deepforge Documentation

Release

Brian Broll

Sep 08, 2017

Getting Started

CHAPTER 1

Getting Started

What is DeepForge?

Deep learning is a very 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 the flexibility of Torch, DeepForge is able to reduce the complexity of using deep learning while still providing advanced features such as defining custom layers.

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!).

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 layers, utilize text environments to maintain the flexibility provided by torch.

Concepts and Terminology

- *Architecture* - neural network architecture composed of torch defined layers
- *Operation* - essentially a function written in torch (such as *SGD*)
- *Pipeline* - directed acyclic graph composed of operations - eg, 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)

CHAPTER 2

Quick Start

The easiest way to get started quickly 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-compose.
↪yml
```

Then start DeepForge using docker-compose:

```
docker-compose up
```

and now DeepForge can be used by opening a browser to http://localhost:8888!

For detailed instructions about deployment installations, check out our deployment installation instructions

CHAPTER 3

First Steps

DeepForge provides an example project for creating a classifier using the CIFAR10 dataset.

When first opening DeepForge in your browser (at *http://localhost:8888* if following the instructions from the quick start), you will be prompted with a list of projects to open and provided the option to create a new project. For this example, let's click "Create new..." and name our project "hello_cifar".

Clicking "Create" will bring us to a prompt for the "seed" for our project. Select "cifar10" from the dropdown and click "Create". This will now create our new project based on the cifar10 example provided with DeepForge.

In this example, we have three main pipelines: `download-normalize`, `train` and `test`. `download-normalize` downloads and prepares our data. The `train` pipeline trains a neural network model on the cifar10 dataset and the `test` pipeline tests our trained model on our test set from the cifar10 dataset.

First, we will have to retrieve and prepare the data by running the `download-normalize` pipeline. This can be done by opening the given pipeline then selecting the *Execute Pipeline* option from the action button in the lower right. As soon as that pipeline finishes, we can now use this data to train a neural network.

Next, we can open the `train` pipeline. Before we execute the pipeline we have to set the input training data that we will be using. This is done by selecting the `Input` operation then clicking the value for the `artifact` field. This will provide all the possible options for the input data; for this example, we will want to select the "trainingdata" artifact. After setting the input, we can click on the `train` operation to inspect the