Hindawi
Scientific Programming
Volume 2020, Article ID 8867380, 15 pages
https://doi.org/10.1155/2020/8867380

Research Article

A Machine Learning Gateway for Scientific Workflow Design

Brian Broll,[1] **Umesh Timalsina**,[1] **Péter Völgyesi**,[1] **Tamás Budavári**,[2] **and Ákos Lédeczi**[1]

[1]*Institute for Software Integrated Systems and Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN, USA*
[2]*Department of Applied Mathematics and Statistics, Department of Physics and Astronomy, and Department of Computer Science, Johns Hopkins University, Baltimore, MD, USA*

Correspondence should be addressed to Brian Broll; brian.broll@vanderbilt.edu

Received 2 June 2020; Accepted 9 September 2020; Published 29 September 2020

Academic Editor: Manuel E. Acacio Sanchez

The paper introduces DeepForge, a gateway to deep learning for scientific computing. DeepForge provides an easy to use, yet powerful visual/textual interface to facilitate the rapid development of deep learning models by novices as well as experts. Utilizing a cloud-based infrastructure, built-in version control, and multiuser collaboration support, DeepForge promotes reproducibility and ease of access and enables remote execution of machine learning pipelines. The tool currently supports TensorFlow/Keras, but its extensible architecture enables easy integration of additional platforms.

1. Introduction

We are witnessing groundbreaking advances at an unprecedented pace in Artificial Neural Network-based (ANN) solutions to application areas ranging from image and speech recognition to natural language processing to hard-to-master games such as Go [1–3]. This Machine Learning (ML) renaissance is driven by a number of factors, including (1) an increase of several orders of magnitude in the computational performance of general purpose Graphics Processing Units (GPUs), (2) the availability of extremely large pre-labeled datasets (image, audio, or text corpus), and (3) the discovery of new activation functions and regularization techniques to overcome the vanishing gradient problem of backpropagation. With the rapid development of freely available open-source neural network training frameworks that support novel network topologies and training techniques (e.g., TensorFlow [4] and PyTorch [5]), researchers can explore new application areas in the natural sciences, social sciences, and even the humanities.

While applications of advanced machine learning techniques are emerging in several scientific domains, including chemistry, materials science, biology, and astronomy, the barrier of entry for a domain scientist is still very high. Conceiving an ML solution to a new problem—that is, choosing a neural network topology, identifying a suitable training technique, and specifying the hyperparameters for the training process—is still more of an art than a science. Although best practices and rules of thumb do exist, one must rely on past experience, intuition, and trial-and-error experimentation to achieve success. The significant amount of contextual mathematical background, the complexity of setting up and using the various ML software packages, the lack of tool-based support for systematic exploration of a design space of parameters and networks, and the difficulty of moving, labeling, and preprocessing data all add to the list of obstacles that hinder wide adoption of ML tools in scientific research.

To address these challenges, we present DeepForge, a Software as a Service (SaaS) platform with a web browser-based visual design interface, which elevates the abstraction of creating ANN workflows and managing training artifacts, thus enabling domain scientists to more easily leverage recent advances in machine learning. DeepForge is built on top of two computing technologies: TensorFlow via Keras [4, 6] and Model Integrated Computing via WebGME [7, 8].

TensorFlow is a high-performance computing framework for machine learning, including the development of deep learning models [4]. It supports a large number of deployment platforms including CPU, GPU, and TPU and can run on a wide variety of hardware configurations from clusters to mobile devices. This broad support makes TensorFlow an ideal backend for the training of deep learning models and promotes the reuse of existing computational resources across scientific domains and organizations.

Model Integrated Computing (MIC) uses models, or domain-specific abstractions, for the rapid design and development of complex systems [7]. The Web-based Generic Modeling Environment (WebGME) is an open-source MIC tool suite developed for creating domain-specific graphical modeling environments [8]. WebGME leverages a number of modern features such as a browser-based user interface, a cloud-based infrastructure, integrated version control, and real-time collaborative editing. MIC in general and WebGME in particular have been successfully used across a wide variety of domains.

DeepForge [9] is built on top of WebGME, extending its standard user interface with custom visualizations and introducing a number of new features. The most important design goals of DeepForge are to provide an extensible, integrative architecture and to build a community around the tool in which users can contribute custom DeepForge extensions, as well as seamlessly share their work and collaborate with one another.

The rest of the paper is organized as follows: Section 2 provides an overview of related works, followed by a brief discussion on MIC in Section 3, which informs the architecture of DeepForge as described in Section 4. Infrastructure integration using DeepForge storage and compute adapters is discussed in Section 5, followed by a description of its visual abstractions in Section 6. Domain integration via libraries/extensions is discussed in Section 8, and Section 9 presents an illustrative example of designing and training a deep learning model in DeepForge.

2. Related Work

While there are a handful of visual environments available to data science developers (e.g., RapidMiner [10] and Knime [11]), as well as tools tailored specifically to deep learning (e.g., Nvidia's DIGITS [12], Google's TensorBoard [4], and Lobe [13]), DeepForge primarily addresses the challenges of developing deep learning models while maintaining a high degree of user flexibility, such as the ability to implement custom operations in Python and to develop neural network architectures using any of the layers available in Keras. This flexibility allows DeepForge to be a valuable tool for solving increasingly complex problems in both academic research and industrial applications, promoting the integration of novel results into production environments.

Both DIGITS and TensorBoard provide support for the execution of machine learning tasks and include visualization utilities to aid in their creation. They also provide web-based user interfaces for monitoring these executions and performing model introspection, such as visualizing the weights of trained layers. Despite these useful graphical functionalities, neither tools provide extensive support for the actual process of creating and defining models, requiring instead that machine learning tasks be written in a textual language (or optionally in Caffe Prototext [14] for DIGITS).

Lobe is perhaps the most similar to DeepForge as it aims to make deep learning more accessible and understandable to non-experts by allowing them to design, train, and evaluate neural networks from the browser. Like DeepForge, Lobe provides a visual interface to design neural network architectures with intermediate feedback during training. In Lobe, users are able to utilize their models interactively from a browser using data from their webcam and can view the results in real time. Additionally, Lobe supports exporting deployable models in the CoreML or TensorFlow formats, while providing a REST API for interacting directly with models from within Lobe itself.

Although DeepForge and Lobe share a number of similarities, there are still significant differences in both their target audiences and levels of customization. DeepForge supports collaborative editing and version control while allowing a higher degree of customization. Users are able to define custom operations in Python, as opposed to only allowing modification of a fixed number of exposed settings. This allows users to customize the intermediate feedback from operations to include custom plots and images. Additionally, execution is more loosely coupled in DeepForge, as training a model will result in the creation of a new execution instance (of which multiples can be created and executed simultaneously) rather than executing the single definition of a given model.

Unlike existing deep learning platforms, DeepForge facilitates reproducibility and collaboration throughout the entire development process. Reproducibility is promoted by ensuring that both the code and data for a workflow can be recovered (described in more detail in Section 7), while real-time collaborative editing capabilities, coupled with advanced versioning features, support enhanced collaboration among users.

3. Model Integrated Computing

Model Driven Engineering (MDE) and a related technique called Model Integrated Computing (MIC) are being applied to an increasing number of domains. MIC advocates the use of visual Domain-Specific Modeling Languages (DSMLs), relying on a tool infrastructure configured automatically by metamodels [15]. In turn, the user builds system models using the DSML, the models are analyzed by various built-in analysis tools, and finally, various artifacts such as configuration files, database schemas, and/or executable code are generated from the models automatically.

The metamodeling language consists of a set of elementary modeling concepts which constitute the building blocks of all derived modeling systems and corresponding tools. A meta-metamodel defines these fundamental concepts, which include composition, inheritance, associations, attributes, and others. The choice of which concepts to include in the modeling language, how to combine them, and how to operate on them are the most important design decisions that affect all aspects of the infrastructure and the domains that will use it.

The Generic Modeling Environment (GME) [15], an open-source desktop application created to support MIC 20 years ago, has been applied successfully to a broad range of domains from medicine to manufacturing [16, 17]. A recent trend in computing is to move away from desktop tools toward cloud- and web-based architectures for better scalability, maintainability, and seamless platform support. The latest-generation GME toolsuite follows this trend and is called WebGME [8]. It is a web-based cyberinfrastructure that supports collaborative modeling, analysis, and synthesis of complex, large-scale information systems. The primary design goal of WebGME is to better support the modeling of highly complex systems, with features targeted specifically for this purpose including collaborative editing, similar to Google Docs, and a Git-like database backend to support model version control. WebGME also introduced a number of novel abstractions to support the scalability of complex system models.

