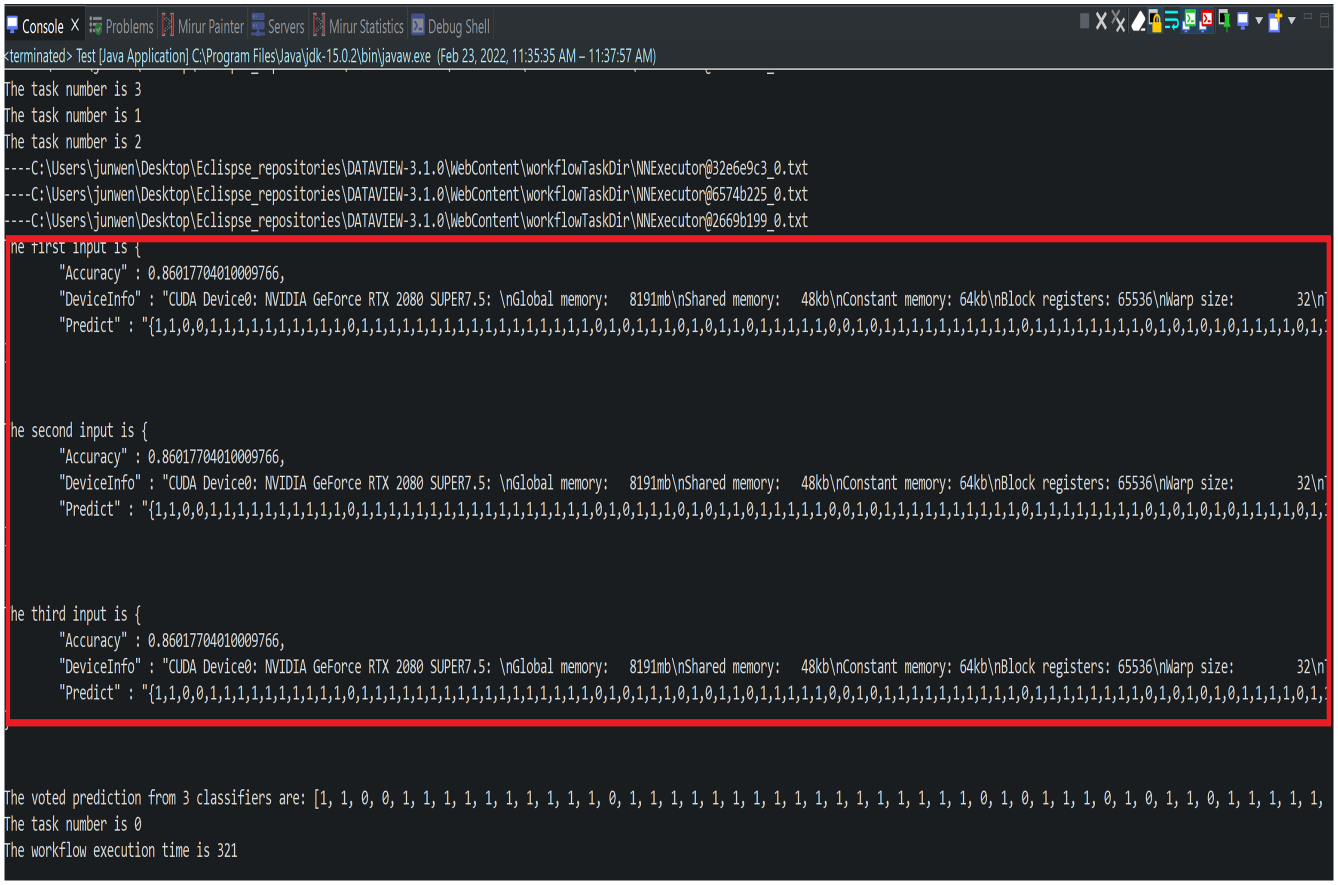


```
NVIDIA Jetson Xavier NX (Developer Kit Version) - Jetpack 4.5 [L4T 32
GPU 0% - 114MHz
GPU [                0%] 114MHz     GPU Temp: 45.0C
Jetson Clocks: inactive
NV Power[5]: 10W DESKTOP
1ALL 2GPU 3CPU 4MEM 5CTRL 6INFO Quit          Raffaello Bonghi

NVIDIA Jetson Nano (Developer Kit Version) - Jetpack 4.5 [L4T 32.5.0]
GPU 62% - 76 MHz
GPU [||||||||||     62%] 76 MHz     GPU Temp: 39.0C
Jetson Clocks: inactive
NV Power[0]: MAXN
ALL 2GPU 3CPU 4MEM 5CTRL 6INFO Quit

NVIDIA Jetson Xavier NX (Developer Kit Version) - Jetpack 4.5 [L4T 32.5.0]
GPU 0% - 114MHz
GPU [                0%] 114MHz     GPU Temp: 44.5C
Jetson Clocks: inactive
NV Power[5]: 10W DESKTOP
1ALL 2GPU 3CPU 4MEM 5CTRL 6INFO Quit          Raffaello Bonghi

NVIDIA Jetson Nano (Developer Kit Version) - Jetpack 4.5 [L4T 32.5.0]
GPU 60% - 76 MHz
GPU [||||||||||     60%] 76 MHz     GPU Temp: 41.0C
Jetson Clocks: inactive
NV Power[0]: MAXN
1ALL 2GPU 3CPU 4MEM 5CTRL 6INFO Quit          Raffaello Bonghi

NVIDIA Jetson Nano (Developer Kit Version) - Jetpack 4.5 [L4T 32.5.0]
GPU 0% - 76 MHz
GPU [|||||||||||    63%] 76 MHz     GPU Temp: 41.0C
Jetson Clocks: inactive
NV Power[0]: MAXN
ALL 2GPU 3CPU 4MEM 5CTRL 6INFO Quit          Raffaello Bonghi

NVIDIA Jetson Nano (Developer Kit Version) - Jetpack 4.5 [L4T 32.5.0]
GPU 0% - 76 MHz
GPU [                0%] 76 MHz     GPU Temp: 38.0C
Jetson Clocks: inactive
NV Power[0]: MAXN
ALL 2GPU 3CPU 4MEM 5CTRL 6INFO Quit          Raffaello Bonghi
```

