Infrastructure-level Support for GPU-Enabled Deep Learning in DATAVIEW

Junwen Liu[a,*], Ziyun Xiao[a], Shiyong Lu[a], Dunren Che[b], Ming Dong[a], Changxin Bai[a]

[a]*Department of Computer Science, Wayne State University, Detroit, MI, USA*
[b]*Department of Computer Science, Southern Illinois University, Carbondale, IL, USA*

Abstract

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 deep learning in broad range of scientific projects, there is a rising need for deep learning support in scientific workflow infrastructures - SWFMSs. However, current GPU-enabled deep learning frameworks are developed separately, not suitable for direct exploitation in SWFMSs, which forces scientists to handle deep learning 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 deep learning as native workflows so that they can conveniently design, train, reuse, and integrate deep learning models in comprehensive workflows. In this paper, We report our latest research progress in supporting GPU-enabled deep learning at infrastructure-level in a popular SWFMS - DATAVIEW, which facilitates: 1) fast design, train, reuse neural networks as native workflows per Deep-Learning-as-a-Workflow (DLaaW) or integrate pre-trained neural network models with ordinary Tasks in one comprehensive workflow via JAVA API or WebBench GUI; 2) flexibly leverage various types of GPU resources for executing deep learning workflows. Our approach and implementations are thoroughly evaluated through experiments that demonstrate the efficacy and efficiency as compared to conventional Pytorch-based implementations.

Keywords: SWFMS, Deep-Learning-as-a-Workflow, Neural Network, GPGPU, CUDA, GPU cluster.

1. Introduction

Scientific workflow modeling and execution using a SWFMS (Scientific Workflow Management System) has become a common practice for scientists to accelerate scientific discoveries across numerous scientific domains. For example, the national Ecological Observatory Network (NEON) [1] 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 [2] 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 [3] has been used by thousands of astronomers for constructing image mosaics of the sky. In the Bioinformatics field, the myExperiment repository [4] currently contains 3935 public scientific workflows shared by 11161 members from 429 groups. The Pegasus workflow system [5] 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, machine learning (ML), especially deep learning (DL), has become increasingly popular and been utilized in broad scientific processes and projects across nearly all scientific domains [6, 7]. 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 [8]. Although there are many GPU-enabled DL libraries available, such as PyTorch [9], Keras/TensorFlow [10], Theano [11] and Mxnet [12], 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 [13], 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 [14] in SWFMS tend to be time-consuming and less efficient; 3) the separate development of DL models and computation-intensive scientific workflows [14] based on different platforms tend to be complicated and error-prone.

One simple idea would be to adopt existing DL APIs (e.g. Pytorch/Keras Java APIs) in a SWFMS with the hope to quickly facilitate production-ready and user-friendly development of the DL functionalities (as components) for scientific workflows. However, as these DL APIs are not designed for scientific workflow environments, the "simple embrace" approach ineluctably bears limitations, of which we shed more light as follow. First, it is very hard to customize any real-time output from intermediate nodes/layers (e.g. feature maps) of developed DL mod-

*Corresponding author
Email addresses: junwen@wayne.edu (Junwen Liu), ziyun.xiao@wayne.edu (Ziyun Xiao), shiyong@wayne.edu (Shiyong Lu), dche@cs.siu.edu (Dunren Che), mdong@wayne.edu (Ming Dong), changxin@wayne.edu (Changxin Bai)

els 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 [15] (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 closed source in their intermediate level APIs. For example, cuDNN and cuBLAS are NVIDIA's intermediate level closed source DL APIs, which are built upon CUDA (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 are unpredictable to both SWFMS developers and workflow users.

Therefore, it is very necessary to provide a SWFMS with infrastructure-level support for GPU-enabled DL that is natively implemented and seamlessly integrated into the SWFMS. In our previous work [16], we proposed the DLaaW (Deep Learning 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 deep learning neural networks as native workflows from insfrastructure level in an integral SWFMS. More specifically, we 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 particular type of GPU Resources (e.g. Local GPU of a host PC, a NVIDIA SoM (System-on-Module)). Accordingly, we implemented a generic *GPU Resource Management* module to leverage diverse GPU resource configurations and maintain great extensibilty for incorporating any new GPU resource types in a long run. In our preliminary work [16], we focused on 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 [16], we have made tremendous progress, fully implemented our novel DLaaW approach that was introduced in [16] and thoroughly tested it. Our new implementations allow workflow users/developers to leverage a full life-cycle deep learning utility to not only design, construct and execute deep neural networks in the form of NNWorkflows, but also reuse pre-trained NNWorkflow models on new datasets for prediction in an ordinary workflow, or integrate these pre-trained models (executed on GPU) with ordinary workflow Tasks (executed on CPU) in one comprehensive workflow. All of above 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 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 available on Github. The current version - *DATAVIEW Release 3.0*, has been released in github.com.

Based on our preliminary work [16], our recent progress reported in this article makes following additional main contributions:

1. We introduce 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 neural networks as native workflows in DATAVIEW.

2. We introduce the graphical WebBench GUI that facilitate NNWorkflow in design, construction, run and reuse in DATAVIEW, or integrate pre-trained models with ordinary Tasks in one comprehensive workflow. The appealing intuitiveness of the GUI enhances the usability of DLaaW and SWFMS as a whole.

3. 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.

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

The rest of this paper is organized as follows: Section 2 reviews recent related work. Section 3 summarizes four research and two engineering challenges of supporting GPU-enabled DL capabilities at infrastructure-level in a general SWFMS framework. Section 4 presents the architecture of our DLaaW approach that addresses all research challenges described in Section 3. Section 5 elaborates on the core implementation of the DLaaW approach, in which we address all engineering challenges described in Section 3. Section 6 reports our experimental results that demonstrate the validity and efficiency of our DLaaW approach and its implementations in DATAVIEW. Section 7 concludes the article and points out some future research directions.

2. Related Work

As the technology of ML, especially DL, is fast advancing, the need for ML/DL across all application areas increases even faster. ML/DL has greatly contributed to scientific discoveries [13] in numerous disciplines. Along the same line, the need and desire for integrating ML/DL capabilities into SWFMSs have risen higher than ever. In this section, we review major

recent related works that we curated from Google Scholar and DBLP in combination with the snowballing method as they are relevant and compared to our work.