As an example, consider Figure 1 that depicts the metamodel of a simple hierarchical signal flow (SF) graph on the left and an example SF domain model on the right. The metamodel shows that *Compute* nodes, *SignalPorts*, and *Flows* are all derived from a single built-in base class called a First Class Object (*FCO*). *Compute* nodes can contain other *Compute* nodes, enabling hierarchical decomposition. *Compute* nodes can also contain *SignalPorts* representing the data interfaces of their parents. The *src* and *dst* pointer specifications from *Flow* to *SignalPort* specify a relationship between two *SignalPorts*, which will then be shown as a connection in the DSML, i.e., a signal flow graph. DeepForge uses similar concepts under the hood, but the default WebGME visual editor has been replaced with one that better fits the domain of scientific workflows.

4. Architecture

DeepForge was designed from the ground-up as a modern web application built on top of WebGME. Its high-level system architecture is shown in Figure 2. This includes a NodeJS server, MongoDB database, and browser-based client implemented as a *single page application* providing core functionality.

Export capabilities and a REST API facilitate extensibility and integration with external applications. Furthermore, the architecture includes adapters for both external compute and storage resources, specifically designed for integration with existing cyberinfrastructures. This is discussed in detail in Section 5.

The DeepForge server hosts the platform, performs neural network model analyses to allow the environment to provide visual feedback about model dimensionality and validation, and manages interactions with all other aspects of the system. The project code and visual models are stored in MongoDB whereas any associated artifacts, such as training data or trained models, are stored using a "storage adapter." Associated artifacts can either be created from an operation or uploaded manually.

When performing a computation such as training a model in DeepForge, the server first generates the corresponding code for the given pipeline. This code contains all of the relevant logic necessary to perform the given operation, including any metadata about associated artifacts that will be either fetched using a storage adapter or retrieved from a local cache. After generating the relevant code, it is then delegated to a connected computational resource for execution (discussed in Section 5). During execution, standard output and any other generated feedback, such as plots, are captured, and updates, are sent back to DeepForge where the user's project will be updated accordingly.

Along with facilitating the development process when designing and training neural networks, it is also important to support users in exporting their work for use outside of DeepForge. In addition to exporting complete final models, it is also helpful to allow users to export modular components of their models, such as specific neural network architectures or a single operation. This gives users the freedom to selectively experiment with different aspects of the platform and promotes interoperability with other tools and environments.

DeepForge has also been designed to support integration with other external tools via a REST API. Although incomplete, this will enable domain experts to script common tasks such as creating new architectures, executing machine learning pipelines, and retrieving artifacts such as trained models and datasets. Providing a flexible interface for scripting also enables collaboration between users with vastly different toolset preferences, as the effects of scripts will be reflected in the visual interface and vice versa. This allows collaborators using external scripts and users interacting with the web-based interface to both benefit from the integrated version control and collaborative capabilities of the underlying platform.

5. Infrastructure Integration

DeepForge is a "Software as a Service" (SaaS) solution whereby all users share the same WebGME backend for model storage. It does, however, externalize its computational infrastructure, where time- and resource-consuming training tasks are executed, as well as the storage of all training data. As such, users must provide their own computational resources, which will be utilized by DeepForge to deploy training jobs, and specify the storage locations of any external training data. To make use of these resources, DeepForge must know what kind of computational specifications are available to the user and their corresponding access mechanisms and any required user access credentials, as well as information about the resources available to store the outputs of a given task, including trained model weights. Access to both computational and storage resources is achieved through use of compute adapters and storage adapters.

5.1. Compute Adapters. Compute adapters enable DeepForge to be used as a design environment for applying machine learning to a given problem while ensuring that users are able utilize an existing compute platform. They expose a common interface which enables execution of

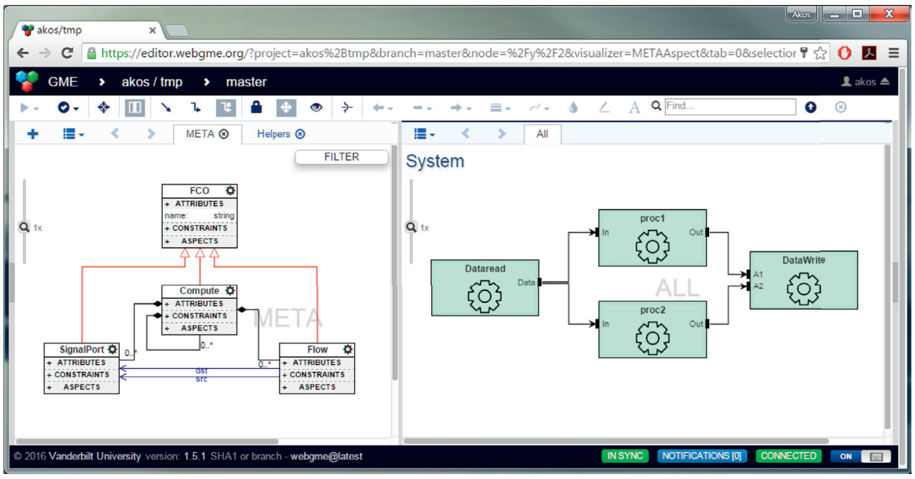

akos/tmp
https://editor.webgme.org/?project=akos%2Btmp&branch=master&node=%2Fy%2F2&visualizer=METAAspect&tab=0&selection
GME
akos / tmp
master
META
Helpers
FILTER
FCO
Compute
SignalPort
Flow
System
Dataread
proc1
proc2
DataWrite
ALL
© 2016 Vanderbilt University version: 1.5.1 SHA1 or branch - webgme@latest
IN SYNC
NOTIFICATIONS [0]
CONNECTED

Figure 1: Hierarchical Signal Flow Graph metamodel (left) and domain model (right).

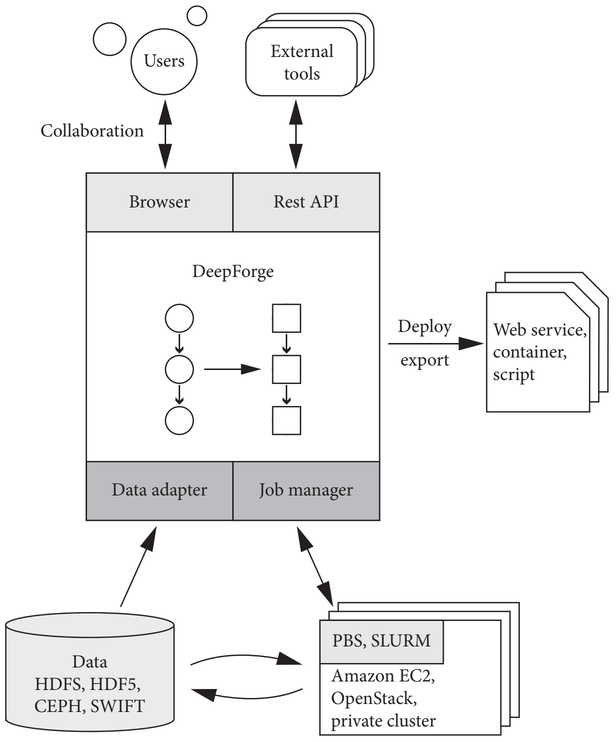


Figure 2: DeepForge architecture.

machine learning pipelines on a variety of external compute resources. An adapter must provide a set of core functionality, including job management and feedback, but may also include definitions for custom configuration information and a GUI for monitoring the compute resources. Support for custom configurations is vital to supporting more sophisticated compute infrastructures. For example, some infrastructures may allow the user to target specific machines or clusters upon which to launch a given job.

Adapters are designed to be simple to ensure support for many different computational resources. They are currently only responsible for execution of the individual jobs in a pipeline; creation of job files and orchestration of the individual jobs are managed by DeepForge. Optional delegation of pipeline orchestration to the compute adapter (if supported by the underlying compute infrastructure) is planned for the future. Currently, compute adapters are available for SciServer [18], on-server execution, and personal local machines.

5.2. Storage Adapters. It is quite common for scientific disciplines to have a variety of existing datasets and storage infrastructures. As such, DeepForge provides capabilities for integrating these resources via storage adapters. As expected, storage adapters provide a common interface to interact with artifacts associated with a DeepForge project, such as datasets or pretrained network weights. As with compute adapters, storage adapters may also specify the configuration parameters necessary to interact with a given data set, such as authentication information for the target storage infrastructure.

To facilitate the integration of data from a variety of sources, storage adapters are specified for each artifact in a given project. As a result, a single project may incorporate data from many different storage locations. Existing storage adapters enable integration with SciServer Files or with any object store having an S3-compatible API.

6. Visual Abstractions

The primary goal of DeepForge is to make machine learning, specifically deep learning, more accessible to those in the natural sciences with limited experience with neural networks. To this end, DeepForge utilizes multiple modalities during project creation: a visual interface for designing workflows using pipelines, and a textual interface for implementing individual operations in the given workflow. (Note that a comprehensive library of common operations is provided by DeepForge, so only application-specific custom computations need to be implemented in a traditional programming language, such as Python). Using these multiple modalities, users are able to define their scientific workflows in an extensible visual modeling language which can then be deployed on their own cyberinfrastructure. Furthermore, extension of the modeling language provides for rich integration into many scientific domains through creation and incorporation of additional modeling tools tailored specifically for that domain.