hyperparameters we are using and the architecture we are training. Selecting the `Output` operation will allow you to change the name of the resulting artifact of this operation (in this case, a trained model). Finally, we can execute this pipeline like before to train the model.

As this operation trains, we can view the status by viewing the running execution. The easiest way to view the running execution is by clicking the given execution from the execution tray in the bottom left when viewing the originating pipeline.

Once the model has been trained, we can test the given model using the `test` pipeline. In this pipeline, we have a few more inputs to set: "testing data", "model to test" and the "human-readable class labels". If you aren't clear which operation provides which input, you can simply hover over it's connected port on the `test` operation. This will provide a tooltip with the full name of the input.

After setting the inputs for the `test` pipeline (using the trained model and data from the first two pipelines), we can simply execute this pipeline to test our model. After executing the `test` pipeline, we can view the execution and

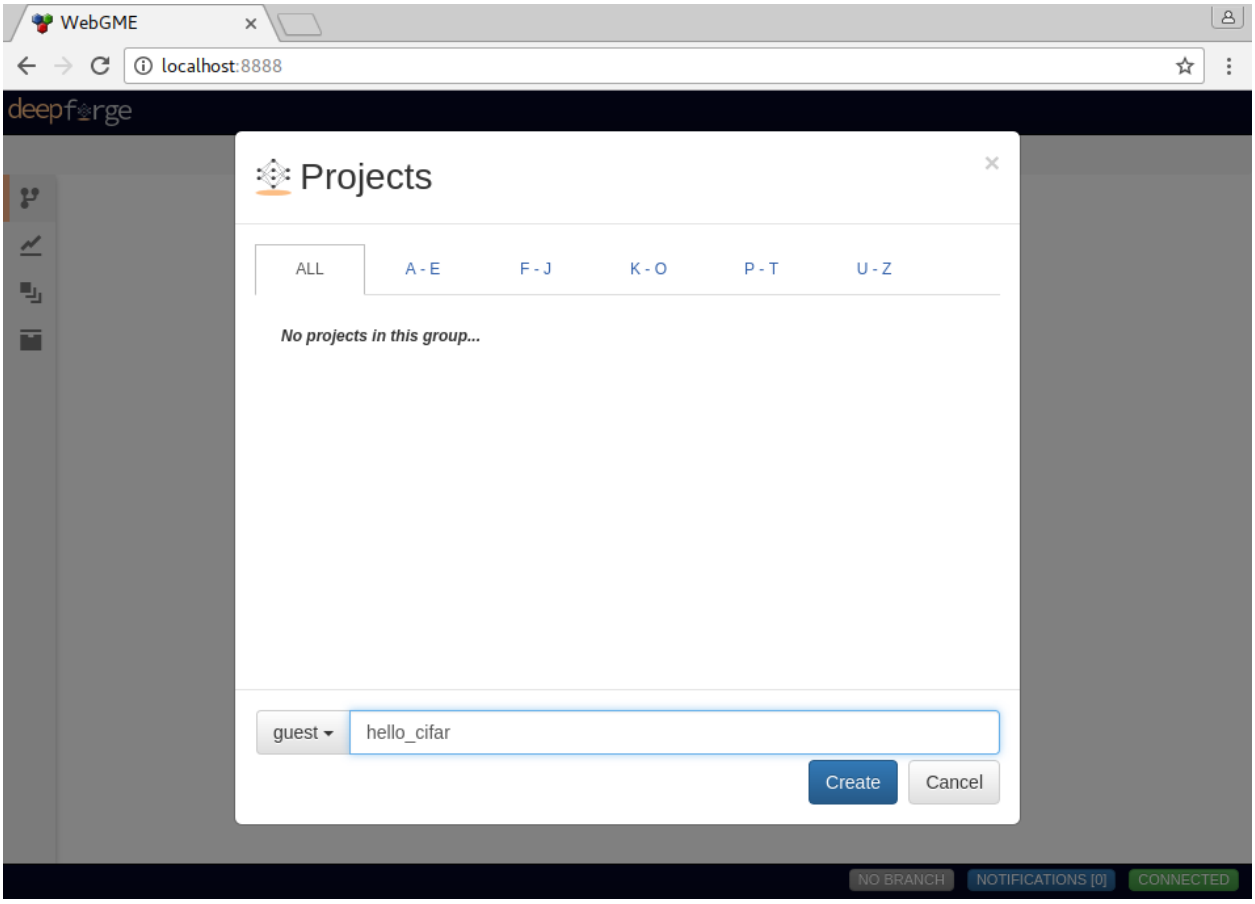

Fig. 3.1: Creating our "hello_cifar" example project

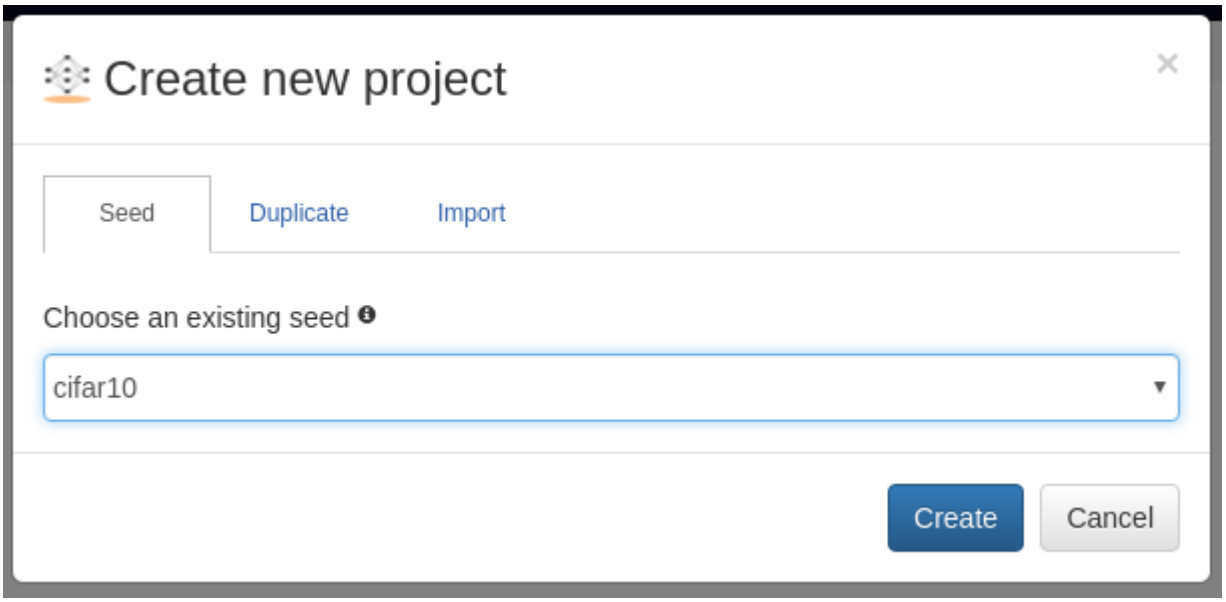

Fig. 3.2: Selecting the "cifar10" example seed

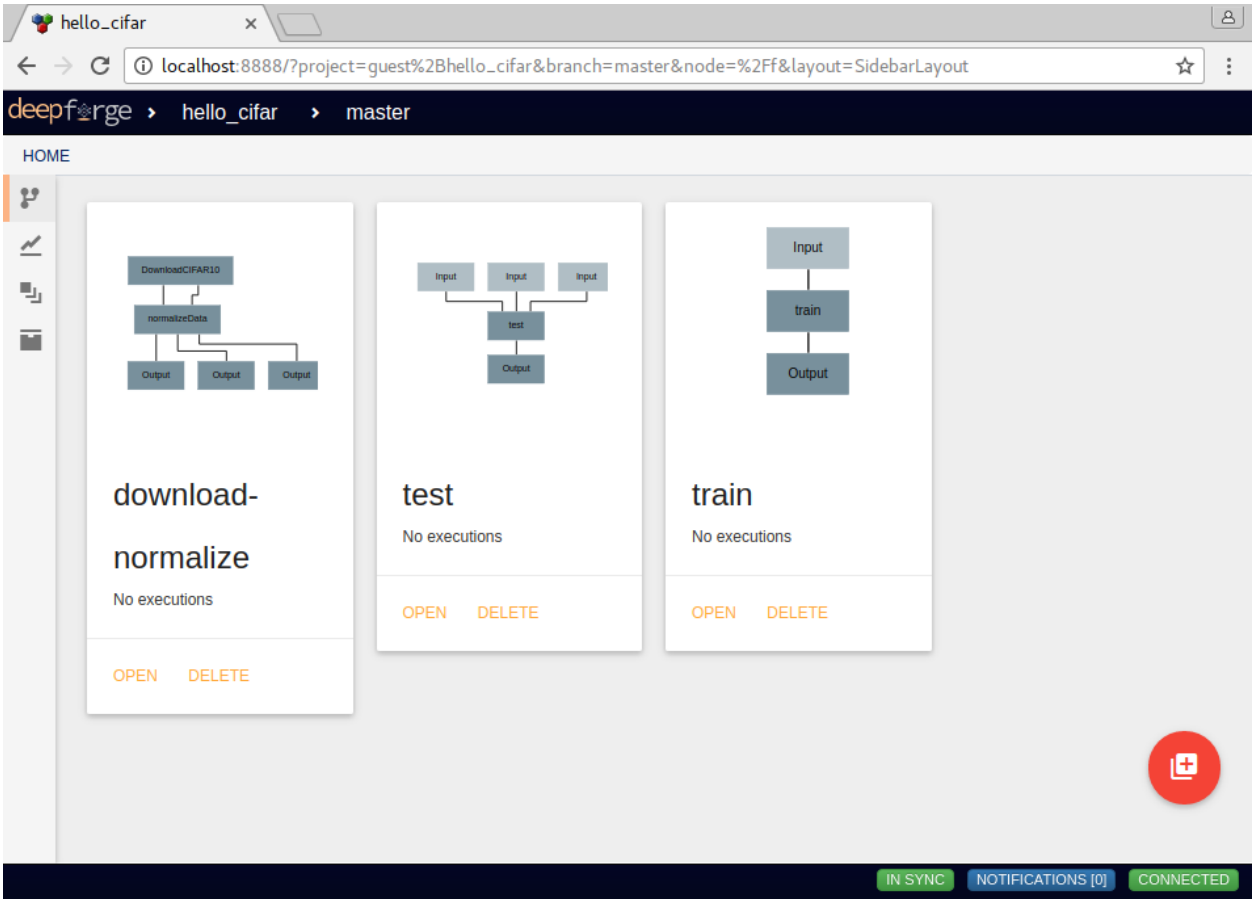

Fig. 3.3: Three main pipelines in the cifar10 example project

