
deepforge Documentation

Brian Broll

Mar 22, 2023

Getting Started

1	Getting Started	1
2	Quick Start	3
3	Interface Overview	5
4	Custom Operations	13
5	Storage and Compute Adapters	19
6	Quick Start	21
7	Overview	23
8	Native Installation	25
9	Introduction	29
10	Creating Pipelines	31
11	Creating Operations	35
12	Creating Neural Networks	39
13	Executing Pipelines	45
14	Viewing Executions	49
15	CIFAR-10 Classifier	53
16	Redshift Estimator	69
17	Tutorial Project - Redshift	87
18	Redshift Estimation	95
19	Command Line Interface	99
20	Configuration	101

1.1 What is DeepForge?

Deep learning is a promising, yet complex, area of machine learning. This complexity can both create a barrier to entry for those wanting to get involved in deep learning as well as slow the development of those already comfortable in deep learning.

DeepForge is a development environment for deep learning focused on alleviating these problems. Leveraging principles from Model-Driven Engineering, DeepForge is able to reduce the complexity of using deep learning while providing an opportunity for integrating with other domain specific modeling environments created with [WebGME](#).

1.2 Design Goals

As mentioned above, DeepForge focuses on two main goals:

1. **Improving the efficiency** of experienced data scientists/researchers in deep learning
2. **Lowering the barrier to entry** for newcomers to deep learning

It is important to highlight that although one of the goals is focused on lowering the barrier to entry, DeepForge is intended to be more than simply an educational tool; that is, it is important not to compromise on flexibility and effectiveness as a research/industry tool in order to provide an easier experience for beginners (that's what forks are for!).

1.3 Overview and Features

DeepForge provides a collaborative, distributed development environment for deep learning. The development environment is a hybrid visual and textual programming environment. Higher levels of abstraction, such as creating architectures, use visual environments to capture the overall structure of the task while lower levels of abstraction, such as defining custom training functions, utilize text environments. DeepForge contains both a pipelining language

and editor for defining the high level operations to perform while training or evaluating your models as well as a language for defining neural networks (through installing a DeepForge extension such as [DeepForge-Keras](#)).

1.3.1 Concepts and Terminology

- *Operation* - essentially a function written in Python (such as training a model, visualizing results, etc)
- *Pipeline* - directed acyclic graph composed of operations - e.g., a training pipeline may retrieve and normalize data, train an architecture and return the trained model
- *Execution* - when a pipeline is run, an “execution” is created and reports the status of each operation as it is run (distributed over a number of worker machines)
- *Artifact* - an artifact represents some data (either user uploaded or created during an execution)
- *Resource* - a domain specific model (provided by a DeepForge extension) to be used by a pipeline such as a neural network architecture

There are two ways to give DeepForge a try: visit the public deployment at <https://editor.deepforge.org>, or spin up your own deployment locally.

2.1 Connecting to the Public Deployment

As of this writing, registration is not yet open to the public and is only available upon request.

After getting an account for <https://editor.deepforge.org>, the only thing required to get up and running with DeepForge is to determine the [compute and storage adapters](#) to use. If you already have an account with one of the existing integrations, then you should be able to use those without any further setup!

If not, the easiest way to get started is to connect your own desktop to use for compute and to use the S3 adapter to store data and trained model weights. Connect your own desktop for computation using the following command (using docker):

```
docker run -it deepforge/worker:latest --host https://editor.deepforge.org -t <access_↵  
↵token>
```

where *<access token>* is an access token for your user (created from the profile page of <https://editor.deepforge.org>).

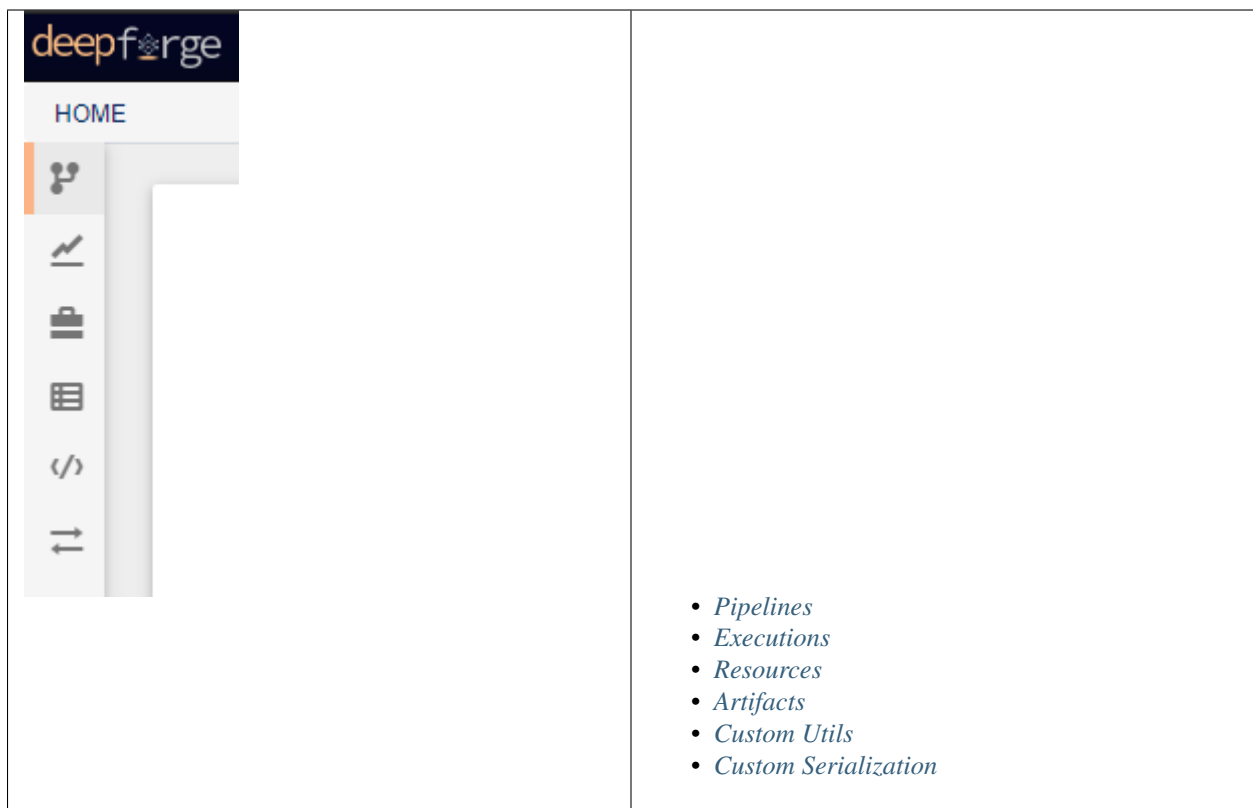
After connecting a machine to use for computation, you can start creating and running pipelines w/o input or output operations! To save artifacts in DeepForge, you will need to connect a storage adapter such as the S3 adapter.

To easily create a custom storage location, [minio](#) is recommended. Simply [spin up an instance of minio](#) on a machine publicly accessible from the internet. Providing the public IP address of the machine (along with any configured credentials) to DeepForge when executing a pipeline will enable you to save any generated artifacts, such as trained model weights, to the minio instance and register it within DeepForge.

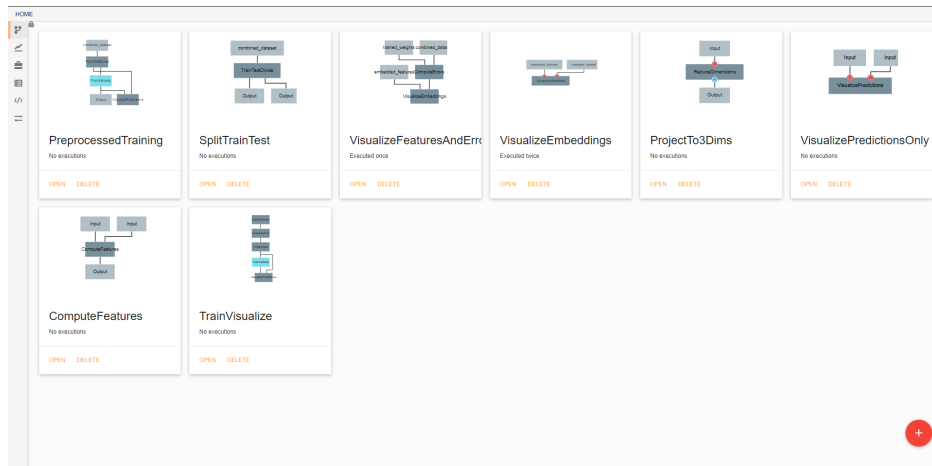
CHAPTER 3

Interface Overview

The DeepForge editor interface is separated into six views for defining all of the necessary features of your desired project. The details of each interface tab are detailed below. You can switch to any of the views at any time by clicking the appropriate icon on the left side of the screen. In order, the tabs are:

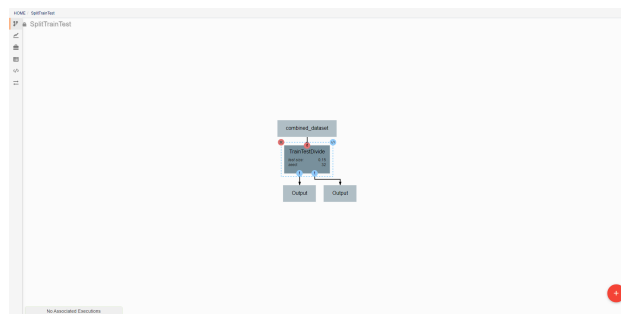


3.1 Pipelines




In the initial view, all pipelines that currently exist in the project are displayed. New pipelines can be created using the floating red button in the bottom right. From this screen, existing pipelines can also be opened for editing, deleted, or renamed.

3.1.1 Pipeline editing



DeepForge pipelines are directed acyclic graphs of operations, where each operation is an isolated python module. Operations are added to a pipeline using the red plus button in the bottom right of the workspace. Any operations that have previously been defined in the project can be added to the pipeline, or new operations can be created when needed. Arrows in the workspace indicate the passing of data between operations. These arrows can be created by clicking on the desired output (bottom circles) of the first operation before clicking on the desired input (top circles) of the second operation. Clicking on a operation also gives the options to delete (red X), edit (blue </>), or change attributes. Information on the editing of operations can be found in [Custom Operations](#)

Pipelines are executed by clicking the yellow play button in the bottom right of the workspace. In the window that appears, you can name the execution, select a computation platform, and select a storage platform. Computation platforms specify what the compute resources used for execution of the operations, such as [SciServer Compute](#), will be. Supported storage platforms, such as endpoints with an S3-compatible API, are used to store intermediate and output data. The provided storage option will be used for storing both the output objects defined in the pipeline, as well as all files used in execution of the pipeline.

 Execute Pipeline (v0.1.0) ✕

Basic Options

Execution name
Optional name for this execution instance

Debug Mode ☐
Allow for operation editing after creation

Compute Options

Compute

Username
SciServer username

Password
SciServer password

Compute Domain
A small job shares resources with up to 4 other jobs and has a max quota for RAM of approx 32GB. A large job runs exclusively and has all CPU cores and RAM available (approx 240GB), however since only one large job will run at a time, there may be a longer wait for the job to start.

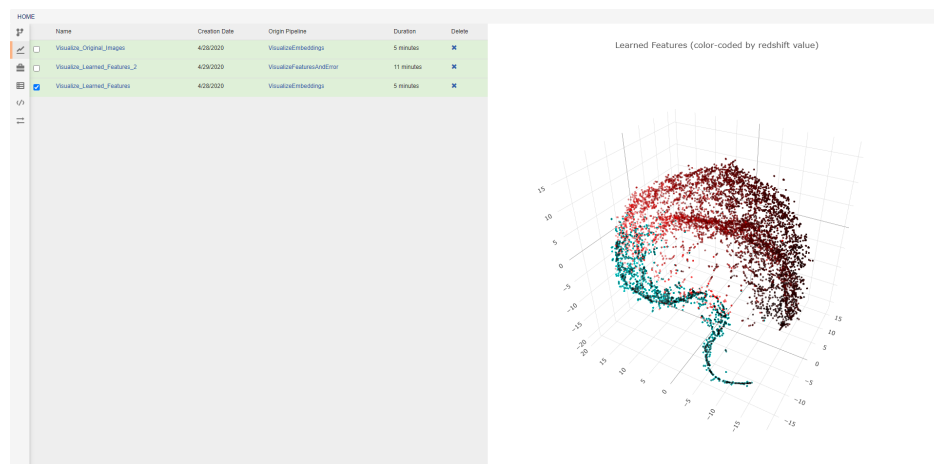
Storage Options

Storage

Username
SciServer username

Password
SciServer password

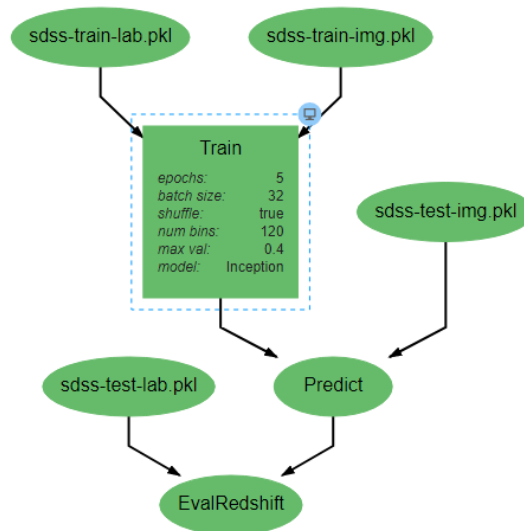
☒ Save these settings in the current user



3.2 Executions

This view allows the review of previous pipeline executions. Clicking on any execution will display any plotted data generated by the pipeline, and selecting multiple executions will display all of the selected plots together. Clicking the provided links will open either the associated pipeline or a trace of the execution (shown below). The blue icon in the top right of every operation allows viewing the text output of that operation. The execution trace can be viewed during execution to check the status of a running job. During execution, the color of a operation indicates its current status. The possible statuses are:

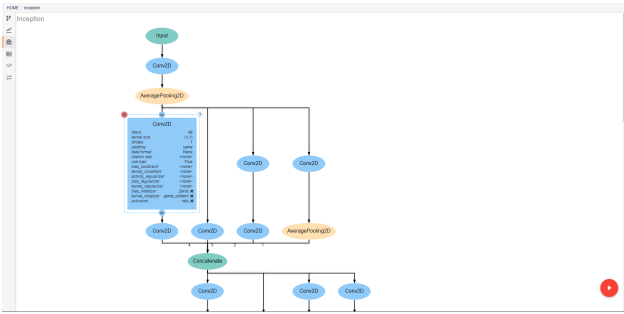
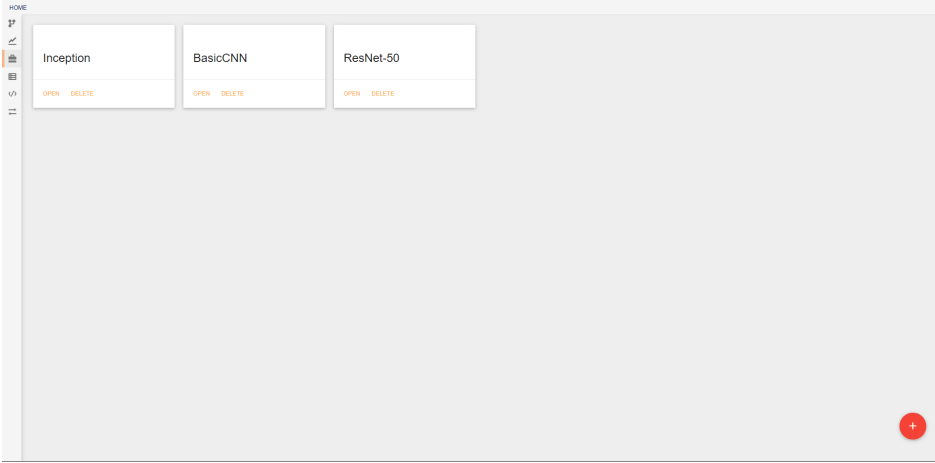
- **Dark gray:** Pending Execution
- **Light gray:** Execution Queued
- **Yellow:** Execution in Progress
- **Orange:** Execution Cancelled
- **Green:** Successfully Finished Execution
- **Red:** Execution Failed



3.3 Resources

This view shows the resources available for use in pipelines. Different types of resources are made available through DeepForge extensions and enable the introduction of new concepts into the project. One such example is [deepforge-keras](#) which enables users to make neural networks architectures with a custom visual editor. The created architectures can then be referenced and used by operations for tasks such as training. From this view, resources can be created, deleted, and renamed.

As with pipelines, the neural networks are depicted as directed graphs. Each node in the graph corresponds to a single layer or operation in the network (information on operations can be found on the [keras website](#)). Clicking on a layer provides the ability to change the attributes of that layer, delete the layer, or add new layers before or after the current layer. Many operations require that certain attributes be defined before use. The Conv2D operation pictured above, for example, requires that the *filters* and *kernel_size* attributes be defined. If these are left as *<none>*, a visual indicator



will show that there is an error to help prevent mistakes. In order to ease analysis and development, hovering over any connecting line will display the shape of the data as it moves between the given layers.

3.4 Artifacts

Name	Type	Size	Creation Date
inception-trained	unknown	152.5 MB	4/24/2020
inception-training-pkl	pickle	688.1 MB	4/25/2020
inception-testing-pkl	pickle	47.8 KB	4/25/2020
inception-validation-pkl	pickle	117.2 MB	4/25/2020
inception-test-pkl	pickle	11.9 KB	4/25/2020
inception-predictions	unknown	270 B	4/25/2020
inception-results	unknown	11.9 KB	4/25/2020
inception-trained	unknown	270 B	4/25/2020
inception-validation-pkl	unknown	17.8 MB	4/25/2020
inception-testing-pkl	unknown	17.8 MB	4/25/2020

In this view, you can see all artifacts that are available to your pipelines. These artifacts can be used in any pipeline through the inclusion of the built in **Input** operation. Artifacts are pieces of saved data that may be associated with some Python data type. Any arbitrary type of data may be used for creating an artifact, but if a data type is not specified, or if a data type is not provided with a *custom serialization*, the artifact will be treated as a *pickle object*. If you have data that cannot be opened with Python’s pickle module, you will need to create a custom serialization as described

below. Some deepforge extensions may also support additional data types by default. DeepForge-Keras, for example, supports saved keras models, in addition to the standard pickle objects, without the need for custom serialization.

A new artifact can be created in one of three ways. First, artifacts are automatically created during the execution of any pipeline that includes the built-in **Output** operation. Second, artifacts can be directly uploaded in this view using the red upload button in the bottom right of the workspace. Using this option will also upload the artifact to the storage platform specified in the popup window. Finally, artifacts that already exist in one of the storage platforms can be imported using the blue import button in the bottom right of the workspace.

Import Existing Data (v0.1.0)

Storage: SciServer Files Service

Username:

Password:

Volume:

Volume Pool: Storage

File Path:

Data Type: pickle

☒ Save these settings in the current user

Run Cancel

Upload Artifact (v0.2.0)

Data to upload:

Data Type: pickle

Storage: SciServer Files Service

Username:

Password:

Volume:

Volume Pool: Storage

☒ Save these settings in the current user

Run Cancel

3.5 Custom Utils

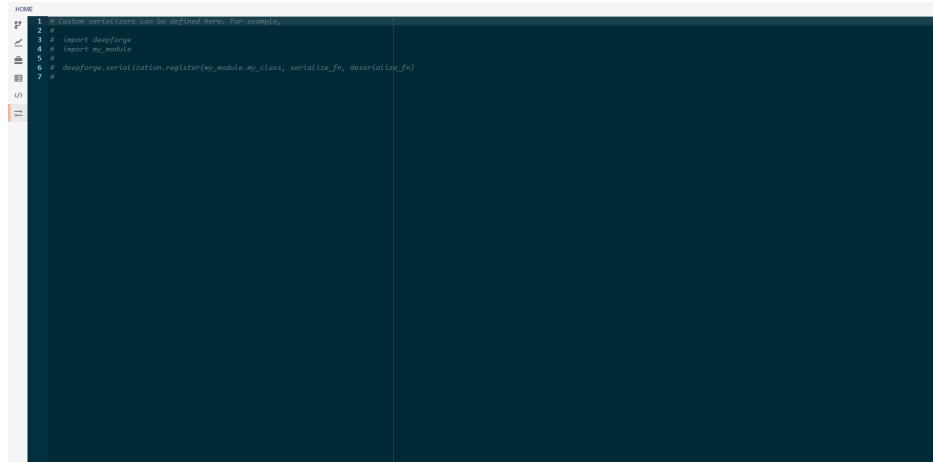
```
HOME
swarp_string.py +
1 # This contains python code accessible from any operation.
2 # Simply write your python code here and then import it elsewhere with:
3 #
4 # from utils.swarp_string import MyCustomCode
5 #
6
7 # This contains python code accessible from any operation.
8 # Simply write your python code here and then import it elsewhere with:
9 #
10 # from utils.swarp_string import MyCustomCode
11 #
12
13 swarp_config_string = """
14 # Default configuration file for Swarp 2.38.0
15 # 18 2019-08-08
16 #
17 #----- Output -----
18 IMAGEOUT_NAME      coadd.fits # Output filename
19 WEIGHTOUT_NAME     coadd.weight.fits # Output weight map filename
20
21 HEADER_ONLY        N          # Only a header at an output file (Y/N)?
22 HEADER_SUFFIX      .head      # Filename extension for additional headers
23
24 #----- Input Weights -----
25
26 WEIGHT_TYPE        NONE        # BACKGROUND, AMP, RMS, RMS_SQRTSPACE
27                               # or MAP_WEIGHT
28 RESCALE_WEIGHTS     Y          # Rescale input weights/variances (Y/N)?
29 WEIGHT_SUFFIX       weight.fits # Suffix to use for weight maps
30 WEIGHT_IMAGE        N          # Weightmap filename (if suffix not used)
31                               # Call on for each weight map?
32 WEIGHT_THRESHOLD    0          # Bad pixel weight threshold
33
34 #----- Co-addition -----
35
36 COMBINE             Y          # Combine resampled images (Y/N)?
37 COMBINE_TYPE        MEDIAN      # MEDIAN, AVERAGE, MIN, MAX, WEIGHTED, CLIPPED
38                               # OR ELUCLUST, RMS, RMS_SQRT, MAP,
39                               # WEIGHTED_WEIGHT, MEDIAN_WEIGHT,
40                               # AND, AND_OR, OR, AND
41 CLIP_AMPRAC         0.3        # Fraction of flux variation allowed
42 CLIP_STDEV          4.0        # RMS across multiple variation allowed
43 CLIP_MEDFLOD        N          # with clipping
44                               # Write output file with coordinates of
45                               # clipped pixels (Y/N)?
46 """
```

This view allows the creation and editing of custom utility modules. Utilities created here can be imported into any pipeline operation. For example, the `swarp_config_string` shown above can be printed out in a operation using the following code:

```
import utils.swarp_string as ss
print(ss.swarp_config_string)
```

3.6 Custom Serialization

In this view, you can create custom serialization protocols for the creation and use of artifacts that are neither python pickle objects nor keras models. To create a serialization, you will need to define two functions, one for serialization and one for deserialization. These functions must then be passed as arguments to the `deepforge.serialization.register`



function as shown in the commented code above. The serializer and deserializer should have the same signatures as the dump and load functions respectively from python's [pickle module](#).

Custom Operations

In this document we will outline the basics of custom operations including the operation editor and operation feedback utilities.

4.1 The Basics

Operations are used in pipelines and have named inputs and outputs. When creating a pipeline, if you don't currently find an operation for the given task, you can easily create your own by selecting the *New Operation...* operation from the add operation dialog. This will create a new operation definition and open it in the operation editor. The operation editor has two main parts, the interface editor and the implementation editor.

The interface editor is provided on the right and presents the interface as a diagram showing the input data and output data as objects flowing into or out of the given operation. Selecting the operation node in the operation interface editor will expand the node and allow the user to add or edit attributes for the given operation. These attributes are exposed when using this operation in a pipeline and can be set at design time - that is, these are set when creating the given pipeline. The interface diagram may also contain light blue nodes flowing into the operation. These nodes represent "references" that the operation accepts as input before running. When using the operation, references will appear alongside the attributes but will allow the user to select from a list of all possible targets when clicked.