6.1. Core Modeling Language. DeepForge provides a core modeling language for designing scientific workflows. This visual modeling language consists of four main concepts for testing and training machine learning models: *Pipelines*, *Operations*, *Executions*, and *Jobs*. The formal specification of these concepts is given in Figure 3(a). Concrete machine learning concepts are provided via custom domain integrations called *libraries*, discussed in detail in Section 8.

Operations are atomic functions that accept one or more named inputs and return one or more named outputs. *Attributes* can be defined for an operation at design time, providing adjustable parameters. At runtime, an operation's attributes are provided as constants to the operation. Operations can also define *references* which, like attributes, are specified at design time. A reference is a pointer to another artifact, such as a neural network architecture.

A *Pipeline* represents a machine learning task composed of operations, such as model training, testing, or data preprocessing. Operations are built by directing the output of a single operation, represented as a port, into the input of one or more other operations. That is, these operations can form acyclic data flow graphs. Pipelines can also contain *Input* and *Output* operations that represent the input and output data for that pipeline. That is, the *Input* operation is an initial node in a pipeline, and the *Output* operation is a terminal node.

Figure 3(b) shows an example of a pipeline with four operations. The first operation downloads and prepares training and testing sets from the MNIST dataset [19]. The training data is then provided to the "Train" operation which is parameterized by a batch size, number of epochs, and a *reference* to the neural network architecture. Clicking the blue icon at the top right of the operation enables the user to view and edit the implementation of the operation and can be used to expose more parameters or extend the operation to perform a hyperparameter search (Pipelines cannot contain cycles so tasks such as hyperparameter optimization must be implemented within a single operation (like recurrent layers in neural network frameworks)). The "Train" operation produces a trained model and passes it to the "Output" and "ScoreModel" operations. The "Output" operation saves the model back to DeepForge and the "ScoreModel" operation evaluates the trained model on the testing data from the first operation.

Running a pipeline results in the creation of an *Execution*. A pipeline's execution is an acyclic data flow graph that is isomorphic with the graph of the originating pipeline. This graph is created by converting each of the pipeline's operations into a *Job* that corresponds to the original operation combined with its execution status and metadata (such as plots generated during the execution).

7. Interactive Editing and Data Provenance

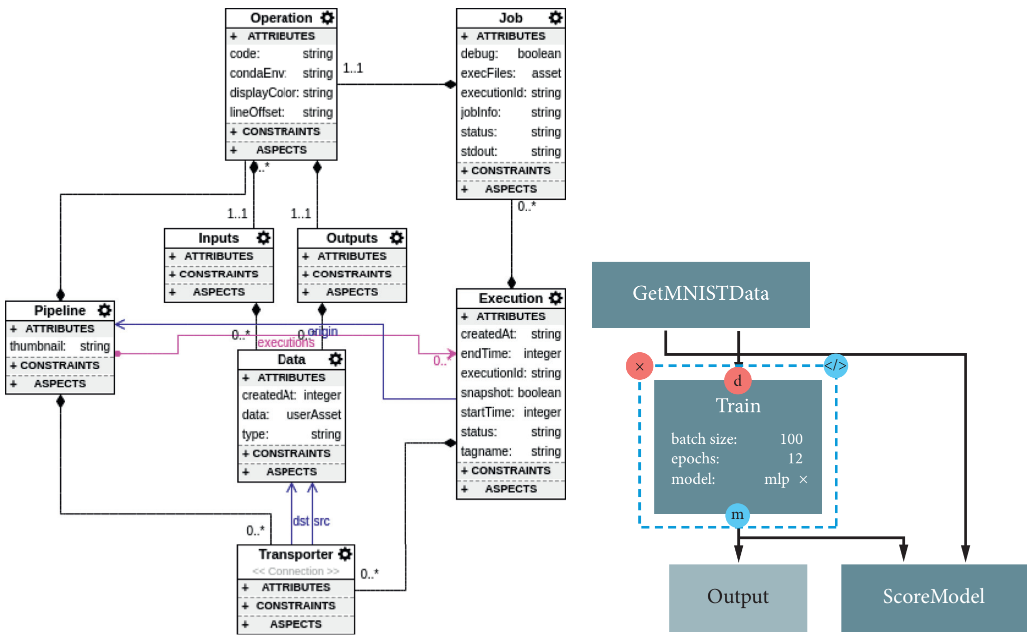
DeepForge was designed to ensure reproducibility while also promoting accessibility to deep learning. To that end, data provenance is captured in a way that supports the type of iterative exploration found in ubiquitous environments like Jupyter Notebook. This ensures that reproducibility, which guarantees that the code and data for workflows can be recovered, is not achieved at the expense of ease of use of the environment. This section presents the design of DeepForge's interactive editing capabilities, followed by a description of its support for data provenance in Section 7.2.

7.1. Interactive Editing. Constructing machine learning pipelines can be an effective abstraction for promoting reusable components and executing long-running tasks. However, this approach can be inconvenient for simple, exploratory tasks where quick, immediate feedback is preferred. Interactivity and immediate feedback have already proven to be powerful mechanisms for promoting accessibility, as evinced by their success in both novice programming environments and common data science environments. The ability to reconstruct an entire history in the form of a pipeline from an artifact makes a compelling case for the interactive, implicit construction of pipelines. Interactive editing in DeepForge is supported through the introduction of "interactive compute sessions," along with visual editors.

An interactive compute session is a connection to a compute resource which can be used for immediate execution of arbitrary commands and is a simple extension of the existing compute infrastructure. Using compute adapters, a custom job is created which establishes a websocket connection to the browser (proxied through the DeepForge server). A bidirectional communication channel supports a relatively simple API which includes importing artifacts or files into the session and spawning processes. Similarly to Jupyter Notebook, the underlying job only executes a single command at a time.

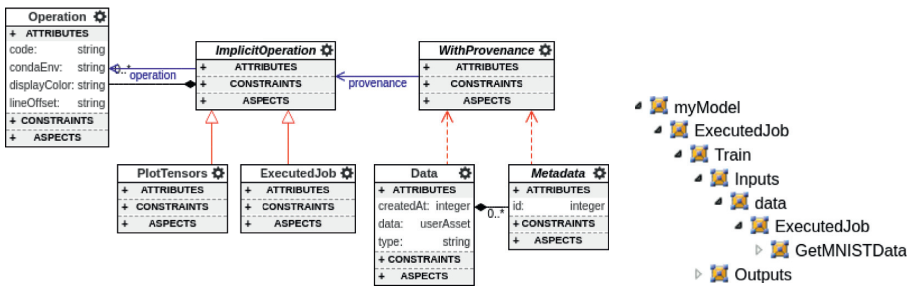
Using the interactive compute capabilities, visual editors can inspect and create artifacts. However, arbitrary interactive editing and code execution can make reproducing workflows difficult. Furthermore, we would like to enable the creation of the provenance of an artifact as an executable pipeline (discussed in Section 7.2). To this end, we introduce the concept of an *ImplicitOperation* as shown in Figure 4.

ImplicitOperations are created and edited by visual editors. An *ImplicitOperation* is a type of task that can be represented as an *Operation* (from Figure 3) which could then be used in a pipeline. For example, the *PlotTensors* node (shown in Figure 4) represents a task which, as the name suggests, creates plots of tensors which can certainly be implemented as an operation. Introducing the concept of an *ImplicitOperation* enables the *PlotTensors* node to use the most appropriate modeling concepts for its representation. Requiring the visual editors to interact directly with *Operation* nodes would restrict them to a format that is convenient to execute but cumbersome to create.



(a) (b)

Figure 3: Modeling language used for specifying scientific workflows. (a) Formal specification of the language (the metamodel). (b) An example workflow training an image classifier on the MNIST dataset.



(a) (b)

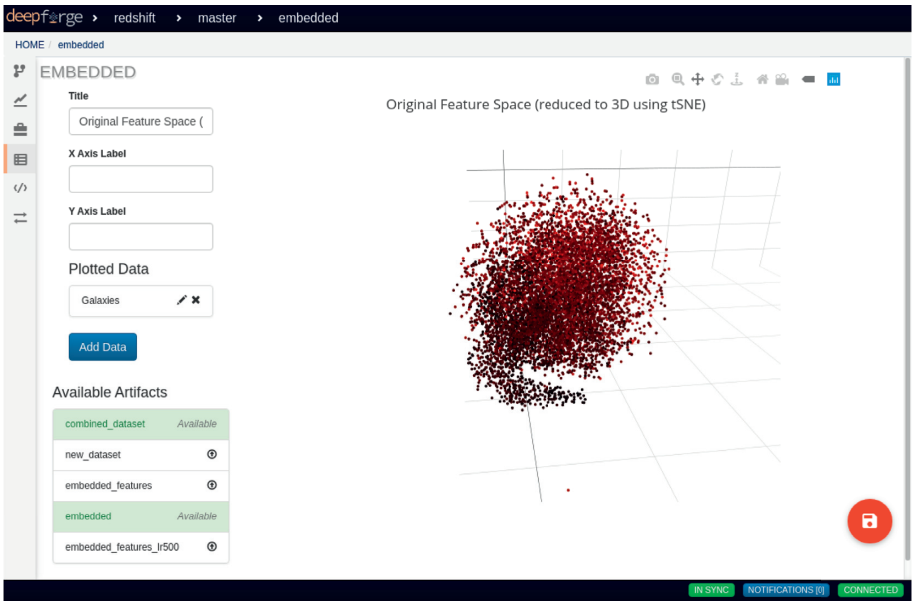
Figure 4: Aspects of the modeling language capturing provenance. (a) Formal specification of data provenance in the metamodel. (b) Recorded provenance of the image classifier trained in Figure 3(b).