CPU-based ML for workflow scheduling: CPU-based ML has been exploited by workflow researchers on deriving better scheduling plans. For example, T.Miu and P. Missier [17] used the C4.5 Decision Tree algorithm on historical input data and makespans to train the model for estimating the makespans of scientific workflows on new input data. A. Nascimento et al. [18] promoted application of Reinforcement Learning (RL) and Q-Learning [19] 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. [20] proposed a task scheduling algorithm called QL-HEFT which combines Q-Learning with the HEFT algorithm [21] 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 we 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 [22] (designed for ordinary workflows) on optimizing NNWorkflows scheduling in the SWFMS.

CPU-based ML in workflow applications: Much work has been done to incorporate CPU-based ML applications in SWFMSs and workflows. A comprehensive survey [23] 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, we comment on more recent representative works published after 2018. I Ahmed et al. [24] implemented a semi-supervised clustering-based diagnosis recommendation model in DATAVIEW [25] SWFMS for improving the diagnosing accuracy via self-training and co-training of the model. N Radosevic et al. [26] utilized CPU-based Decision Tree in solar radiation modeling in the KNIME [27] SWFMS to increase reproducibility and warrantability of environmental models. 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.

Exascale GPU-based DL: Interesting works [28, 29, 30] have been done to exscale distributed deep learning based on existing DL libraries/APIs (as mentioned in Section 1). One particularly interesting project related to workflows is the CANDLE [30] project carried out at Argonne National laboratory. This project aims to develop exascaled deep learning networks (trained on massive datasets) for accelerating cancer research. CANDLE is built upon Swift/T [31] which is a well-known SWFMS centered on the Swift language, involving various HPC schedulers to leverage DOE super computing resources for exascaling deep learning instances. Under the hood, CANDLE distributes time-consuming deep learning 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, workflow users inevitably need to face the inherited limitations/issues from third-party intermediate level DL APIs in SWFMSs (as we pointed out in Section 1).

Unified Ecosystem for AI applications: There are works dedicated to construction of unified frameworks/systems for developing AI applications. For example, Bazaar [32], 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. WOLF [33], is an automated machine learning workflow management framework designed to simultaneously automate the process of selecting the best algorithm and searching for the optimum hyperparameters. In contrast, our work emphasizes on something bigger – not merely implementing ML/DL as workflows but facilitating ML/DL workflows' further and seamless integration into general scientific workflows that require ML/DL capabilities. On the other hand, Agora [34], 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 [35] 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.

3. Challenges of infrastructural support on GPU-enabled DL in SWFMSs

In order to allow workflow users, i.e., scientists and engineers, to be able to fully leverage the power of ML/DL, a seamless integration of DL capabilities into existing SWFMSs is needed, which brings several important benefits, 1) the programmatic and graphical design interfaces, which a SWFMS such as DATAVIEW typically has, are made readily reusable for the designs of NNWorkflows; 2) supporting neural network represented as a native workflow in the SWFMS, executed and reused in the same manner as an ordinary scientific workflow, which in turn facilitates seamless integration of NNWorkflows (as components) into comprehensive scientific workflows; 3) the existing optimization and scheduling algorithms [22] shall potentially be extended to NNWorkflows for expedited execution. To achieve these benefits, 4 major research challenges (3.1-3.4) and 2 major engineering challenges (3.5-3.6) are identified and need to be overcome. These challenges are discussed in the subsequent subsections:

3.1. NNWorkflows construction Challenge

To construct a neural network as a native workflow in a SWFMS, firstly we need well-defined neural network tasks (or NNTasks) which constitute a workflow. Typically, a scientific workflow is constructed by connecting/pipelining constituent Tasks through their input/output ports, and dispatched to suitable hardware resources for execution by workflow executors. Traditional scientific workflows are formally defined and described as directed acyclic graphs (DAGs) [36], in which data always flow from the starting node(s) to the ending node(s) in an acyclic manner and each task is visited exactly once. However in DL, a neural network can be executed in a certain number of epochs, this means that each task needs to be revisited (during training) a certain number of times, and with each visit the relevant weights (of the learning model) are updated and saved until all epochs are finished. Generally, there are at least three granularity options for the construction of NNTasks: 1) A-Layer-as-a-Task, consisting of multiple neurons laid out in one layer is constructed as a task in an NNWorkflow and will carry out the forward/backward propagation of the neural network in a data-driven manner (where the execution is triggered whenever the set data dependencies become ready); 2) A-Neuron-as-a-Task, in which each single neuron is constructed as a task in an NNWorkflow, and each neuron would independently take care of its input and output data in a data-driven manner; and 3) A-Whole-Neural-Network-as-a-Task, in which a neural network is constructed as a single task in an enclosing scientific workflow. The decision can greatly affect the complexity of the NNWorkflow construction, the mapping and the implementation of in-house low level DL APIs for GPU enabled deep learning.

3.2. CPU&GPU communication Challenge

GPUs were originally designed to create images for computer graphics and video games, but since the early 2010's, GPUs has been increasingly used to accelerate calculations involving massive amounts of data. As the representation of data in neural networks are tensors and the computation on which basically consists of massive repeated operations, thus modern GPUs are optimized for training DL neural networks due to their superior capability on simultaneously processing thousands of cores in parallel, which can greatly boost the performance on the training process of a neural network. However, conducting such General-purpose GPU (GPGPU) computing is still rather abstruse due to the substantial difference between CPU and GPU computing in hardware architecture, computing mechanisms and programming languages [37]. Thus, communication between CPU and GPU becomes a big challenge involving the following two aspects: 1) implementing proper interfaces to bridge the gap between native SWFMS and the GPGPU computing, since modern SWFMSs are all built upon the CPU infrastructure (i.e. CPU based hardware and operating systems) [22]; 2) lubricating the CPU&GPU collaboration in GPGPU computing, as all GPU computing are initiated, coordinated by and finally reduced to CPU. Such process involves loads of synchronization, data copy, etc.