The operation editor also provides an interface to specify operation python dependencies. DeepForge uses `conda` to manage python dependencies for an operation. This pairs well with the integration of various compute platforms that available to the user and the only requirement for a user is to have Conda installed in their computing platform. You can specify operation dependencies using a conda environment `file` as shown in the diagram below:

To the left of the operation editor is the implementation editor. The implementation editor is a code editor specially tailored for programming the implementations of operations in DeepForge. It also is synchronized with the interface editor. A section of the implementation is shown below:

```
import numpy as np
from sklearn.model_selection import train_test_split
import keras
import time
from matplotlib import pyplot as plt
```

(continues on next page)

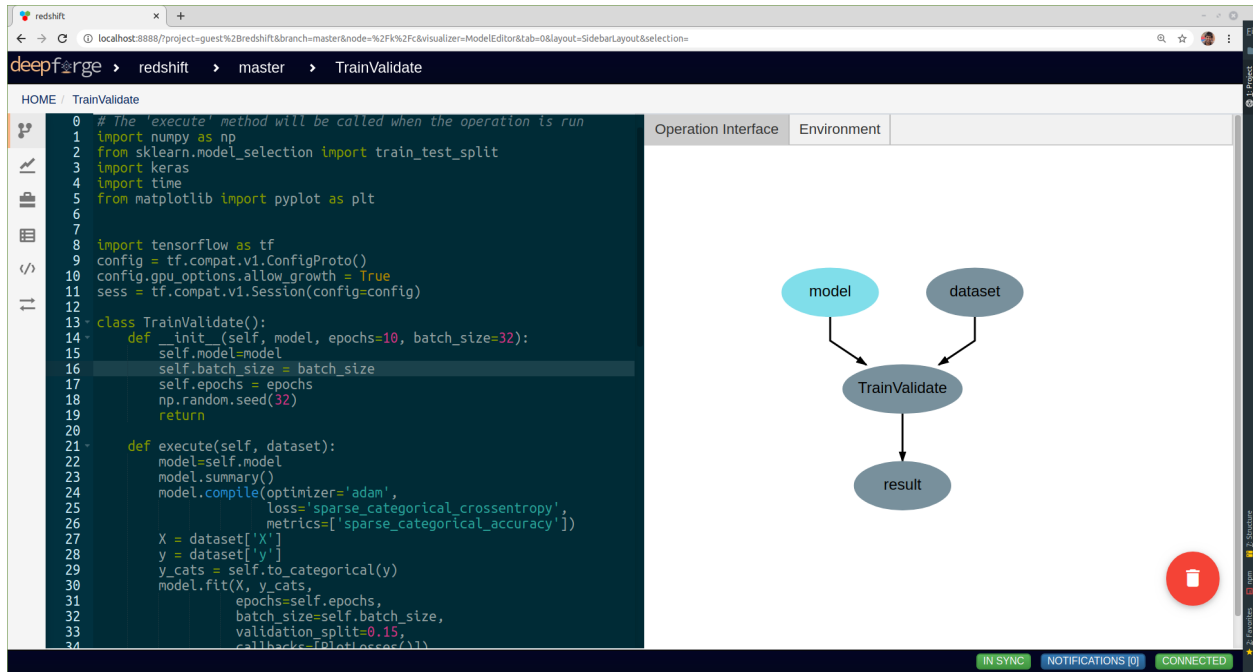


Fig. 1: Editing the “TrainValidate” operation from the “redshift” example

(continued from previous page)

```

import tensorflow as tf

import tensorflow as tf
config = tf.compat.v1.ConfigProto()
config.gpu_options.allow_growth = True
sess = tf.compat.v1.Session(config=config)

class TrainValidate():
    def __init__(self, model, epochs=10, batch_size=32):
        self.model=model
        self.batch_size = batch_size
        self.epochs = epochs
        np.random.seed(32)
        return

    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()])
  
```

(continues on next page)

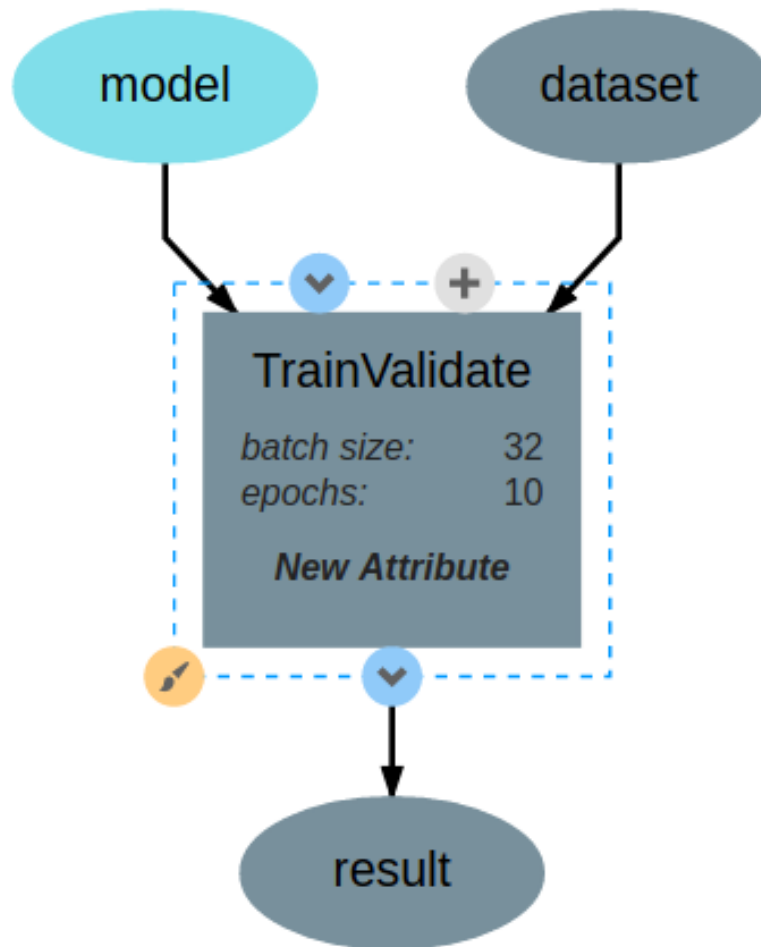


Fig. 2: The TrainValidate operation accepts training data, a model and attributes for setting the batch size, and the number of epochs.

Operation Interface	Environment
<pre> 1 # Conda environment to use when executing the given operation. 2 # For more information, check out https://docs.conda.io/projects/conda 3 dependencies: 4 - python=3.7 5 - pip: 6 - tensorflow==1.14 7 - keras=2.2.5 8 - matplotlib 9 - numpy 10 - scikit-learn 11 </pre>	

Fig. 3: The operation environment contains python dependencies for the given operation.

(continued from previous page)

```

        return model.get_weights()

    def to_categorical(self, y, max_y=0.4, num_possible_classes=32):
        one_step = max_y / num_possible_classes
        y_cats = []
        for values in y:
            y_cats.append(int(values[0] / one_step))
        return y_cats

    def datagen(self, X, y):
        # Generates a batch of data
        X1, y1 = list(), list()
        n = 0
        while 1:
            for sample, label in zip(X, y):
                n += 1
                X1.append(sample)
                y1.append(label)
                if n == self.batch_size:
                    yield [np.array(X1), y1]
                    n = 0
                    X1, y1 = list(), list()

class PlotLosses(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.i = 0
        self.x = []
        self.losses = []

    def on_epoch_end(self, epoch, logs={}):
        self.x.append(self.i)
        self.losses.append(logs.get('loss'))
        self.i += 1

        self.update()

    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()

```

The “TrainValidate” operation uses capabilities from the `keras` package to train the neural network. This operation sets all the parameters using values provided to the operation as either attributes or references. In the implementation, attributes are provided as arguments to the constructor making the user defined attributes accessible from within the implementation. References are treated similarly to operation inputs and are also arguments to the constructor. This can be seen with the `model` constructor argument. Finally, operations return their outputs in the `execute` method; in this example, it returns a single output named `model`, that is, the trained neural network.

After defining the interface and implementation, we can now use the “TrainValidate” operation in our pipelines! An example is shown below.

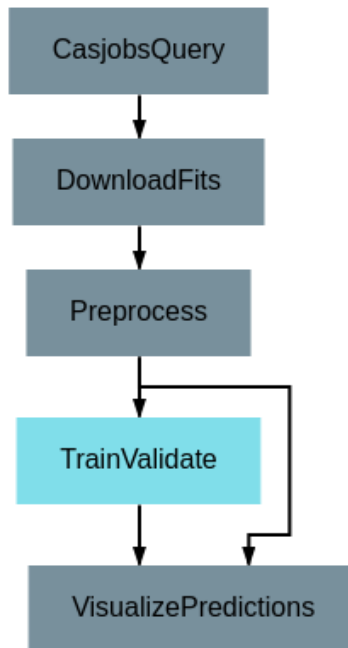


Fig. 4: Using the “TrainValidate” operation in a pipeline

4.2 Operation Feedback

Operations in DeepForge can generate metadata about its execution. This metadata is generated during the execution and provided back to the user in real-time. An example of this includes providing real-time plotting feedback. When implementing an operation in DeepForge, this metadata can be created using the `matplotlib` plotting capabilities.



Fig. 5: An example graph of the loss function while training a neural network.

Storage and Compute Adapters

DeepForge is designed to integrate with existing computational and storage resources and is not intended to be a competitor to existing HPC or object storage frameworks. This integration is made possible through the use of compute and storage adapters. This section provides a brief description of these adapters as well as currently supported integrations.

5.1 Storage Adapters

Projects in DeepForge may contain artifacts which reference datasets, trained model weights, or other associated binary data. Although the project code, pipelines, and models are stored in MongoDB, this associated data is stored using a storage adapter. Storage adapters enable DeepForge to store this associated data using an appropriate storage resource, such as a object store w/ an S3-compatible API. This also enables users to “bring their own storage” as they can connect their existing cyberinfrastructure to a public deployment of DeepForge. Currently, DeepForge supports 3 different storage adapters:

1. S3 Storage: Object storage with an S3-compatible API such as [minio](#) or [AWS S3](#)
2. SciServer Files Service : Files service from [SciServer](#)
3. WebGME Blob Server : Blob storage provided by [WebGME](#)

5.2 Compute Adapters

Similar to storage adapters, compute adapters enable DeepForge to integrate with existing cyberinfrastructure used for executing some computation or workflow. This is designed to allow users to leverage their existing HPC or other computational resources with DeepForge. Compute adapters provide an interface through which DeepForge is able to execute workflows (e.g., training a neural network) on external machines.

Currently, the following compute adapters are available:

1. WebGME Worker: A worker machine which polls for jobs via the [WebGME Executor Framework](#). Registered users can connect their own compute machines enabling them to use their personal desktops with DeepForge.
2. SciServer-Compute: Compute service offered by [SciServer](#)

3. Server Compute: Execute the job on the server machine. This is similar to the execution model used by Jupyter notebook servers.

CHAPTER 6

Quick Start

The recommended (and easiest) way to get started with DeepForge is using docker-compose. First, install [docker](#) and [docker-compose](#).

Next, download the docker-compose file for DeepForge:

```
wget https://raw.githubusercontent.com/deepforge-dev/deepforge/master/docker/docker-  
compose.yml
```

Next, you must decide if you would like authentication to be enabled. For production deployments, this is certainly recommended. However, if you just want to spin up DeepForge to “kick the tires”, this is certainly not necessary.

6.1 Without User Accounts

Start the docker containers with `docker-compose run`:

```
docker-compose --file docker-compose.yml run -p 8888:8888 -p 8889:8889 -e "NODE_  
ENV=default" server
```

6.2 User Authentication Enabled

First, generate a public and private key pair

```
mkdir -p deepforge_keys  
openssl genrsa -out deepforge_keys/private_key  
openssl rsa -in deepforge_keys/private_key -pubout > deepforge_keys/public_key  
export TOKEN_KEYS_DIR="$(pwd)/deepforge_keys"
```

Then start DeepForge using `docker-compose run`:

```
docker-compose --file docker-compose.yml run -v "${TOKEN_KEYS_DIR}:/token_keys" -p ↵  
↵8888:8888 -p 8889:8889 server
```

Finally, create the admin user by connecting to the server's docker container. First, get the ID of the container using:

```
docker ps
```

Then, connect to the running container:

```
docker exec -it <container ID> /bin/bash
```

and create the admin account

```
./bin/deepforge users useradd admin <admin email> <password> -c -s
```

After setting up DeepForge (with or without user accounts), it can be used by opening a browser to <http://localhost:8888>!

For detailed instructions about deployment installations, check out our [deployment installation instructions](#). An example of customizing a deployment using docker-compose can be found [here](#).

7.1 DeepForge Component Overview

DeepForge is composed of four main elements:

- *Client*: The connected browsers working on DeepForge projects.
- *Server*: Main component hosting all the project information and is connected to by the clients.
- *Compute*: Connected computational resources used for executing pipelines.
- *Storage*: Connected storage resources used for storing project data artifacts such as datasets or trained model weights.

7.2 Component Dependencies

The following dependencies are required for each component:

- *Server* (NodeJS LTS)
- *Database* (MongoDB v3.0.7)
- *Client*: We recommend using Google Chrome and are not supporting other browsers (for now). In other words, other browsers can be used at your own risk.

7.3 Configuration

After installing DeepForge, it can be helpful to check out [configuring DeepForge](#)

Native Installation

8.1 Dependencies

First, install [NodeJS](#) (LTS) and [MongoDB](#). You may also need to install git if you haven't already.

Next, you can install DeepForge using npm:

```
npm install -g deepforge
```

Now, you can check that it installed correctly:

```
deepforge --version
```

After installing DeepForge, it is recommended to install the [deepforge-keras](#) extension which provides capabilities for modeling neural network architectures:

```
deepforge extensions add deepforge-keras
```

DeepForge can now be started with:

```
deepforge start
```

8.1.1 Database

Download and install MongoDB from the [website](#). If you are planning on running MongoDB locally on the same machine as DeepForge, simply start *mongod* and continue to setting up DeepForge.

If you are planning on running MongoDB remotely, set the environment variable “MONGO_URI” to the URI of the Mongo instance that DeepForge will be using:

```
MONGO_URI="mongodb://pathToMyMongo.com:27017/myCollection" deepforge start
```

8.1.2 Server

The DeepForge server is included with the deepforge cli and can be started simply with

```
deepforge start --server
```

By default, DeepForge will start on `http://localhost:8888`. However, the port can be specified with the `-port` option. For example:

```
deepforge start --server --port 3000
```

8.1.3 Worker

The DeepForge worker (used with WebGME compute) can be used to enable users to connect their own machines to use for any required computation. This can be installed from <https://github.com/deepforge-dev/worker>. It is recommended to install [Conda](#) on the worker machine so any dependencies can be automatically installed.

8.1.4 Updating

DeepForge can be updated with the command line interface rather simply:

```
deepforge update
```

```
deepforge update --server
```

For more update options, check out `deepforge update --help`!

8.2 Manual Installation (Development)

Installing DeepForge for development is essentially cloning the repository and then using `npm` (node package manager) to run the various start, test, etc, commands (including starting the individual components). The deepforge cli can still be used but must be referenced from `./bin/deepforge`. That is, `deepforge start` becomes `./bin/deepforge start` (from the project root).

8.2.1 DeepForge Server

First, clone the repository:

```
git clone https://github.com/dfst/deepforge.git
```

Then install the project dependencies:

```
npm install
```

To run all components locally start with

```
./bin/deepforge start
```

and navigate to `http://localhost:8888` to start using DeepForge!

Alternatively, if jobs are going to be executed on an external worker, run `./bin/deepforge start -s` locally and navigate to `http://localhost:8888`.

8.2.2 Updating

Updating can be done the same as any other git project; that is, by running *git pull* from the project root. Sometimes, the dependencies need to be updated so it is recommended to run *npm install* following *git pull*.

8.3 Manual Installation (Production)

To deploy a deepforge server in a production environment, follow the following steps. These steps are for using a Linux server and if you are using a platform other than Linux, we recommend using a dockerized deployment.

1. Make sure you have a working installation of [Conda](#) in your server.
2. Clone this repository to your production server.

```
git clone https://github.com/deepforge-dev/deepforge.git
```

3. Install dependencies and add extensions:

```
cd deepforge && npm install
./bin/deepforge extensions add deepforge-keras
```

2. Generate token keys for user-management (required for user management).

```
chmod +x utils/generate_token_keys.sh
./utils/generate_token_keys.sh
```

Warning: The token keys are generated in the root of the project by default. If the token keys are stored in the project root, they are accessible via */extlib*, which is a security risk. So, please make sure you move the created token keys out of the project root.

3. Configure your environment variables:

```
export MONGO_URI=mongodb://mongo:port/deepforge_database_name
export DEEPFORGE_HOST=https://url.of.server
export DEEPFORGE_PUBLIC_KEY=/path/to/public_key
export DEEPFORGE_PRIVATE_KEY=/path/to/private_key
```

4. Add a site-admin account by using *deepforge-users* command:

```
./bin/deepforge-users useradd -c -s admin_username admin_email admin_password
```

5. Now you should be ready to deploy a production server which can be done using *deepforge* command.

```
NODE_ENV=production ./bin/deepforge start --server
```

Note: The default port for a deepforge server is 8888. It can be changed using the option *-p* in the command above.

CHAPTER 9

Introduction

This tutorial provides detailed instructions for creating a complete DeepForge project from scratch. The motivating examples for this walkthrough will be a simple image classification task using [CIFAR-10](#) as our dataset and a more complex astronomical redshift estimation task using [Sloan Digital Sky Survey](#) as our dataset.

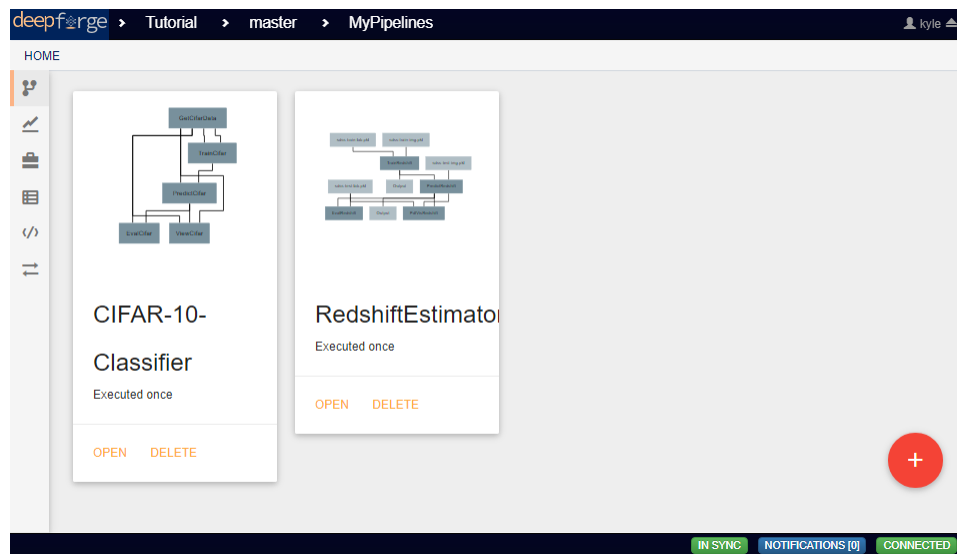
The overall process of creating projects is centered around the creation of data processing **pipelines** that will be executed to generate the data, visualizations, models, etc. that we need. This guide begins with a detailed walkthrough on how to create pipelines and all their constituent parts. After this introductory walkthrough will be detailed walkthroughs on how to create a pair of useful pipelines using the motivating examples.

The screenshot displays the DeepForge web application interface. At the top, a dark navigation bar contains the 'deepforge' logo and a breadcrumb trail: 'Tutorial > master > MyPipelines'. A user profile icon for 'kyle' is in the top right. Below the navigation bar, a 'HOME' header is visible. On the left, a vertical sidebar shows icons for home, a chart, a list, a document, and a double arrow. The main content area features two pipeline cards. The first card, 'CIFAR-10-Classifier', shows a flowchart with steps: 'GetCIFARData', 'TrainClassifier', 'PredictClassifier', 'EvalClassifier', and 'ViewClassifier'. It notes 'Executed once' and has 'OPEN' and 'DELETE' buttons. The second card, 'RedshiftEstimator', shows a more complex flowchart with steps like 'GetSDSSData', 'TrainRedshiftModel', 'PredictRedshift', 'EvalRedshift', and 'ViewRedshift'. It also notes 'Executed once' and has 'OPEN' and 'DELETE' buttons. A red circular button with a white '+' sign is in the bottom right. At the very bottom, a dark status bar shows 'IN SYNC', 'NOTIFICATIONS [0]', and 'CONNECTED'.

CHAPTER 10

Creating Pipelines

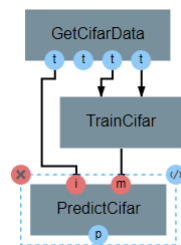
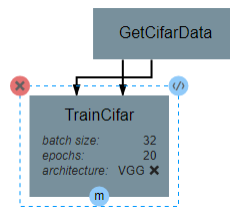
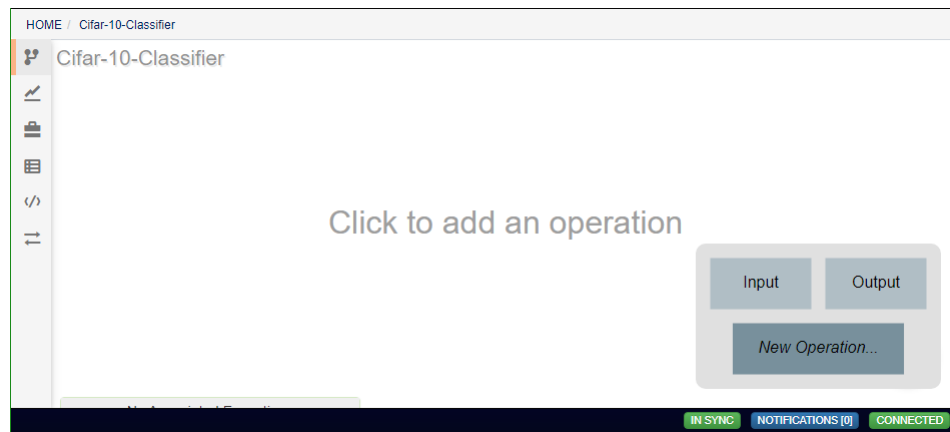
From the home view in the *Pipelines* tab, you are presented with a list of the pipelines that have already been created. Clicking on any of these pipelines will allow editing of that pipeline. To create a new, empty pipeline, click on the red button in the bottom right corner of the workspace.



The basic unit of work in a pipeline is the operation. Operations can be added using the red button in the bottom right corner of the workspace.

After an operation has been added, the attributes of that operation can be changed by clicking on the operation and then clicking on the current value of that attribute.

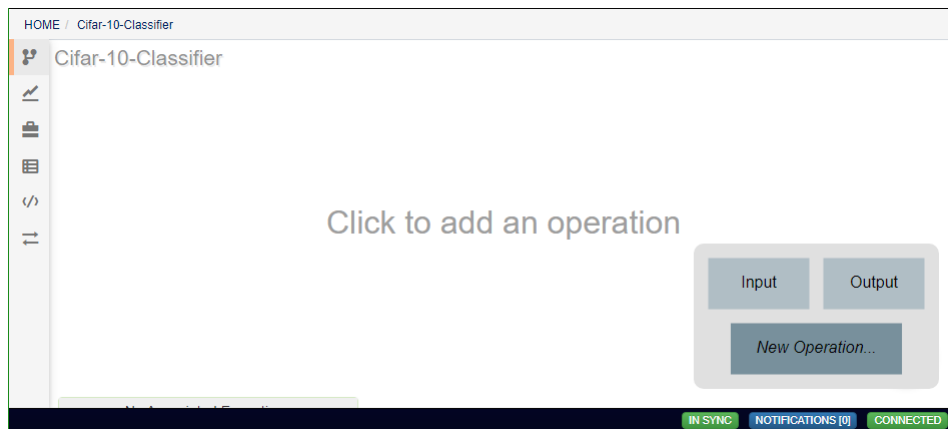
Operation inputs and outputs are represented by blue circles that are visible after clicking on the operation. Blue circles on the top on the operation represent inputs, while circles on the bottom represent outputs. The red X circle can be clicked to remove an operation from the pipeline. This does not remove it from the set of available operations. The `</>` icon will open the operation editor view. Holding alt while clicking this icon will instead create a copy of the operation and open the new copy's editor.