The *ImplicitOperation* abstraction is very powerful in this context as it enables the creation of custom editors which define the appropriate higher-level abstractions to solve a given task. These visual editors can then present the most suitable modality and visualization techniques to interact with the given concepts. Upon saving the data, an instance of the corresponding *ImplicitOperation* type is created and stored as the data's provenance. The given node can then be mapped onto an executable operation which ensures that the workflow can be reproduced. This enables the user to first interactively prototype a single solution and then automatically generate a pipeline to be generalized for broader application.

Figure 5 shows an example of interactive editing in DeepForge with a simple editor for exploring a dataset. In this example, spectral input data is being explored for a regression problem in which the original inputs have been reduced to 3-dimensional representations using t-Distributed Stochastic Neighbor Embedding (t-SNE) [20] and then color-coded by their ground truth labels (see Section 9). In this case, the data appears to have 4 outliers, and there is not an obvious pattern in the original input space (aside from a slight dark "tail" at the bottom of the cluster).

This example enables users to import arbitrary artifacts into the current session and then explore various aspects of the data. It was designed to help users gain insight into their data and detect potential issues such as class imbalance quickly. Generated plots include provenance information; this ensures that the same plotting operation can be replicated and generalized to other datasets. This enables users to implicitly construct a workflow through interactively editing a concrete example. Once the user is satisfied with the results, the implicit workflow can be reified and generalized.

7.2. Data Provenance. Automatic version control is inherited by DeepForge from WebGME, ensuring that any historical version of a project can easily be reloaded or


deepforge › redshift › master › embedded
HOME / embedded
EMBEDDED
Original Feature Space (reduced to 3D using tSNE)
Add Data
IN SYNC
NOTIFICATIONS [0]
CONNECTED

Title: Original Feature Space (
X Axis Label:
Y Axis Label:
Plotted Data: Galaxies

Available Artifacts

combined_dataset	Available
new_dataset	
embedded_features	
embedded	Available
embedded_features_lr500	

Figure 5: Example of interactively exploring the dataset used in Section 9. Generated data or plots are saved with their data provenance to ensure the workflow can be reproduced.

forked. DeepForge builds upon these existing capabilities to enable detailed provenance information for individual artifacts within a project. Although integrated version control ensures that the data is available somewhere in a historical version of the project, the extension of provenance ensures that the history of an individual artifact can be recovered without searching the project history. As both of these features automatically track data as well as code, the reproducibility of experiments is established by ensuring that the workflow producing any given artifact can be recovered. Furthermore, when combined with interactive editing, this enables a powerful approach to experimentation via interactively prototyping and then generalizing the workflow used for the initial prototype. That is, an initial prototype can be created interactively and then the provenance of the prototype can be reified as a pipeline and generalized.

Provenance is added to the modeling language through a simple extension of the core concepts presented in Figure 3. This extension defines the central abstractions used for recording provenance, and these abstractions then form the building blocks from which the entire history of a given artifact can be reconstructed. Copies of the data itself are not stored; in the case where the initial data has been deleted, the reconstructed history will simply specify this as an unset input to the pipeline. When data is generated in DeepForge, such as during pipeline execution and interactive editing, the provenance for the newly created data is automatically stored in DeepForge's representation of the generated data.

At a high level, data provenance is captured in the modeling language through the introduction of a general concept used to represent pipeline operations or edits made via interactive editing. This concept is referred to as an *ImplicitOperation*, shown in Figure 4. It is used to record how data in DeepForge was created and is self-contained; any required data inputs or references are contained entirely within the given node. In turn, the data inputs also will contain a record of their own provenance ensuring that the entire provenance is captured for a given artifact.

Implicit operations can also be converted into executable operations. As a result, data provenance is not only a record of the history of a given artifact but can be transformed into an executable pipeline, thus enabling prototypical generalization while facilitating reproducibility and tinkerability. An example artifact with recorded data provenance is presented in Figure 4(b). This figure shows the containment hierarchy of *myModel*, a trained neural network model produced from the pipeline presented in Figure 3(b). The trained model was created by executing the *Train* job. This is shown in the figure by the contained implicit operation *ExecutedJob* which in turn contains a snapshot of the originating operation. The *Train* operation contains a single input, *data*, which was generated from the *GetMNISTData* operation as evident by its contained provenance nodes.

8. Domain Integration via DeepForge Libraries

One important design decision that enables better integration of other domains into DeepForge lies in the introduction of DeepForge libraries. A library is an extension which enables users to introduce custom tooling and domain concepts, or resources, into their workflows. For example, new neural network frameworks can be integrated into DeepForge using libraries.

The introduction of a new resource into DeepForge requires three main components: a formal specification of the resource, mechanisms for user creation and manipulation of the resource, and mechanisms for computational usage. To this end, a DeepForge library consists of a metamodel, an editor, a code generator, and optional initialization logic, if required. The metamodel is a specification of the domain concepts and their relationships, and it defines the domain-specific modeling language provided by the library. The editor defines a method of interaction with the resource which can be used to provide validation and feedback during design time. Additionally, the editor can be used to enforce other domain constraints such as ensuring that the constructed domain model is a directed acyclic graph. Editors may include any combination of visual and textual components, allowing the domain extension to leverage the most appropriate modality for a given task.

Along with a metamodel and editor, libraries must define a code generator. For example, in a library adding support for defining neural network architectures, the code generator maps the architecture to the source code that implements a necessary computation. This enables the resources provided by the library to be converted into Python code and then used by any referencing operation.

Finally, a library can also define custom initialization logic for the provided resources to be executed before any of the subsequent operation tasks. One common use-case is the need for defining custom serialization logic. This may be the case when the provided resource requires multiple steps before it can be serialized or deserialized appropriately.

This decoupling of the resources and pipeline concepts is very powerful both for supporting rich integration with scientific domains and also for enabling integration with any number of deep learning frameworks. This makes DeepForge agnostic to the deep learning framework of choice and allows it to easily be extended to support many different frameworks.

8.1. DeepForge-Keras Library. Currently, deep learning in DeepForge is supported through the DeepForge-Keras library. This library provides a rich modeling language for defining neural network architectures using Keras and is designed to lower the learning curve for new users while still providing valuable support for more experienced users.

The metamodel specifies the modeling concepts for defining neural network architectures using DeepForge-Keras. This includes both abstract core concepts as well as specific layer definitions. The core concepts consist of *Architecture*, *Layer* (including concepts representing layer inputs and outputs), and concepts for different types of functions such as activation and regularization. The formal specification of these core concepts is presented in Figure 6.

As shown in the metamodel, neural network architectures contain a set of input and output layers. Each layer contains a set of inputs consisting of *LayerInput* nodes which correspond to arguments accepted by the layer during inference or training time. Similarly, *LayerOutput* nodes are used to represent the outputs produced by the layer. Layers are connected by adding a reference from an output of the preceding layer to an input of the subsequent layer in the neural network. As DeepForge-Keras provides its own visualizations, this relationship can then be depicted as a connection between layers.

The majority of the metamodel consists of concrete layer definitions for the many supported layer types in Keras. As manually updating the metamodel to contain these definitions would be tedious, the metamodel is generated programmatically by parsing the Keras source code. Figure 7 shows the generated formal specification for the *Dense* layer alongside an instance.

As shown in the figure, configurable parameters that reference a function, such as initializer or activation function, are defined as pointers in the metamodel. When editing an instance from the visual editor, these references can be modified in a similar fashion as primitive values. Furthermore, layers also contain additional metadata in the form of (1) documentation and (2) the argument order to the constructor, stored in the *docstring* and *ctor_arg_order* attributes, respectively.

Code generation capabilities are included in which Python code is synthesized from the user-defined specification of the neural network architecture. As the library supports customization of activation functions, two-step serialization is required for the resultant models.

During deserialization, any custom activation functions are first deserialized followed by deserialization of the Keras architecture itself. The library initialization code defined by DeepForge-Keras then registers these serialization methods for Keras models within DeepForge.