3.3. CPU&GPU I/O overhead Challenge

Though the interests in leveraging the infrastructure-level support of GPUs in SWFMS are high, the data communication between GPU and CPU are expensive and can be a potential bottleneck of performance. As the input&output ports of NNWorkflow tasks are residing on the CPU side, pipelining the input&output of relevant tasks into&out from GPU involves excessive I/O overhead, considering there may be hundreds or thousands of neurons in a neural network. The accumulated I/O cost between CPU and GPU thus can be enormous and overwhelming, and remains as a stumbling block preventing traditional SWFMSs from incorporating the GPU computing power at the infrastructure level. For this reason, designing an efficient data transportation mechanism for NNWorkflows that minimizes the I/O communication between CPU and GPU for NNWorkflow tasks is a major challenge that is needs to be solved for adequately leveraging GPU enabled executions in a native workflow in traditional SWFMSs.

3.4. 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 may be composed of arbitrary types, arbitrary number and in arbitrary order of neural network neurons and layers, such CPU-to-GPU mapping mechanism needs to be generic and dynamic so that any legitimate 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 specification mapping mechanism is another major challenge for incorporating GPU enabled execution in traditional SWFMSs.

3.5. GPGPU DL services implementation Challenge

Currently, there exist several popular computing platforms and models for GPGPU computing: 1) NVIDIA's CUDA, 2) OpenCL, and 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, construction and execution of neural network on any one of these GPU computing platforms remains a great challenge, which includes implementing various neural network layers, constructing the architecture of neural network, conducting forward&backward propagation across layers, etc. These are all non-trivial issues that need to be addressed by a SWFMS in order for DL to become readily available functionalities for any scientific workflows.

3.6. Challenge of uniformly supporting diverse GPU resources

Recognizing the fact that different GPU resources require different parallel 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 implementation of these interfaces (e.g. message passing interface, procedural calls, Java native interface) that bridge CPU with a particular GPU, on receiving the same NNWorkflow specification, all GPU services on different infrastructures should uniformly construct the same neural network and conduct execution accordingly. Thus, implementing a standardized protocol for various GPU Services backed by various types of GPU resources is yet another major challenge in our way.

4. Architecture

In order to bring DL as a native functionality into modern SWFMSs and to address challenges outlined in the Section 2 based on DATAVIEW, we propose 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, we particularly address 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.

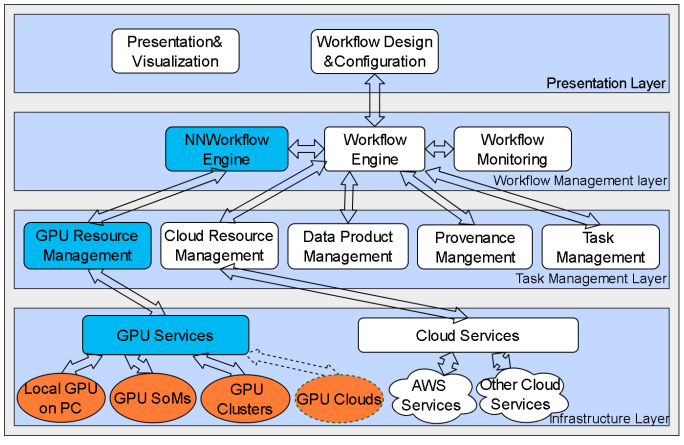

Figure 1: DATAVIEW's new architecture with inherent support for deep-learning-as-a-workflow.

Fig. 1 shows the new architecture of DATAVIEW which provides inherent support for DL through the DLaaW approach. In the design and implementation of this architecture, we have made great effort to decouple each component from others to enable it greater extensibility not only for the good of our own system but for broader extensibility of work to other SWFMSs. The original architecture of DATAVIEW consists of the following main components (shown in white rectangles in Fig. 1): 1) the *Workflow Design and Configuration* component, which provides intuitive programmatic and 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 DATAVIEW's original architecture, the new architectural design adds the following new DL-specific components (shown in light blue rectangles in Fig. 1): 1) *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) *GPUResourceManagement* Component, which, upon receiving the NNWorkflow specification, acts as a unified gateway to route the specification to the target GPU services; 3) *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. As preliminary implementation, we have not done any special modifications in the original provenance management component for supporting DL-specific components at the time of this writing. Instead, we leave it as a possible future task to leverage special provenance data for more efficient support DL functions.

Below, we 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.

4.1. 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 *NeuralNetworkExecutors*. 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.

In DATAVIEW, we adopt 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, by considering

the trade-off between the granular control over the GPU management and the ease of network specification from the user perspective, 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. In this way, our system would provide workflow users a similar level of abstraction to Keras or Pytorch, and meanwhile preserve great flexibility of customization for SWFMS developers. Up to now, our system has no support for existing Keras/Pytorch code to be wrapped and reused directly in scientific workflows, we are planning to provide multiple cross-platform parsers in future releases to support mapping users' existing DL code (e.g. Keras, Pytorch) automatically to NNWorkflows, which may give users the greatest convenience to adapt their existing work to our system.

The *NeuralNetworkTrainers* 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. At a 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 *GPUResourceManagement* 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 pretrained 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 *GPU Resource Management* 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, we can satisfactorily address Challenge 3.1 and 3.4 as discussed earlier Section 3.

4.2. GPU Resource Management Component

Upon receiving the aggregated NNWorkflow specification from the *NNWorkflowEngine* component, the *GPU Resource Management* 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 *NNWorkflowEngine* 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.

In this way, NNTasks (layers) with their input/output ports would be mapped from the CPU side to the GPU side, thus, 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.

4.3. 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 *NNWorkflow Engine* and *GPU Services* components to gain implementation independence for salient 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.