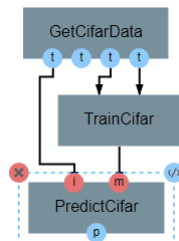
Operations can be connected by clicking on an input or output of an operation before clicking on an output or input respectively of another operation. All input and output connections are optional, though missing outputs may give errors depending upon the operation's internal logic.

Creating Operations

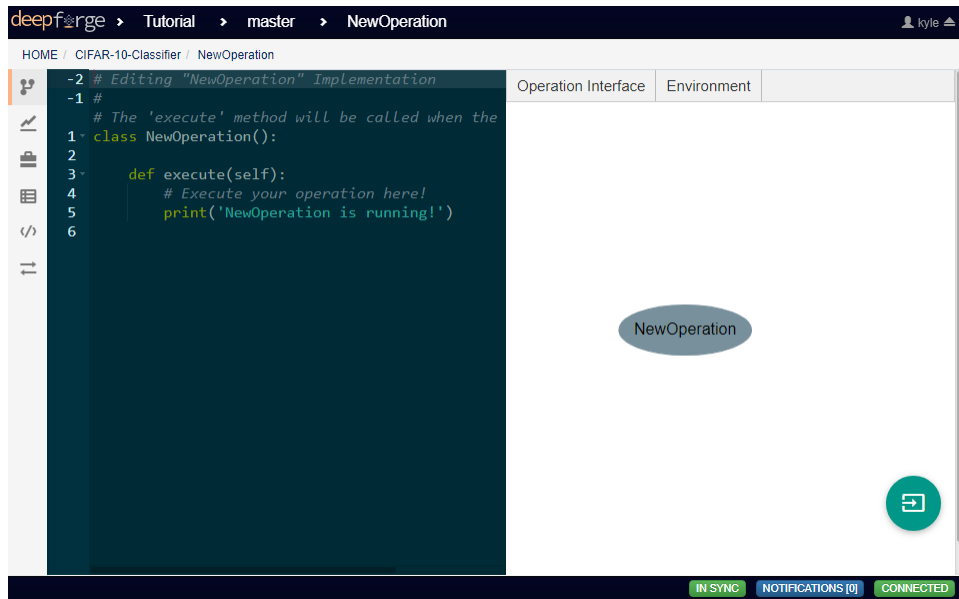
When adding an operation to a pipeline, new operations can be created by clicking the *New Operation* option. This will open the operation editor for the new operation.



This editor can also be reached for existing operations by clicking the `</>` icon when editing an operation's attributes.



This editor has two primary views for editing the operation. The left view allows editing the underlying code of the operation directly. The right view provides a graphical means of adding inputs, outputs, and attributes.



11.1 Editing the Operation Interface



Clicking on the operation in the right view will allow editing the operation interface. The operation interface consists of design time parameters (attributes and references) as well as inputs and outputs generated at runtime.

Attributes can be added by clicking the *New Attribute* label, which will open a dialog box where you can define the name, type, default value, and other metadata about that attribute. This dialog box can be viewed again to edit the attribute by clicking on the name of the attribute in the right-side view.

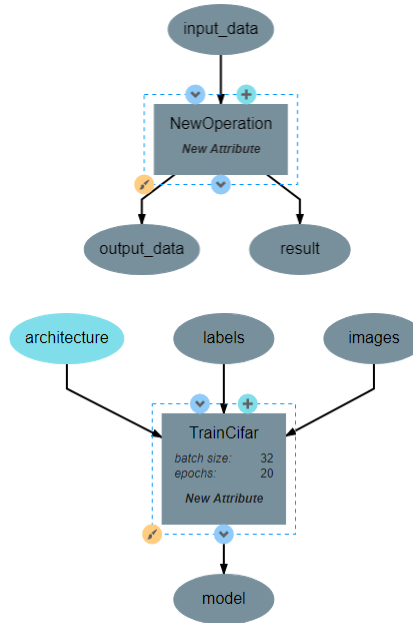
The screenshot shows a dialog box titled 'Edit attribute parameters...'. It contains the following fields and options:

- Name:** A text input field containing 'epochs'.
- Description:** A text area with the placeholder text 'Enter an optional description...'.
- Type:** A dropdown menu currently set to 'integer'.
- Read-only:** An unchecked checkbox.
- Hidden:** An unchecked checkbox.
- Enumeration:** An unchecked checkbox.
- Default value:** A text input field containing '20'.
- Inclusive Value Range:** Two text input fields labeled 'minimum' and 'maximum'.

At the bottom of the dialog, there are three buttons: 'Delete' (red), 'Save' (blue), and 'Cancel' (grey).

Inputs and outputs can be added using the blue arrow icons. Any number of inputs and outputs can be added to an operation, but each should be given a unique name.

Using the plus icon, references to resources can be added to the operation. These resources will usually be some form of neural network. As with inputs and outputs, any number of resources can be added to an operation.



The paint brush icon allows editing the color of the operation, but this is purely aesthetic and does not affect the operation's underlying logic.

11.2 Implementing the Operation

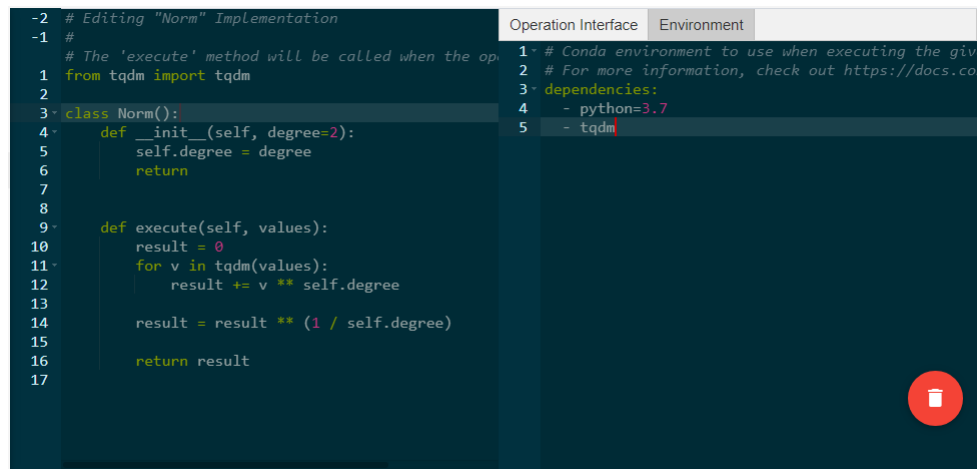
```

-2 # Editing "Norm" Implementation
-1 #
# The 'execute' method will be called when the operation is run
1 class Norm():
2     def __init__(self, degree=2):
3         self.degree = degree
4         return
5
6
7     def execute(self, values):
8         result = 0
9         for v in values:
10             result += v ** self.degree
11
12         result = result ** (1 / self.degree)
13
14         return result
15

```

In the left-side view, the underlying logic of the operation is implemented using Python. The code here can be edited freely. All operations are defined by a class with the same name as the operation. This class has two primary functions associated with it. The first is the `__init__` function, which will appear automatically when creating the operation's first attribute. This function will run when the operation is initialized and is primarily used for the creation of class variables and the processing of attributes. Note that operation attributes will not be accessible from other functions and must be assigned to a class variable in this function to be utilized elsewhere. The second primary function is the `execute` function. This is the function that is executed when the operation is running. Any number of other classes and functions can be created in the code editor, but they will not be executed if they are not called within the `execute` function. The outputs of the `execute` function will also be the outputs of the operation.

11.3 Importing Libraries



The screenshot displays the DeepForge IDE interface. On the left, the 'Operation Interface' tab is active, showing a Python class named `Norm`. The code includes a docstring, an import for `tqdm`, and two methods: `__init__` and `execute`. The `execute` method uses `tqdm` for a loop. On the right, the 'Environment' tab is active, showing a list of dependencies for the operation's environment. The dependencies are `python=3.7` and `tqdm`. A red trash icon is visible in the bottom right corner of the IDE window.

```
-2 # Editing "Norm" Implementation
-1 #
1 # The 'execute' method will be called when the op
1 from tqdm import tqdm
2
3 class Norm():
4     def __init__(self, degree=2):
5         self.degree = degree
6         return
7
8
9     def execute(self, values):
10        result = 0
11        for v in tqdm(values):
12            result += v ** self.degree
13
14        result = result ** (1 / self.degree)
15
16        return result
17
```

```
1 # Conda environment to use when executing the giv
2 # For more information, check out https://docs.co
3 dependencies:
4 - python=3.7
5 - tqdm
```

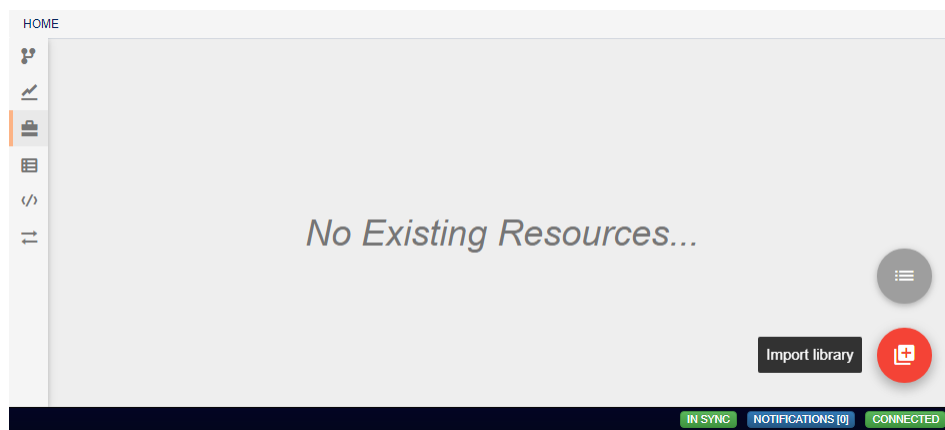
Python libraries can be used within an operation by importing them, which is usually done above the operation class. Any library that is installed on the compute backend's python environment can be imported as normal, but more niche libraries that are available through pip or anaconda need to be specified as dependencies for the operation by clicking the *Environment* tab on the right side. The dependencies described here should be defined using the same syntax as in a [conda environment file](#).

Creating Neural Networks

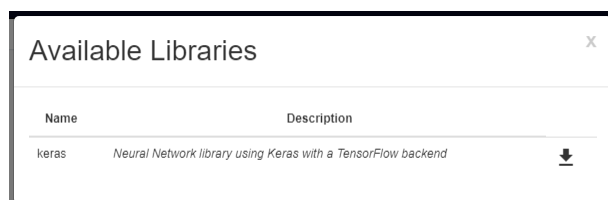
This page will guide you through the steps needed to create a neural network architecture for use in pipelines.

12.1 Importing Resource Libraries

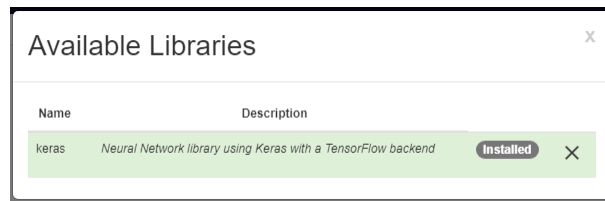
Neural networks and other models can be created from the *Resources* tab on the sidebar. Before any models can be created in this tab, you must import the associated library into the project using the red floating button in the bottom right of the workspace.



In the box that appears, you will see a list of libraries that are available for import.

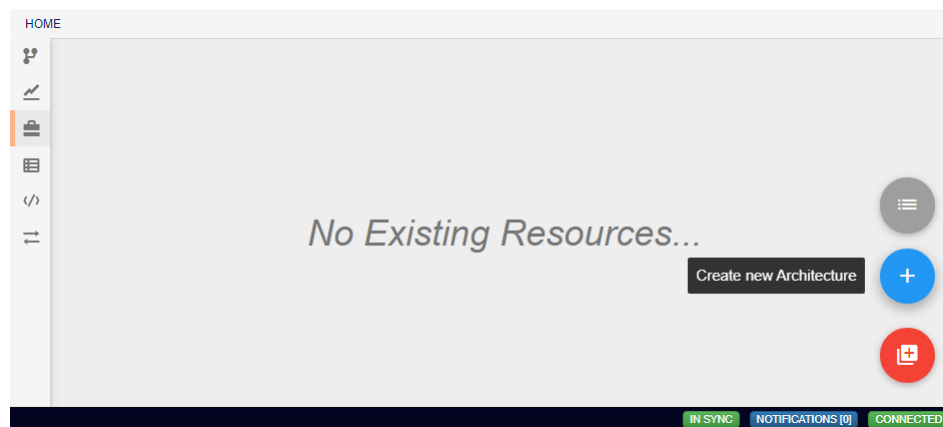


Clicking the download icon will install that library and allow creation of associated models. The keras library, for instance, allows for the creation of neural network models.

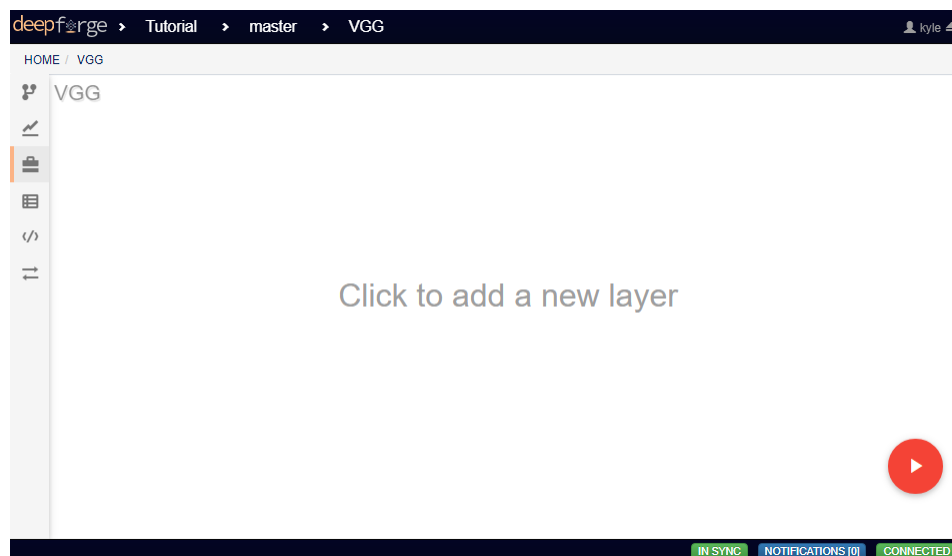


12.2 Creating a New Architecture

After any library has been imported, new models can be created by hovering over the import library button and clicking the floating blue button that appears. This will generate a blank model and automatically open up that model's workspace.

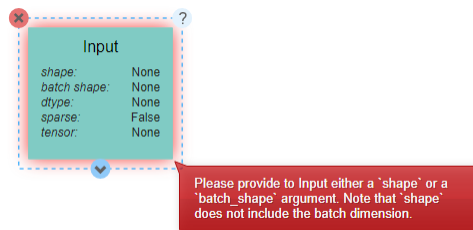


Clicking anywhere in the workspace will add the first layer of the architecture, which will always be an input layer. Just as with pipelines, these architectures are represented by a flowchart, with each node representing a single layer in the neural network.

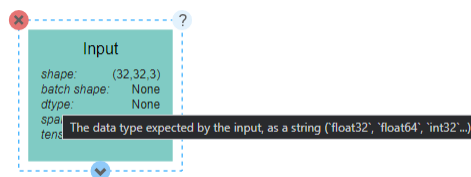


12.3 Editing Network Layers

Clicking on a layer allows for changing the parameters of that layer. Many of these parameters can be left undefined, but many layers require that some specific parameters be given. If a layer has not been supplied with the necessary parameters, or if there is some other error encountered at that layer when building the network, the layer will be highlighted with a red border. Hovering the mouse over the layer will reveal the error. Hovering over a newly created **Input** layer, for example, shows us that the layer requires that either the `shape` or `batch_shape` parameter must be defined.



Some parameters may not be immediately clear on their effects from the name alone. For unfamiliar parameters, hovering over the name of the parameter will reveal a short description.



In addition, clicking on the ? icon in the top right of the expanded layer will display documentation on the layer as a whole, including descriptions of all available parameters.

Input Documentation

`Input()` is used to instantiate a Keras tensor. A Keras tensor is a tensor object from the underlying backend (Theano, TensorFlow or CNTK), which we augment with certain attributes that allow us to build a Keras model just by knowing the inputs and outputs of the model. For instance, if `a`, `b` and `c` are Keras tensors, it becomes possible to do: `model = Model(input=[a, b], output=c)`. The added Keras attributes are: `_keras_shape`: Integer shape tuple propagated via Keras-side shape inference. `_keras_history`: Last layer applied to the tensor; the entire layer graph is retrievable from that layer, recursively.

Arguments

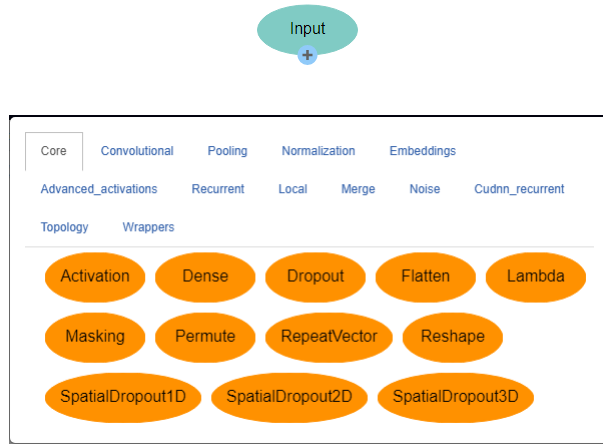
- `shape`: A shape tuple (integer), not including the batch size. For instance, `shape=(32,)` indicates that the expected input will be batches of 32-dimensional vectors.
- `batch_shape`: A shape tuple (integer), including the batch size. For instance, `batch_shape=(10, 32)` indicates that the expected input will be batches of 10 32-dimensional vectors. `batch_shape=(None, 32)` indicates batches of an arbitrary number of 32-dimensional vectors.
- `name`: An optional name string for the layer. Should be unique in a model (do not reuse the same name twice). It will be autogenerated if it isn't provided.
- `dtype`: The data type expected by the input, as a string (`'float32'`, `'float64'`, `'int32'`...).
- `sparse`: A boolean specifying whether the placeholder to be created is sparse.
- `tensor`: Optional existing tensor to wrap into the `Input` layer. If set, the layer will not create a placeholder tensor.

Returns

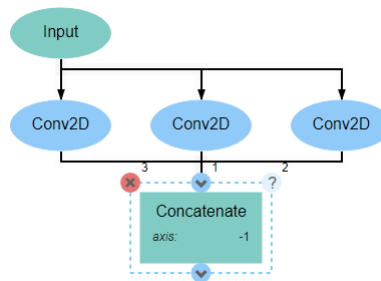
A tensor.

12.4 Adding Additional Layers

To add additional layers, you can click on the arrow icons on the top or bottom of any layer. The icon should become a + icon and clicking again will open a menu from which the desired layer type can be chosen.



Layers can also be removed from the network by expanding the layer and clicking the red X icon in the top left. Two layers that already exist in the network can also be linked by clicking on the output icon on one layer and the input icon on another. A given layer can have any number of other layers as inputs or outputs. Some layers, such as the **Dense** layer, however, only expect one input and will give an error when multiple inputs are detected.



It is optional, though recommended, that the network be concluded with an **Output** layer. A network may include multiple outputs, in which case all outputs must be given an **Output** layer. If no **Output** layer is included, the last layer in the network will be treated as the sole output.

12.5 Connections Between Layers

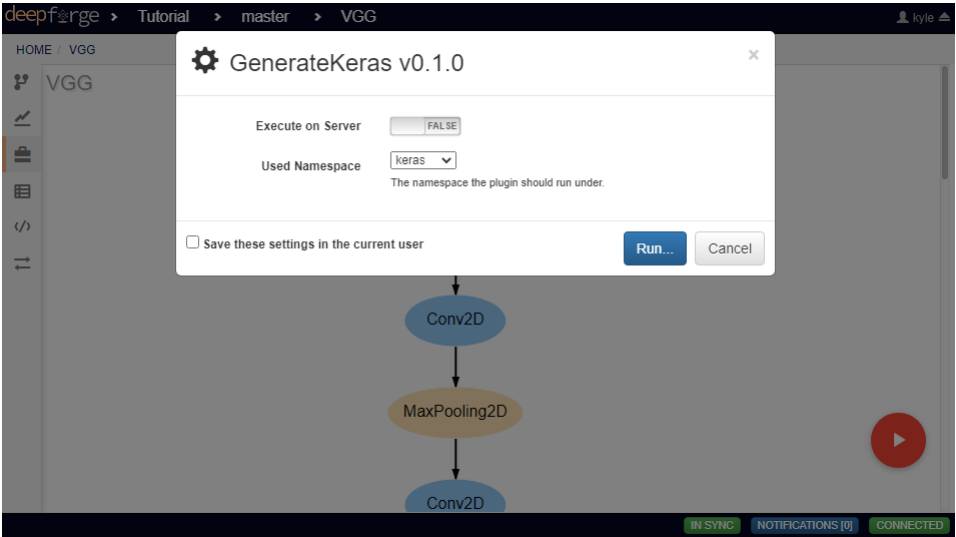
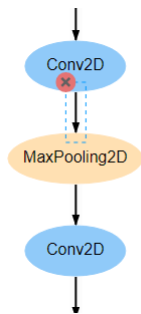
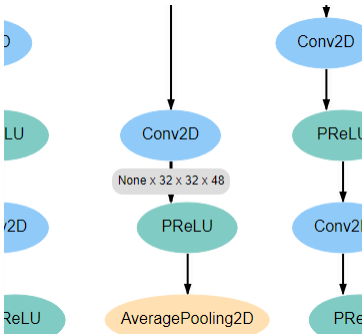
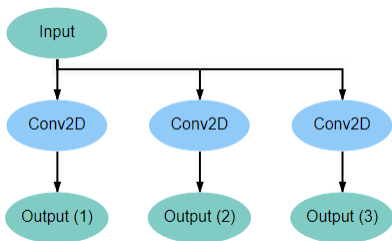
When two layers are connected, they will be joined by a black arrow that indicates the flow of data through the network. Hovering over these arrows will reveal the shape of the data, which can help with analyzing the network to ensure that the data is being transformed as desired.

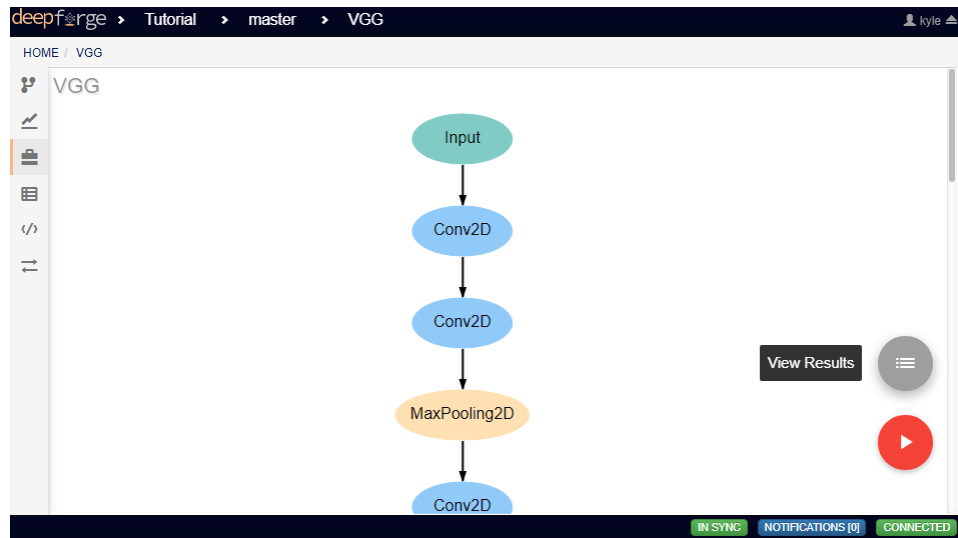
Connections can also be removed and layers separated by clicking on the unwanted arrow and then clicking on the red X icon that appears.