Additionally, DeepForge-Keras contains a visual editor for designing neural network architectures which provides rich feedback to support both novice and experienced users. Documentation about individual neural network layers and layer arguments is generated automatically from the Python source code, which is also used to generate the Keras official documentation.

Interactive feedback is also given to the user with respect to dimensionality information and errors encountered during construction of the neural network architecture. An example of error feedback is shown in Figure 8.

9. Illustrative Example

Machine learning and, in particular, deep learning, have recently succeeded in achieving state-of-the-art performance on various tasks in astronomy. There is a great deal of interest amongst the astronomical community in various powerful learning models, especially given the exponential rise in quantity and quality of available data. Redshift estimation, a task which is traditionally very time- and resource-intensive, is one problem that has seen noteworthy improvement through the use of deep learning methods [21].

In this section, we present an example demonstrating how DeepForge can be leveraged to create, train, and visualize redshift predictions for a *Photometric Redshift Estimation Model* [21] using a deep-convolutional neural network and images from the Sloan Digital Sky Survey (SDSS). The example performs the following steps to curate the dataset and train the neural network:

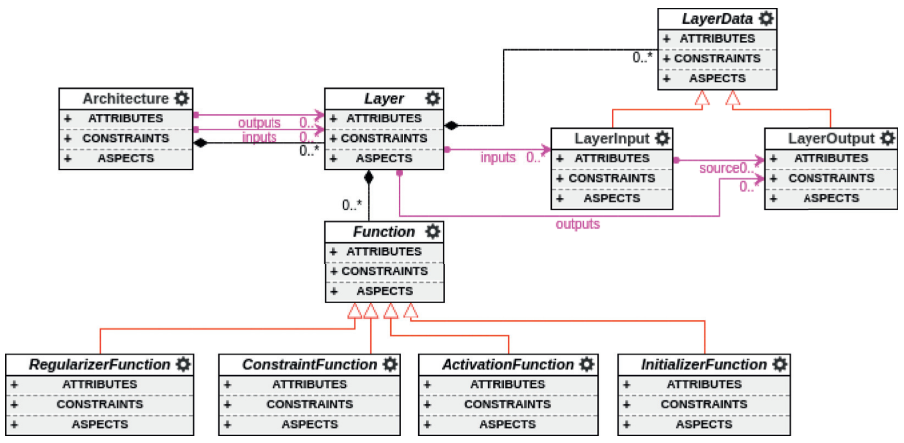
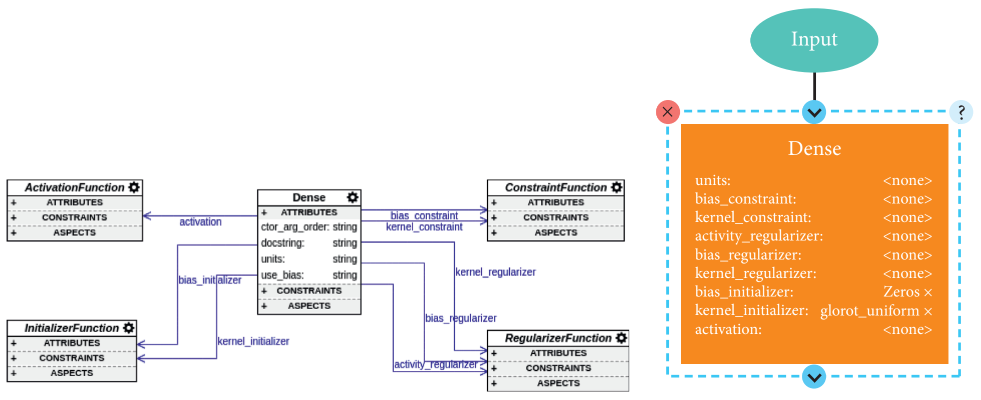


Figure 6: Core concepts in DeepForge-Keras modeling language.

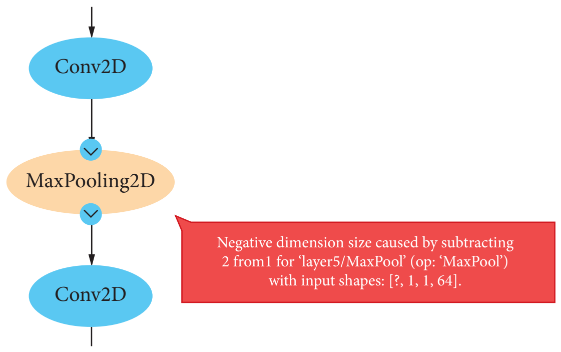


Dense

units:	<none>
bias_constraint:	<none>
kernel_constraint:	<none>
activity_regularizer:	<none>
bias_regularizer:	<none>
kernel_regularizer:	<none>
bias_initializer:	Zeros ×
kernel_initializer:	glorot_uniform ×
activation:	<none>

(a) (b)

Figure 7: An example of the formal specification for a neural network layer in Keras alongside an instance of the given layer. (a) Formal specification for a neural network dense layer and (b) an instance of the dense layer.



Negative dimension size caused by subtracting 2 from1 for 'layer5/MaxPool' (op: 'MaxPool') with input shapes: [?, 1, 1, 64].

Figure 8: Design-time error feedback in the DeepForge-Keras editor.

(1) Query the CasJobs server [22] for galaxies that have redshift values between 0 and 0.4.

(2) From the query results, obtain the download URL for FITS files of images in each of the five spectroscopic bands captured by SDSS for each galaxy and then download the images.

(3) For every galaxy with images in the five bands, resample each image around the galactic centers to (64×64) pixels and combine images of each band into a $(64 \times 64 \times 5)$ datacube.

(4) Provide these datacube images as inputs to a stacked inception network, and train this network to predict their redshift values using cross-entropy loss. At inference time, the mathematical expectation of the output probability distributions is used for the redshift value, as in [21].

In this example, we first define the architecture of the neural network using functionality provided by DeepForge-Keras. We then discuss the individual operations defined in Python, corresponding generally to the enumerated steps in the training procedure. Finally, we present the composition of these individual operations in a machine learning pipeline and discuss their evaluation.

9.1. Model Architecture. This example uses a deep-convolutional neural network similar to the architecture used in [21]. The network starts with a convolution layer that takes an input image of size 64×64 with 5 channels, followed by an average pooling layer using a 2×2 kernel. The output of the pooling layer is then passed to an inception module [23] as shown in Figure 9. The output from one inception block is

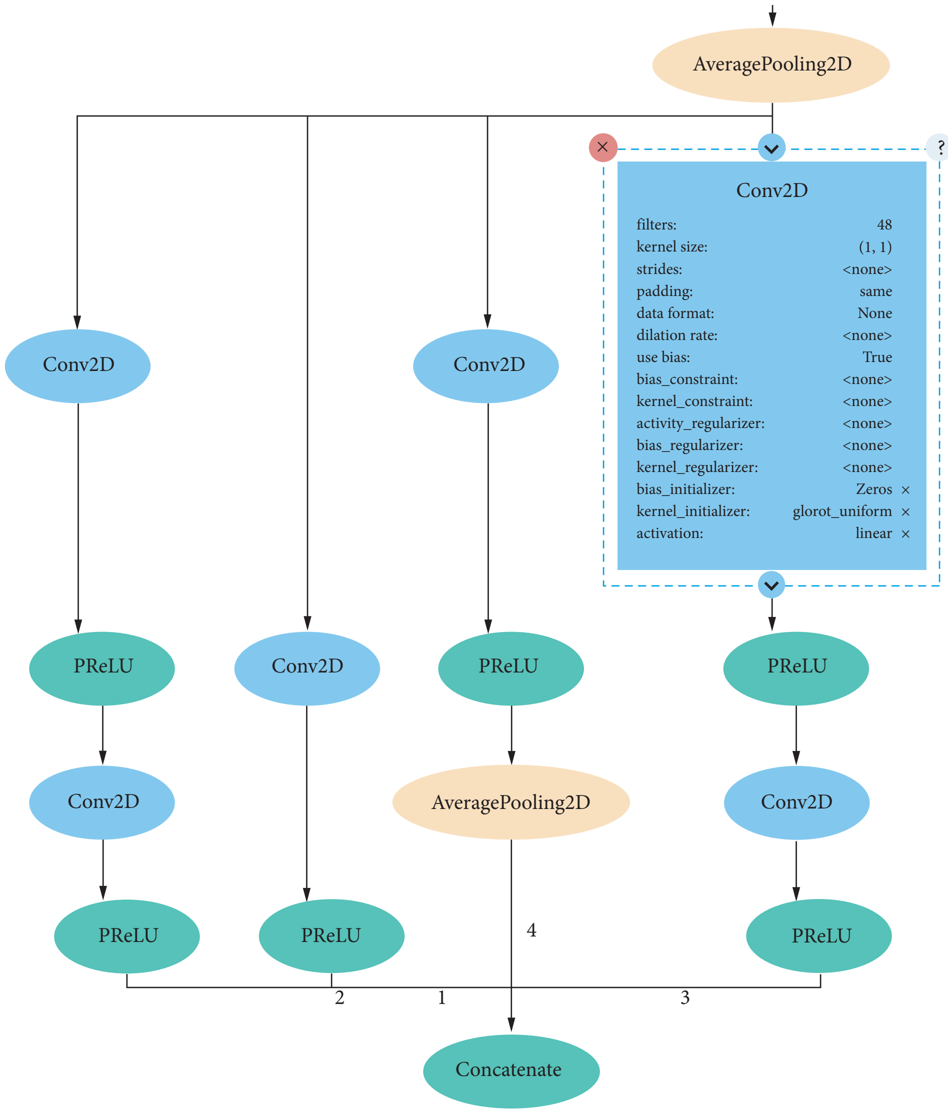
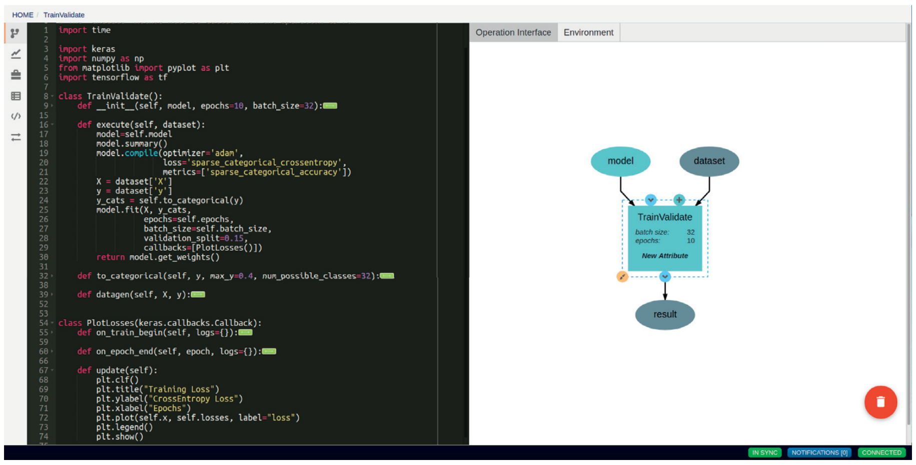