Figure 26: Memory utilization of each GPU node during distributed 5-fold cross validation

APPENDIX E

The intermediate outputs from 3 weak classifiers, which are served as the inputs of the Ensemble_Vote Task in the neural network ensemble learning workflow are printed out in Figure 27.



```
Console X  Problems  Mirur Painter  Servers  Mirur Statistics  Debug Shell
<terminated> Test [Java Application] C:\Program Files\Java\jdk-15.0.2\bin\javaw.exe (Feb 23, 2022, 11:35:35 AM – 11:37:57 AM)
The task number is 3
The task number is 1
The task number is 2
----C:\Users\junwen\Desktop\Eclispse_repositories\DATAVIEW-3.1.0\WebContent\workflowTaskDir\NNExecutor@32e6e9c3_0.txt
----C:\Users\junwen\Desktop\Eclispse_repositories\DATAVIEW-3.1.0\WebContent\workflowTaskDir\NNExecutor@6574b225_0.txt
----C:\Users\junwen\Desktop\Eclispse_repositories\DATAVIEW-3.1.0\WebContent\workflowTaskDir\NNExecutor@2669b199_0.txt
he first input is {
        "Accuracy" : 0.86017704010009766,
        "DeviceInfo" : "CUDA Device0: NVIDIA GeForce RTX 2080 SUPER7.5: \nGlobal memory:   8191mb\nShared memory:   48kb\nConstant memory: 64kb\nBlock registers: 65536\nWarp size:        32\n
        "Predict" : "{1,1,0,0,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,0,1,1,1,0,1,0,1,1,0,1,1,1,1,1,0,0,1,0,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,0,1,0,1,0,1,1,1,1,0,1,


he second input is {
        "Accuracy" : 0.86017704010009766,
        "DeviceInfo" : "CUDA Device0: NVIDIA GeForce RTX 2080 SUPER7.5: \nGlobal memory:   8191mb\nShared memory:   48kb\nConstant memory: 64kb\nBlock registers: 65536\nWarp size:        32\n
        "Predict" : "{1,1,0,0,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,0,1,1,1,0,1,0,1,1,0,1,1,1,1,1,0,0,1,0,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,0,1,0,1,0,1,1,1,1,0,1,


he third input is {
        "Accuracy" : 0.86017704010009766,
        "DeviceInfo" : "CUDA Device0: NVIDIA GeForce RTX 2080 SUPER7.5: \nGlobal memory:   8191mb\nShared memory:   48kb\nConstant memory: 64kb\nBlock registers: 65536\nWarp size:        32\n
        "Predict" : "{1,1,0,0,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,0,1,1,1,0,1,0,1,1,0,1,1,1,1,1,0,0,1,0,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,0,1,0,1,0,1,1,1,1,0,1,


The voted prediction from 3 classifiers are: [1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1,
The task number is 0
The workflow execution time is 321
```