5. Implementation

In DATAVIEW, native deep learning capability is implemented according to our DLaaW approach (i.e., a deep learning network as a workflow) and the design choice of a-layer-as-a-task. A number of new architectural components (as shown in Fig. 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 deep learning not only supports scientists to conveniently

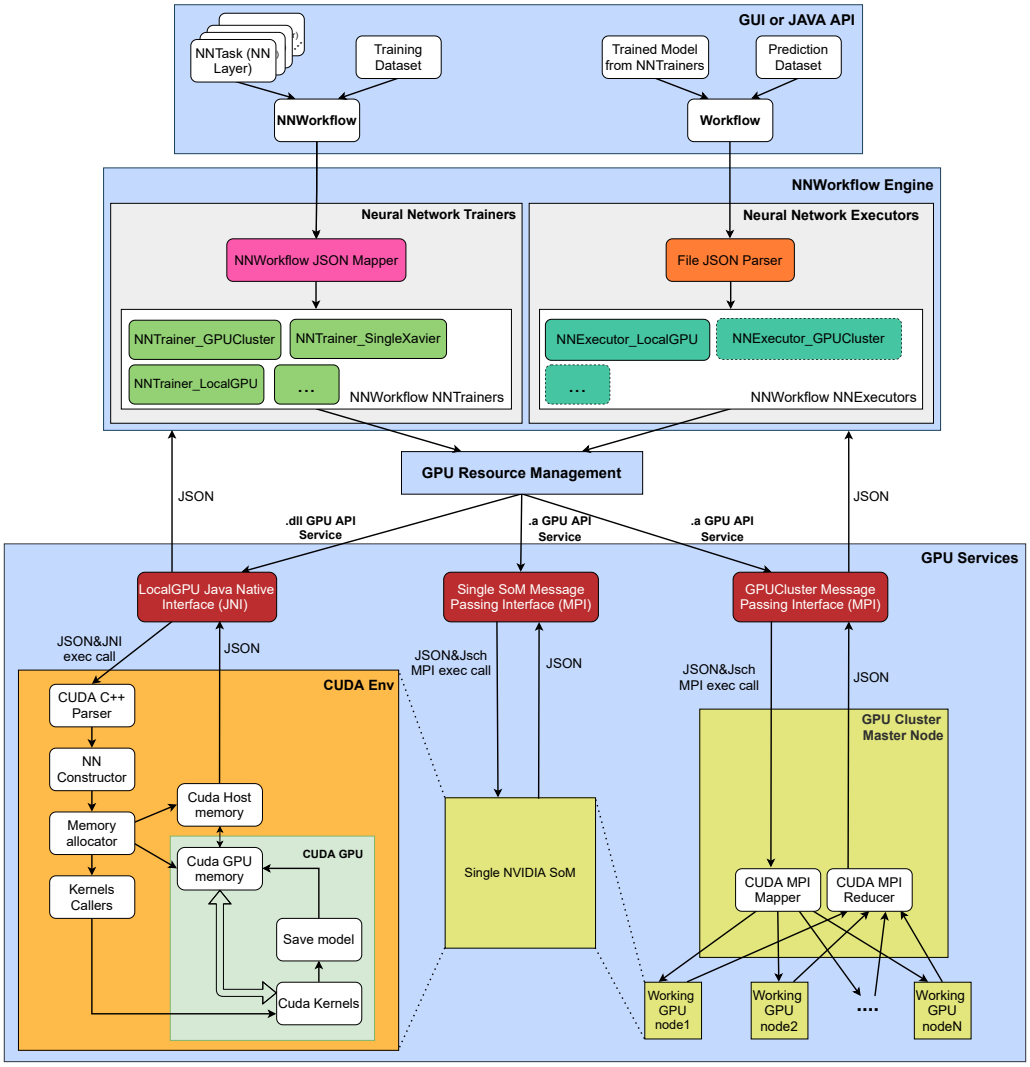

Figure 2: Procedures to construct, execute and reuse a deep-learning-as-a-workflow in DATAVIEW.

build and orchestrate any NNWorkflow at any scale, but also aid them to configure suitable execution plans on available GPU resources for particular DLaaWs.

Fig. 2 shows the procedural layout of our current implementation of the DLaaW in DATAVIEW, which lays out the details of the three new components illustrated in Fig. 1. In this section we discuss the implementation details of the DLaaW approach and explain how the 2 major engineering challenges (3.5- 3.6) are addressed in the implementation.

5.1. 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 and 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. Listing 1 shows a sample Java code of the SampleNNWorkflow class (a subclass of NNWorkflow) to construct a simple NNWorkflow that models on the input dataset Breast_cancer_data.csv and output the modeling results to the output.txt. This NNWorkflow 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. The input dataset will be split into 5 batches and the model will be trained in 1000 epochs (as specified in the Sequential() call). Fig. 3 shows the visualization of this sample NNWorkflow produced by the *Presentation&Visualization* component in DATAVIEW.

Listing 1: Construct a sample NNWorkflow with 4 layers:

```
wins[0] = "Breast_cancer_data.csv";
wouts[0] = "output.txt";

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, 5, 1000);
}
```

Listing 2: Construct a sample Workflow that reuse the NNWorkflow trained model:

```
wins[0] = new DATAVIEW_BigFile("model@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);
}
```

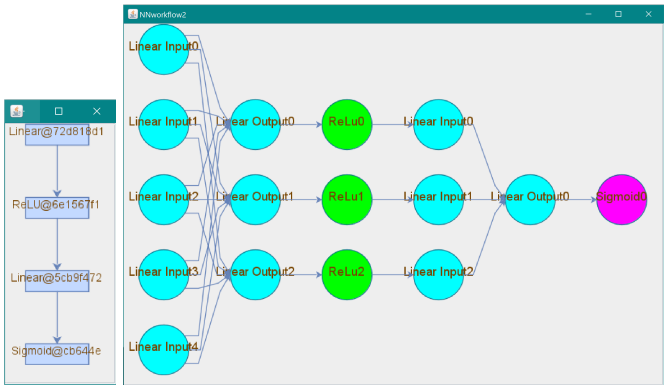

Figure 3: NNWorkflow Visualization by DATAVIEW: i) The sample NNWorkflow and ii) its Neuron-level architecture.

Alternatively, the same SampleNNWorkflow can be designed, constructed and run using DATAVIEW's webBench GUI in a drag-and-drop manner, as showed in Fig. 4. 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.

In addition, a deep-learning enabled 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 2 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, model@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 Fig. 5. As pre-trained NNWorkflow models are designed, constructed and run as an ordinary workflow, they can leverage all existing ordinary workflow executors (for ordinary workflows) in DATAVIEW, including WorkflowExecutor_Beta [38], WorkflowExecutor_Local [39], etc.