Figure 9: An inception module.



```
import time

import keras
import numpy as np
from matplotlib import pyplot as plt
import tensorflow as tf

class TrainValidate():
    def __init__(self, model, epochs=10, batch_size=32):

    def execute(self, dataset):
        model=self.model
        model.summary()
        model.compile(optimizer='adam',
                      loss='sparse_categorical_crossentropy',
                      metrics=['sparse_categorical_accuracy'])
        X = dataset['X']
        y = dataset['y']
        y_cats = self.to_categorical(y)
        model.fit(X, y_cats,
                  epochs=self.epochs,
                  batch_size=self.batch_size,
                  validation_split=0.15,
                  callbacks=[PlotLosses()])
        return model.get_weights()

    def to_categorical(self, y, max_y=0.4, num_possible_classes=32):

    def datagen(self, X, y):


class PlotLosses(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):

    def on_epoch_end(self, epoch, logs={}):

    def update(self):
        plt.clf()
        plt.title("Training Loss")
        plt.ylabel("CrossEntropy Loss")
        plt.xlabel("Epochs")
        plt.plot(self.x, self.losses, label="loss")
        plt.legend()
        plt.show()
```

Figure 10: Code/Interface editor for the operation TrainValidate. The left side shows the Python code to train a model, plot losses for each epoch (implemented via Keras Callback), and returns the model weights upon completion. The right side shows the operation inputs (the dataset and the model), outputs (model weights), and attributes (epochs, batch size).

successively passed to similar inception blocks (with varying filter size, kernel size, and strides for each block), and the final classification is performed using a fully connected layer with softmax activation.

Figure 9 shows the first inception block presented in [21]. The block consists of six 2D convolution layers. The input to the block comes from the `AveragePooling` layer shown at the top of the figure. Integer labels correspond to the ordering of the inputs to the subsequent layer and can be seen with the inputs to the `Concatenate` layer. The attributes of a specific layer such as the activation function, regularization, and constraints, among others, can be modified by clicking on the layer (as shown in the top right `Conv2D` layer in Figure 9). The full model consists of five different inception blocks stacked together.

9.2. Operations. The major operations used in this machine learning workflow are described below. The first three operations are used for fetching and preprocessing the dataset. The next operation is used for training the neural network, and the final three operations are used for visualization, dimensionality reduction, and manual verification. Detailed descriptions of the individual operations are given below:

(a) CasJobsQuery: CasJobs [22] is a MySQL-like interface to the metadata in SDSS images which provides a Python API to perform queries and download query results in the desired format. The CasJobsQuery operation is a simple operation that takes a query string and performs the query in CasJobs. Upon successful completion of the operation, the results of the query are passed as the output of the operation.

(b) DownloadFits and Preprocess: In order to download the images in the FITS format, we utilize the artifact generated by the CasJobsQuery operation, which provides the location for the FITS files for the galaxy. After downloading the FITS images, the preprocess operation is applied according to the metadata generated by the query for the galaxy by the CasJobsQuery operation. Together, these operations ready the dataset to train the neural network using the architecture presented in Section 9.1.

(c) TrainValidate: Given a neural network architecture and a dataset, this operation trains a neural network and plots the training loss. In this example, the dataset is the preprocessed image files from SciServer, and the neural network architecture is the aforementioned deep-convolutional neural network. After training is complete, the trained neural network weights can be saved as an artifact or passed along to a subsequent operation.

(d) VisualizePredictions: After the training is complete, this operation is used to visualize the predictions for a few of the inputs for manual verification. Specifically, this operation will select five inputs and plot the softmax outputs for the given inputs to be compared with the ground truth values.

(e) ReduceDimensions: This operation runs t-Distributed Stochastic Neighbor Embedding to perform a dimension reduction on the inputs and is used to get a 3D embedded representation of the dataset's learned features for visualization.

(f) VisualizeEmbeddings: This operation generates a 3D scatter plot of embeddings produced by the ReduceDimensions operation, color-coded based on the original dataset labels/error values.

DeepForge provides a Python interface to write custom operations that can leverage the architectures, artifacts, and other assets in a project. For example, the DeepForge-Keras neural network training process may take a number of approaches, including using different learning rates and optimizers, implementing an early stop, or reporting epoch metrics via Keras Callbacks, among others. Since the operations are defined in Python, it is straightforward to customize them to fit specific pipeline and project needs. Figure 10 shows the TrainValidate operation (described above) in a built-in text editor side-by-side with its graphical representation in DeepForge.

9.3. Pipeline Design and Evaluation. As shown in Figure 11(a), we connect the operations described in Section 9.2 to form an end-to-end training pipeline. These operations can be reused by providing different inputs each time we want to run a different execution for the pipeline. The operations *TrainValidate* and *VisualizePredictions* use the neural network architecture described above.

This example trains the model with 10,000 images for 10 epochs. Considering the number of images, the redshift values between 0 and 0.4 are divided into 32 distinct bins. Cross-entropy loss is used to train the network to predict the correct bin for each image. This results in the execution shown in Figure 11(b) where each job is color-coded based on its state of execution. In the figure, *CasJobsQuery*, *DownloadFits*, and *Preprocess* have completed, and *TrainValidate* is currently training the neural network. Although the execution can be monitored in DeepForge (as shown in Figure 11(b)), the actual computation is performed on a compute infrastructure using compute adapters as discussed in Section 5.

In this example, *TrainValidate* generates two distinct metadata during execution: console logs and a plot of the training loss. Figure 12 shows both forms of feedback during execution of the given job. The graph remains visible and updates as the job executes. In this case, it is updated after each training epoch.