Figure 27: Intermediate outputs in the neural network ensemble learning workflow

REFERENCES

[1] MatConvNet: Convolutional Neural Networks for MATLAB.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv preprint arXiv:1603.04467*, 2016.

[3] R. Abbott, T. Abbott, S. Abraham, F. Acernese, K. Ackley, A. Adams, C. Adams, R. Adhikari, V. Adya, C. Affeldt, et al. GWTC-2: Compact Binary Coalescences Observed by LIGO and Virgo during the First Half of the Third Observing Run. *Physical Review X*, 11(2):021053, 2021.

[4] I. Ahmed, S. Lu, C. Bai, and F. A. Bhuyan. Diagnosis Recommendation Using Machine Learning Scientific Workflows. In *2018 IEEE International Congress on Big Data (BigData Congress)*, pages 82–90. IEEE, 2018.

[5] Z. Akkus, J. Cai, A. Boonrod, A. Zeinoddini, A. D. Weston, K. A. Philbrick, and B. J. Erickson. A Survey of Deep-Learning Applications in Ultrasound: Artificial Intelligence–Powered Ultrasound for Improving Clinical Workflow. *Journal of the American College of Radiology*, 16(9):1318–1328, 2019.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010.

[7] C. Bai, J. Liu, I. Ahmed, and S. Lu. DATAVIEW Release 2.1. `https://github.com/shiyonglu/DATAVIEW/releases/tag/2.1`, 2019.

[8] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron,

N. Bouchard, D. Warde-Farley, and Y. Bengio. Theano: New features and speed improvements. *arXiv preprint arXiv:1211.5590*, 2012.

[9] D. Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[10] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, K. Thiel, and B. Wiswedel. KNIME-the Konstanz Information Miner: Version 2.0 and Beyond. *AcM SIGKDD explorations Newsletter*, 11(1):26–31, 2009.

[11] E. Bisong. *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A comprehensive guide for beginners*. Apress, 2019.

[12] A. Bouguettaya, M. Singh, M. Huhns, Q. Z. Sheng, H. Dong, Q. Yu, A. G. Neiat, S. Mistry, B. Benatallah, B. Medjahed, et al. A Service Computing Manifesto: The Next 10 Years. *Communications of the ACM*, 60(4):64–72, 2017.

[13] J. Chen and X. Ran. Deep Learning With Edge Computing: A Review. *Proceedings of the IEEE*, 107(8):1655–1674, 2019.

[14] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv preprint arXiv:1512.01274*, 2015.

[15] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *arXiv preprint arXiv:1410.0759*, 2014.

[16] D. M. Chitty. A Data Parallel Approach to Genetic Programming Using Programmable Graphics Hardware. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1566–1573, 2007.

[17] G. P. Consortium. A Global Reference For Human Genetic Variation. *Nature*, 526(7571):68, 2015.