12.6 Exporting Architectures

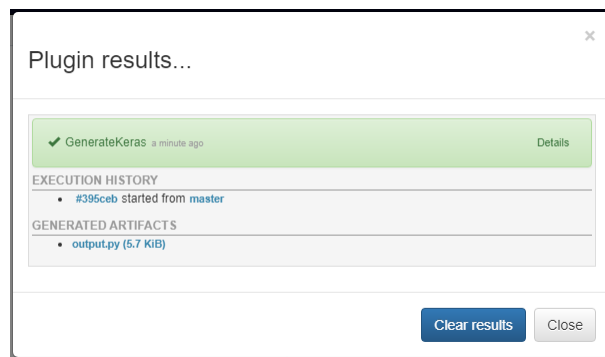
With keras models, another feature exists to export the model as python code. Clicking the red arrow button in the bottom right of the workspace will display a window generating the code. After making any optional changes to the configuration, clicking run will generate the code.

After successful generation, hovering over the red arrow button and clicking on the floating gray list button will provide a list of all exported architectures.





Clicking on *Details* will provide some metadata about the export, as well as a link to download the generated file. This file can then be incorporated into a python project.



Executing Pipelines

This page will guide you through the steps needed to execute a finished pipeline.

13.1 Executing within DeepForge

Finished pipelines can be conveniently executed from within DeepForge. To do so, navigate to the desired pipeline's workspace, hover over the red + button in the bottom right, and click on the blue arrow button. This will open a dialog box for defining how to execute the pipeline. The configuration options are split into several sections. Once all information has been provided, clicking the blue *Run* button will begin execution. The information provided can also be saved for future executions by checking the box in the bottom left.

13.1.1 Basic Options

Here you will define the name of the execution. Execution names are unique identifiers and cannot be repeated. In the case that a name is given that has already been used for that project, an index will be added to the pipeline name automatically (i.e. *test* becomes *test_2*). Upon starting execution, the execution name will also be added to the project version history as a tag.

The pipeline can also be chosen to run in debug mode here. This will allow editing the operations and re-running the pipelines with the edited operations after creation. Alternatively, the execution will only use the version of each operation that existed when the pipeline was first executed. This can be helpful when creating and testing pipelines before deployment.

13.1.2 Credentials for Pipeline Inputs

Pipeline inputs may be stored in different storage backends, such as S3 or SciServer Files, which may in turn require authorization for access. If authorization is required, this section of the dialog will be populated with input fields for the expected credentials. If no input artifacts are used in the pipeline, this section will not be present.

In the figure below, the pipeline inputs are stored using SciServer Files so a dropdown containing the linked SciServer accounts is shown.

Execute Pipeline (v0.1.0)

Basic Options

Execution name

run-cifar

Optional name for this execution instance

Debug Mode

FALSE

Allow for operation editing after creation

Compute Options

Compute

SciServer Compute

Username

kmoore

SciServer username

Password

SciServer password

Compute Domain

Small Jobs Domain

A small job shares resources with up to 4 other jobs and has a max quota for RAM of approx 32GB. A large job runs exclusively and has all CPU cores and RAM available (approx 240GB), however since only one large job will run at a time, there may be a longer wait for the job to start.

Storage Options

Storage

SciServer Files Service

Username

kmoore

SciServer username

Password

☒ Save these settings in the current user

Run...

Cancel

Basic Options

Execution name

run-cifar

Optional name for this execution instance

Debug Mode

FALSE

Allow for operation editing after creation

Credentials for Pipeline Inputs
dataset.pkl (SciServer Files Service):

Username

brillb

SciServer account to use

13.1.3 Compute Options

In this section, you will select from the available compute backends. Each compute backend may require additional information, such as login credentials or computation resources that should be used.

In the figure below, SciServer Compute is selected. SciServer Compute requires the user to select a linked account and a compute domain; this example is using the account, brollb, on the “Small Jobs Domain”.

Compute Options

Compute

Username

SciServer username

Compute Domain

A small job shares resources with up to 4 other jobs and has a max quota for RAM of approx 32GB. A large job runs exclusively and has all CPU cores and RAM available (approx 240GB), however since only one large job will run at a time, there may be a longer wait for the job to start.

13.1.4 Storage Options

Here, the storage backend must be chosen from the available options. Each backend may require additional input, such as login credentials and the desired storage location. This storage backend and location will be where all files created during execution will be stored. This will include both files used during execution, such as data passed between operations, as well as artifacts created from Output operations.

The figure below shows an example using SciServer Files. Like SciServer Compute, the user must first select the linked account to use. Then the user must provide the volume and volume pool to be used. In this example, the “brollb” account is used with the “brollb/deepforge_data” volume on the “Storage” volume pool. More information can be found in the [SciServer User Guide](#).

Storage Options

Storage

Username

SciServer account to use

Volume

Volume to use for upload.

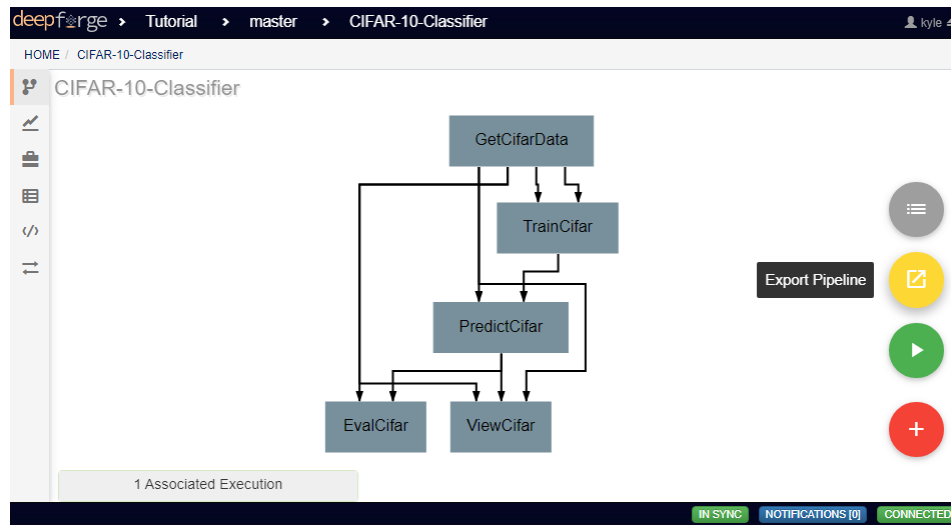
Volume Pool

Folders and files in User Volumes under “Storage” will be backed up and permanent, but there is a quota limit of 10GB. Folders and files in User Volumes under “Temporary” are not backed up, and will be deleted after a particular time period.

13.2 Export Pipeline

If desired, pipelines can be exported as a command line utility for execution locally. Hovering over the red + icon in the pipeline’s workspace and clicking the yellow export button that appears will open a dialog box for exporting the pipeline.

The dialog will prompt the user to select which of the pipeline inputs should be treated as *static artifacts*. A static artifact is an input of the pipeline that should be fixed to the current value. For example, if you have an evaluation pipeline which accepts two inputs, testing data and a model, you may want to set the model as a static artifact to export a pipeline which allows you to easily evaluate the model on other datasets. Alternatively, setting the testing data as a static artifact would allow you to evaluate different models on the same data. Any static artifacts will require the login credentials for the backend and account where the artifact is stored so it can be retrieved and bundled in the exported zip archive.



Clicking the blue *Run* button in the bottom right will generate the execution files for the pipeline and automatically download them in a zip file. In this zip folder are all the files normally generated for execution. The simplest way to execute this pipeline is to run the top-level *main.py* file.

The "Export (v1.0.0)" dialog box is shown. It contains a section for "Static Artifacts" with four items, each with a "TRUE" toggle and a description: "sdss-test-img.pkl", "sdss-test-lab.pkl", "sdss-train-img.pkl", and "sdss-train-lab.pkl". Below this is a section for "Storage Credentials (for static artifacts)" for "sdss-train-img.pkl (SciServer Files Service)". It includes a "Username" field with the value "kmoore" and a "SciServer username" field. At the bottom, there is a checkbox "Save these settings in the current user" which is checked, and two buttons: "Run..." and "Cancel".

Viewing Executions

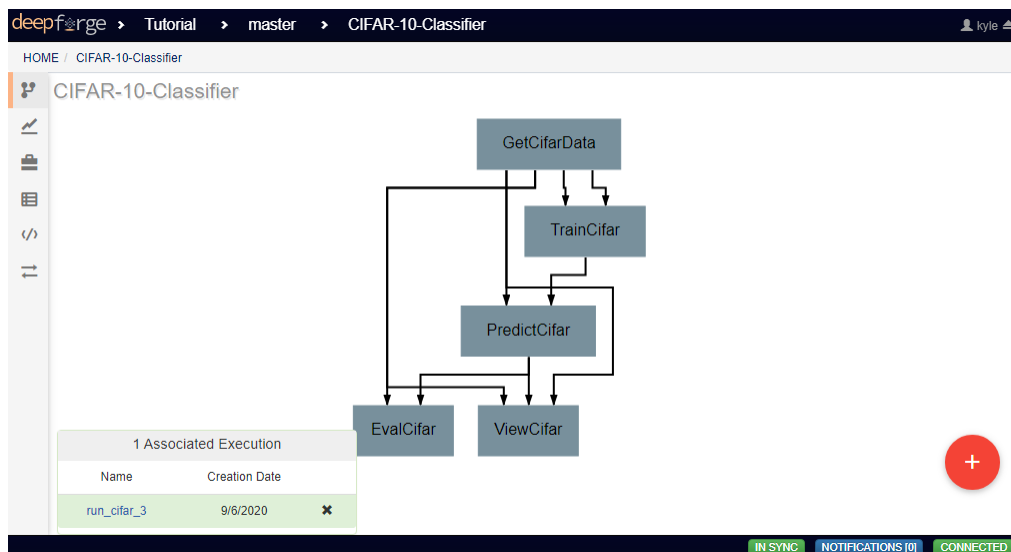
This page will guide you through monitoring the execution of pipelines and viewing the output of finished executions.

14.1 Monitoring Executions

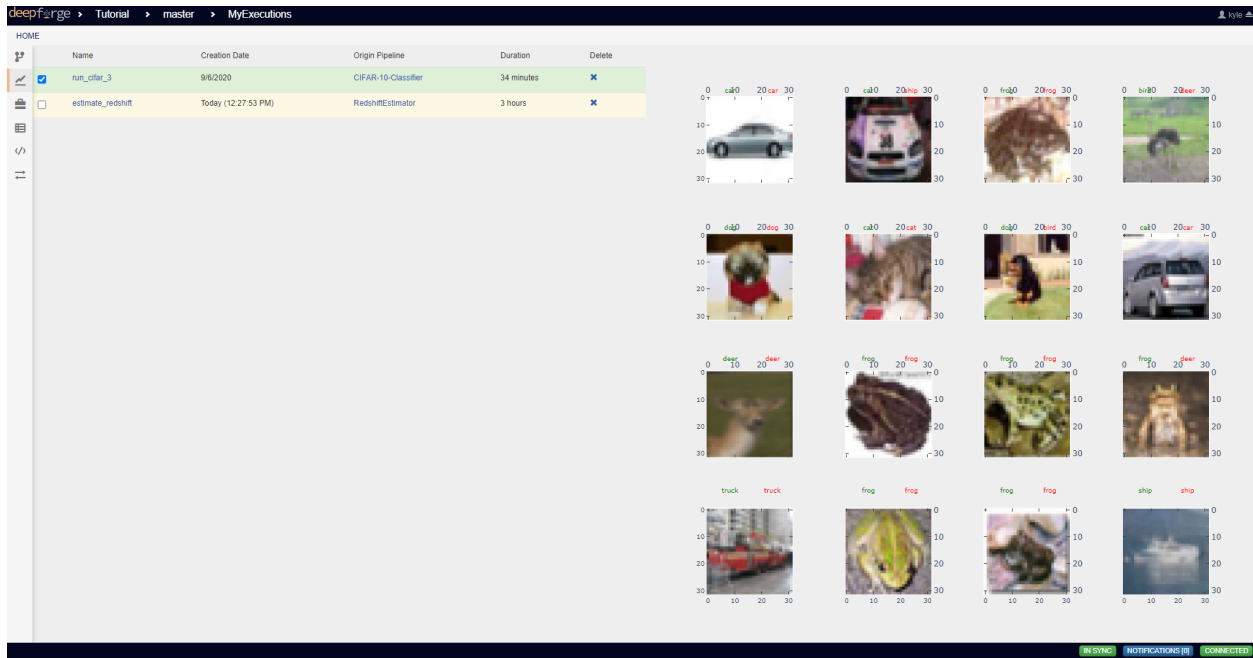
After execution has been started through DeepForge, the status of an execution can be checked using numerous methods.

14.1.1 Viewing Execution Status

While in the workspace for a pipeline, the bottom left corner shows a list of all executions associated with the pipeline. Clicking on the name of an execution will open the status tracker for that execution.

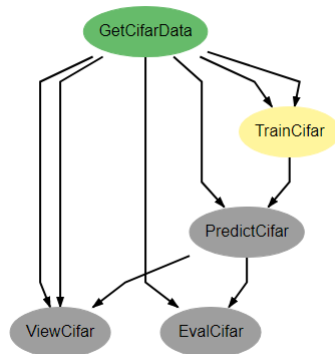


An alternative method of getting to this screen is to go to the *Executions* tab for a list of all executions for the current project. In this view, clicking the name of the desired execution will also open the status tracker.

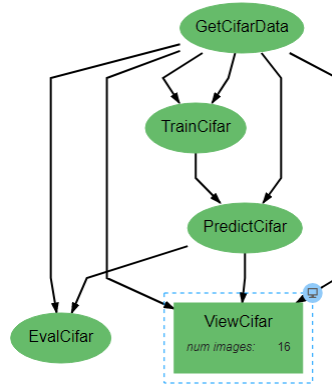


In the status tracker view, the current status for the execution is displayed on an operation level. Each operation is colored based upon its status, with the following possible states:

- Gray - Awaiting Execution
- Yellow - Currently Executing
- Green - Execution Finished Successfully
- Orange - Execution Cancelled
- Red - Error Encountered During Execution

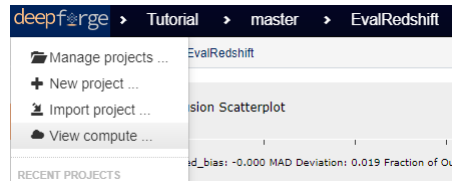


Also in this view, clicking on an operation will reveal the attribute values used for this execution. Clicking the blue monitor icon in the top right of a selected operation will open the console output for that operation.

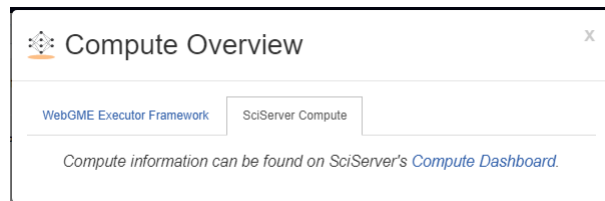


14.1.2 Viewing the Compute Dashboard

In the top left of the webpage, clicking the DeepForge logo will open the DeepForge main dropdown menu. In this menu is an option named *View Compute*.



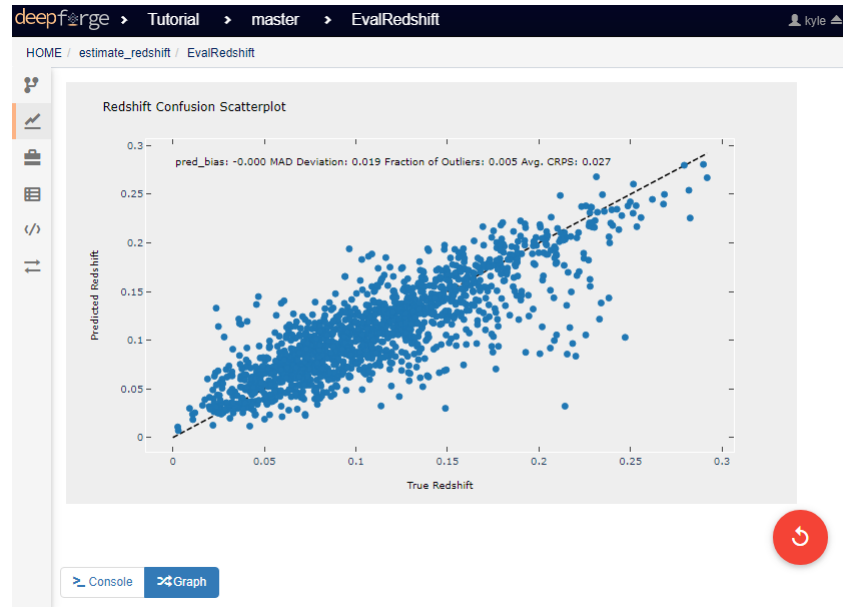
This option opens a dialog box that displays the current execution status of the connected compute backends. Each backend will have its own tab with information specific to that backend. The SciServer Compute tab includes a direct link to SciServer's Compute Dashboard, where the status and output of current and past executions can be viewed.



14.2 Viewing Execution Output

Execution output can be viewed in one of two major ways. Textual output that is printed to the console can be viewed by going to the [execution status tracker](#), selecting the operation that produces the desired output, and clicking on the blue monitor icon in the top right of the operation. For operations generating matplotlib figures, a set of buttons in the bottom left will allow swapping between console and matplotlib figures.

Graphical output, which will generally be generated using a graphical library like [Matplotlib](#), can be viewed from the *Executions* tab on the sidebar. Beside each execution is a checkbox. Activating a checkbox will display the graphical output generated during that execution. Selecting multiple boxes will display the output from all selected executions together.



deepforge > Tutorial > master > MyExecutions

HOME

Name	Creation Date	Origin Pipeline	Duration	Delete
<input checked="" type="checkbox"/> run_cifar_3	9/6/2020	CIFAR-10-Classifer	34 minutes	<input checked="" type="checkbox"/>
<input type="checkbox"/> estimate_redshift	Today (12:27:53 PM)	RedshiftEstimator	3 hours	<input checked="" type="checkbox"/>

truck truck frog frog frog frog ship ship

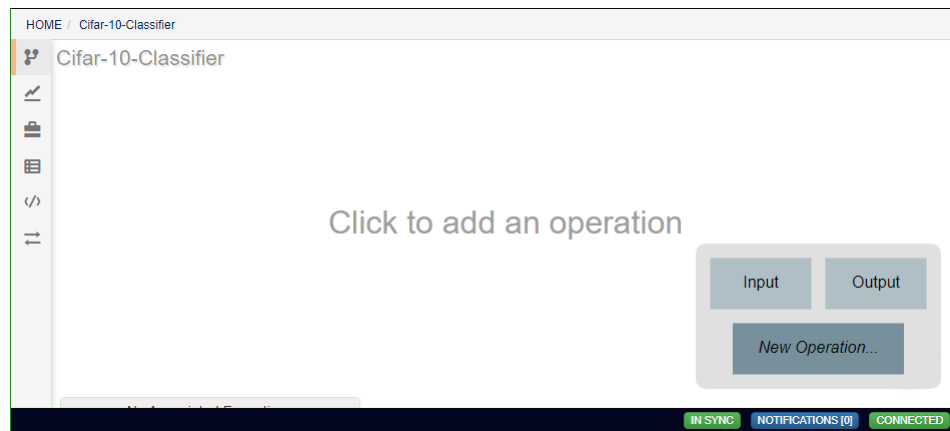
[IN SYNC] [NOTIFICATIONS (0)] [CONNECTED]

CHAPTER 15

CIFAR-10 Classifier

This guide provides step-by-step instructions on how to create full pipeline for training and evaluating a simple image classification neural network. This example uses the [CIFAR-10 dataset](#). This guide assumes that the reader has a basic understanding of the DeepForge interface. New users are recommended to review the [step-by-step guides](#) before attempting the process described in this guide.

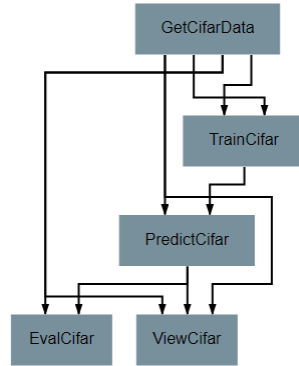
15.1 Pipeline Overview



This guild will give a step-by-step process beginning with a new, blank pipeline (shown above) and ending with the pipeline shown below that will create, train, and evaluate a CIFAR-10 classifier.

15.2 GetCifarData Operation

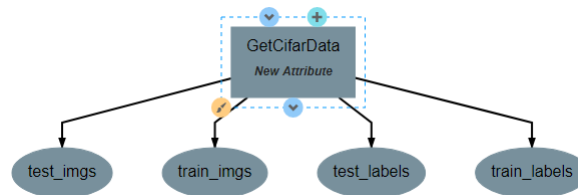
To create your first operation, click on the floating red button in the bottom right of the pipeline editor workspace, and click on the *New Operation* option that appears.



This operation provides the pipeline with the training and testing data that will be used by later operations. In many cases, this will be accomplished with *Input* operations, but it may be preferable in some cases to retrieve the data programmatically.

The first step in any operation should be giving it a name, defining its attributes, and defining its inputs and outputs. These steps are best performed in the right-side panel in the operation editor.

Our GetCifarData operation will produce four outputs, representing the images and labels from the training and testing sets. This operation does not require any inputs or attributes.



The next step in creating any operation is defining its implementation. This is performed in the left panel using the python programming language. Every operation is defined as a python class that must include an *execute* function. Any arbitrary code can be included and run in association with this operation, but an *execute* function must be included and external code will only run if called from within the *execute* function.

CIFAR-10 is a very common benchmarking dataset. As such, the common keras neural network python library provides a simple method for directly downloading and using the data. The code for doing this is relatively straightforward and is shown below.

```

from keras.datasets import cifar10

class GetCifarData():

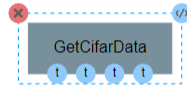
    def execute(self):
        print('Retrieving CIFAR-10 train_imgs')

        # Retrieve CIFAR-10 data. load_data() returns a 2-tuple of 2-tuples. The
        # left hand side decomposes these tuples into four separate variables.
        (train_imgs, train_labels), (test_imgs, test_labels) = cifar10.load_data()

        print('CIFAR-10 train_imgs successfully retrieved')
        print('Training set shape: {shape}'.format(shape=train_imgs.shape))
        print('Testing set shape: {shape}'.format(shape=test_imgs.shape))

        return train_labels, test_imgs, test_labels, train_imgs
  
```

When finished, return to the pipeline and use the add operation button again to add the new operation to the pipeline. At this point, you should have a single operation with four outputs, as shown below:



15.3 TrainCifar Operation

The next operation will create and train the neural network classifier.

Once again, our first step after naming is to define the inputs and outputs of the operation. Unlike the previous operation, two attributes should be added: *batch_size* and *epochs*. Batch size is the number of training samples that the model will be trained on at a time and epochs is the number of times that each training sample will be given to the model. Both are important hyperparameters for a neural network. For this guide, the attributes are defined as shown below, but the exact number used for default values can be changed as desired by the reader.

Edit attribute parameters...

Edit attribute parameters...

Name

batch_size

Description

Enter an optional description..

Type

integer

☐ Read-only
 ☐ Hidden
 ☐ Enumeration

Default value

32

Inclusive Value Range

minimum

maximum

Name

epochs

Description

Enter an optional description..