Furthermore, NNTasks can be integrated as a part of one comprehensive workflow, which consists of both NNTasks (executed in GPU) and ordinary Tasks (executed in CPU). We also provides two options (programmtically&graphically) for workflow users to construct comprehensive workflows. For example, a simple use case of Ensemble Learning on neural networks includes 3 weak NNWorkflow classifiers (pre-trained NNWorkflow models) and one voting task. By leveraging these weak classifiers to independently output their predictions on any particular data entry, the final prediction of that data entry can be derived by a voting process by majority votes of 3 classifiers.

Listing 3 shows sample Java Code of above comprehensive neural network ensemble learning workflow, which consists of 1) 4 inputs, including three pre-trained NNWorkflow models (model@749702, @749540, @749503) 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 output file (output.txt).

Listing 3: Construct a comprehensive ensemble learning neural network workflow:

```
wins[0] = new DATAVIEW_BigFile("model@749702");
wins[1] = new DATAVIEW_BigFile("model@749540");
wins[2] = new DATAVIEW_BigFile("model@749503");
wins[3] = new DATAVIEW_BigFile("Breast_cancer_data.
    csv");
wouts[0] = new DATAVIEW_BigFile("output.txt");

public void design()
{
    Task stage1 = addTask("NNExecutor");
    Task stage2 = addTask("NNExecutor");
    Task stage3 = addTask("NNExecutor");
    Task stage4 = addTask("Ensemble_vote");

    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);
}
```

Fig. 6 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. After clicking the Run button, the prediction will be carried on in GPU and the results will be showed in a separated pop-up window, in which each value represents the voted outcome (based on the 3 weak classifiers' independent predictions) for a particular row/record in the input dataset.

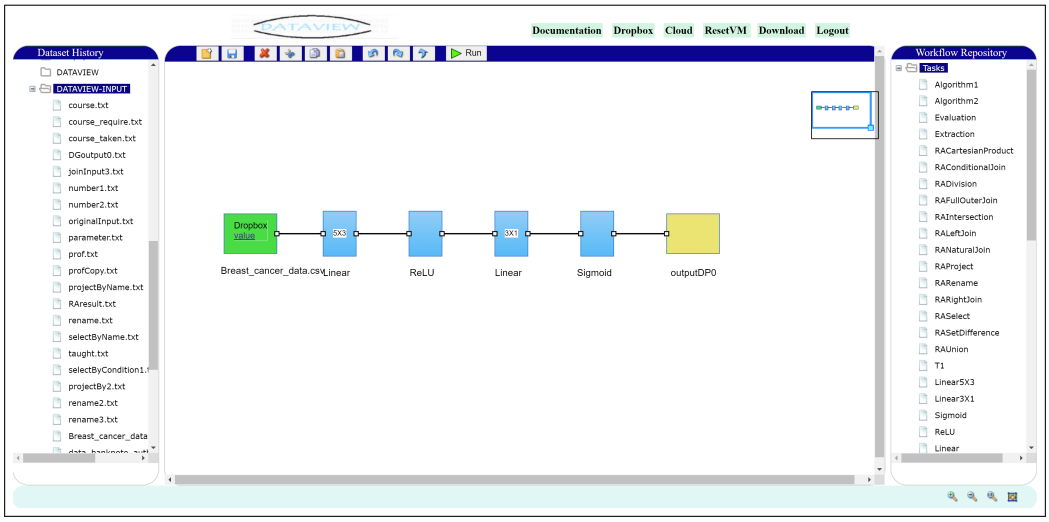

Figure 4: Design and construct an NNWorkflow in DATAVIEW webBench.

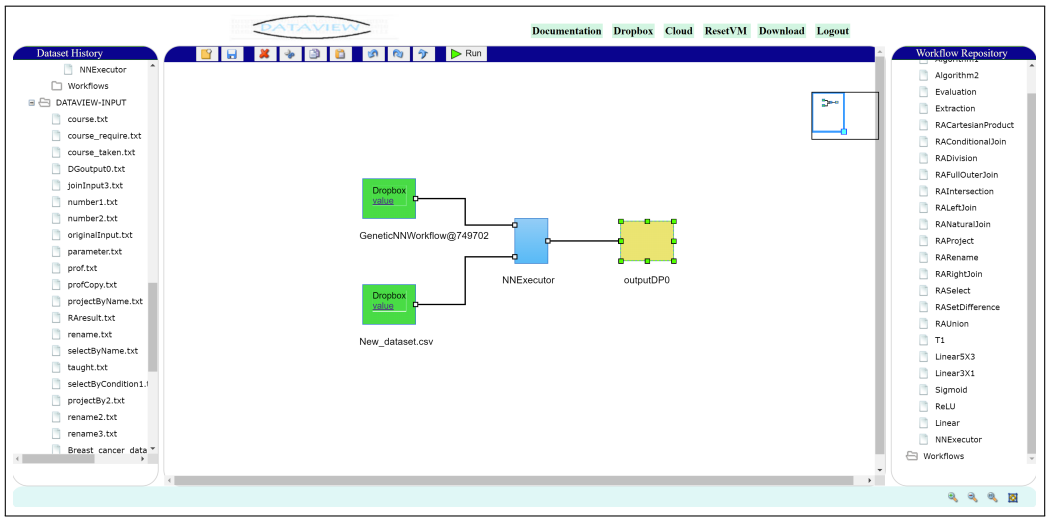

Figure 5: Reuse a trained NNWorkflow model on new dataset for prediction in DATAVIEW webBench.

The above three samples show that the established usability of DATAVIEW [25] (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.

5.2. NNWorkflow Engine: from native NNWorkflow to GPU recognizable specification

5.2.1. 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. A sample GPU recognizable specification mapped from the native NNWorkflow is included in Appendix A. At the time of this writing, the NNWorkflow Mapper contains one specific mapper - NNWorkflow JSON mapper, alternative mappers will be investigated in the future.