[18] P. Covas, A. Effler, E. Goetz, P. Meyers, A. Neunzert, M. Oliver, B. Pearlstone, V. Roma, R. Schofield, V. Adya, et al. Identification and Mitigation of Narrow Spectral Artifacts that Degrade Searches for Persistent Gravitational Waves in the first two Observing Runs of Advanced LIGO. *Physical Review D*, 97(8):082002, 2018.

[19] R. F. da Silva, R. Filgueira, I. Pietri, M. Jiang, R. Sakellariou, and E. Deelman. A Characterization of Workflow Management Systems for Extreme-scale Applications. *Future Generation Computer Systems*, 75:228–238, 2017.

[20] S. Dargan, M. Kumar, M. R. Ayyagari, and G. Kumar. A Survey of Deep Learning and Its Applications: A New Paradigm to Machine Learning. *Archives of Computational Methods in Engineering*, 27(4):1071–1092, 2020.

[21] C. DeBrusk. The Risk of Machine Learning Bias (And How to Prevent It). *MIT Sloan Management Review*, 2018.

[22] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus: Mapping Scientific Workflows onto the Grid. In *European Across Grids Conference*, pages 11–20. Springer, 2004.

[23] E. Deelman, R. F. da Silva, K. Vahi, M. Rynge, R. Mayani, R. Tanaka, W. Whitcup, and M. Livny. The Pegasus Workflow Management System: Translational Computer Science in Practice. *Journal of Computational Science*, 52:101200, 2021.

[24] E. Deelman, A. Mandal, M. Jiang, and R. Sakellariou. The Role of Machine Learning in Scientific Workflows. *The International Journal of High Performance Computing Applications*, 33(6):1128–1139, 2019.

[25] E. Deelman, T. Peterka, I. Altintas, C. D. Carothers, K. K. van Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, and J. Vetter. The Future of Scientific Workflows. *The International Journal of High Performance Computing Applications*, 32(1):159–175, 2018.

[26] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The Cost of Doing Science on the Cloud: the Montage example. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12. Ieee, 2008.

[27] J. Ding, M. Nemati, C. Ranaweera, and J. Choi. IoT Connectivity Technologies and Applications: A Survey. *arXiv preprint arXiv:2002.12646*, 2020.

[28] F. K. Došilović, M. Brčić, and N. Hlupić. Explainable Artificial Intelligence: A Survey. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 0210–0215. IEEE, 2018.

[29] D. Dua, C. Graff, et al. UCI Machine Learning Repository. 2017.

[30] J. Freire, D. Koop, F. Chirigati, and C. T. Silva. Reproducibility Using VisTrails. In *Implementing Reproducible Research*, pages 33–56. Chapman and Hall/CRC, 2018.

[31] J. Fung and S. Mann. Computer Vision Signal Processing on Graphics Processing Units. In *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages V–93. IEEE, 2004.

[32] J. Fung, F. Tang, and S. Mann. Mediated Reality using Computer Graphics Hardware for Computer Vision. In *Proceedings. Sixth International Symposium on Wearable Computers,*, pages 83–89. IEEE, 2002.

[33] C. A. Goble, J. Bhagat, S. Aleksejevs, D. Cruickshank, D. Michaelides, D. Newman, M. Borkum, S. Bechhofer, M. Roos, P. Li, et al. myExperiment: A Repository and

Social Network for the Sharing of Bioinformatics Workflows. *Nucleic acids research*, 38(suppl_2):W677–W682, 2010.

[34] J. Goecks et al. Galaxy: A comprehensive Approach for Supporting Accessible, Reproducible, and Transparent Computational Research in the Life Sciences. *Genome Biology*, 11(8):R86, 2010.

[35] K. J. Goodman, S. M. Parker, J. W. Edmonds, and L. H. Zeglin. Expanding the scale of aquatic sciences: the role of the National Ecological Observatory Network (NEON). *Freshwater Science*, 34(1):377–385, 2015.

[36] A. Gulli, A. Kapoor, and S. Pal. *Deep learning with TensorFlow 2 and Keras: regression, ConvNets, GANs, RNNs, NLP, and more with TensorFlow 2 and the Keras API*. Packt Publishing Ltd, 2019.