Type

integer

☐ Read-only
 ☐ Hidden
 ☐ Enumeration

Default value

20

Inclusive Value Range

minimum

maximum

Delete

Save

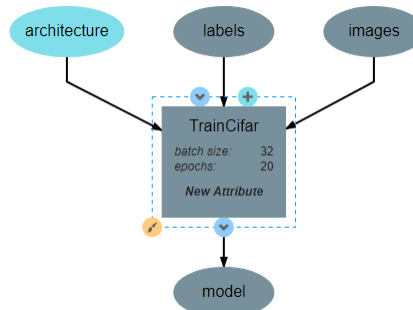
Cancel

Delete

Save

Cancel

This operation will require two inputs (images and labels) and a neural network architecture. Finally, the operation produces one output, which is the trained classifier model. After all inputs, outputs, and attributes have been added, the structure of the operation should appear similar to the following:



The code for this operation follows the standard procedure for creating and training a Keras network. The code for this process is shown below. Note that the attributes must be assigned as class variables in the `__init__` function in order to be used in the `execute` function. Also note that we do not need to import the keras library explicitly here. This is because the architecture object already comes with all the currently needed keras functions attached.

```
class TrainCifar():

    # Runs when preparing the operation for execution
    def __init__(self, architecture, batch_size=32, epochs=20):
        print("Initializing Trainer")

        # Saves attributes as class variables for later use
        self.arch = architecture
        self.epochs = epochs
        self.batch_size = batch_size
        return

    # Runs when the operation is actually executed
    def execute(self, images, labels):
        print("Initializing Model")

        # Creates an instance of the neural network architecure. Other
        # losses and optimizers can be used as desired
        self.arch.compile(loss='sparse_categorical_crossentropy',
                          optimizer='adam',
                          metrics=['sparse_categorical_accuracy'])
        print("Model Initialized Successfully")

        print("Beginning Training")
        print("Training images shape:", images.shape)
        print("Training labels shape:", labels.shape)

        # Train the model on the given inputs (images) and outputs (labels)
        # using the specified training options.
        self.arch.fit(images,
                      labels,
                      batch_size=self.batch_size,
                      epochs=self.epochs,
                      verbose=2)

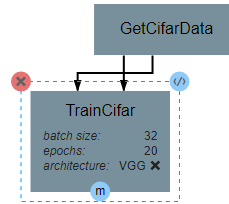
        print("Training Complete")

        # Saves the model in a new variable. This is necessary so that the
        # output of the operation is named 'model'
        model = self.arch

        return model
```

After the operation is fully defined, it needs to be added to the workspace and connected to the **GetCifarData** operation as shown below. Specifically, the *train_images* and *train_labels* outputs from **GetCifarData** should be connected to the *images* and *labels* inputs to **TrainCifar** respectively. Hovering over the circles representing each input or output will display the full name of that element. This should help to ensure that the correct inputs and outputs are matched together.

Note that the architecture selected from within the pipeline editor until after the *Neural Network Architecture* section of this guide is completed.



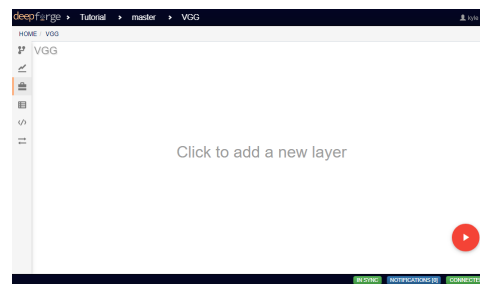
15.4 Neural Network Architecture

This section will describe how to create a simple, but effective, Convolutional Neural Network for classifying CIFAR-10 images. In particular, this section gives instructions on creating a slightly simplified [VGG network](#). The basic structure of this network is a series of four feature detection blocks, followed by a densely connected classifier block.

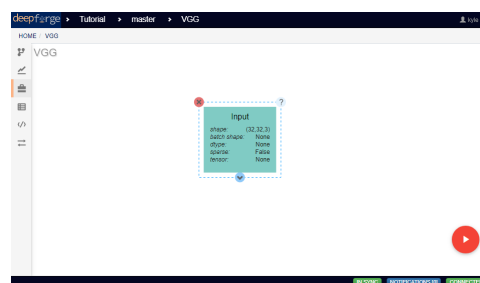
For specifics on how to create a new network how to use the neural network editor interface, consult the [Creating Neural Networks](#) walkthrough.

Beginning from a blank network, the first step when building a network is to create an Input layer by clicking anywhere on the workspace.

For reference during design, the full architecture can be found [here](#).



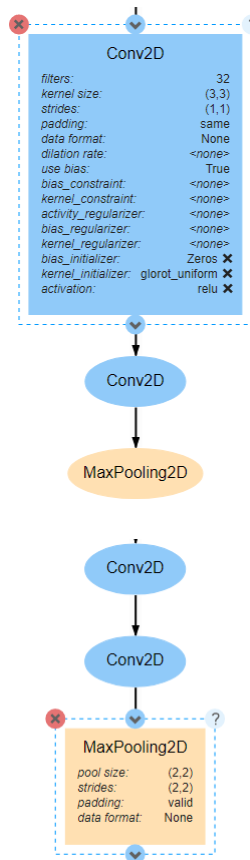
This Input layer requires that either the *shape* or *batch_shape* attributes be defined. Because our data is composed of 32*32 pixel RGB images, the *shape* of our input should be (32,32,3).



The four feature detector blocks are each composed of two **Conv2D** layers followed by a **MaxPooling2D** layer. The settings for the first **Conv2D** and **MaxPooling2D** layers are shown below.

Every **Conv2D** layer requires that the *filters* and *kernel_size* attributes be defined. Each **Conv2D** layer in this network will use a *kernel_size* (window size) of (3,3), a stride of (1,1), and will use ReLU as the activation function. They should all also use *same* as the padding so that the size of the input does not change during convolution. For the first pair of **Conv2D** layers, the number of filters will be 32.

Every **MaxPooling2D** layer requires that the *pool_size* (window size) attribute be defined. In this network, all **MaxPooling2D** layers will use a *pool_size* of (2,2), a stride of (2,2), and padding set to *valid*. These settings will result in the size of the image being cut in half at every pooling.



A total of four of these convolutional blocks should be created in sequence. The only difference between each block is that the number of filters used in the **Conv2D** layers in each block should double after each pooling. In other words, the value of *filters* should be 32 for the first **Conv2D** layer, 64 for the third **Conv2D** layer, 128 for the fifth, and so on.

After the last convolutional block comes the classifier block. The first layer in this block is a **Flatten** layer, which converts the convolved image into a 1D vector that can be fed into the following **Dense** layers. The **Flatten** layer has no attributes to change.

There are a total of three **Dense** layers in this classifier, with the first two using the same attribute values. Every **Dense** layer requires that the *units* (output length) attribute be defined.

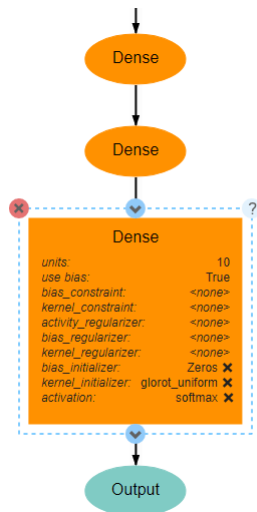
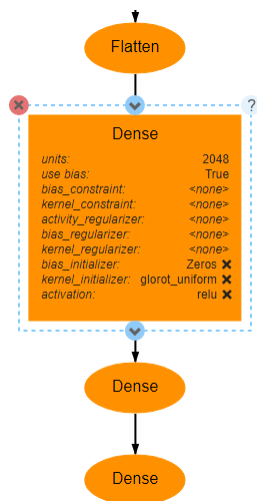
For the first two **Dense** layers, the number of units used will be 2048, and the activation function used will be ReLU, as shown below.

The final **Dense** layer will actually provide the output probability density function for the model. As such, the number of units should be the number of categories in the data (in this case 10). This last layer also uses the *softmax* activation function, which ensures that the output is a vector whose sum is 1.

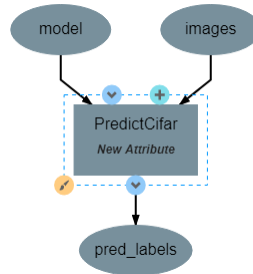
Optionally, an **Output** layer may be added after the final **Dense** layer. This layer explicitly marks the output of a model, but may be excluded when there is only one output. When there is only one output, such as in this network, the lowest layer in the model will be assumed to be the output layer.

15.5 PredictCifar Operation

This operation uses the model created by **TrainCifar** to predict the class of a set on input images. This operation has no attributes, takes a model and images as input and produces a set of predicted labels (named *pred_labels*), resulting



in the following structure:



The code for this operation is short and straightforward with only one peculiarity. The *predict* function does not provide a prediction directly, instead providing a **probability density function (pdf)** over the available classes. For example, a CIFAR-10 classifier's output for a single input may be [0, 0.03, 0.9, 0.02, 0, 0, 0.05, 0, 0, 0], which indicates that the model is predicting that the likelihood that the image falls into each category is 0% for category 1, 3% for category 2, 90% for category 3, and so on. This requires taking the argmax of every output of the model to determine which class has been ruled the most likely.

```
import numpy as np

class PredictCifar():

    def execute(self, images, model):
        print('Predicting Image Categories')

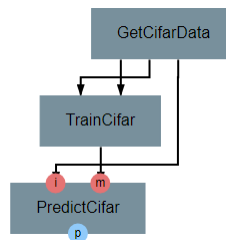
        # Predicts the PDF for the input images
        pred_labels = model.predict(images)

        # Converts PDFs into scalar predictions
        pred_labels = np.argmax(pred_labels, axis=1)

        print('Predictions Generated')

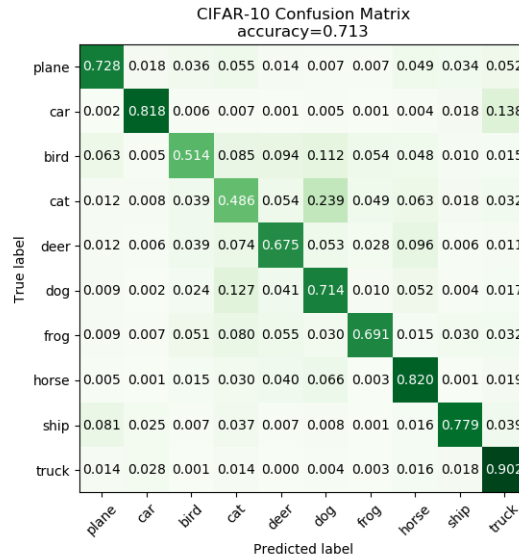
        return pred_labels
```

After the operation is fully defined, it needs to be added to the workspace and connected to the previous operations as shown below. Specifically, the *test_images* outputs from **GetCifarData** and the *model* output from **TrainCifar** should be connected to the *images* and *model* inputs to **PredictCifar** respectively.

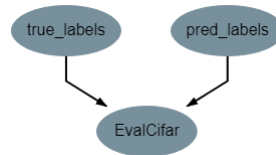


15.6 EvalCifar Operation

This operation evaluates the outputs from the classifier and produces a confusion matrix that could be helpful for determining where the shortcomings of the model lie.



This operation requires no attributes and produces no output variables. It requires two inputs in the form of *true_labels* and *pred_labels*. The structure of this operation is shown below:



With this operation, the code becomes a bit more complex as we build the visualization with the tools provided by the `matplotlib.pyplot` library. The code below is annotated with comments describing the purpose of all graphing commands. Also of note is that the expected input *true_labels* is a 2-dimensional array, where the second dimension is of length 1. This is because of a quirk of keras that requires this structure for training and automatic evaluation. To ease calculations, the first step taken is to flatten this array to one dimension.

```

import matplotlib.pyplot as plt
import numpy as np

class EvalCifar():

    def execute(self, pred_labels, true_labels):

        # Reduces the dimensionality of true_labels by 1
        # ex. [[1],[4],[5],[2]] becomes [1, 4, 5, 2]
        true_labels = true_labels[:,0]

        # Builds a confusion matrix from the lists of labels
        cm = self.buildConfustionMatrix(pred_labels, true_labels)

        #normalize values to range [0,1]
        cm = cm / cm.sum(axis=1)

        # Calculates the overall accuracy of the model
        # acc = (# correct) / (# samples)

```

(continues on next page)

(continued from previous page)

```

acc = np.trace(cm) / np.sum(cm)

# Display the confusion matrix as a grayscale image, mapping the
# intensities to a green colorscale rather than the default gray
plt.imshow(cm, cmap=plt.get_cmap('Greens'))

# Adds a title to the image. Also reports accuracy below the title
plt.title('CIFAR-10 Confusion Matrix\naccuracy={:0.3f}'.format(acc))

# Labels the ticks on the two axes (placed at positions [0,1,2,...,9]) with
# the category names
bins = np.arange(10)
catName = ['plane', 'car', 'bird',
           'cat', 'deer', 'dog', 'frog',
           'horse', 'ship', 'truck']
plt.xticks(bins, catName, rotation=45)
plt.yticks(bins, catName)

# Determines value at the center of the color scale
mid = (cm.max() + cm.min()) / 2

for i in range(10):
    for j in range(10):
        # Prints the value of each cell to three decimal places.
        # Colors text so that white text is printed on dark cells
        # and black text on light cells
        plt.text(j, i, '{:0.3f}'.format(cm[i, j]),
                 ha='center', va='center',
                 color='white' if cm[i, j] > mid else 'black')

# Labels the two axes
plt.ylabel('True label')
plt.xlabel('Predicted label')

plt.tight_layout()

# Displays the plot
plt.show()

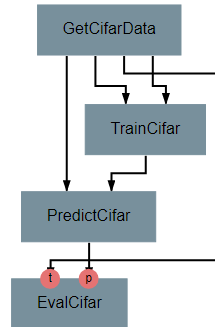
def buildConfustionMatrix(self, pred_labels, true_labels):
    # Creates an empty matrix of size 10 x 10
    mat = np.zeros((10,10))

    # Computes count of times that image with true label t is
    # assigned predicted label p
    for p, t in zip(pred_labels, true_labels):
        mat[t][p] += 1

    return mat

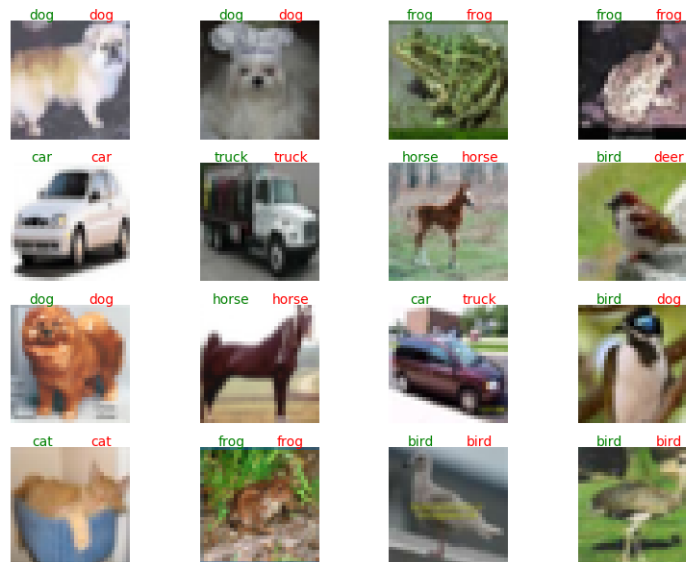
```

After the operation is fully defined, it needs to be added to the workspace and connected to the previous operations as shown below. Specifically, the `test_labels` outputs from **GetCifarData** and the `pred_labels` output from **PredictCifar** should be connected to the `true_labels` and `pred_labels` inputs to **EvalCifar** respectively.



15.7 ViewCifar Operation

This operation displays a random subset of images, along with the predicted and actual categories in which those images belong. Such a visualization might be helpful for seeing what kind of images are being misclassified and for what reason.



This operation includes an attribute `num_images` for specifying the number of images that should be drawn from the testing set and displayed. As with the attributes in `TrainCifar`, this attribute should be given a type of integer and will be given the default value of 16.

This operation produces no outputs and requires three inputs: the images, the associated true labels, and the associated predicted labels. The overall structure is shown.

As with the previous operation, the code for this operation gets slightly complicated and has been annotated with comments describing each command.

```

from matplotlib import pyplot as plt
import numpy as np
import math

```

(continues on next page)

Edit attribute parameters...

✕

Name

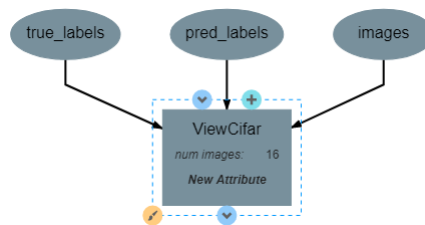
Description

Type

☐ Read-only ☐ Hidden ☐ Enumeration

Default value

Inclusive Value Range



(continued from previous page)

```
class ViewCifar():
    def __init__(self, num_images=16):
        self.num_images = num_images

    return

    def execute(self, pred_labels, true_labels, images):
        # Reduces the dimensionality of true_labels by 1
        # ex. [[1],[4],[5],[2]] becomes [1, 4, 5, 2]
        true_labels = true_labels[:,0]

        # Chooses a random selection of indices representing the chosen images
        orig_indices = np.arange(len(images))
        indices = np.random.choice(orig_indices, self.num_images, replace=False)

        # Extracts the images and labels represented by the chosen indices
        images = np.take(images, indices, axis=0)
        pred_labels = np.take(pred_labels, indices, axis=0)
        true_labels = np.take(true_labels, indices, axis=0)

        # Calculates the number of rows and columns needed to arrange the images in
        # as square of a shape as possible
        num_cols = math.ceil(math.sqrt(self.num_images))
        num_rows = math.ceil(self.num_images / num_cols)

        # Creates a collection of subplots, with one cell per image
        fig, splts = plt.subplots(num_rows, num_cols, sharex=True, sharey=True)

        catName = ['plane', 'car', 'bird',
```

(continues on next page)

(continued from previous page)

```

        'cat','deer','dog','frog',
        'horse','ship','truck']

    for i in range(self.num_images):

        # Determines the current row and column location
        col = i % num_cols
        row = i // num_cols

        # Displays the current image
        splts[row,col].imshow(images[i])
        splts[row,col].axis('off')

        # Retrieves the text label equivalent of the numerical labels
        p_cat = catName[pred_labels[i]]
        t_cat = catName[true_labels[i]]

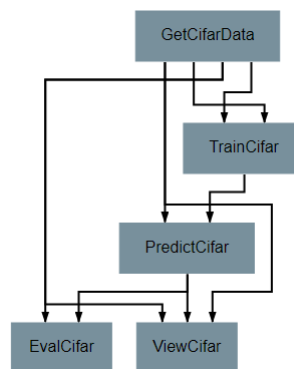
        # Displays the category labels, with the true label colored green and in
        # the top-left corner and the predicted label colored red and in the
        # top-right corner
        splts[row,col].text(8,0,t_cat,ha='center',va='bottom',color='green')
        splts[row,col].text(24,0,p_cat,ha='center',va='bottom',color='red')

    # Displays the figure
    plt.show()

```

After the operation is fully defined, it needs to be added to the workspace and connected to the previous operations as shown below. Specifically, the `test_labels` outputs from **GetCifarData**, the `test_images` from **GetCifarData**, and the `pred_labels` output from **PredictCifar** should be connected to the `true_labels`, `images`, and `pred_labels` inputs to **ViewCifar** respectively.

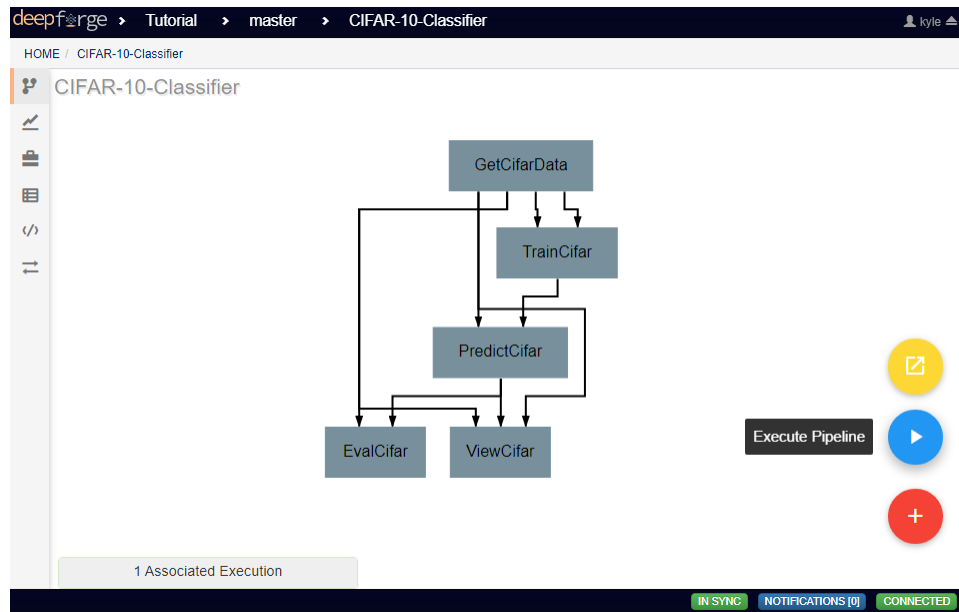
With this, we have a full pipeline ready for execution.



15.8 Execution and Results

With the pipeline fully prepared, it is time to execute the pipeline. To do this, go to the pipeline editor workspace, hover over the red *Add Operation* button and click the floating blue *Execute Pipeline* button

A dialog box will open where the settings for the current execution must be defined. All inputs are required are detailed below.



Execute Pipeline (v0.1.0)

Basic Options

Execution name:
Optional name for this execution instance

Debug Mode: ☐
Allow for operation editing after creation

Compute Options

Compute:

Username:
SciServer username

Password:
SciServer password

Compute Domain:
A small job shares resources with up to 4 other jobs and has a max quota for RAM of approx 32GB. A large job runs exclusively and has all CPU cores and RAM available (approx 240GB), however since only one large job will run at a time, there may be a longer wait for the job to start.

Storage Options

Storage:

Username:
SciServer username

Password:

☒ Save these settings in the current user

The *Basic Options* section includes two settings. The first is the name to be used for identifying the execution. An execution's name must be unique within the project and if a name is given here that has already been used for an execution in the same project, a number will be appended to the given name automatically. The debug option allows for individual operations to be edited and rerun after execution. This is useful during pipeline development and allows for easier debugging or tuning.

Basic Options

Execution name
Optional name for this execution instance

Debug Mode ☐
Allow for operation editing after creation

The *Compute Options* section allows configuration of the compute backend to be used for execution. The specific inputs required here will vary with the selected compute backend. For instance, the [SciServer Compute](#) backend requires login credentials and the selection of a compute domain.

Compute Options

Compute

Username
SciServer username

Password
SciServer password

Compute Domain
A small job shares resources with up to 4 other jobs and has a max quota for RAM of approx 32GB. A large job runs exclusively and has all CPU cores and RAM available (approx 240GB), however since only one large job will run at a time, there may be a longer wait for the job to start.

The *Storage Options* section allows configuration of the storage backend to be used during execution. This backend will be where all files used during execution and created as output from the pipeline will be stored. The specific inputs required here will vary with the selected compute backend. For instance, the **SciServer Files Service** backend requires login credentials, the selection of a storage volume, and the type of the volume.

Storage Options

Storage

Username
SciServer username

Password
SciServer password

Volume
Volume to use for upload.

Volume Pool
Folders and files in User Volumes under "Storage" will be backed up and permanent, but there is a quota limit of 10GB. Folders and files in User Volumes under "Temporary" are not backed up, and will be deleted after a particular time period.

When all settings have been specified, click **Run** to begin execution. For information on how to check execution status, consult the [Viewing Executions](#) walkthrough.

To view the output of the execution, go to the *Executions* tab and check the box next to the desired execution.

