Deep-Learning-as-a-Workflow (DLaaW): An Innovative Approach to Enabling Deep Learning in Scientific Workflows

Junwen Liu*, Ziyun Xiao*, Shiyong Lu*
Department of Computer Science
*Wayne State University**
Detroit, Michigan, USA
*Email: {junwen, ziyun.xiao, shiyong}@wayne.edu**

Dunren Che†
School of Computing
Southern Illinois University†
Carbondale, Illinois, USA
Email: dche@cs.siu.edu†

***Abstract*—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 deep learning models in many scientific applications, there is a rising need for infrastructure-level support for deep learning technologies in scientific workflow cyberinfrastructures. However, current scientific workflow cyberinfrastructures and GPU-enabled deep learning frameworks are developed separately, neither alone can be a satisfactory choice. In this paper, We propose the Deep-Learning-as-a-Workflow approach in DATAVIEW, which for the first time incorporates native infrastructure level support for GPU-enabled deep learning in a scientific workflow management system and enables the fast training and execution of neural networks as workflows (NNWorkflows) leveraging various types of GPU resource configurations. Our experiments demonstrate the salient usability feature of DATAVIEW in providing seamless infrastructure-level support to both scientific and deep learning workflows in one system, while delivering competitive (better in most cases) learning efficiency compared to the conventional implementations based on Keras.**

***Keywords*-DATAVIEW; Workflow Management System; Workflow; DLaaW; Deep-Learning-as-a-Workflow; Deep learning; Neural Network; GPGPU; CUDA; NVIDIA GPUs;**

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 [1]. In the CyberShake project, more than 230 scientific workflows were used by seismologists to generate seismic hazards maps in one year alone [2]. In Bioinformatics, the myExperiment website currently contains 3935 public scientific workflows shared by 11161 members from 429 groups [3]. The Pegasus workflow system [4] 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, machine learning (ML), especially deep learning (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 [5] and PyTorch [6], 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 [7], 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 [8] from a Scientific Workflow Management System (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 [9] 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 paper address the above limitations and makes the following contributions:

1) 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 deep learning neural network as a *native* workflow in a workflow management system.
2) 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.
3) We implement a generic *GPU Resource Management* module, to leverage various GPU resource configurations. Currently we provide 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.

2. Challenges of integrating neural networks into 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 [10] 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.

A. 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) [11], 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.

B. 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 [12]. 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) [10], 2) smoothing the collaboration between CPU and GPU since GPU computing is initiated and coordinated by and finally reduced to CPU.

C. 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.

D. CPU&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.

E. 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.

F. 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. 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 – we extend DATAVIEW's prior architecture with novel dedicated components. Based on this extended and new architectural design, we particularly emphasize 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.

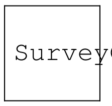

Figure 1: DATAVIEW's new architecture with inherent support for Deep-Learning-as-a-Workflow.

Fig. 1 shows the new architecture of DATAVIEW which provides inherent support for DL through the DLaaW approach, 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 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 $NNWorkflow\ Engine$ 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.

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 [13] 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 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);
}
```

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 inhouse 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:

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

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.

4. Experiments

In order to evaluate our proposed approach and implementation, we conducted experiments 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.

We 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 I). 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.

A. Hardware and Datasets

In our preliminary implementation of DLaaW, we adopted the following hardware: 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

Table I: 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 [14], 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

SurveyOfModernSWF/Images/modelingacc5.pdf

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

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.

We selected and adopted 5 datasets (as showed in Table I) by considering 1) their popularity, all the datasets are with high popularity among Machine Learning 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 machine learning repository, which is one of the most popular machine learning repositories with more than 3400 citations [15]; 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.

B. Experiment Results

The results of experiments are showed in Fig. 2. 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 I) 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 [16] 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 competitiveness in comparison to the conventional implementations of neural networks. We are excited about this result as it will function as a cornerstone for our ongoing research that tries to leverage GPU-enabled deep learning to benefit broad scientific workflows in DATAVIEW.

The bar charts in the bottom of Fig. 2 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.

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

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.

5. Conclusions and future work

In this paper, we propose the DLaaW approach which creates, executes and reuses any neural networks as native workflows in a general scientific workflow management system – DATAVIEW. Our work makes DATAVIEW the first scientific workflow management system that supports GPU-enabled deep learning on various GPU resources at the infrastructure level. Through carefully designed comparative experiments with the Keras-based counterpart implementations, we 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.

Acknowledgement

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

References

[1] 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*. Ieee, 2008, pp. 1–12.

[2] 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*, vol. 48, pp. 1–18, 2015.

[3] 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*, vol. 38, no. suppl_2, pp. W677–W682, 2010.

[4] 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*. Springer, 2004, pp. 11–20.

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

[6] 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*, vol. 32, pp. 8026–8037, 2019.

[7] 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*, vol. 16, no. 9, pp. 1318–1328, 2019.

[8] 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*, vol. 75, pp. 228–238, 2017.

[9] 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*, vol. 19, no. 18, pp. 59–69, 2018.

[10] 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)*. IEEE, 2020, pp. 132–141.

[11] J. Liu, E. Pacitti, P. Valduriez, and M. Mattoso, "A survey of data-intensive scientific workflow management," *Journal of Grid Computing*, vol. 13, no. 4, pp. 457–493, 2015.

[12] 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*, vol. 26, no. 1. Wiley Online Library, 2007, pp. 80–113.

[13] A. Kashlev *et al.*, "Big data workflows: A reference architecture and the DATAVIEW system," *STBD*, vol. 4, no. 1, pp. 1–19, 2017.

[14] S. K. Kumar, "On weight initialization in deep neural networks," *arXiv preprint arXiv:1704.08863*, 2017.

[15] D. Dua, C. Graff *et al.*, "Uci machine learning repository," 2017.

[16] C. DeBrusk, "The risk of machine-learning bias (and how to prevent it)," *MIT Sloan Management Review*, 2018.