[37] A. Gulli and S. Pal. *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[38] O. Gupta and R. Raskar. Distributed Learning of Deep Neural Network over Multiple Agents. *Journal of Network and Computer Applications*, 116:1–8, 2018.

[39] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An Update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[40] J. Herbst and D. Karagiannis. Integrating Machine Learning and Workflow Management to Support Acquisition and Adaptation of Workflow Models. *Intelligent Systems in Accounting, Finance & Management*, 9(2):67–92, 2000.

[41] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In

Proceedings of the 22nd ACM international conference on Multimedia, pages 675–678, 2014.

[42] Y. Kano et al. U-Compare: A modular NLP workflow construction and evaluation system. *J. Res. Dev*, 55(3):11–1, 2011.

[43] D. Kaplan, J. Powell, and T. Woller. AMD Memory Encryption. *White paper*, 2016.

[44] A. Kashlev et al. Big data workflows: A reference architecture and the DATAVIEW system. *STBD*, 4(1):1–19, 2017.

[45] A. Kashlev, S. Lu, and A. Chebotko. Typetheoretic Approach to the Shimming Problem in Scientific Workflows. *IEEE Transactions on Services Computing*, 8(5):795–809, 2014.

[46] P. Kim. Matlab Deep Learning. *With Machine Learning, Neural Networks and Artificial Intelligence*, 130(21), 2017.

[47] J. Kranjc, R. Orač, et al. ClowdFlows: Online workflows for distributed big data mining. *FGCS*, 68:38–58, 2017.

[48] S. K. Kumar. On Weight Initialization in Deep Neural Networks. *arXiv preprint arXiv:1704.08863*, 2017.

[49] F. Li and F. Song. Building A Scientific Workflow Framework to Enable Real-Time Machine Learning and Visualization. *CCPE*, 31(16):e4703, 2019.

[50] C. Lin, S. Lu, et al. A Reference Architecture for Scientific Workflow Management Systems and the VIEW SOA Solution. *TSC*, 2(1):79–92, 2009.

[51] C. Lin, S. Lu, X. Fei, D. Pai, and J. Hua. A Task Abstraction and Mapping Approach to the Shimming Problem in Scientific Workflows. In *2009 IEEE International Conference on Services Computing*, pages 284–291. IEEE, 2009.

[52] F. Liu, G. Tang, Y. Li, Z. Cai, X. Zhang, and T. Zhou. A Survey on Edge Computing Systems and Tools. *Proceedings of the IEEE*, 107(8):1537–1562, 2019.

[53] J. Liu, C. Bai, and S. Lu. DATAVIEW Release 3.0. `https://github.com/shiyonglu/DATAVIEW/releases/tag/3.0`, 2021.

[54] J. Liu, S. Lu, and D. Che. A Survey of Modern Scientific Workflow Scheduling Algorithms and Systems in the Era of Big Data. In *2020 IEEE International Conference on Services Computing (SCC)*, pages 132–141. IEEE, 2020.

[55] J. Liu, E. Pacitti, P. Valduriez, and M. Mattoso. A Survey of Data-intensive Scientific Workflow Management. *Journal of Grid Computing*, 13(4):457–493, 2015.

[56] J. Liu, Z. Xiao, S. Lu, and D. Che. Deep-Learning-as-a-Workflow (DLaaW): An Innovative Approach to Enabling Deep Learning in Scientific Workflows. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 3101–3106. IEEE, 2021.

[57] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and computation: Practice and experience*, 18(10):1039–1065, 2006.

[58] E. Lyons, G. Papadimitriou, C. Wang, K. Thareja, P. Ruth, J. Villalobos, I. Rodero, E. Deelman, M. Zink, and A. Mandal. Toward a Dynamic Network-Centric Distributed Cloud Platform for Scientific Workflows: A Case Study for Adaptive Weather Sensing. In *15th IEEE eScience Conference*, 2019.

[59] G. S. Mahmood, D. J. Huang, and B. A. Jaleel. Achieving an Effective, Confidentiality and Integrity of Data in Cloud Computing. *IJ Network Security*, 21(2):326–332, 2019.