For a more detailed and larger view of individual figures, click on the name of the execution to view its status page and open the console output for the desired operation. In the bottom left is a set of buttons for switching between console output and graph output for that operation.

deepforge > Tutorial > master > MyExecutions

HOME

Name	Creation Date	Origin Pipeline	Duration	Delete
<input checked="" type="checkbox"/> run_cifar_3	9/6/2020	CIFAR-10-Classfier	34 minutes	
<input type="checkbox"/> estimate_redshift	Today (12:27:53 PM)	RedshiftEstimator	3 hours	

IN SYNC NOTIFICATIONS ON CONNECTED

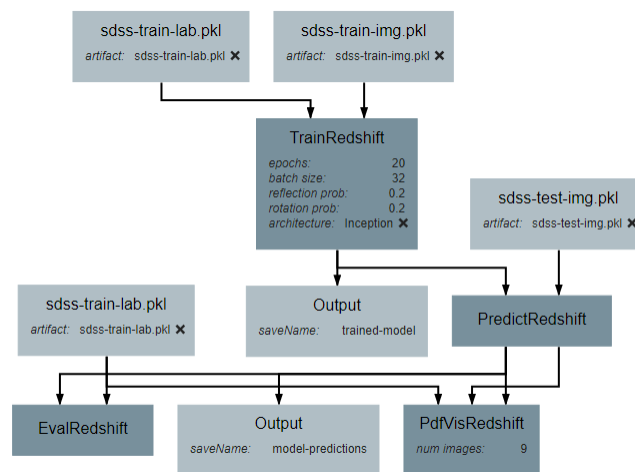


Redshift Estimator

This guide provides instructions on how to create a full pipeline for training and evaluating a convolutional neural network on the task of predicting astronomical redshift values given images of galaxies. It provides an approach that is simplified from work by [Pasquet et. al.](#) The data referenced and used in this guide was obtained from the [Sloan Digital Sky Survey Data Release 3](#), obtained via [SciServer's CasJobs Service](#), and processed using [Astromatic's SWarp tool](#). This guide assumes that the reader has a basic understanding of the DeepForge interface and how to create basic pipelines. New users are recommended to review the [step-by-step guides](#) before attempting the process described in this guide.

16.1 Pipeline Overview

This guild will give instruction on creating a pipeline that will create, train, and evaluate a model that estimates photometric redshift. Each of the sections below details how to create a piece of the final pipeline and how each piece is connected.



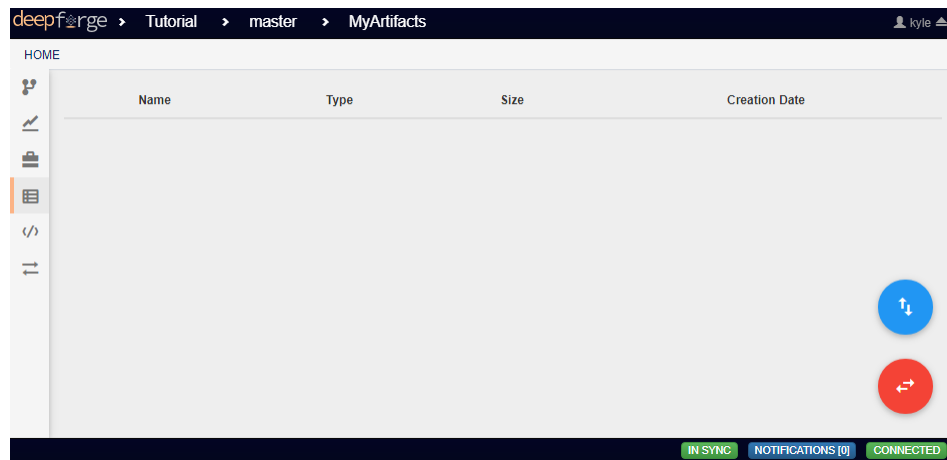
16.2 Input Operations

While it is possible to retrieve the data needed for model creation programmatically in many cases, this guide makes use of the **Input** operations to load preprocessed data. This is in the interest of both simplicity and generalizability to datasets and data sources different than those used in this tutorial.

This pipeline uses four **Input** operations. These operations provide the training input images, training output values, testing input images, and testing output values. For the purposes of this tutorial, the structure of the input images is a 4D numpy array of shape (N, 64, 64, 5), where N is the number of images. The outputs are 1D numpy arrays of length N. The process described in this tutorial will work for images that are not 64*64 pixels in size and that use any number of color channels, requiring only a slight change in the neural network.



Each **Input** operation requires an artifact to have been added to the project. To do this, go to the artifacts view and click either of the two floating buttons in the bottom right of the workspace (one button only appears on hover).



With these buttons, you can either upload a local file to one of the storage backends or import a file that already exists within a storage backend.

By default, all artifacts are treated as python [pickle objects](#). Using other forms of serialized data, such as [FITS](#) or [numpy](#) files, requires defining a custom serializer in the *Custom Serialization* view, which is not covered in this tutorial.

16.3 TrainRedshift Operation

The first custom operation will create and train the neural network classifier.

Two attributes should be added: *batch_size* and *epochs*. Batch size is the number of training samples that the model will be trained on at a time and epochs is the number of times that each training sample will be given to the model. Both are important hyperparameters for training a neural network. For this guide, the attributes are defined as shown below, but the exact number used for default values can be changed as desired by the reader.

This operation will require two inputs (images and labels) and a neural network architecture. Finally, the operation produces one output, which is the trained classifier model. After all inputs, outputs, and attributes have been added, the structure of the operation should appear similar to the following:

Upload Artifact (v0.3.0)

New Artifact

Data to upload

sdss-test-lab.pkl

Storage

SciServer Files Service

Username

kmoore

SciServer username

Password

SciServer password

Volume

kmoore/sdss-data

Volume to use for upload.

Volume Pool

Storage

Folders and files in User Volumes under "Storage" will be backed up and permanent, but there is a quota limit of 10GB. Folders and files in User Volumes under "Temporary" are not backed up, and will be deleted after a particular time period.

Data Type

pickle

Import Existing Data (v0.1.0)

Storage

SciServer Files Service

Username

kmoore

SciServer username

Password

SciServer password

Volume

kmoore/sdss-data

Volume to use for upload.

Volume Pool

Storage

Folders and files in User Volumes under "Storage" will be backed up and permanent, but there is a quota limit of 10GB. Folders and files in User Volumes under "Temporary" are not backed up, and will be deleted after a particular time period.

File Path

sdss-train-lab.pkl

Path to an existing artifact in the storage backend

Data Type

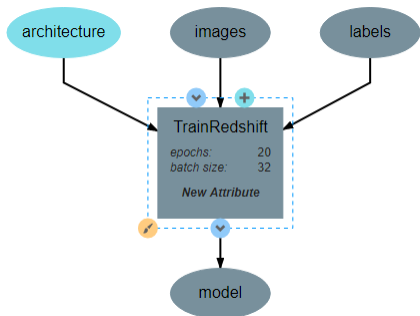
pickle

☒ Save these settings in the current user

Run... Cancel

☒ Save these settings in the current user

Run... Cancel



The code for this operation follows the standard procedure for creating and training a Keras network with one minor caveat. The method used by Pasquet et al. on which this pipeline is based formulates redshift prediction as a classification problem. Because the labels used in this tutorial are floating point values, they must be converted into a categorical format. This is the purpose of the `to_categorical` function. The code for this operation is shown below.

```
import numpy as np

class TrainRedshift():
    def __init__(self, architecture,
                 epochs=20,
                 batch_size=32):
        self.arch = architecture
        self.epochs = epochs
        self.batch_size = batch_size

        # Maximum expected redshift value and number of bins to be used in
        # classification
        # step. The max_val will need to change to be reasonably close to the maximum
        # redshift of your dataset. The number of bins must match the output shape of
        # the
        # architecture but may be tuned as a hyperparameter. Both can optionally be
        # made
        # attributes of the operation.
        self.max_val = 0.4
        self.num_bins = 180
        return

    def execute(self, images, labels):
        print(type(labels))
        print("Initializing Model")

        # Initialize the model
        self.arch.compile(loss='sparse_categorical_crossentropy',
                          optimizer='adam',
                          metrics=['sparse_categorical_accuracy'])
        print("Model Initialized Successfully")

        print("Beginning Training")
        print("Training images shape:", images.shape)
        print("Training labels shape:", labels.shape)

        # Train the model on the images and the labels. Labels are converted to
        # categorical
        # data because the architecture expects an index to an output vector of
        # length 180
        self.arch.fit(images,
                      self.to_categorical(labels),
                      epochs=self.epochs,
                      verbose=2)

        print("Training Complete")

        # Saves the model in a new variable. This is necessary so that the
        # output of the operation is named 'model'
        model = self.arch
        return model
```

(continues on next page)

(continued from previous page)

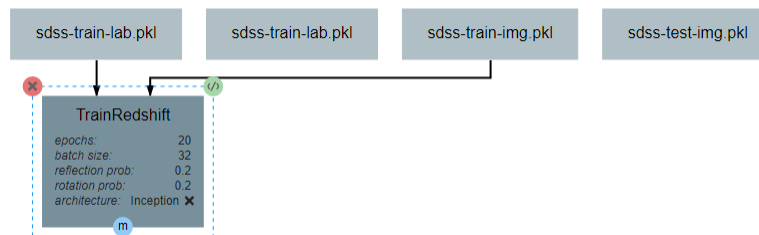
```

# Converts floating point labels to categorical vectors. The result for a given_
↪ input
# label is a 1D vector of length 1 whose value is the index representing the_
↪ range in
# which the label falls. For example, if the max_val is 0.4 and the num_bins is 4,
↪ the
# possible indices are 0-3, representing the ranges [0,0.1), [0.1,0.2), [0.2,0.3),
↪ and
# [0.3,0.4] respectively. So, a label of 0.12 results in an output of [1]
def to_categorical(self, labels):
    return np.array(labels) // (self.max_val / self.num_bins)

```

After the operation is fully defined, it needs to be added to the workspace and connected to the **Input** operations as shown below. Specifically, the training images and training outputs should be connected to the *images* and *labels* inputs of **TrainRedshift** respectively.

Note that the architecture selected from within the pipeline editor until after the *Neural Network Architecture* section of this guide is completed.



16.4 Neural Network Architecture

This section will describe how to create a convolutional neural network for estimating redshift from images. In particular, this section gives instructions on creating an **Inception-v1 network**. The basic structure of this network is an input block, a series of five inception blocks, followed by a densely connected classifier block. These blocks are each described in order below.

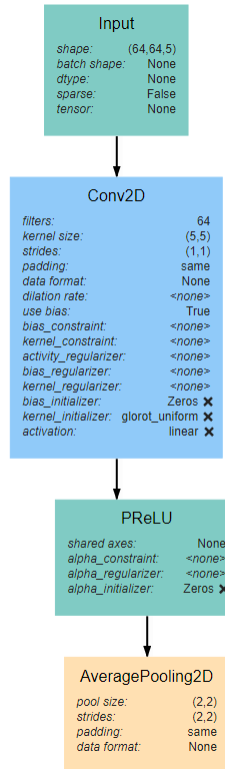
For reference during design, the full architecture can be found [here](#).

16.4.1 Input Block

The input block begins, as with all network architectures, with an **Input** layer. The shape of this layer should be the shape of the input images (64*64*3 in this case). This input feeds into a 5*5 **Conv2D** layer with 64 filters and linear activation. The activation here is linear because the layer is to be activated by the **PReLU** layer that follows. The Input block is finished with an **AveragePooling2D** layer with a window size and stride of 2. Note that all layers use *same* padding to prevent changes in data shape due to the window size.

16.4.2 Inception Blocks

The five inception blocks fall into one of three designs. Blocks 1 and 3 share the same design, as do blocks 2 and 4. Each of the three designs are described more detail below. Take note throughout these subsections that every **Conv2D** layer is followed by a **PReLU** layer using the default attribute values. In addition, all **AveragePooling2D** layers will



use have the attribute values of (2,2) for both *pool_size* and *strides* and *same* for *padding*. In the interest of brevity, this will not be pointed out in each subsection.

Inception Blocks 1 and 3

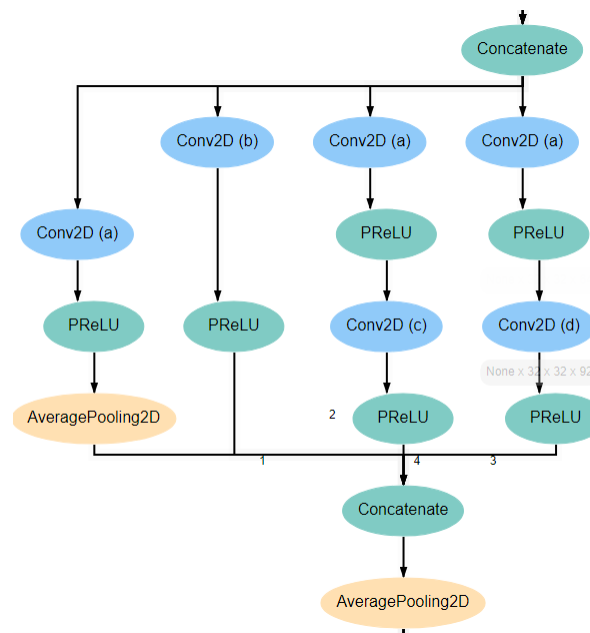
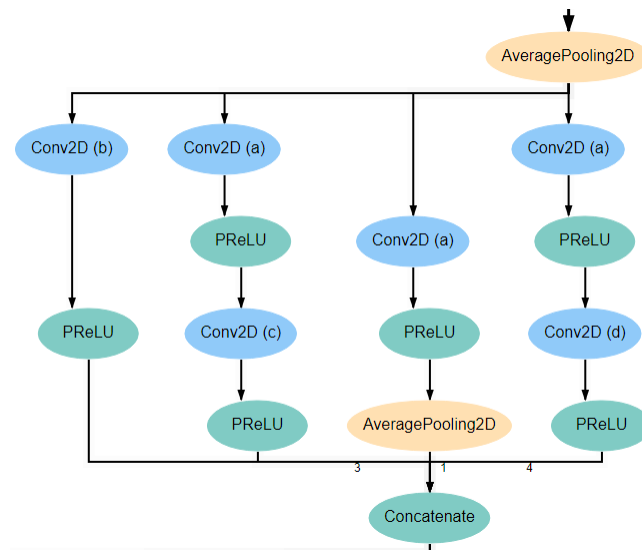
Blocks 1 and 3 each begins with an **AveragePooling2D** layer. This is the same layer pictured at the bottom of the input block and blocks 2 and 4. The output of this layer is fed into 4 separate **Conv2D** layers that all have a *kernel_size* of 1*1. Two of these new layers feed into another **Conv2D** layer, one with *kernel_size* 3*3 and another with *kernel_size* 5*5. Another of the original **Conv2D** layers feeds into an **AveragePooling2D** layer. Finally, the remaining original **Conv2D** layer, along with the **AveragePooling2D** layer and the two new **Conv2D** layers all feed into a **Concatenate** layer. For reference, the expected structure is shown below.

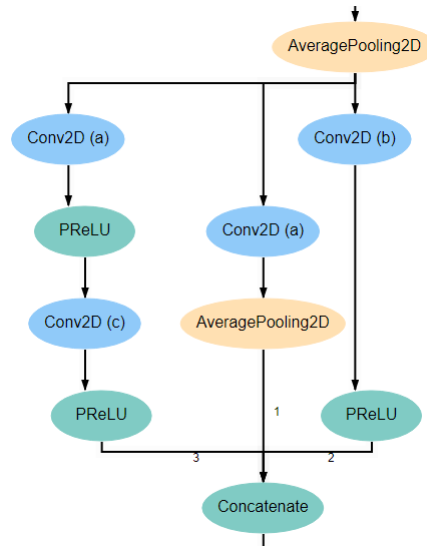
Inception Blocks 2 and 4

Blocks 2 and 4 are laid out mostly identically to blocks 1 and 3, with the exception of the first and last layers. The first layer in these blocks is the **Concatenate** layer from the end of the previous block. In addition, another **AveragePooling2D** layer is added after the **Concatenate** layer at the end of the block. For reference, the expected structure is shown below.

Inception Block 5

Block 5 is laid out mostly identically to blocks 1 and 3. The only difference is that one of the two branches with two **Conv2D** layers is omitted. Specifically, the branch in which the second layer has a *kernel_size* of 5*5 is left out. For reference, the expected structure is shown below.





Conv2D Attributes

All **Conv2D** layers in the architecture use a stride of 1, use *same* padding, and use a *linear* activation function. The only attributes that vary between the various layers are the number of *filters* and the *kernel_size*. Notice in the diagrams above that every **Conv2D** layer is marked with an identifying letter. The table below gives the correct values for *filters* and *kernel_size* for every layer in each inception block.

	Block 1		Block 2		Block 3		Block 4		Block 5	
Con2D layer	filters	kernel	filters	kernel	filters	kernel	filters	kernel	filters	kernel
a	48	(1,1)	64	(1,1)	92	(1,1)	92	(1,1)	92	(1,1)
b	64	(1,1)	92	(1,1)	128	(1,1)	128	(1,1)	128	(1,1)
c		(3,3)		(3,3)		(3,3)		(3,3)		(3,3)
d		(5,5)		(5,5)		(5,5)		(5,5)		

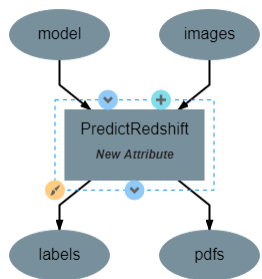
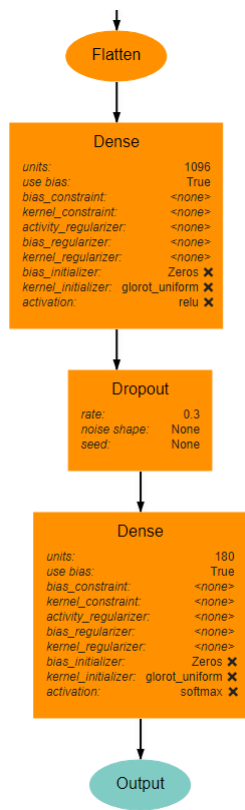
16.4.3 Classifier Block

The classifier block begins with a **Flatten** layer to reshape the data into a 1D vector. This feeds into a **Dense** layer with 1096 units and ReLU activation. The next layer is a **Dropout** layer intended to help prevent overfitting. The dropout rate used here is 0.3, but this may require tuning to fit the dataset most appropriately. . Finally, a **Dense** layer using softmax activation produces the final output. This final layer must use the value for *units* as the *num_bins* variable used in various operations. An optional **Output** layer may also be included but is unnecessary as long as the **Dense** layer is the lowest layer in the architecture.

16.5 PredictRedshift Operation

This operation uses the model created by **TrainRedshift** to predict the values of a set on input images. This operation has no attributes, takes a model and a set of images as input and produces a set of predicted values (named *labels*) and the associates probability density functions that resulted in those values (named *pdfs*). The structure of the operation is as shown below:

The *model.predict* function results in a probability density function (PDF) over all redshift values in the allowed range [0,0.4]. In order to get scalar values for predictions, a weighted average is taken for each PDF where the value being



averaged is the redshift value represented by that bin and the weight is the PDF value at that bin (i.e. how likely it is that the value represented by that bin is the actual redshift value).

```
import numpy as np

class PredictRedshift():

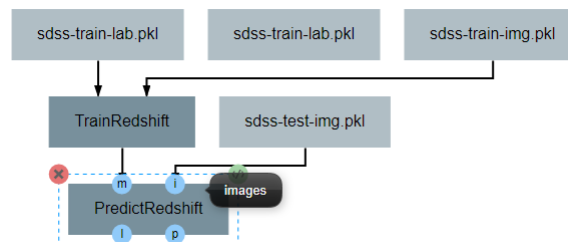
    def execute(self, images, model):
        # See first comment in PredictRedshift()
        max_val = 0.4
        num_bins = 180
        step = max_val / num_bins

        # Generates PDF for the redshift of each image
        pdfs = model.predict(images)
        bin_starts = np.arange(0, max_val, step)

        # Regresses prediction to a scalar value. Essentially a weighted average
        # where the weights are the pdf values for each bin and the values are
        # the beginning of the range represented by each bin.
        labels = np.sum((bin_starts + (step / 2)) * pdfs, axis=1)

        return pdfs, labels
```

After the operation is fully defined, it needs to be added to the workspace and connected to the previous operations as shown below. Specifically, the *test images* **Input** operation and the *model* output from **TrainRedshift** should be connected to the *images* and *model* inputs to **PredictRedshift** respectively.



16.6 EvalRedshift Operation

This operation creates a figure for evaluating the accuracy of the redshift model. The resulting figure (shown on the right in the image below) plots the true redshift value against the predicted value. The further a point falls away from the diagonal dotted line, the more incorrect that prediction.

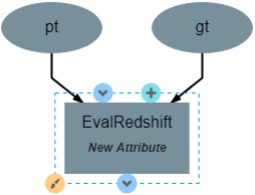
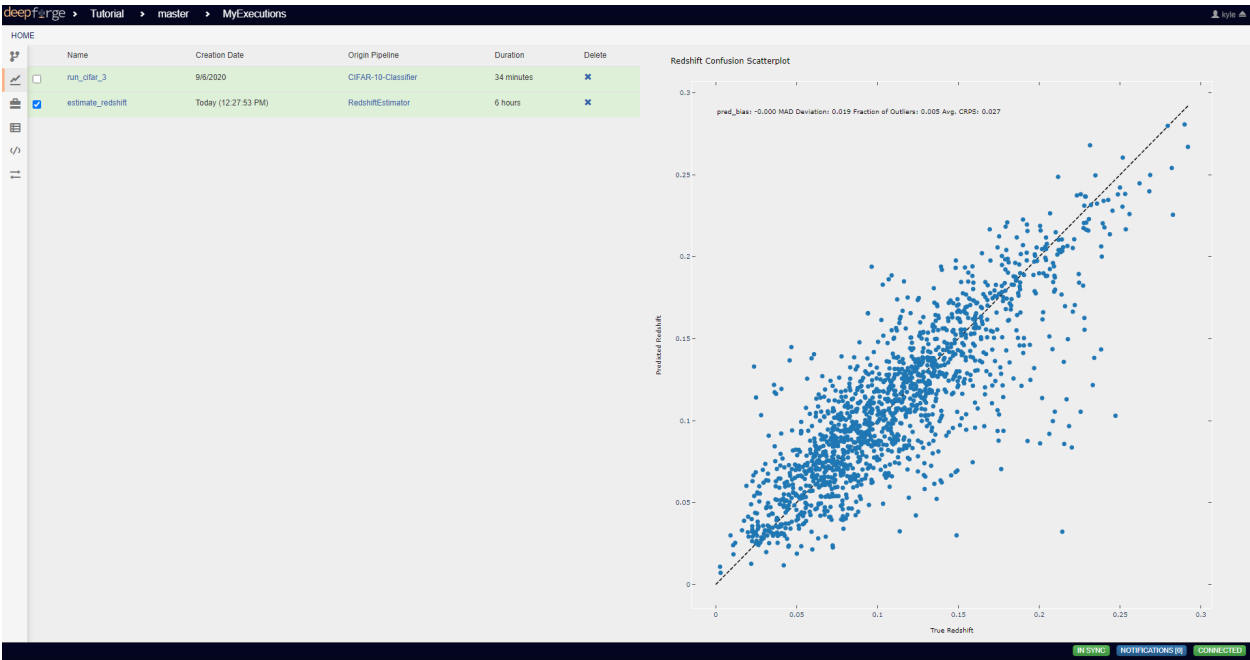
This operation has no attributes and produces no output. It requires two inputs in the form of a list of predicted redshift values (*pt*) and a list of actual redshift values (*gt*). The structure of the operation is as shown below:

The code for this operation is below and is heavily annotated to explain the various graphing functions.

```
import numpy as np
from properscoring import crps_gaussian
import matplotlib.pyplot as plt

class EvalRedshift():
```

(continues on next page)



(continued from previous page)

```

def execute(self, gt, pt):
    print('Evaluating model')

    # Calculates various metrics for later display. For more info, see section 4.
    → 1 of
        # of Pasquet et. al.
        residuals = (pt - gt) / (gt + 1)
        pred_bias = np.average(residuals)
        dev_MAD = np.median(np.abs(residuals - np.median(residuals))) * 1.4826
        frac_outliers = np.count_nonzero(np.abs(residuals) > (dev_MAD * 5)) /
    → len(residuals)
        crps = np.average(crps_gaussian(pt, np.mean(pt), np.std(pt)))

        # Creates the figure and gives it a title
        plt.figure()
        plt.title('Redshift Confusion Scatterplot')

        # Plots all galaxies where the x-value is the true redshift of a galaxy and
    → the
        # y-value is the predicted redshift value of a galaxy
        plt.scatter(gt, pt)

        # Creates a dashed black line representing the line on which a perfect
    → prediction
        # would lie. This line has a slope of 1 and goes from the origin to the
    → maximum
        # redshift (predicted or actual)
        maxRS = max(max(gt), max(pt))
        endpoints = [0, maxRS]
        plt.plot(endpoints, endpoints, '--k')

        # Creates a formatted string with one metric per line. Prints metrics to three
        # decimal places
        metricStr = 'pred_bias: {pb:.03f}\n' + \
                    'MAD Deviation: {dm:.03f}\n' + \
                    'Fraction of Outliers: {fo:.03f}\n' + \
                    'Avg. CRPS: {ac:.03f}'
        formattedMetrics = metricStr.format(pb=pred_bias,
                                           dm=dev_MAD,
                                           fo=frac_outliers,
                                           ac=crps)

        # Prints the metrics string at the top left of the figure
        plt.text(0, maxRS, formattedMetrics, va='top')

        # Labels axes and displays figure
        plt.ylabel('Predicted Redshift')
        plt.xlabel('True Redshift')
        plt.show()

    return

```

Notice in the above code that there is a new library used to calculate one of the metrics. This library is not standard and is not included in many default environments. Because of this, the library needs to be added to the environment at runtime by going to the *Environment* tab in the operation editor and defining the operation dependencies as shown below. Operation dependencies are defined in the style of a [conda environment file](#).

```

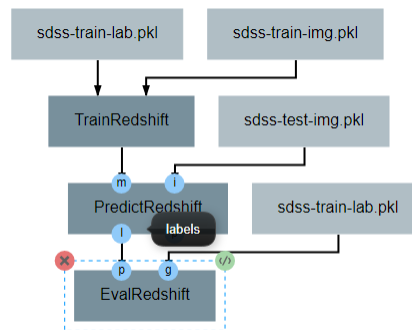
deepforge > Tutorial > master > EvalRedshift
HOME / RedshiftEstimator / EvalRedshift

-2 # Editing "EvalRedshift" Implementation
-1 #
1 import numpy as np
2 from properscoring import crps_gaussian
3 import matplotlib.pyplot as plt
4
5 class EvalRedshift():
6
7     def execute(self, gt, pt):
8         print('Evaluating model')
9
10        # For more info, see section __ of P
11        residuals = (pt - gt) / (gt + 1)
12        pred_bias = np.average(residuals)
13        dev_MAD = np.median(np.abs(residuals))
14        frac_outliers = np.count_nonzero(np.
15        crps = np.average(crps_gaussian(pt, i
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

Operation Interface  Environment
1 # Conda environment to use when executing the
2 # For more information, check out https://doc
3 dependencies:
4 - python=3.7
5 - pip:
6 - properscoring
7
IN SYNC NOTIFICATIONS (0) CONNECTED

```

After the operation is fully defined, it needs to be added to the workspace and connected to the previous operations as shown below. Specifically, the test values **Input** operation and the *labels* output from **PredictRedshift** should be connected to the *gt* and *pt* inputs to **EvalRedshift** respectively.