Fig. 3.4: Selecting the training data for the input to the training pipeline

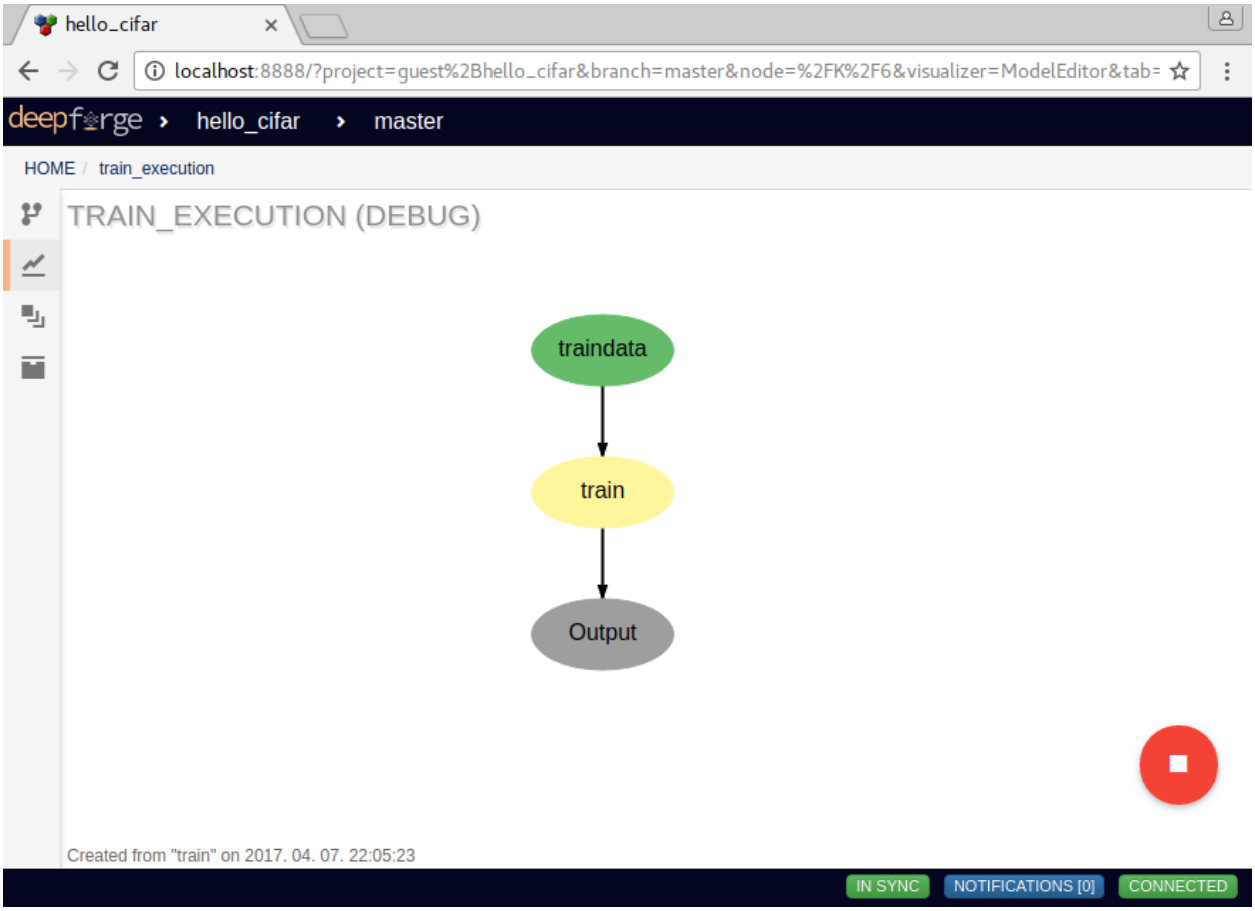

Fig. 3.5: Viewing the execution of the training pipeline

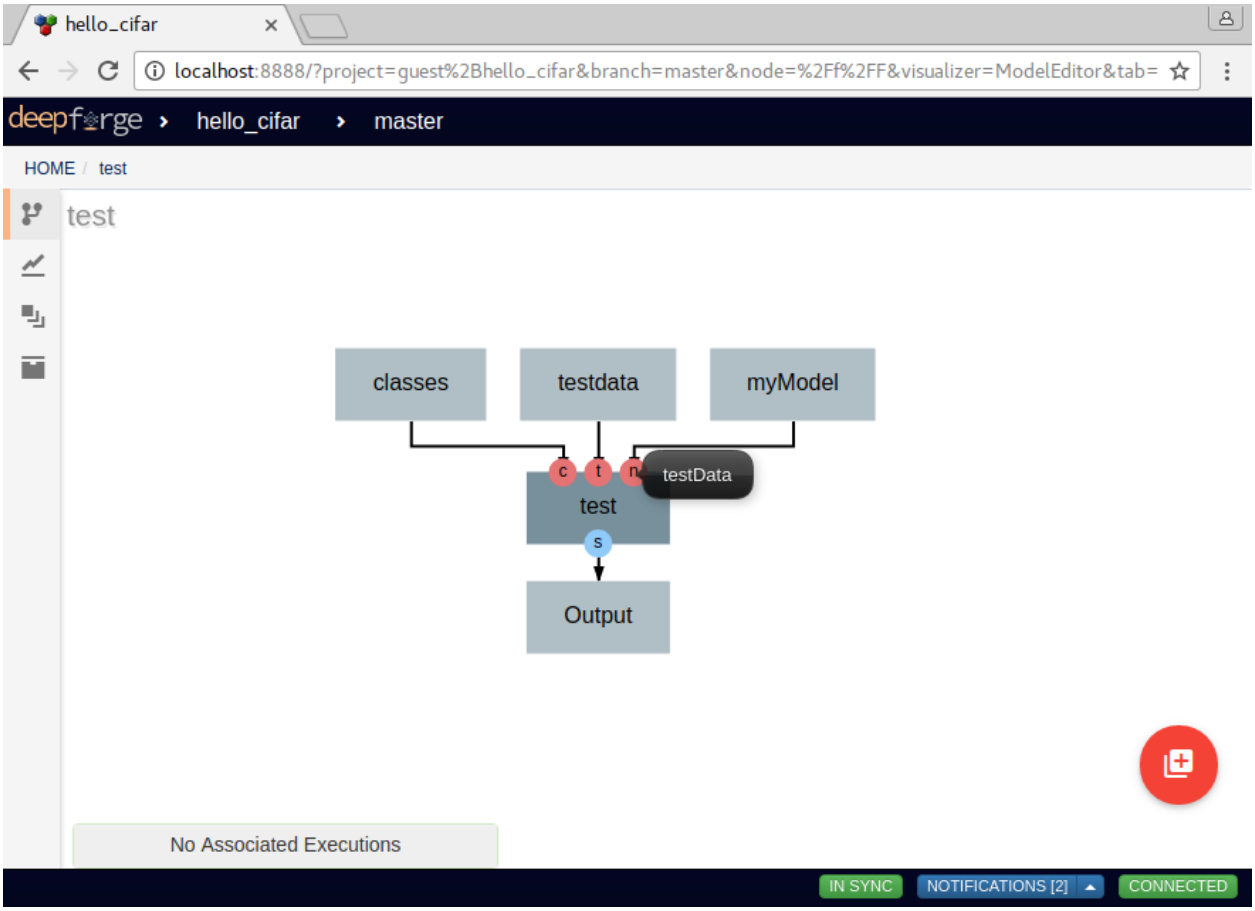

Fig. 3.6: Viewing the execution of the testing pipeline

open the `test` job to view the stdout for the given job. In the `test` operation, this will allow us to view the printed accuracies of the model over each class.

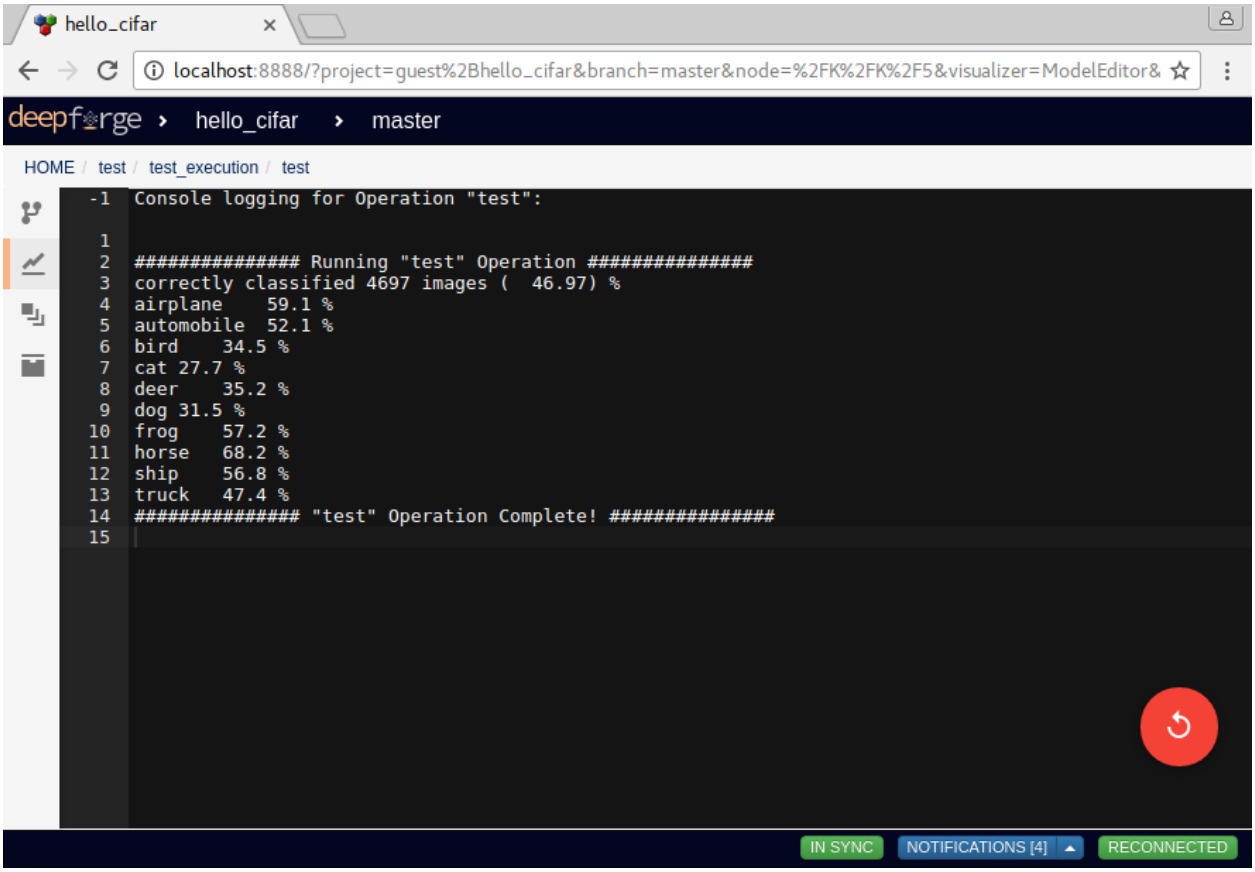

Fig. 3.7: Viewing the results of the testing operation

And that's it! We have just trained and tested our first neural network model using DeepForge. Although there are still a lot more advanced features that can be used, this should at least familiarize us with some of the core concepts in DeepForge.

CHAPTER 4

Custom Operations

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

The Basics

Operations are used in pipelines and have named, typed 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 left 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.

On the right 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. This includes some autocomplete support for common globals in this context like the `deepforge` and `torch` globals. It also is synchronized with the interface editor and will provide input to the interface editor about unused variables, etc. These errors will present themselves as error or warning highlights on the data in the interface editor. A section of the implementation is shown below:

```
trainer = nn.StochasticGradient(net, criterion)
trainer.learningRate = attributes.learningRate
trainer.maxIteration = attributes.maxIterations

print('training for ' .. tostring(attributes.maxIterations) .. ' iterations (max)')
print('learning rate is ' .. tostring(attributes.learningRate))
print(trainer)
```

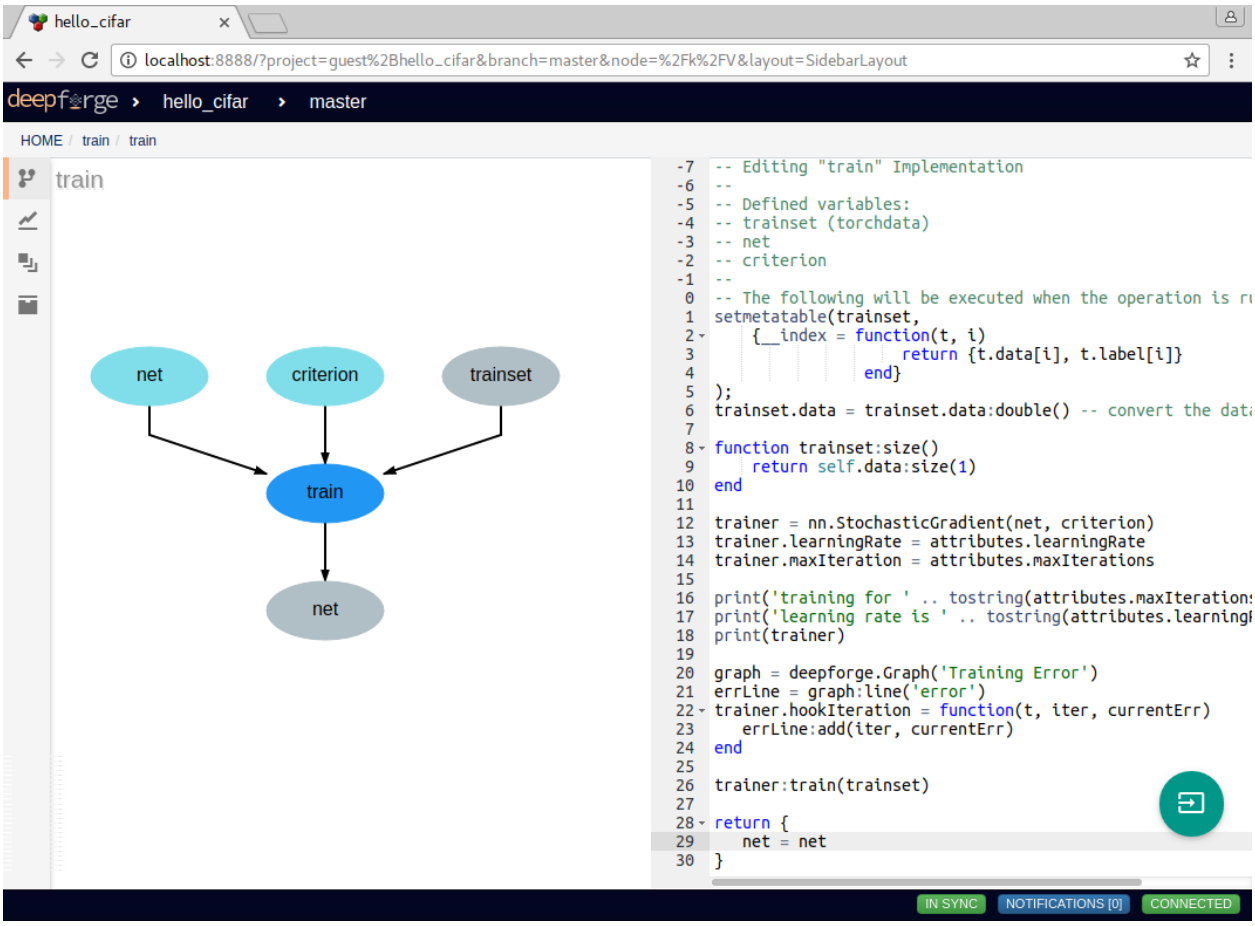

Fig. 4.1: Editing the "train" operation provided in the "First Steps" section

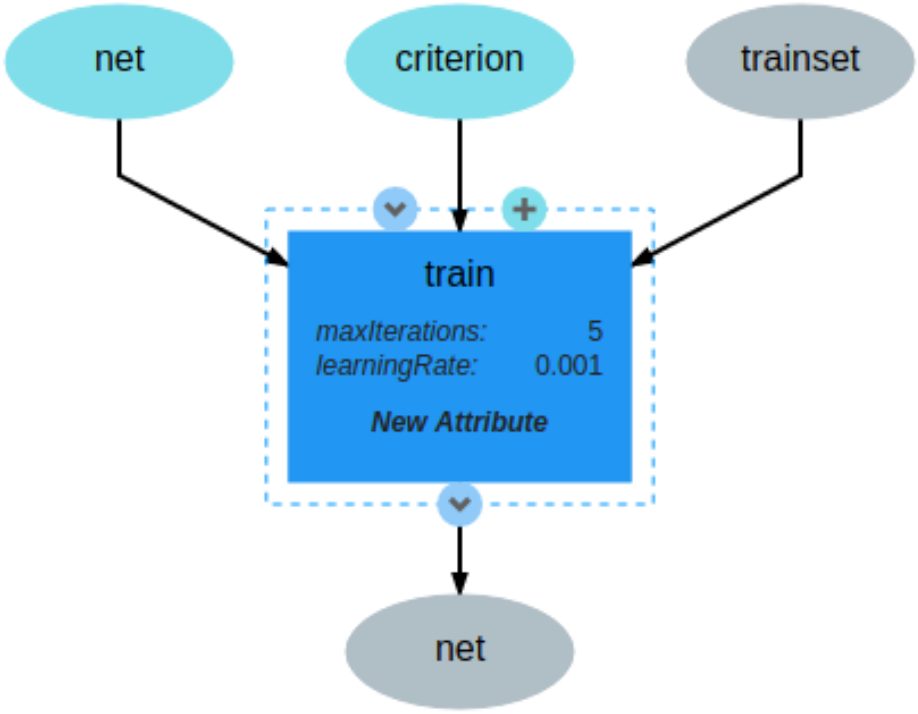

Fig. 4.2: The train operation accepts training data, an architecture and criterion and returns a trained model