[60] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski. Algorithms for Cost- and Deadline-constrained Provisioning for Scientific Workflow Ensembles in IaaS clouds. *Future Generation Computer Systems*, 48:1–18, 2015.

[61] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas. Intel® Software Guard Extensions (intel® sgx) Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, pages 1–9. 2016.

[62] R. Mitchell, L. Pottier, S. Jacobs, R. F. da Silva, M. Rynge, K. Vahi, and E. Deelman. Exploration of Workflow Management Systems Emerging Features from Users Perspectives. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 4537–4544. IEEE, 2019.

[63] T. Miu and P. Missier. Predicting the Execution Time of Workflow Activities Based on Their Input Features. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 64–72. IEEE, 2012.

[64] S. Mofrad, I. Ahmed, S. Lu, P. Yang, H. Cui, and F. Zhang. SecDATAVIEW: A Secure Big Data Workflow Management System for Heterogeneous Computing Environments. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 390–403. ACM, 2019.

[65] A. Mohan, A. Kashlev, C. Bai, and S. Lu. DATAVIEW Release 2.0. `https://github.com/shiyonglu/DATAVIEW/releases/tag/2.0`, 2019.

[66] M. L. Mondelli, T. Magalhães, G. Loss, M. Wilde, I. Foster, M. Mattoso, D. Katz, H. Barbosa, A. T. R. de Vasconcelos, K. Ocaña, et al. BioWorkbench: A High-

Performance Framework for Managing and Analyzing Bioinformatics Experiments. *PeerJ*, 6:e5551, 2018.

[67] F. Moradi, R. Stadler, and A. Johnsson. Performance Prediction in Dynamic Clouds using Transfer Learning. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 242–250. IEEE, 2019.

[68] A. Nascimento, V. Olimpio, V. Silva, A. Paes, and D. de Oliveira. A Reinforcement Learning Scheduling Strategy for Parallel Cloud-Based Workflows. In *2019 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 817–824. IEEE, 2019.

[69] C. Nvidia. CUBLAS library programming guide. *NVIDIA Corporation. edit*, 1:1–237, 2007.

[70] P. Nyström et al. The TimeStudio Project: An open source scientific workflow system for the behavioral and brain sciences. *Behavior research methods*, 48(2):542–552, 2016.

[71] T. Oinn, M. Addis, et al. Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[72] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[73] J. Ozik, N. T. Collier, J. M. Wozniak, and C. Spagnuolo. From desktop to Large-Scale Model Exploration with Swift/T. In *2016 Winter Simulation Conference (WSC)*, pages 206–220. IEEE, 2016.

[74] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. 2017.

[75] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

[76] M. Perovšek, J. Kranjc, et al. TextFlows: A Visual Programming Platform for Text Mining and Natural Language Processing. *Science of Computer Programming*, 121:128–152, 2016.

[77] W. L. Poehlman, M. Rynge, C. Branton, D. Balamurugan, and F. A. Feltus. OSG-GEM: Gene Expression Matrix Construction Using the Open Science Grid. *Bioinformatics and Biology insights*, 10:BBI–S38193, 2016.

[78] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. S. Iyengar. A Survey on Deep Learning: Algorithms, Techniques, and Applications. *ACM Computing Surveys (CSUR)*, 51(5):1–36, 2018.

[79] N. Radosevic, M. Duckham, G.-J. Liu, and Q. Sun. Solar radiation modeling with KNIME and Solar Analyst: Increasing environmental model reproducibility using scientific workflows. *Environmental Modelling & Software*, 132:104780, 2020.

[80] M. Raghu and E. Schmidt. A Survey of Deep Learning for Scientific Discovery. *arXiv preprint arXiv:2003.11755*, 2020.

[81] M. A. Rodriguez and R. Buyya. A Responsive Knapsack-based Algorithm for Resource Provisioning and Scheduling of Scientific Workflows in Clouds. In *ICPP*, pages 839–848. IEEE, 2015.

[82] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[83] D. Saxena, R. Gupta, and A. K. Singh. A Survey and Comparative Study on Multi-Cloud Architectures: Emerging Issues And Challenges For Cloud Federation. *arXiv preprint arXiv:2108.12831*, 2021.