Listing 4: Sample code of selecting runing NNTrainers for a NNWorkflow:

```
{
NNTrainer_LocalGPU trainer1 = new NNTrainer_LocalGPU
    (w);
NNTrainer_crossValOnGPUCluster trainer2 = new
    NNTrainer_crossValOnGPUCluster(w);

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

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 specification. The aggregated NNWorkflow specification is then sent to the *GPU Resource Management* component by calling the *train*() method in the NNTrainer. Listing 4 is the sample code showing a sample NNWorkflow w, being fed into two NNTrainers to respectively trigger their *train*() methods. DATAVIEW 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 NNWorkflow w,

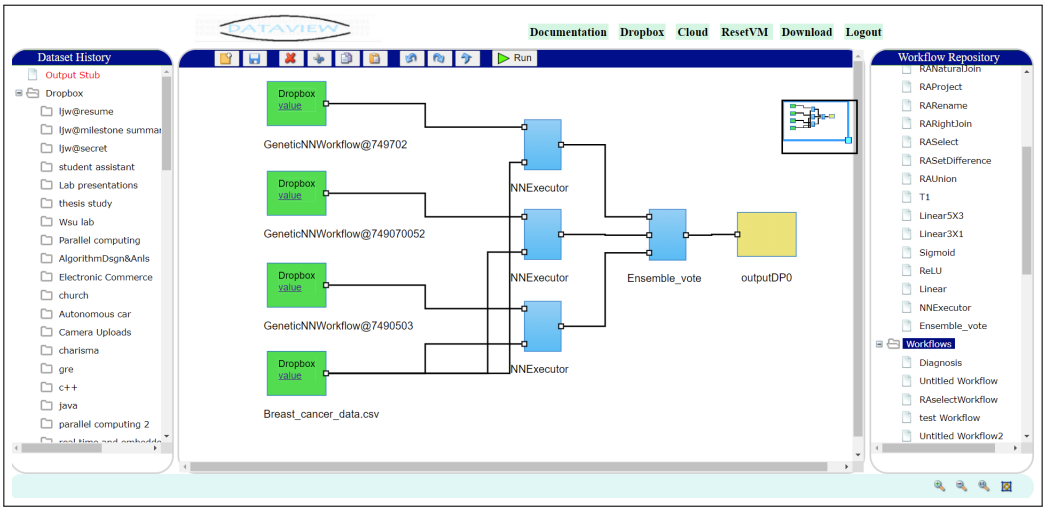

Figure 6: Construct and run neural network ensemble learning workflow in DATAVIEW webBench.

NNTrainer_LocalGPU construct a regular train&test plan running on the local GPU of a host 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.

5.2.2. *Neural Network Executors*

The File JSON Parser module (Written in Java) reads and parses primitives of pre-trained NNWorkflow models into the counterparts that are recognizable by the backend GPU services. The GPU recognizable specification mapped from the trained model from model@749702 in subsection 5.1 is included in Appendix B. At present, our system supports one specific parser - File JSON Parser. we plan to support more alternative parsers in future releases.

Currently, DATAVIEW supports one NNExecutor - NNExecutor_LocalGPU, which is capable to apply any pre-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, we have provided a complete life-cycle DL use case (from design to reuse on new datasets) in DATAVIEW via both JAVA API and WebBench GUI.

5.3. *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).

5.4. *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, we support three types of GPU infrastructure for the execution of NNWorkflows in DATAVIEW as follow:

5.4.1. *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:

$$X_{new} = (X - X_{min})/(X_{max} - X_{min}) \quad (1)$$

Finally, the actual training is kicked off for a user-specified number of epochs through forward and backward propagations, during which respective CUDA kernels are called within their layer objects to carry out parallel computation on thousands of GPU cores. 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 via JNI.

On the other hand, to reuse a pre-trained NNWorkflow model on a given new dataset, the trained model (i.e. model architecture, saved weights and bias in each layer) gets reloaded in the target GPU resource. The execution of neural network is carried out just like an ordinary workflow but leveraging the GPU environment. 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.

5.4.2. 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.

5.4.3. 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 utilize 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.

6. Experiments

In order to validate our proposed approach, we conducted three groups of experiments, including: 1) two groups of experiments to compare the performance of DLaaW implementation and the counterpart python implementation on Pytorch in the same 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; and 2) one group of experiments to analyze different values of hyperparameters for each network under the same DLaaW infrastructural setting, in order to compare different configurations of the neural network for the same problem and pick up the best for the problem. These three groups of experiments are as follows:

i We adopt 4 infrastructures, to train and test 5 neural networks respectively designed for 5 popular binary classification datasets with hyperparameter set 1 (characterized in table 1). 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 first setting utilizes JNI and the last two leverage MPI calls to pass information between CPU and GPU.

ii We adopt 5 infrastructures to conduct 5-fold cross validation on the same neural networks, using the same datasets with hyperparameter set 1 as showed in table 1. 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).

iii We adopt 4 different hyperparameter sets - hyperparameter set 1 to 4 as showed in table 1. Under the same DLaaW infrastructural setting, we conduct 5-fold cross validation on these sets of hyperparameters for each neural network on the local GPU of a host PC of our system.

6.1. Hardware

In our preliminary implementation of DLaaW in DATAVIEW, the hardware we adopted include: 1) A x64-based Windows Desktop with AMD Ryzen 5 3600 6-core CPU, 16GB DDR4 RAM, 500GB SSD, NVIDIA GeForce RTX 2080 Super GPU with 3072 CUDA cores and 8GB GDDRR6 memory; 2) A heterogeneous GPU cluster, 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; and 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.

6.2. Datasets

We adopted 5 representative datasets 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

Table 1: 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 5 features and 569 instances	Pima Indians Diabetes Dataset with 8 features and 768 instances	Data Banknote Authentication Dataset with 4 features and 1372 instances	Electrical Grid Stability Dataset with 13 features and 10000 instances	Bank Marketing Dataset with 16 features and 45211 instances
Initialization	Xavier weight init, 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
Hyperparameter set 1	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
Hyperparameter set 2	learningRate=0.01, momentumSGD =0.9	learningRate=0.01, momentumSGD =0.9	learningRate=0.01, momentumSGD =0.9	learningRate=0.01, momentumSGD =0.9	learningRate=0.01, momentumSGD =0.9
Hyperparameter set 3	learningRate=0.01, momentumSGD =0.1	learningRate=0.01, momentumSGD =0.1	learningRate=0.01, momentumSGD =0.1	learningRate=0.01, momentumSGD =0.1	learningRate=0.01, momentumSGD =0.1
Hyperparameter set 4	learningRate=0.1, momentumSGD =0.1	learningRate=0.1, momentumSGD =0.1	learningRate=0.1, momentumSGD =0.1	learningRate=0.1, momentumSGD =0.1	learningRate=0.1, momentumSGD =0.1
# of batches	6	6	6	6	6
# of Epochs	1000	1000	1000	1000	1000

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.