Both forms of feedback shown in Figure 12 follow standard Python conventions and promote portability of the operation implementations to execution outside of DeepForge. The plot is implemented using matplotlib from within the *TrainValidate* operation. Execution of the operation within DeepForge results in the creation of the interactive plot shown in Figure 12 (through use of a custom matplotlib backend). If executed outside of DeepForge, such as from the command line or in a Jupyter notebook, the plot would be

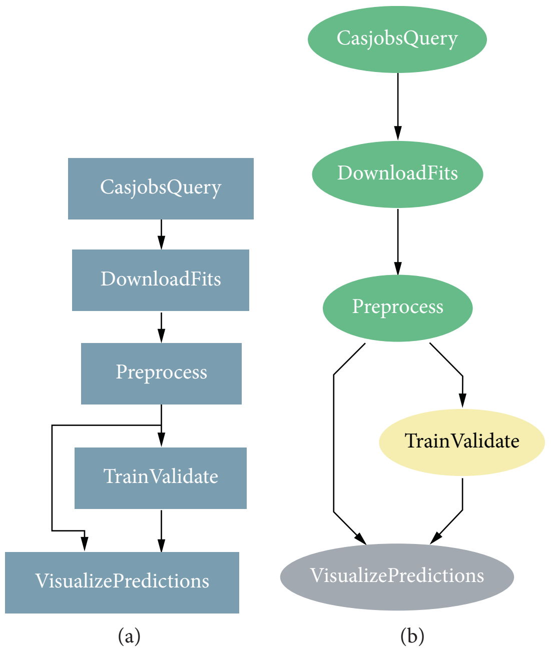
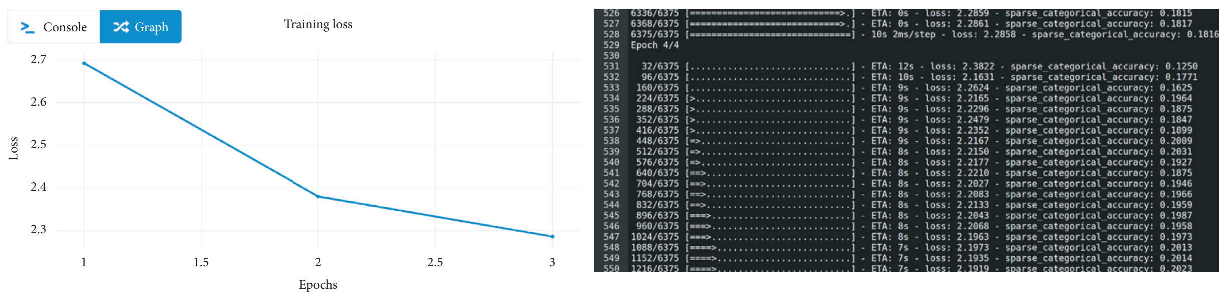


Figure 11: Training pipeline and its execution. The complete training pipeline is shown in (a). Execution of the pipeline is shown in (b). Jobs are color-coded by their execution status. Green jobs have completed, yellow jobs are currently running, and grey jobs have yet to run.



```
6336/6375 [============================>.] - ETA: 0s - loss: 2.2859 - sparse_categorical_accuracy: 0.1815
6368/6375 [============================>.] - ETA: 0s - loss: 2.2861 - sparse_categorical_accuracy: 0.1817
6375/6375 [==============================] - 10s 2ms/step - loss: 2.2858 - sparse_categorical_accuracy: 0.1816
Epoch 4/4

  32/6375 [..............................] - ETA: 12s - loss: 2.3822 - sparse_categorical_accuracy: 0.1250
  96/6375 [..............................] - ETA: 10s - loss: 2.1631 - sparse_categorical_accuracy: 0.1771
 160/6375 [..............................] - ETA: 9s - loss: 2.2624 - sparse_categorical_accuracy: 0.1625
 224/6375 [>.............................] - ETA: 9s - loss: 2.2165 - sparse_categorical_accuracy: 0.1964
 288/6375 [>.............................] - ETA: 9s - loss: 2.2296 - sparse_categorical_accuracy: 0.1875
 352/6375 [>.............................] - ETA: 9s - loss: 2.2479 - sparse_categorical_accuracy: 0.1847
 416/6375 [>.............................] - ETA: 9s - loss: 2.2352 - sparse_categorical_accuracy: 0.1899
 448/6375 [=>............................] - ETA: 9s - loss: 2.2167 - sparse_categorical_accuracy: 0.2009
 512/6375 [=>............................] - ETA: 8s - loss: 2.2150 - sparse_categorical_accuracy: 0.2031
 576/6375 [=>............................] - ETA: 8s - loss: 2.2177 - sparse_categorical_accuracy: 0.1927
 640/6375 [==>...........................] - ETA: 8s - loss: 2.2210 - sparse_categorical_accuracy: 0.1875
 704/6375 [==>...........................] - ETA: 8s - loss: 2.2027 - sparse_categorical_accuracy: 0.1946
 768/6375 [==>...........................] - ETA: 8s - loss: 2.2083 - sparse_categorical_accuracy: 0.1966
 832/6375 [==>...........................] - ETA: 8s - loss: 2.2133 - sparse_categorical_accuracy: 0.1959
 896/6375 [===>..........................] - ETA: 8s - loss: 2.2043 - sparse_categorical_accuracy: 0.1987
 960/6375 [===>..........................] - ETA: 8s - loss: 2.2068 - sparse_categorical_accuracy: 0.1958
1024/6375 [===>..........................] - ETA: 8s - loss: 2.1963 - sparse_categorical_accuracy: 0.1973
1088/6375 [====>.........................] - ETA: 7s - loss: 2.1973 - sparse_categorical_accuracy: 0.2013
1152/6375 [====>.........................] - ETA: 7s - loss: 2.1935 - sparse_categorical_accuracy: 0.2014
1216/6375 [====>.........................] - ETA: 7s - loss: 2.1919 - sparse_categorical_accuracy: 0.2023
```

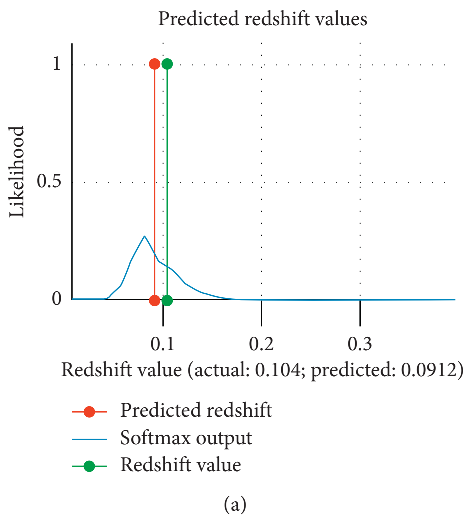
Figure 12: Real-time feedback for the *TrainValidate* job. A plot of the loss during training is shown on the left, and the console output is shown on the right.

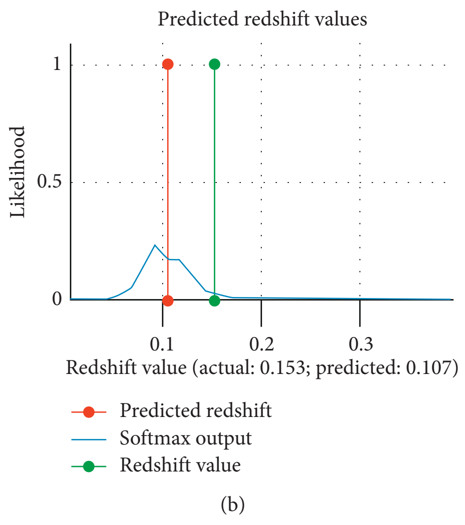
generated appropriately for the given environment without any changes to the operation's implementation. This ensures that the logic for the operation is also portable, as exported pipelines can simply use the appropriate matplotlib backend and behave as expected automatically in the new execution environment.

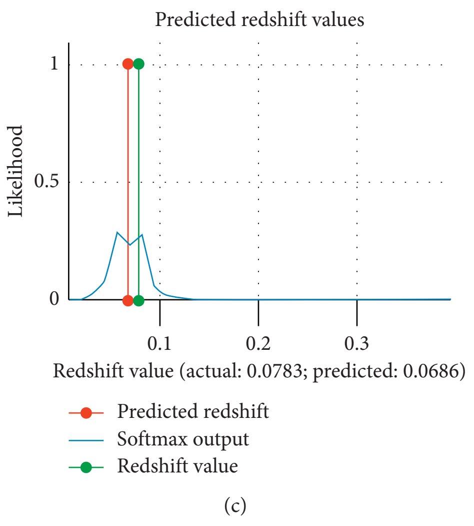
After 10 epochs of training, we use the trained model to predict the redshift values for the given galaxy images. As the model was trained on a classification task, inference is performed by first predicting the probability distribution for the 32 bins used in training and then computing the expected value of this distribution. Figures of the probability distributions are generated in the *VisualizePredictions* job for manual evaluation. One of these images is shown in Figure 13.

The learned features from the neural network for each galaxy are visualized using the *VisualizeEmbeddings* job by designing a pipeline that takes the learned weights (saved as an artifact) from the Train pipeline to run the neural network in inference mode and calculate the error in redshift prediction for each galaxy, which forms the basis for color-coding the embeddings of the dataset. This is shown in Figure 14. The 3D embeddings of the dataset are obtained by *ReduceDimensions*.

This example demonstrates how DeepForge can be used to design a convolutional neural network using the visual editor, as well as how to develop custom operations in Python for fetching and preprocessing the dataset. The example also demonstrates the construction of an entire end-to-end pipeline and execution on a connected cyberinfrastructure through the use of compute and storage adapters. Monitoring of real-time feedback is also shown, including the use of matplotlib to provide a familiar, portable API for creating custom figures during execution.