[84] Y. L. Simmhan, B. Plale, and D. Gannon. A Survey of Data Provenance in e-Science. *ACM Sigmod Record*, 34(3):31–36, 2005.

[85] M. J. Smith, C. Sala, J. M. Kanter, and K. Veeramachaneni. The Machine Learning Bazaar: Harnessing the ML Ecosystem for Effective System Development. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 785–800, 2020.

[86] B. C. Stahl. *Artificial Intelligence for a Better Future: An Ecosystem Perspective on the Ethics of AI and Emerging Digital Technologies*. Springer Nature, 2021.

[87] E. Stevens, L. Antiga, and T. Viehmann. *Deep Learning with PyTorch*. Manning Publications, 2020.

[88] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, 12(3):66, 2010.

[89] T. T. D. Team, R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, et al. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 2016.

[90] Z. Tong, X. Deng, H. Chen, J. Mei, and H. Liu. QL-HEFT: A Novel Machine Learning Scheduling Scheme base on Cloud Computing Environment. *Neural Computing &*

Applications, 32(10), 2020.

[91] H. Topcuoglu, S. Hariri, et al. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *TPDS*, 13(3):260–274, 2002.

[92] J. Traub, Z. Kaoudi, J.-A. Quiané-Ruiz, and V. Markl. Agora: Bringing Together Datasets, Algorithms, Models and More in a Unified Ecosystem [Vision]. *ACM SIGMOD Record*, 49(4):6–11, 2021.

[93] J. Traub, J.-A. Quiané-Ruiz, Z. Kaoudi, and V. Markl. Agora: A Unified Asset Ecosystem Going Beyond Marketplaces and Cloud Services. *arXiv preprint arXiv:1909.03026*, 2019.

[94] W. A. Warr. Scientific workflow systems: Pipeline Pilot and KNIME. *Journal of computer-aided molecular design*, 26(7):801–804, 2012.

[95] C. J. Watkins and P. Dayan. Q-Learning. *Machine learning*, 8(3-4):279–292, 1992.

[96] D. Weitzel, B. Bockelman, D. A. Brown, P. Couvares, F. Würthwein, and E. F. Hernandez. Data Access for LIGO on the OSG. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, pages 1–6. 2017.

[97] P. Wieder, J. M. Butler, W. Theilmann, and R. Yahyapour. *Service Level Agreements for Cloud Computing*. Springer Science & Business Media, 2011.

[98] S. Wienke, P. Springer, C. Terboven, and D. an Mey. OpenACC — First Experiences with Real-World Applications. In *European Conference on Parallel Processing*, pages 859–870. Springer, 2012.

[99] M. Wilde et al. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

[100] G. L. Wojcik, M. Graff, K. K. Nishimura, R. Tao, J. Haessler, C. R. Gignoux, H. M. Highland, Y. M. Patel, E. P. Sorokin, C. L. Avery, et al. Genetic Analyses of Diverse Populations Improves Discovery for Complex Traits. *Nature*, 570(7762):514–518, 2019.

[101] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster. Swift/T: Large-Scale Application Composition via Distributed-Memory Dataflow Processing. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 95–102. IEEE, 2013.

[102] J. M. Wozniak, R. Jain, P. Balaprakash, J. Ozik, N. T. Collier, J. Bauer, F. Xia, T. Brettin, R. Stevens, J. Mohd-Yusof, et al. CANDLE/Supervisor: A Workflow Framework for Machine Learning Applied to Cancer Research. *BMC bioinformatics*, 19(18):59–69, 2018.

[103] Q. Zhang, L. T. Yang, Z. Chen, and P. Li. A Survey on Deep Learning for Big Data. *Information Fusion*, 42:146–157, 2018.

[104] C. Zheng, B. Tovar, and D. Thain. Deploying high throughput Scientific Workflows on container schedulers with Makeflow and MESOS. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 130–139. IEEE, 2017.