We selected and adopted the 5 datasets by considering their 1) popularity, all the datasets are with high popularity among Machine Learning 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 machine learning repository, which is one of the most popular machine learning repositories with more than 3400 citations [40]; 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.

6.3. Experiment Results

6.3.1. 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 Fig. 7. The bar charts on the top 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 1) implemented through DLaaW as NNWorkflows. Compared to Pytorch-based implementations, NNWorkflows consistently outperform in 4/5 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 we 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 we list out 3 most inclined reasons: 1) our in-house CUDA services wrapped 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 DATAVIEW and Pytorch adopts Xavier weights initialization [41], the random initialization can result in the same neural network being trained to different local optimals (in a non-convex optimization problem). On the other hand, Pytorch-based implementation outpaces DLaaW on accuracy in the third dataset, for the reason that we are not able to look into the close source intermediate level APIs in Pytorch, there can be many reasons that lead this exception, for example, different data shuffle, different weight initialization, etc. Thus, we won't be able to precisely locate the root of cause of this exception. 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 deep learning and provide the functionality to broad scientific workflows in

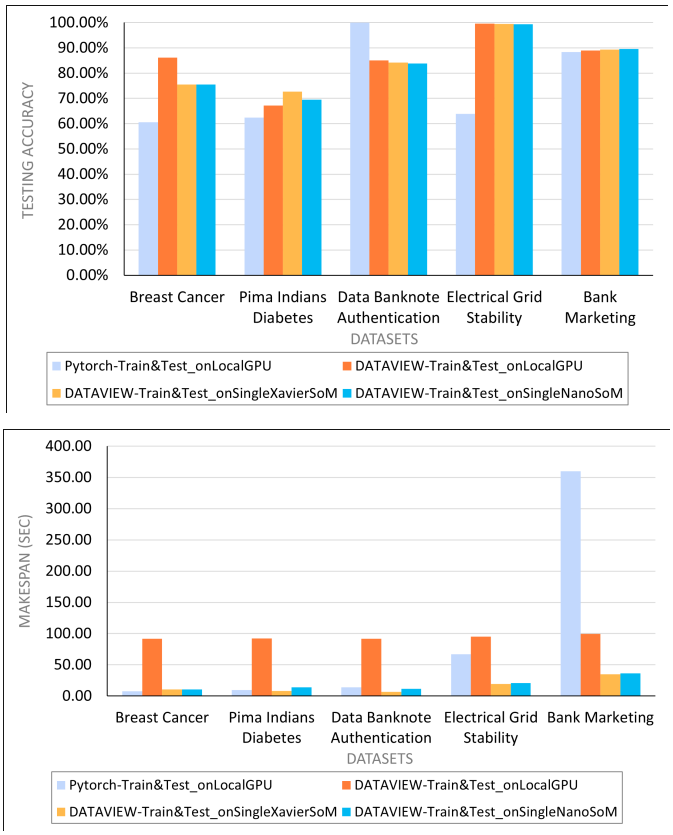

Figure 7: Regular train and test: i) Testing accuracies and ii) Timespans (in seconds)

Figure 8: 5-fold cross validations: i) Average testing accuracies on 5 folds; ii) Timespans (in seconds)

DATAVIEW.

The bar charts in the bottom of Fig. 7 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 Pytorch-based implementation of neural networks may suffer from bad scalability (in terms of data size), which is also caught by other Keras developers in [16] and Pytorch developers[1,2]. Having this same issue in both Keras and Pytorch may suggest that the root cause of this scalability issue potentially be persisted in their commonly used closed-source intermediate level APIs (e.g. cuDNN, cuBLAS). The scalability issue of Pytorch 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 in exponential rate as the data size grows larger. In contrast, NNWorkflows implemented per our DLaaW approach enjoy great scalability. Across the different datasets of varied sizes, the DLaaW implementations on local GPU shows pretty consistent but slighly larger timespans than other DLaaW implementations, such observation likely suggests that the communication through JNI is less efficient than communication through MPI based on our best understanding. 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 and leads to much better scalability compared with DLaaW implementation on local GPU and Pytorch-based implementation. Apart from the more efficient MPI-based communication, our DLaaW implementation on a single SoM gained extra improvement on timespan as compared to the local implementation owing to the Jetson zero-copy mechanism adopted by NVIDIA Jetson SoMs, where CPU and GPU physically share the same system memory so that the synchronization overhead between CPU and GPU can be greatly alleviated.

6.3.2. *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. We thus 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 implemen-

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

[2]Pytorch issue #2829 on Github.com,`https://github.com/pytorch/fairseq/issues/2829`

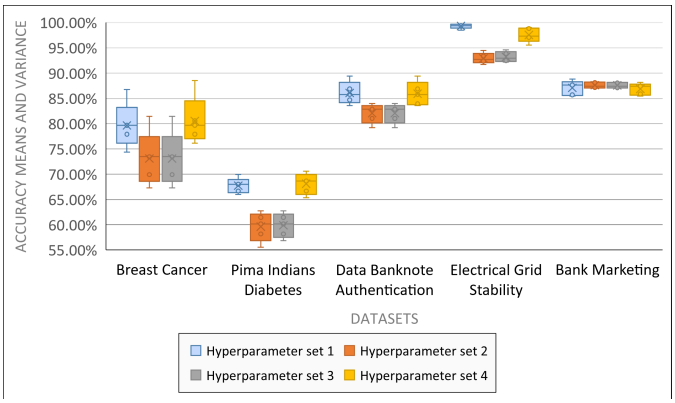

Figure 9: Hyperparameter sets testing accuracies comparison

tation 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 results of this group of experiments are showed in Fig. 8. The top chart shows the mean and variance of validation accuracies on each neural network across 5-fold cross validation. All DLaaW implementations in DATAVIEW consistently deliver competitive 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 the 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 5th) across all infrastructural settings, which is likely to suggest the data bias [42] exerts higher impact on smaller datasets while conducting train and test on different portion of data.

The bottom chart in Fig. 8 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 5-fold 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, which sets a solid foundation and an attractive incentive on further exploiting more advanced features of deep learning in the SWFMS as future work.

6.3.3. Experiment Group III

This group of experiments analyzes different values of hyperparameters for each neural network, and this capability of our system helps users evaluate different configurations of neural networks and pick up the best suitable.

The results of this group of experiment are showed in Fig. 9, where the mean and variance of 5-fold cross validation accuracies on each hyperparameter set of the neural networks are presented. We can see hyperparameter sets 1 and 4 (with learning rate = 0.1) derive higher mean and lower variance of accuracies than hyperparameter sets 2 and 3 (with learning rate = 0.01) in the first two datasets. For the third and fourth datasets, hyperparameter sets 1 and 4 still preserve higher accuracy, but we observe that hyperparameter sets 2 and 3 are getting lower variance as the size of dataset grows larger (compared with the first two datasets). Hyperparameter sets 2 and 3 finally reach comparable accuracies and lower variance compared with hyperparameter sets 1 and 4 in the fifth dataset, which indicates hyperparameter set 2 and 3 would produce comparable (in terms of accuracy) but more stable models in the last dataset. One explanation to such scenario is that the learning rate in hyperparameter sets 2 and 3 is much smaller than that in hyperparameter sets 1 and 4, which causes the resulted models from hyperparameter sets 2 and 3 likely to be underfitted in small datasets – in other words, they are not fully trained because of small learning steps and small amount of training data. However, such case could be ameliorated when data size grows larger, and ultimately hyperparameter set 2 and 3 could reach very comparable accuracy to sets 1 and 4. Moreover, smaller learning rate is likely to help with a smaller variance as the learning steps are more fine-grained, which can be helpful to avoid overshooting the local optimal during the stochastic gradient descent.