```
-- Adding the error graph
graph = deepforge.Graph('Training Error') -- creating graph feedback
errLine = graph:line('error')
trainer.hookIteration = function(t, iter, currentErr)
   errLine:add(iter, currentErr) -- reporting the current error (will update in real␣
↪time in DeepForge)
end

trainer:train(trainset)

return {
   net = net
}
```

The "train" operation uses the `StochasticGradient` functionality from the `nn` package to perform stochastic gradient descent. This operation sets all the parameters using values provided to the operation as either attributes or references. In the implementation, attributes are provided by the `attributes` variable and provides access to the user defined attributes from within the implementation. References are treated similarly to operation inputs and are defined in variables of the same name. This can be seen with the `net` and `criterion` variables in the first line. Finally, operations return a table of their named outputs; in this example, it returns a single output named `net`, that is, the trained neural network.

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

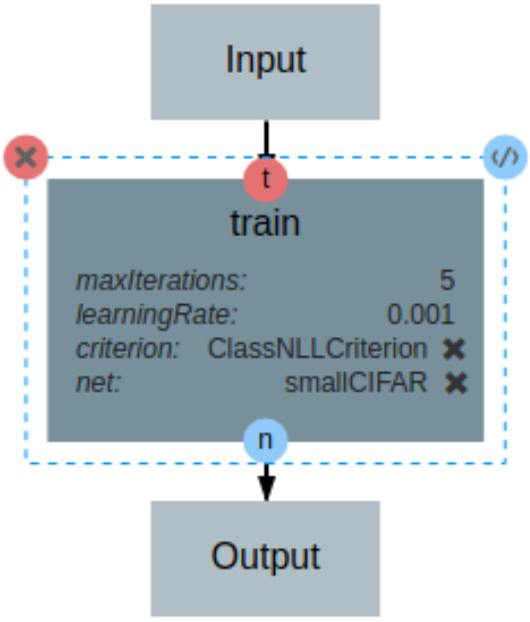

Fig. 4.3: Using the custom "train" operation in a pipeline

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 of the loss function of a model while training. When implementing an operation in DeepForge, this metadata can be created using the `deepforge` global.

Detailed information about the available operation metadata types can be found in the reference.

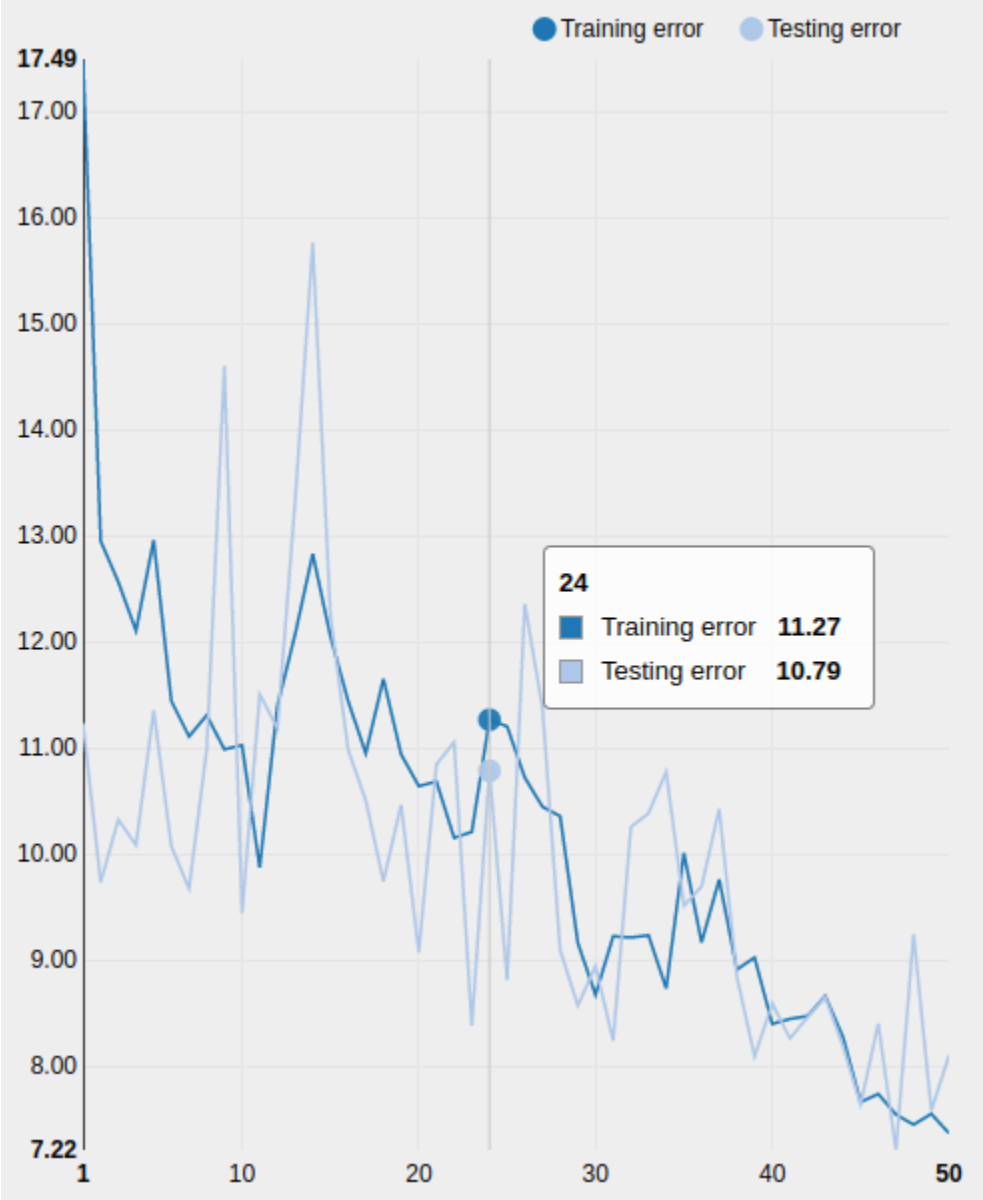

Fig. 4.4: An example graph of the loss function while training a neural network

CHAPTER 5

Custom Layers

DeepForge supports the creation of custom neural network layers using Torch7 and the easy usage of these layers in the visual architecture editor. Before creating custom layers, it is recommended to read about creating custom layers in Torch7.

A new custom layer can be created from the "add layer dialog" in the architecture editor. When creating a layer, DeepForge provides a code editor for creating custom neural network layers prepopulated with a basic template for defining the custom layer.

After defining the layer in the layer editor, DeepForge will provide this layer in the architecture editor and expose any configurable attributes for the layer. These attributes are parsed from the layer definition.

Best Practices

Here are a couple best practices to keep in mind when defining custom neural network layers:

- Use type assertions for layer, boolean attributes
- Return `self` when defining setter functions

Type assertions should be used when defining layer attributes (ie, constructor arguments or arguments to a setter function). For example, consider the following layer definition for `RecurrentAttention` which accepts an `action` layer argument to its constructor.

```
local RecurrentAttention, parent = torch.class("nn.RecurrentAttention", "nn.
↪AbstractSequencer")

function RecurrentAttention:__init(rnn, action, nStep, hiddenSize)
   parent.__init(self)
   assert(torch.isTypeOf(action, 'nn.Module'))
   assert(torch.type(nStep) == 'number')
   assert(torch.type(hiddenSize) == 'table')
   assert(torch.type(hiddenSize[1]) == 'number', "Does not support table hidden layers
↪" )
```

```
   self.rnn = rnn
   -- we can decorate the module with a Recursor to make it AbstractRecurrent
   self.rnn = (not torch.isTypeOf(rnn, 'nn.AbstractRecurrent')) and nn.Recursor(rnn)␣
↪or rnn

   -- samples an x,y actions for each example
   self.action =  (not torch.isTypeOf(action, 'nn.AbstractRecurrent')) and nn.
↪Recursor(action) or action
   self.hiddenSize = hiddenSize
   self.nStep = nStep

   self.modules = {self.rnn, self.action}

   self.output = {} -- rnn output
   self.actions = {} -- action output

   self.forwardActions = false

   self.gradHidden = {}
end
```

In this example, `assert(torch.isTypeOf(action, 'nn.Module'))` enforces that the `action` variable is another neural network layer. After defining the layer, DeepForge will parse the layer definition and create a visual representation for use in the architecture editor. As this assertion enforces that `action` is a neural network layer, DeepForge will update itself accordingly; in this case, editing the attribute will allow the user to hierarchically create nested neural network architectures to be passed as the `action` argument to the constructor.

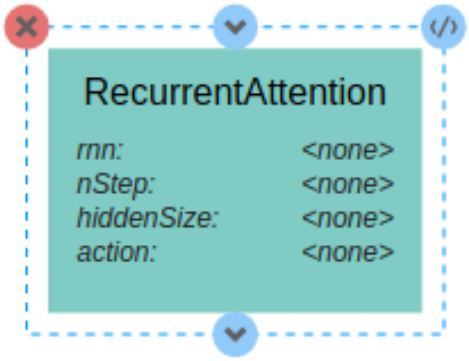

Fig. 5.1: RecurrentAttention has attributes for each of the constructor arguments

An example of the generated visual model for the `RecurrentAttention` is provided above. This layer has attributes for each of the constructor arguments defined in its definition. Clicking on the `<none>` value for the `action` attribute will then allow the user to provide layer inputs as shown below.

The second best practice is to make sure to **return self in any setter functions**. An example of this can be found in the setters in the `SpatialMaxPooling` layer shown below:

```
function SpatialMaxPooling:ceil()
  self.ceil_mode = true
  return self
end

function SpatialMaxPooling:floor()
```

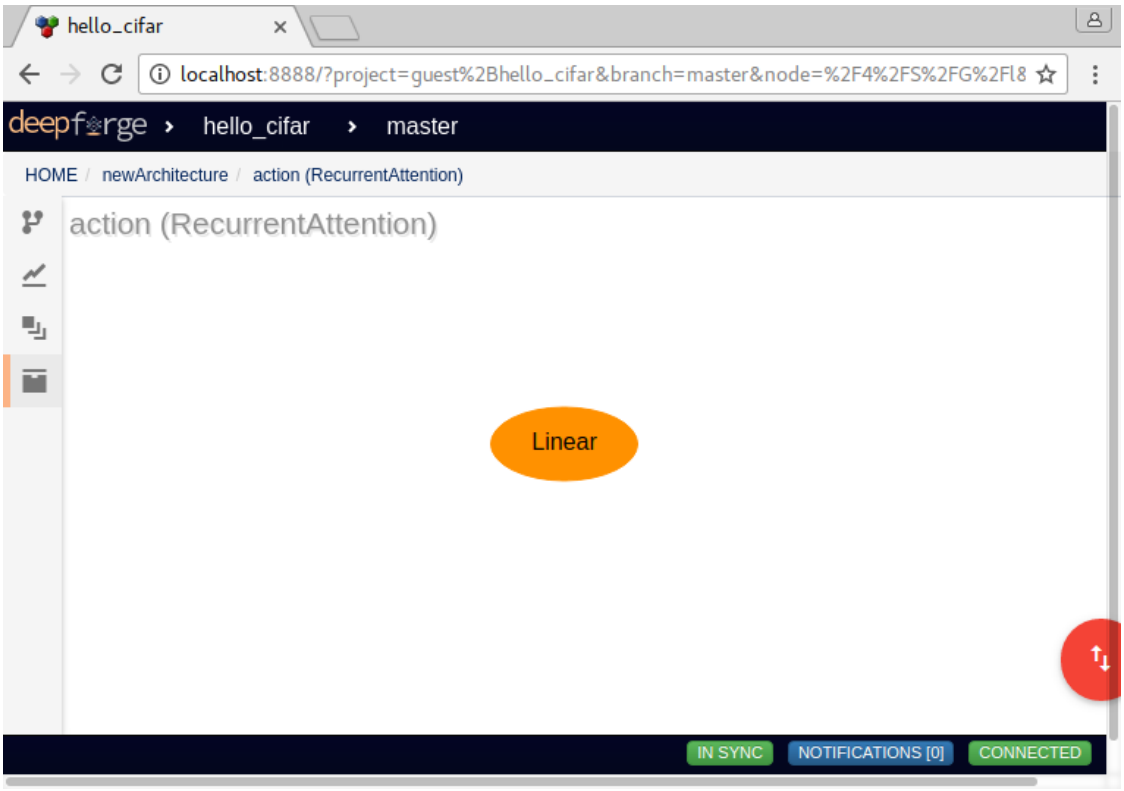

Fig. 5.2: Creating layer inputs for the "action" variable

```
  self.ceil_mode = false
  return self
end
```

Returning `self` in setter functions is a good convention when defining neural network layers in Torch7 as it promotes simple and legible code such as