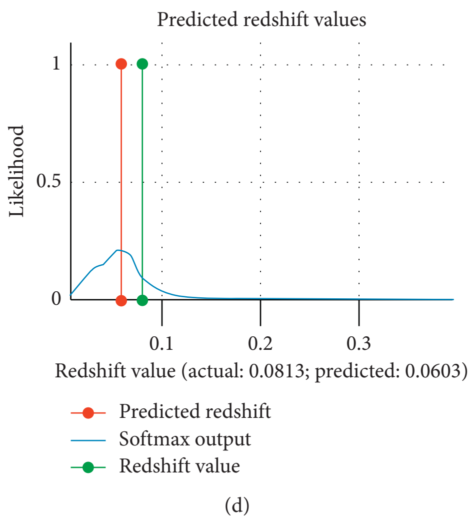
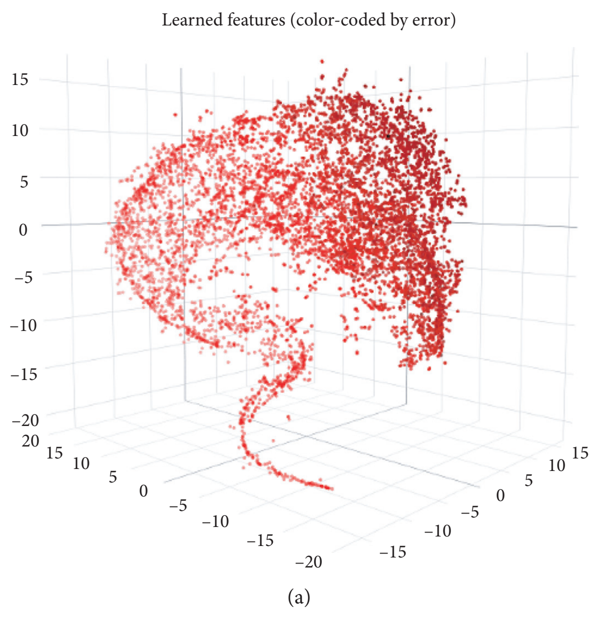


FIGURE 13: Visualization of the redshift probability distribution for four random galaxy images generated by the *VisualizePredictions* job. The probability distribution is generated from the neural network architecture, and support is given by the 32 bins over the range of redshift values. The predicted redshift value is the expected value of this probability distribution.



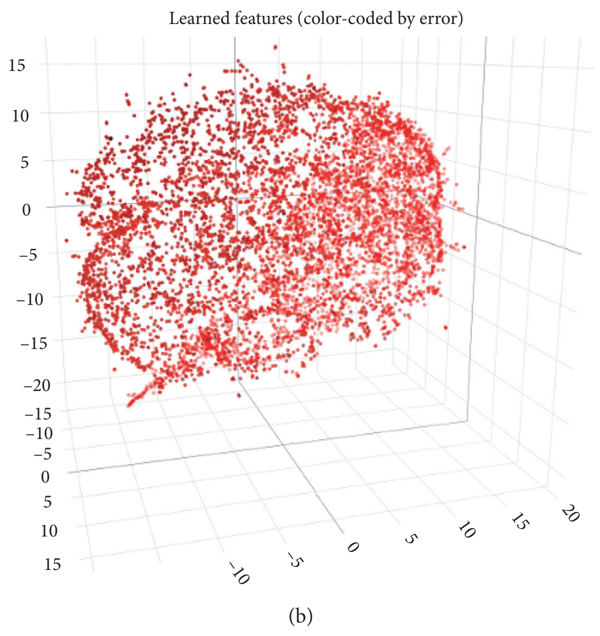


FIGURE 14: Visualization of dataset embeddings using *VisualizeEmbeddings*. Flipped (along the horizontal direction) views of the 3D plot of the embeddings (color-coded by error values where darker points have smaller error values) are shown in (a) and (b).

10. Conclusions

This paper presented DeepForge, a browser-based development environment for designing machine learning workflows. DeepForge has a flexible computational architecture and rich support for integration of external domains including existing compute and storage infrastructures. Utilizing the strengths of Model Integrated Computing, including model analysis and manipulation, this platform provides an intuitive visual interface for enforcing domain semantics and promoting accessibility to deep learning. Furthermore, DeepForge has been designed to facilitate ease of use, reproducibility, and productivity.

Future work includes developing adapters for additional cyberinfrastructures such as NERSC and DOE supercomputers and Globus Connect [24]. Hierarchical support to pipelines to facilitate tasks such as hyperparameter optimization, adding support for additional deep learning libraries, such as PyTorch, as well as creating a registry for hosting operation definitions, architectures, and trained models are also planned. We intend to enhance the existing extension architecture to support custom data visualization capabilities along with model introspection and visualization techniques [25, 26].

Data Availability

DeepForge is an open-source project under the Apache 2.0 license. The source code is available as a GitHub repository (https://github.com/deepforge-dev/deepforge). The example project presented in Section 9 can be accessed from GitHub repository available at https://github.com/deepforge-dev/examples/tree/master/redshift.

Disclosure

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Conflicts of Interest

The authors declare that there is no conflicts of interest regarding publication of this paper.

Acknowledgments

This work was supported by The National Science Foundation under Grant no. 1740151.

References

[1] E. L. Denton, S. Chintala, R. Fergus et al., "Deep generative image models using a laplacian pyramid of adversarial networks," in *Proceedings of the Advances in neural information Processing Systems*, pp. 1486–1494, Montreal, Canada, December 2015.

[2] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the Advances in neural information Processing Systems*, pp. 3111–3119, Lake Tahoe, NV, USA, December 2013.

[3] D. Silver, A. Huang, C. J. Maddison et al., "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[4] M. Abadi, A. Agarwal, P. Barham et al., "TensorFlow: large-scale machine learning on heterogeneous systems," November 2015, https://www.tensorflow.org/.

[5] A. Paszke, S. Gross, F. Massa et al., "Pytorch: An imperative style, high-performance deep learning library," vol. 32, pp. 8024–8035, in *Proceedings of the Advances in Neural Information Processing Systems*, vol. 32, Curran Associates, Inc., Vancouver, Canada, December 2019.

[6] F. Chollet, "Keras," 2015, https://keras.io.

[7] J. Sztipanovits and G. Karsai, "Model-integrated computing," *Computer*, vol. 30, no. 4, pp. 110-111, 1997.

[8] M. Maróti, T. Kecskés, R. Kereskényi et al., "Next generation (meta) modeling: web-and cloud-based collaborative tool infrastructure," in *Proceedings of the 8th Multi-Paradigm Modeling Workshop*, pp. 41–60, Valencia, Spain, September 2014.

[9] "DeepForge Website," 2020, https://deepforge.org.

[10] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler, "Yale: rapid prototyping for complex data mining tasks," in *Proceedings of the 12th ACM SIGKDD international Conference on Knowledge Discovery and Data Mining*, pp. 935–940, ACM, Philadelphia, PA, USA, August 2006.

[11] M. R. Berthold, N. Cebron, F. Dill et al., "KNIME: the konstanz information miner," in *Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007)* Springer, Berlin, Germany, 2007.

[12] Nvidia, "Nvidia DIGITS: Interactive deep learning gpu training system," 2019, https://developer.nvidia.com/digits.

[13] Lobe, https://lobe.ai, 2020.

[14] Y. Jia, E. Shelhamer, J. Donahue et al., "Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international Conference on Multimedia*, pp. 675–678, Orlando, FL, USA, October 2014.

[15] Á. Lédeczi, A. Bakay, M. Maróti et al., "Composing domain-specific design environments," *Computer*, vol. 34, no. 11, 2001.

[16] J. L. Máthe, J. B. Martin, P. Miller et al., "A model-integrated, guideline-driven, clinical decision-support system," *IEEE Software*, vol. 26, no. 4, pp. 54–61, 2009.

[17] E. Long, A. Misra, and J. Sztipanovits, "Increasing productivity at saturn," *Computer*, vol. 31, no. 8, pp. 35–43, 1998.

[18] "SciServer Website," 2020, http://www.sciserver.org/.

[19] L. Deng, "The MNIST database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141-142, 2012.

[20] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.

[21] J. Pasquet, E. Bertin, M. Treyer, S. Arnouts, and D. Fouchez, "Photometric redshifts from sdss images using a convolutional neural network," *Astronomy & Astrophysics*, vol. 621, p. A26, 2019.

[22] M. Taghizadeh-Popp, J. W. Kim, G. Lemson et al., "Sciserver: a science platform for astronomy and beyond," 2020, http://arxiv.org/abs/2001.08619.

[23] C. Szegedy, W. Liu, Y. Jia et al., "Going deeper with convolutions," in *Proceedings of the 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, Boston, MA, USA, June 2015.

[24] I. Foster and C. Kesselman, “Globus: a metacomputing infrastructure toolkit,” *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, no. 2, pp. 115–128, 1997.

[25] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson, “Understanding neural networks through deep visualization,” 2015, http://arxiv.org/abs/1506.06579.

[26] M. D. Zeiler and R. Fergus, “Visualizing and understanding convolutional networks,” in *European Conference on Computer Vision*, pp. 818–833, Springer, Berlin, Germany, 2014.