16.7 PdfVisRedshift Operation

This operation creates another figure for evaluating the accuracy of the redshift model as shown below. Compared to the output of the **EvalRedshift** operation, this figure provides a more zoomed in picture of individual predictions. Each of the subplots is a plotting of the probability density function for a randomly chosen input image. The red and green lines indicate the predicted and actual value of the image's redshift value respectively.

This operation has one attribute, *num_images* and produces no output. It requires three inputs in the form of a list of predicted redshift values (*pt*), a list of actual redshift values (*gt*), and a list of probability density functions (*pdfs*). The structure of the operation is as shown below:

The code for this operation is below and is heavily annotated to explain the various graphing functions.

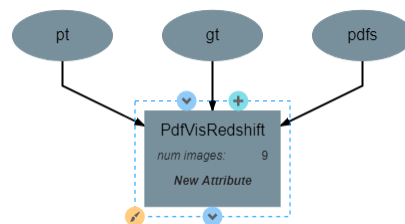
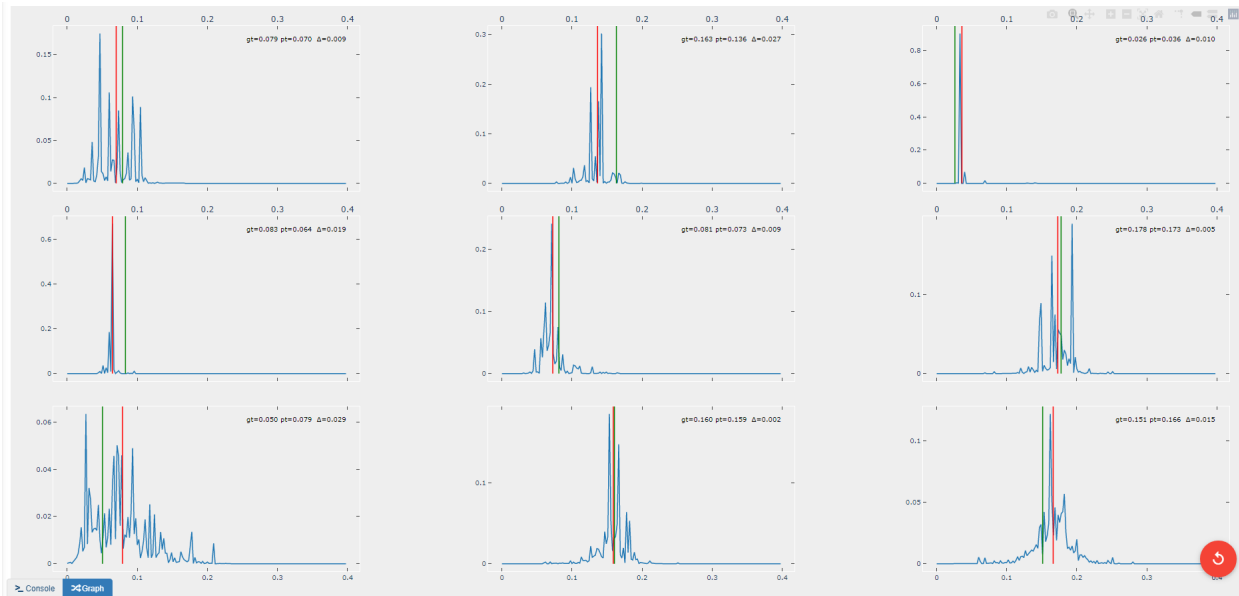
```

import numpy as np
import matplotlib.pyplot as plt
import math

class PdfVisRedshift():
    def __init__(self, num_images=9):

```

(continues on next page)



(continued from previous page)

```

# Calculates the number of rows and columns needed to arrange the images in
# as square of a shape as possible
self.num_images = num_images
self.num_cols = math.ceil(math.sqrt(num_images))
self.num_rows = math.ceil(num_images / self.num_cols)

self.max_val = 0.4
return

def execute(self, gt, pt, pdfs):

    # Creates a collection of subfigures. Because each predicition uses the same_
    ↪bins,
    # x-axes are shared.
    fig, splts = plt.subplots(self.num_rows,
                              self.num_cols,
                              sharex=True,
                              sharey=False)

    # Chooses a random selection of indices representing the chosen images
    random_indices = np.random.choice(np.arange(len(pt)),
                                      self.num_images,
                                      replace=False)

    # Extracts the pdfs and redshifts represented by the chosen indices
    s_pdfs = np.take(pdfs, random_indices, axis=0)
    s_pt = np.take(pt, random_indices, axis=0)
    s_gt = np.take(gt, random_indices, axis=0)

    # Creates a list of the lower end of the ranges represented by each bin
    x_range = np.arange(0, self.max_val, self.max_val / pdfs.shape[1])

    for i in range(self.num_images):
        col = i % self.num_cols
        row = i // self.num_cols

        # Creates a line graph from the current image's pdf
        splts[row,col].plot(x_range, s_pdfs[i], '-')

        # Creates two vertical lines to represent the predicted value (red) and_
        ↪the
        # actual value (green)
        splts[row,col].axvline(s_pt[i], color='red')
        splts[row,col].axvline(s_gt[i], color='green')

        # Creates a formatted string with one metric per line. Prints metrics to_
        ↪three
        # decimal places. d (delta) is how far off the prediction was from the_
        ↪actual value
        metricString = 'gt={gt:.03f}\npt={pt:.03f}\n \u0394={d:.03f}'
        metricString = metricString.format(gt = s_gt[i],
                                           pt = s_pt[i],
                                           d = abs(s_gt[i]-s_pt[i]))

        # Determines whether the metrics should be printed on the left or right_
        ↪of the

```

(continues on next page)

(continued from previous page)

```

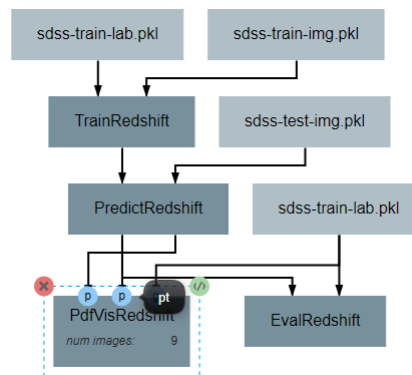
→clear      # figure. If prediction is on the left end, the right side should be more_
            # and should be the chosen side.
            alignRight = s_pt[i] <= self.max_val / 2

            # Adds the metric string to the figure at the top of the subfigure (which_
→is the     # max value of that pdf)
            splts[row,col].text(self.max_val if alignRight else 0,
                                np.max(s_pdfs[i]),
                                metricString,
                                va='top',
                                ha='right' if alignRight else 'left')

            # Automatically tweaks margins and positioning of the graph
            plt.tight_layout()
            plt.show()

```

After the operation is fully defined, it needs to be added to the workspace and connected to the previous operations as shown below. Specifically, the *labels* and *pdfs* output from **PredictRedshift** and the test values **Input** operation should be connected to the *pt*, *pdfs* and *pt* inputs to **PdfVisRedshift** respectively.



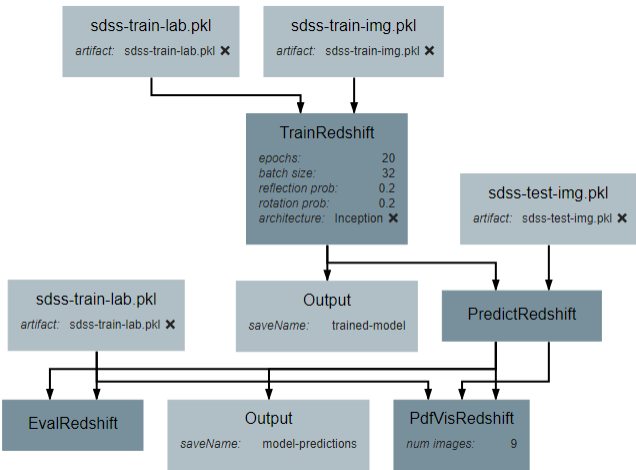
16.8 Output Operations

Output operations are special operations that allow saving python objects generated during execution. For instance, in this tutorial, it might be useful to save the trained model and the generated predictions for later use or analysis. Shown below is the result of adding two **Output** operations to the pipeline to save these two objects.

Objects created in this way will be saved in the execution working directory (defined in *Execution Options* when executing a pipeline) under the name given to the operation's *saveName* attribute. Objects saved in this manner will also be automatically added to the list of available artifacts for use in other pipelines.

16.9 Execution and Results

As with all pipelines, this pipeline can be executed using the red floating button in the bottom right of the pipeline editor view. In addition to the normal settings that are always included, this pipeline (as with any pipeline using **Input** operations) required additional credentials for each artifact being used.



deepforge > Tutorial > master > MyArtifacts kyle

HOME

Name	Type	Size	Creation Date	
model-predictions	numpy.ndarray	11.9 KiB	9/7/2020	🔍 ✕ 📄
trained-model	keras.engine.training.Model	294.3 MiB	9/7/2020	🔍 ✕ 📄
sdss-train-img.pkl	pickle	468.8 MiB	9/7/2020	🔍 ✕ 📄
sdss-train-lab.pkl	pickle	47.0 KiB	9/7/2020	🔍 ✕ 📄
sdss-test-lab.pkl	pickle	11.9 KiB	9/7/2020	🔍 ✕ 📄
sdss-test-img.pkl	pickle	117.2 MiB	9/7/2020	🔍 ✕ 📄

IN SYNC NOTIFICATIONS (0) CONNECTED

Credentials for Pipeline Inputs

sdss-train-img.pkl (SciServer Files Service):

Username
SciServer username

Password
SciServer password

sdss-test-img.pkl (SciServer Files Service):

Username
SciServer username

Password
SciServer password

sdss-test-lab.pkl (SciServer Files Service):

Username
SciServer username

Password
SciServer password

sdss-train-lab.pkl (SciServer Files Service):

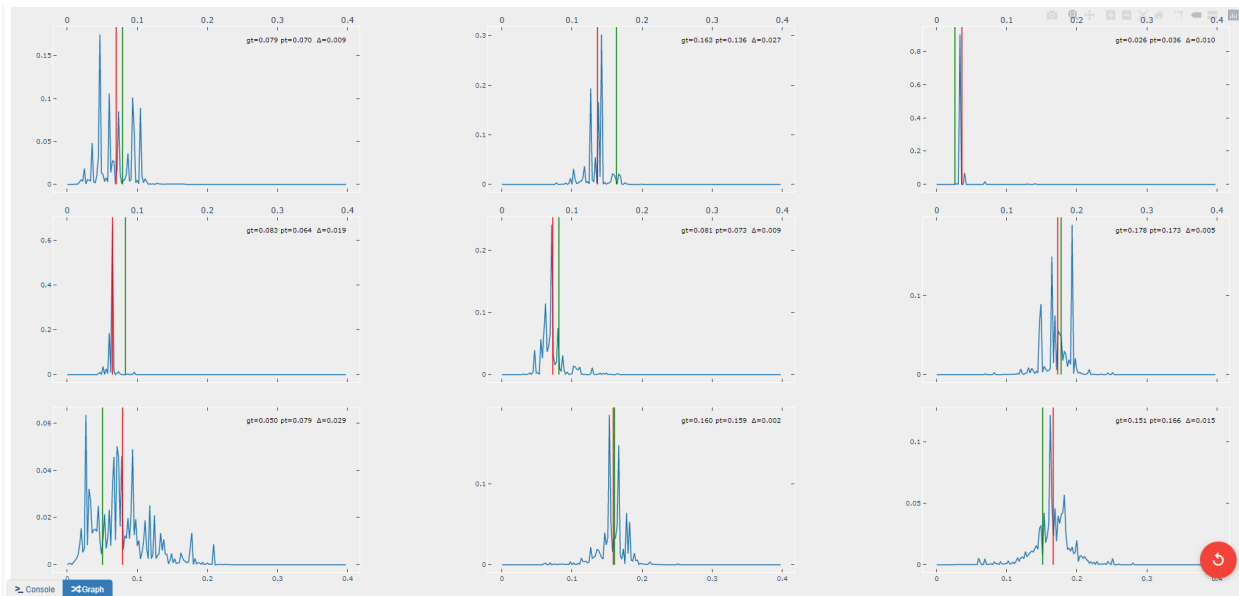
Username
SciServer username

Password
SciServer password

To view the output of the execution, go to the *Executions* tab and check the box next to the desired execution.



For a more detailed and larger view of individual figures, click on the name of the execution to view its status page and open the console output for the desired operation. In the bottom left is a set of buttons for switching between console output and graph output for that operation.



The project described on this page can be found in the [examples repo](#) on GitHub under the name **Redshift-Tutorial.webgmex**

17.1 Pipeline Overview

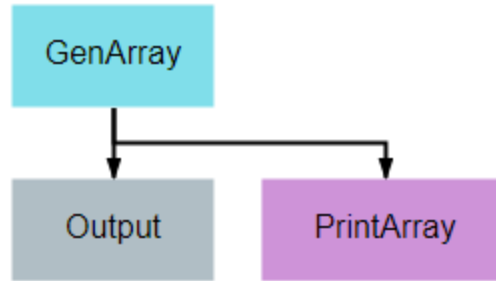
1. *Basic Input/Output*
2. *Display Random Image*
3. *Display Random CIFAR-10*
4. *Train CIFAR-10*
5. *Train-Test*
6. *Train-Test-Compare*
7. *Train-PredVis*
8. *Download-Train-Evaluate*

17.2 Pipelines

17.2.1 Basic Input/Output

This pipeline provides one of the simplest examples of a pipeline possible in DeepForge. Its sole purpose is to create an array of numbers, pass the array from the first node to the second node, and print the array to the output console.

The **Output** operation shown is a special built-in operation that will save the data that is provided to it to the selected storage backend. This data will then be available within the same project as an artifact and can be accessed by other pipelines using the special built-in **Input** operation.

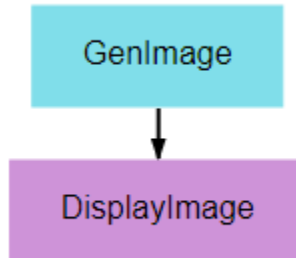


```
import numpy

class GenArray():
    def __init__(self, length=10):
        self.length = length
        return

    def execute(self):
        arr = list(numpy.random.rand(self.length))
        return arr
```

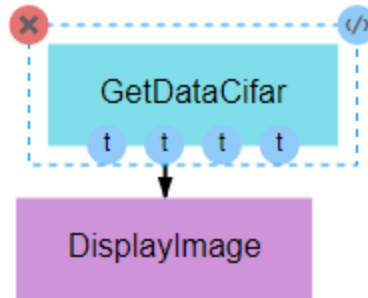
17.2.2 Display Random Image



This pipeline's primary purpose is to show how graphics can be output and viewed. A random noise image is generated and displayed using matplotlib's pyplot library. Any graphic displayed using the **plt.show()** function can be viewed in the executions tab.

```
from matplotlib import pyplot as plt
from random import randint

class DisplayImage():
    def execute(self, image):
        if len(image.shape) == 4:
            image = image[randint(0, image.shape[0] - 1)]
        plt.imshow(image)
        plt.show()
```



17.2.3 Display Random CIFAR-10

As with the previous pipeline, this pipeline simply displays a single image. The image from this pipeline, however, is more meaningful, as it is drawn from the commonly used [CIFAR-10 dataset](#). This pipeline seeks to provide an example of the input being used in the next pipeline while providing an example of how the data can be obtained. This is important for users who seek to develop their own pipelines, as CIFAR-10 data generally serves as an effective baseline for testing and development of new CNN architectures or training processes.

Also note, as shown in the figure above, that it is not necessary to utilize all of the outputs of a given node. Unless specifically handled, however, it is generally inappropriate for an input to be left undefined.

```
from keras.datasets import cifar10

class GetDataCifar():
    def execute(self):
        ((train_imgs, train_labels),
         (test_imgs, test_labels)) = cifar10.load_data()
        return train_imgs, train_labels, test_imgs, test_labels
```

17.2.4 Train CIFAR-10

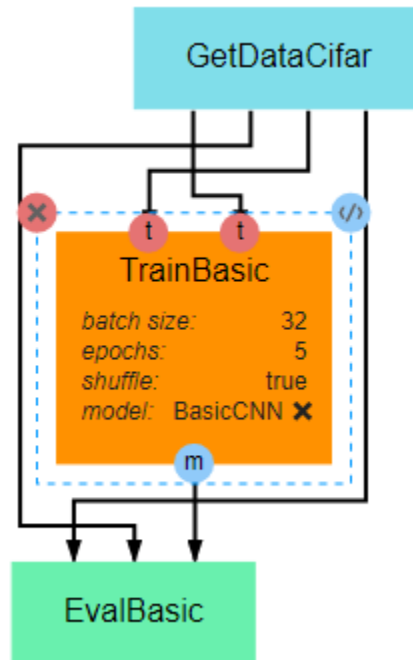
This pipeline gives a very basic example of how to create, train, and evaluate a simple CNN. The primary takeaway from this pipeline should be the overall structure of a training pipeline, which should follow the following steps in most cases:

1. Load data
2. Define the loss, optimizer, and other metrics
3. Compile model, with loss, metrics, and optimizer, using the **compile()** method
4. Train model using the **fit()** method, which requires the training inputs and outputs
5. Output the trained model for serialization and/or utilization in subsequent nodes

```
import numpy as np
import keras

class TrainBasic():
    def __init__(self, model, epochs=20, batch_size=32, shuffle=True):
        self.model = model
        self.epochs = epochs
        self.batch_size = batch_size
        self.shuffle = shuffle
```

(continues on next page)



(continued from previous page)

```

return

def execute(self, train_imgs, train_labels):
    opt = keras.optimizers.rmsprop(lr=0.001)
    self.model.compile(loss='sparse_categorical_crossentropy',
                       optimizer=opt,
                       metrics=['sparse_categorical_accuracy'])
    self.model.fit(train_imgs,
                   train_labels,
                   batch_size=self.batch_size,
                   epochs=self.epochs,
                   shuffle=self.shuffle,
                   verbose=2)
    model = self.model
    return model

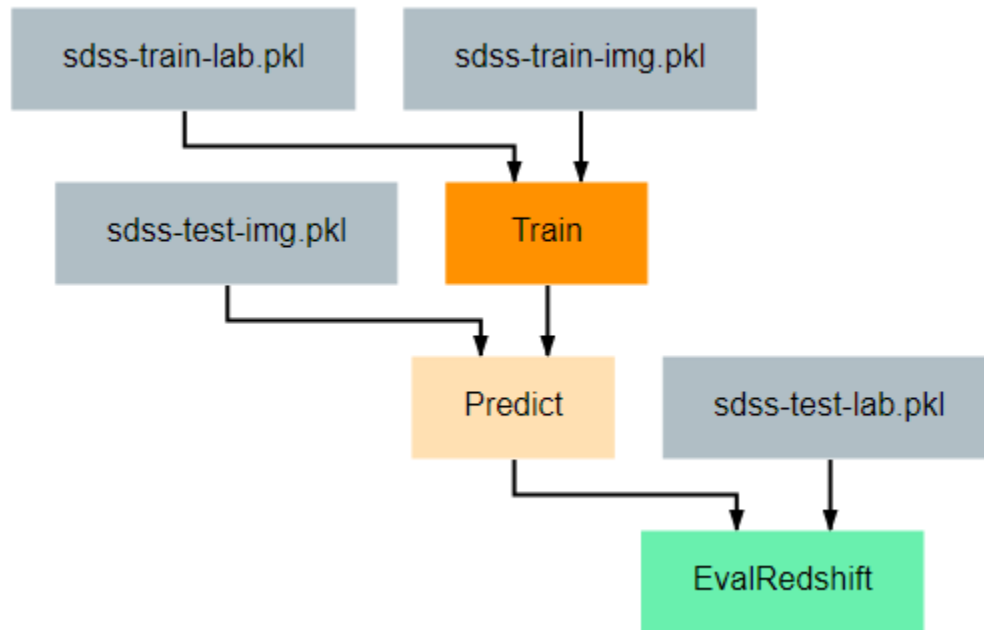
```

```

class EvalBasic():
    def __init__(self):
        return

    def execute(self, model, test_imgs, test_labels):
        results = model.evaluate(test_imgs, test_labels, verbose=0)
        for i, metric in enumerate(model.metrics_names):
            print(metric, '-', results[i])
        return results

```



17.2.5 Train-Test

This pipeline provides an example of how one might train and evaluate a redshift estimation model. In particular, the procedure implemented here is a simplified version of work by [Pasquet et. al. \(2018\)](#). For readers unfamiliar with cosmological redshift, [this article](#) provides a simple and brief introduction to the topic. For the training process, there are two primary additions that should be noted.

First, the **Train** class has been given a function named **to_categorical**. In line with the Paquet et. al. method linked above, this tutorial uses a classification model rather than a regression model for estimation. Because we are using classification models, the keras model expects the output labels to be either one-hot vectors or a single integer where the position/value indicates the range in which the true redshift value falls. This function converts the continuous redshift values into the necessary discrete, categorical format.