ABSTRACT

DEEP LEARNING AS NATIVE SCIENTIFIC WORKFLOWS IN THE MODERN SWFMS - DATAVIEW

by

JUNWEN LIU

August 2022

Advisor:	Dr. Shiyong Lu
Major:	Computer Science
Degree:	Doctor of Philosophy

Scientific workflow has become a common practice for scientists to effectively formalize and structure complex scientific processes to accelerate many scientific discoveries in numerous research fields. With the recent thriving of deep learning and it application in broad scientific projects, there is a rising need for deep learning support in scientific workflow infrastructures. However, current GPU-enabled deep learning frameworks are developed separately, and their capabilities are not immediately accessible to scientific workflow management systems (SWFMSs). Consequently, scientists have to handle deep learning as a powerful tool outside of SWFMSs and then integrate it into workflows in an ad-hoc manner. What workflow users pressingly need today is a user-friendly and well-integrated SWFMS with built-in support of GPU-enabled deep learning as native (sub-)workflows so that they can conveniently design, train, reuse, share, and include deep learning models seamlessly into their scientific workflow projects. In this dissertation, I demonstrate our research outcome in supporting GPU-enabled deep learning at the infrastructure level in a popular SWFMS - DATAVIEW, which facilitates: 1) fast design, train and reuse neural

networks as native workflows per Deep-Learning-as-a-Workflow (DLaaW) via JAVA API or WebBench GUI; 2) flexibly leverage various types of GPU resources for executing deep learning workflows. 3) conveniently integrate NNTasks (tasks implementing neural network capabilities) with ordinary workflow tasks in one comprehensive workflow through JAVA API or GUI web interface. Our approach and implementations are thoroughly evaluated through experiments that demonstrate the efficacy and efficiency as compared to conventional PyTorch-based and Keras-based implementations.

AUTOBIOGRAPHICAL STATEMENT

EDUCATION

- Doctor of Philosophy (Computer Science), April 2022
  Wayne State University, Detroit, Michigan, United States
- Master of Science (Computer Science), August 2014
  Wayne State University, Detroit, Michigan, United States

PUBLICATION

- **Junwen Liu**, Ziyun Xiao, Shiyong Lu, and Dunren Che. "Deep-Learning-as-a-Workflow (DLaaW): An Innovative Approach to Enabling Deep Learning in Scientific Workflows." In 2021 IEEE International Conference on Big Data (Big Data), pp. 3101-3106. IEEE, 2021.
- **Junwen Liu**, Shiyong Lu, and Dunren Che. "A Survey of Modern Scientific Workflow Scheduling Algorithms and Systems in the Era of Big Data." In 2020 IEEE International Conference on Services Computing (SCC), pp. 132-141. IEEE, 2020.
- **Junwen Liu**, Ziyun Xiao, Shiyong Lu, Dunren Che, Ming Dong, and Changxin Bai. "Infrastructure-level Support for GPU-Enabled Deep Learning in DATAVIEW." *In Proc. of the IEEE Transactions on Services Computing Journal*, 2022. (submitted).
- **Junwen Liu**, Shiyong Lu, and Dunren Che. "A Survey on Scientific Workflow Scheduling Algorithms in the Cloud." *In Proc. of the Future Generation Computer Systems Journal*, 2022. (submitted).
- Changxin Bai, Shiyong Lu, Dunren Che, and **Junwen Liu**. "Deadline-Constrained Big Data Workflow Scheduling in the Cloud: the LPOD Algorithm." *In Proc. of the IEEE Transactions on Services Computing Journal*, 2021. (submitted).
- Changxin Bai, **Junwen Liu**, Shiyong Lu, and Dunren Che. "A Generic Efficient Workflow Executor for the Optimizations of Run-time Execution Of Workflow Schedules." *In Proc. of the IEEE International Conference on Services Computing (SCC)*, 2022. (submitted).

TEACHING

- CSC4110: Software Engineering, Spring/Summer 2021
  Wayne State University, Detroit, Michigan, United States
- CSC4710: Introduction to Database Management Systems, Spring/Summer 2020
  Wayne State University, Detroit, Michigan, United States