7. Conclusions and future work

In this paper, we presented DLaaW, a novel approach to implementation of deep learning as native workflows in a scientific workflow environment for seamless integration with broad scientific workflows. We demonstrated its efficacy through system architectural design, implementation, and extensive empirical evaluation. Our goal is to enable workflow users to conveniently structure, execute and reuse any GPU-enabled neural networks as native standalone workflows or integrate into more comprehensive scientific workflows via either Java API or a graphical webBench interface in a general scientific workflow management system – DATAVIEW. This work resulted in DATAVIEW as the first scientific workflow management system that supports GPU-enabled deep learning at the infrastructure level. Currently, we 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 carefully designed experiments, we validated our proposed DLaaW approach and the correctness of its implementations on different GPU infrastructures, and demonstrated the effectiveness and efficiency (in terms of prediction accuracy and execution timespan) by comparing with the counterpart Python implementations on the cutting-edge DL library - Pytorch. We

show that DLaaW can gracefully leverage the superior parallelism offered by SWFMSs on boosting the DL performance. We conducted additional experiments demonstrating the usage of our proposed system on analyzing different values of hyperparameters through 5-fold cross validation on various neural networks and datasets.

As future work, we plan to investigate and incorporate more GPU services, enrich CUDA APIs implementations, and provide DLaaW as an open service for extended usability beyond our own SWFMS – DATAVIEW – and our own community.

Acknowledgements

This work is partially supported by National Science Foundation under grant CNS-1747095 and OAC-1738929.

References

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

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

[3] E. Deelman, G. Singh, M. Livny, B. Berriman, 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, Ieee, 2008, pp. 1–12.

[4] 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) (2010) W677–W682.

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

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

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

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

[9] 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 (2019) 8026–8037.

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

[11] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, Y. Bengio, Theano: New features and speed improvements, arXiv preprint arXiv:1211.5590 (2012).

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

[13] M. Raghu, E. Schmidt, A Survey of Deep Learning for Scientific Discovery, arXiv preprint arXiv:2003.11755 (2020).

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

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

[16] J. Liu, Z. Xiao, S. Lu, 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), IEEE, 2021, pp. 3101–3106.

[17] T. Miu, 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, IEEE, 2012, pp. 64–72.

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

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

[20] Z. Tong, X. Deng, H. Chen, J. Mei, H. Liu, QL-HEFT: A Novel Machine Learning Scheduling Scheme base on Cloud Computing Environment, Neural Computing & Applications (2020) 32 (10).

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

[22] J. Liu, S. Lu, 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), IEEE, 2020, pp. 132–141.

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

[24] I. Ahmed, S. Lu, C. Bai, F. A. Bhuyan, Diagnosis Recommendation Using Machine Learning Scientific Workflows, in: 2018 IEEE International Congress on Big Data (BigData Congress), IEEE, 2018, pp. 82–90.

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

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

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

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

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

[30] 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) (2018) 59–69.

[31] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, 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, IEEE, 2013, pp. 95–102.

[32] M. J. Smith, C. Sala, J. M. Kanter, 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, 2020, pp. 785–800.

[33] S. Kiani, S. Awan, J. Huan, F. Li, B. Luo, Wolf: automated machine learning workflow management framework for malware detection and other applications, in: Proceedings of the 7th Symposium on Hot Topics in the Science of Security, 2020, pp. 1–8.

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

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

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

[37] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, T. J. Purcell, A Survey of General-Purpose Computation on Graphics Hardware, in: Computer graphics forum, Vol. 26, Wiley Online Library, 2007, pp. 80–113.

[38] A. Mohan, A. Kashlev, C. Bai, S. Lu, DATAVIEW Release 2.0, `https:`

//github.com/shiyonglu/DATAVIEW/releases/tag/2.0 (2019).
[39] C. Bai, J. Liu, I. Ahmed, S. Lu, DATAVIEW Release 2.1, https://github.com/shiyonglu/DATAVIEW/releases/tag/2.1 (2019).
[40] D. Dua, C. Graff, et al., UCI Machine Learning Repository (2007).
[41] S. K. Kumar, On Weight Initialization in Deep Neural Networks, arXiv preprint arXiv:1704.08863 (2017).
[42] C. DeBrusk, The Risk of Machine Learning Bias (And How to Prevent It), MIT Sloan Management Review (2018).