Second, a class has been provided to give examples of how researchers may define their own [keras Sequence](#) for training. Sequences are helpful in that they allow alterations to be made to the data during training. In the example given here, the **SdssSequence** class provides the ability to rotate or flip images before every epoch, which will hopefully improve the robustness of the final model.

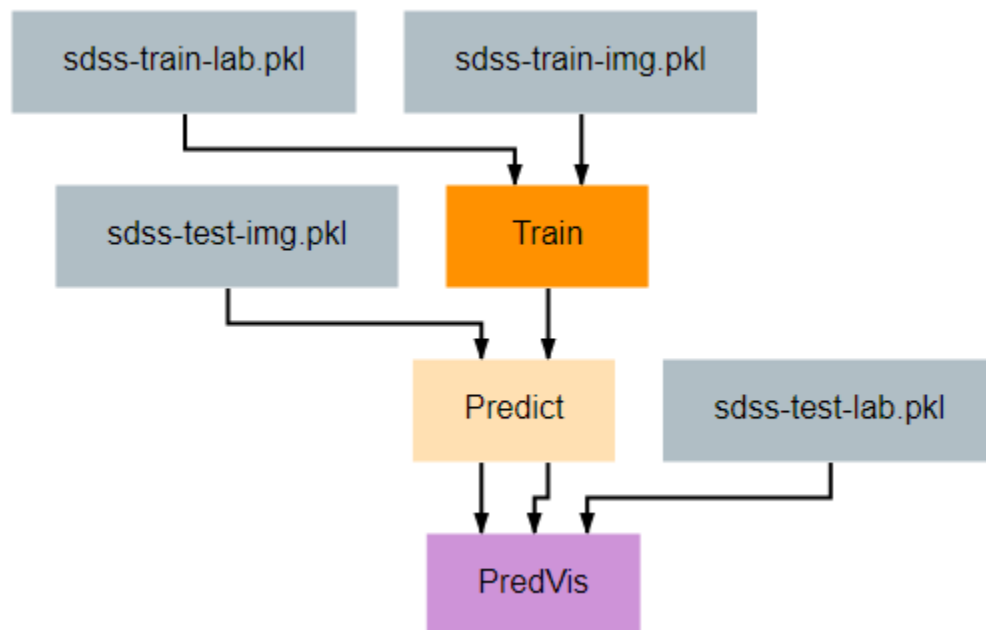
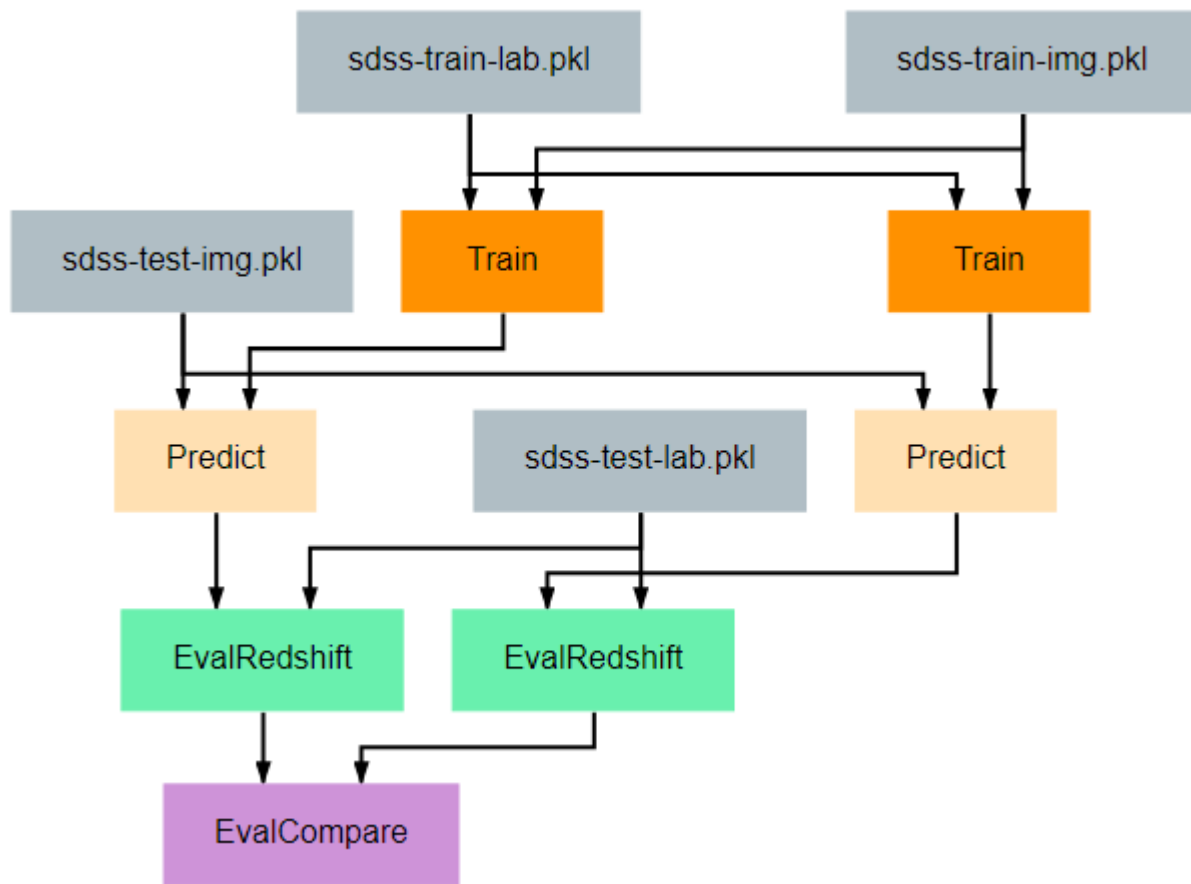
The evaluation node has also been updated to provide metrics more in line with redshift estimation. Specifically, it calculates the fraction of outlier predictions, the model's prediction bias, the deviation in the MAD scores of the model output, and the average Continuous Ranked Probability Score (CRPS) of the output.

17.2.6 Train-Test-Compare

This pipeline gives a more complicated example of how to create visualizations that may be helpful for understanding the effectiveness of a model. The **EvalCompare** node provides a simple comparison visualization of two models.

17.2.7 Train-PredVis

This pipeline shows another more complex and useful visualization example that can be helpful for understanding the effectiveness of your redshift estimation model. It generates a set of graphs like the one below that show the output



probability distribution function (pdf) for the redshift values of a set of random galaxies' images. A pair of vertical lines in each subplot indicate the actual redshift value (green) and the predicted redshift value (red) for that galaxy.

As shown in this example, any visualization that can be created using the `matplotlib.pyplot` python library can be created and displayed by a pipeline. Displaying these visualizations can be accomplished by calling the `pyplot.show()` function after building the visualization. They can then be viewed from the [Executions view](#).

```
import numpy as np
from matplotlib import pyplot as plt

class PredVis():
    def __init__(self, num_bins=180, num_rows=1, num_cols=1, max_val=0.4):
        self.num_rows = num_rows
        self.num_cols = num_cols
        self.xrange = np.arange(0, max_val, max_val / num_bins)
        return

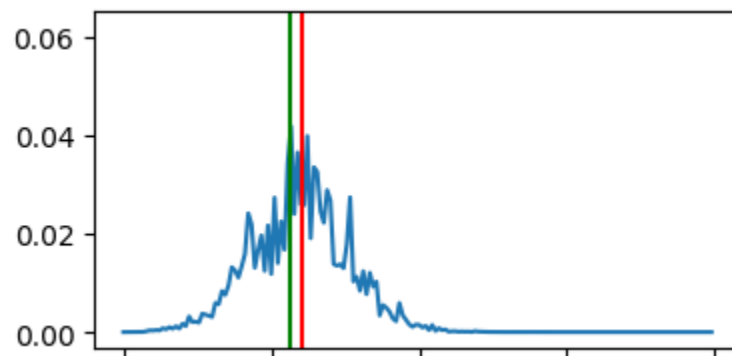
    def execute(self, pt, gt, pdfs):
        fig, splts = plt.subplots(self.num_rows, self.num_cols, sharex=True,
        ↪sharey=True)

        num_samples = self.num_rows * self.num_cols

        random_indices = np.random.choice(list(range(len(gt))), num_samples,
        ↪replace=False)

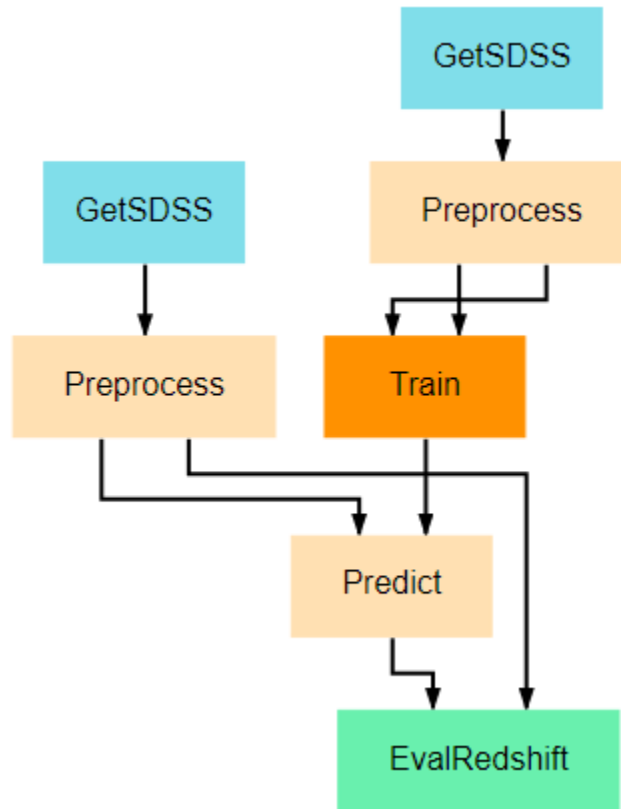
        s_pdfs = np.take(pdfs, random_indices, axis=0)
        s_pt = np.take(pt, random_indices, axis=0)
        s_gt = np.take(gt, random_indices, axis=0)

        for i in range(num_samples):
            col = i % self.num_cols
            row = i // self.num_cols
            splts[row,col].plot(self.xrange, s_pdfs[i], '-')
            splts[row,col].axvline(s_pt[i], color='red')
            splts[row,col].axvline(s_gt[i], color='green')
        plt.show()
```



17.2.8 Download-Train-Evaluate

This pipeline provides an example of how data can be retrieved and utilized in the same pipeline. The previous pipelines use manually uploaded artifacts. In many real cases, users may desire to retrieve novel data or more specific data using SciServer's CasJobs API. In such cases, the **DownloadSDSS** node here makes downloading data relatively



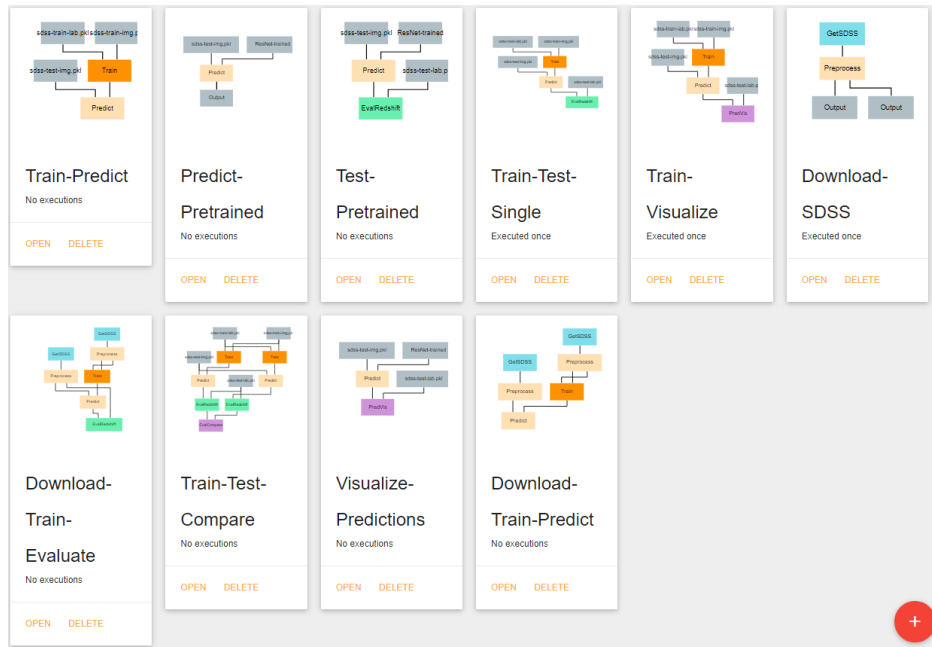
simple for users. It should be noted that the data downloaded is not in a form easily usable by our models and first requires moderate preprocessing, which is performed in the **Preprocessing** node. This general structure of download-process-train is a common pattern, as data is rarely supplied in a clean, immediately usable format.

The project described on this page can be found in the [examples repo](#) on GitHub under the name **Redshift-Application.webgmex**

This project provides a small collection of generalized pipelines for the training and utilization of redshift estimation models. This project is designed to allow simple use by only requiring that the configuration parameters of individual nodes be defined where necessary. The most involved alterations that should be necessary for most users is the definition of additional architectures in the **Resources** tab. It should be noted that any newly defined architecture should have an output length and input shape that match the *num_bins* and *input_shape* configuration parameters being used in the various pipelines.

18.1 Pipeline Overview

- *Train Test Single*
- *Train Test Compare*
- *Download Train Evaluate*
- *Train Predict*
- *Predict Pretrained*
- *Test Pretrained*
- *Download SDSS*
- *Download Train Predict*
- *Visualize Predictions*



18.2 Pipelines

18.2.1 Train Test Single

Trains and evaluates a single CNN model. Uses predefined artifacts that contain the training and testing data. For this and all training pipelines, the artifacts should each contain a single numpy array. Input arrays should be a 4D array of shape **(n, y, x, c)** where n=number of images, y=image height, x=image width, and c=number of color channels. Output (label) arrays should be of shape **(n,)**.

18.2.2 Train Test Compare

Trains and evaluates two CNN models and compares effectiveness of the models.

18.2.3 Download Train Evaluate

Downloads SDSS images, trains a model on the images, and evaluates the model on a separate set of downloaded images. Care should be taken when defining your own CasJobs query to ensure that all queried galaxies for training have a redshift value below the **Train** node's `max_val` configuration parameter's value.

18.2.4 Train Predict

Trains a single CNN model and uses the newly trained model to predict the redshift value of another set of galaxies.

18.2.5 Predict Pretrained

Predicts the redshift value of a set of galaxies using a pre-existing model that is saved as an artifact.

18.2.6 Test Pretrained

Evaluates the performance of a pre-existing model that is saved as an artifact.

18.2.7 Download SDSS

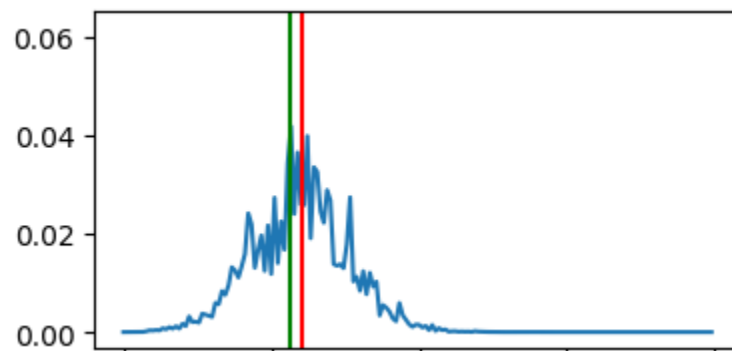
Download SDSS images and save them as artifacts. Can be used in conjunction with the other pipelines that rely on artifacts rather than images retrieved at execution time.

18.2.8 Download Train Predict

Download SDSS images and use some images to train a model before using the model to predict the redshift value of the remaining galaxies.

18.2.9 Visualize Predictions

This pipeline produces a visualization that can be helpful for understanding the effectiveness of your redshift estimation model. It generates a set of graphs like the one below that show the output probability distribution function (pdf) for the redshift values of a set of random galaxies' images. A pair of vertical lines in each subplot indicate the actual redshift value (green) and the predicted redshift value (red) for that galaxy. This allows users to see how far the model's predictions are from the correct answers and can help with identifying biases or weak-points the model may have (for example, consistently underestimation or inaccuracy with galaxies in a specific redshift range).



Command Line Interface

This document outlines the functionality of the `deepforge` command line interface (provided after installing `deepforge` with `npm install -g deepforge`).

- Installation Configuration
- Starting DeepForge or Components
- Update or Uninstall DeepForge
- Managing Extensions

19.1 Installation Configuration

Installation configuration can be edited using the `deepforge config` command as shown in the following examples:

Printing all the configuration settings:

```
deepforge config
```

Printing the value of a configuration setting:

```
deepforge config blob.dir
```

Setting a configuration option, such as `blob.dir` can be done with:

```
deepforge config blob.dir /some/new/directory
```

For more information about the configuration settings, check out the [configuration](#) page.

19.2 Starting DeepForge Components

The DeepForge server can be started with the `deepforge start` command. By default, this command will start both the server and a mongo database (if applicable).

The server can be started by itself using

```
deepforge start --server
```

19.3 Update/Uninstall DeepForge

DeepForge can be updated or uninstalled using

```
deepforge update
```

DeepForge can be uninstalled using `deepforge uninstall`

19.4 Managing Extensions

DeepForge extensions can be installed and removed using the `deepforge extensions` subcommand. Extensions can be added, removed and listed as shown below

```
deepforge extensions add https://github.com/example/some-extension
deepforge extensions remove some-extension
deepforge extensions list
```

Configuration

Configuration of deepforge is done through the *deepforge config* command from the command line interface. To see all config options, simply run *deepforge config* with no additional arguments. This will print a JSON representation of the configuration settings similar to:

```
Current config:
{
  "blob": {
    "dir": "/home/irishninja/.deepforge/blob"
  },
  "mongo": {
    "dir": "~/.deepforge/data"
  }
}
```

Setting an attribute, say *blob.dir*, is done as follows

```
deepforge config blob.dir /tmp
```

20.1 Environment Variables

Most settings have a corresponding environment variable which can be used to override the value set in the cli's configuration. This allows the values to be temporarily set for a single run. For example, starting the server with a different blob location can be accomplished by setting *blob.dir* can be done with:

```
DEEPFORGE_BLOB_DIR=/tmp deepforge start -s
```

The complete list of the environment variable overrides for the configuration options can be found [here](#).

20.2 Settings

20.2.1 blob.dir

The path to the blob (large file storage containing models, datasets, etc) to be used by the deepforge server.

This can be overridden with the *DEEPFORGE_BLOB_DIR* environment variable.

20.2.2 mongo.dir

The path to use for the *-dbpath* option of mongo if starting mongo using the command line interface. That is, if the *MONGO_URI* is set to a local uri and the cli is starting the deepforge server, the cli will check to verify that an instance of mongo is running locally. If not, it will start it on the given port and use this setting for the *-dbpath* setting of mongod.

Operations can provide a variety of forms of real-time feedback including subplots, 2D and 3D plots and images using matplotlib.

21.1 Graphs

The following example shows a sample 3D scatter plot and its rendering in DeepForge.

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

class Scatter3DPlots():

    def execute(self):
        # Set random seed for reproducibility
        np.random.seed(19680801)

        def randrange(n, vmin, vmax):
            '''
            Helper function to make an array of random numbers having shape (n, )
            with each number distributed Uniform(vmin, vmax).
            '''
            return (vmax - vmin)*np.random.rand(n) + vmin

        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')

        n = 100

        # For each set of style and range settings, plot n random points in the box
        # defined by x in [23, 32], y in [0, 100], z in [zlow, zhigh].
```

(continues on next page)

(continued from previous page)

```

for m, zlow, zhigh in [('o', -50, -25), ('^', -30, -5)]:
    xs = randrange(n, 23, 32)
    ys = randrange(n, 0, 100)
    zs = randrange(n, zlow, zhigh)
    ax.scatter(xs, ys, zs, marker=m)

ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')
plt.show()

```

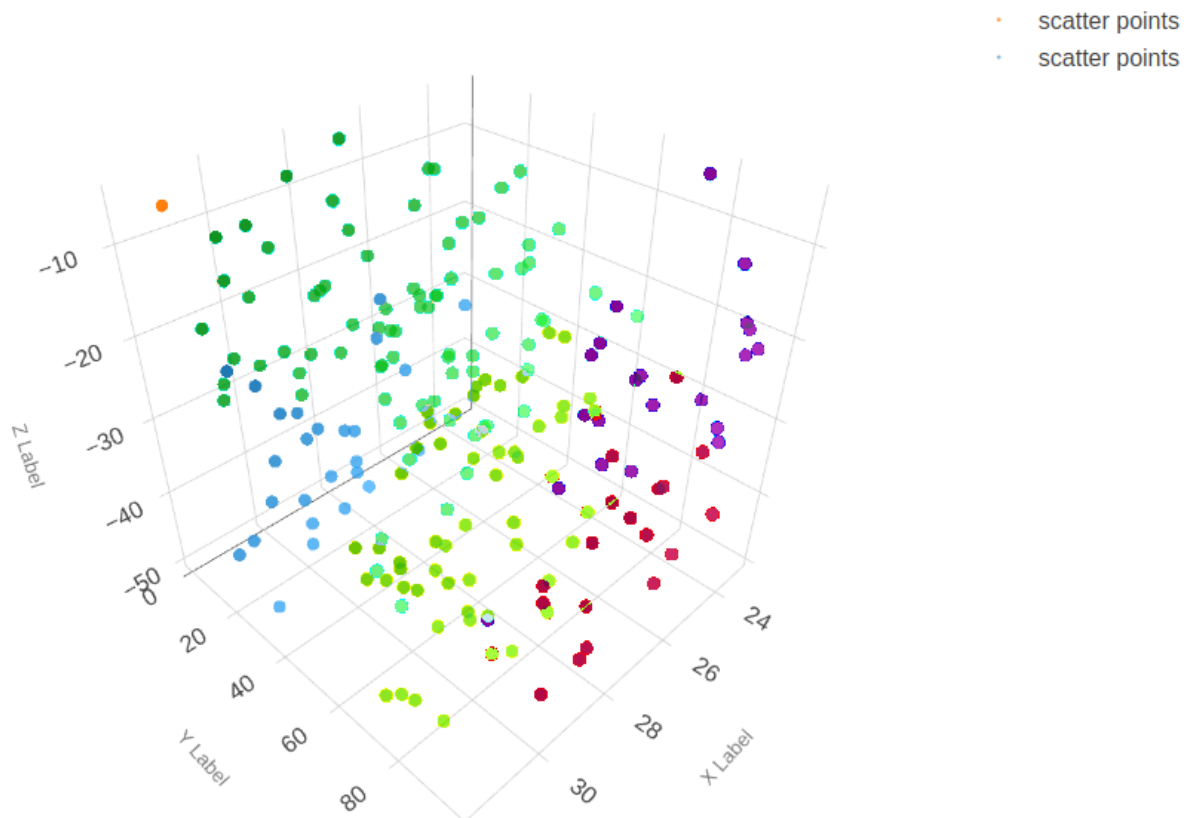


Fig. 1: Example of a 3D scatter plot using matplotlib in DeepForge

21.2 Images

Visualizing images using *matplotlib* is also supported. The following example shows images from the *MNIST fashion dataset*.

```
from matplotlib import pyplot
from keras.datasets import fashion_mnist

class MnistFashion():

    def execute(self):

        (trainX, trainy), (testX, testy) = fashion_mnist.load_data()
        # summarize loaded dataset
        print('Train: X=%s, y=%s' % (trainX.shape, trainy.shape))
        print('Test: X=%s, y=%s' % (testX.shape, testy.shape))
        for i in range(9):
            pyplot.subplot(330 + 1 + i) # define subplot
            pyplot.imshow(trainX[i], cmap=pyplot.get_cmap('gray')) # plot raw pixel_
↪data
        pyplot.show()
```