```
net:add(nn.SpatialMaxPooling(5, 5, 2, 2):ceil())
```

where `net` is a container like a `Sequential` layer. DeepForge enforces this convention and, if it finds a setter function (which also returns `self`) in the layer definition will expose the internal variable (in this case `ceil_mode`) to the user in the visual editor.

CHAPTER 6

Custom Data Types

As operation inputs and outputs are strongly typed, DeepForge supports the creation of custom data types to promote flexibility when designing complex pipelines and operations. DeepForge data types can be either primitive types or custom classes. Custom DeepForge primitive types are relatively straight-forward; they can inherit from other types and must implement a serialization and deserialization methods (which may be as simple as `torch.save` and `torch.load`). Custom classes are also relatively simple to define but actually contain their own methods along with serialization and deserialization functions.

New data types can be defined from the operation editor from the dialog for selecting input or output data for the operation. After defining a new class, this class is available from within any of the operations in the DeepForge project.

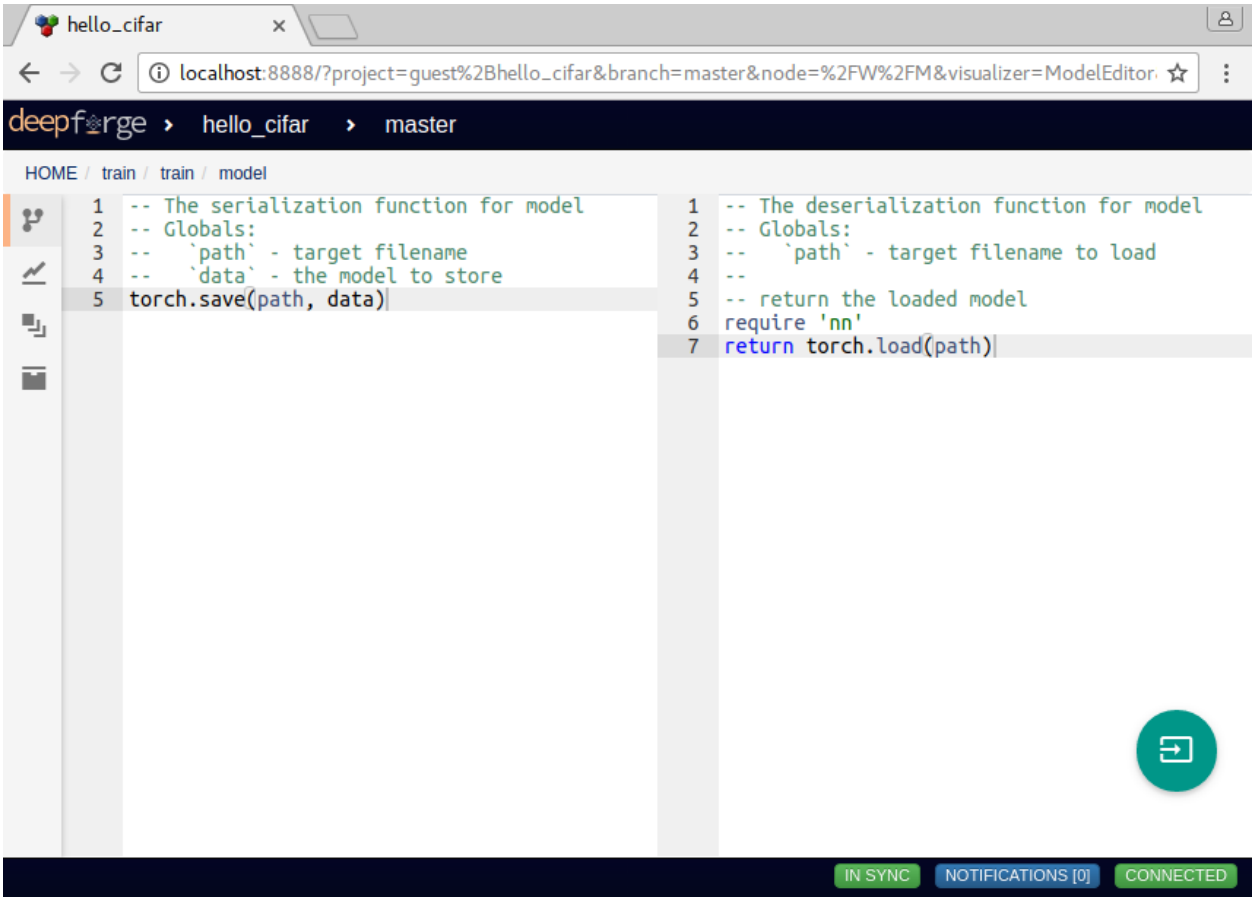

Fig. 6.1: Editing the serialization and deserialization for the “model” type

CHAPTER 7

Overview

DeepForge Component Overview

DeepForge is composed of four main elements:

- *Server*: Main component hosting all the project information and is connected to by the clients
- *Database*: MongoDB database containing DeepForge, job queue for the workers, etc
- *Worker*: Slave machine performing the actual machine learning computation
- *Client*: The connected browsers working on DeepForge projects.

Of course, only the *Server*, *Database* (MongoDB) and *Worker* need to be installed. If you are not going to execute any machine learning pipelines, installing the *Worker* can be skipped.

Component Dependencies

The following dependencies are required for each component:

- *Server* (NodeJS v6.2.1)
- *Database* (MongoDB v3.0.7)
- *Worker*: NodeJS v6.2.1 (used for job management logic) and Torch (this will be installed automatically by the cli when needed)
- *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.

Configuration

After installing DeepForge, it can be helpful to check out configuring DeepForge

CHAPTER 8

Native Installation

Dependencies

First, install NodeJS (v6) 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
```

DeepForge can now be started with:

```
deepforge start
```

However, the first time DeepForge is started, it will make sure that the deep learning framework is installed (if it isn't found on the host system). This may require you to start DeepForge a couple times; the first time it starts it will install Torch7 and require a terminal restart to update a couple environment variables (like *PATH*). The second time it starts it will install additional torch packages but will not require a terminal restart. Finally, DeepForge will start with all the required dependencies.

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
```

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
```

Worker

The DeepForge worker can be started with

```
deepforge start --worker
```

The worker will install dependencies the first time it is run (including torch, if it is not already installed).

To connect to a remote deepforge instance, add the url of the DeepForge server:

```
deepforge start --worker http://myaddress.com:1234
```

Updating

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

```
deepforge update
```

By default, this will update both DeepForge and the local torch installation. To only update DeepForge, add the *–server* flag:

```
deepforge update --server
```

For more update options, check out *deepforge update –help*!

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).

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*.

DeepForge Worker

If you are using *./bin/deepforge start -s* you will need to set up a DeepForge worker (*./bin/deepforge start* starts a local worker for you!). DeepForge workers are slave machines connected to DeepForge which execute the provided jobs. This allows the jobs to access the GPU, etc, and provides a number of benefits over trying to perform deep learning tasks in the browser.

Once DeepForge is installed on the worker, start it with

```
./bin/deepforge start -w
```

Note: If you are running the worker on a different machine, put the address of the DeepForge server as an argument to the command. For example:

```
./bin/deepforge start -w http://myaddress.com:1234
```

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*.

CHAPTER 9

Dockerized Installation

Each of the components are also available as docker containers. This page outlines the running of each of the main components as docker containers and connecting them as necessary.

Database

First, you can start the mongo container using:

```
docker run -d -v /abs/path/to/data:/data/db mongo
```

where `/abs/path/to/data` is the path to the mongo data location on the host. If running the database in a container, you will need to get the ip address of the given container:

```
docker inspect <container id>  | grep IPAddr
```

The `<container id>` is the value returned from the original `docker run` command.

When running mongo in a docker container, it is important to mount an external volume (using the `-v` flag) to be used for the actual data (otherwise the data will be lost when the container is stopped).

Server

The DeepForge server can be started with

```
docker run -d -v $HOME/.deepforge/blob:/data/blob \
-p 8888:8888 -e MONGO_URI=mongodb://172.17.0.2:27017/deepforge \
deepforge/server
```

where `172.17.0.2` is the ip address of the mongo container and `$HOME/.deepforge/blob` is the path to use for binary DeepForge data on the host. Of course, if the mongo instance is locating at a different location, `MONGO_URI` can be set to this address as well. Also, the first port (`8888`) can be replaced with the desired port to expose on the host.

Worker

As workers may require GPU access, they will need to use the nvidia-docker plugin. Workers can be created using

```
nvidia-docker run -d deepforge/worker http://172.17.0.1:8888
```

where `http://172.17.0.1:8888` is the location of the DeepForge server to which to connect.

Note: The `deepforge/worker` image is packaged with cuda 7.5. Depending upon your hardware and nvidia version, you may need to build your own docker image or run the worker natively.

CHAPTER 10

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
- Installing and Upgrading Torch
- Update or Uninstall DeepForge
- Managing Extensions

Installation Configuration

Installation configuration including the installation location of Torch7 and data storage locations. These 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 torch.dir
```

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

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

For more information about the configuration settings, check out the configuration page.

Starting DeepForge Components

DeepForge components, such as the server or the workers, can be started with the `deepforge start` command. By default, this command will start all the necessary components to run including the server, a mongo database (if applicable) and a worker.

The server can be started by itself using

```
deepforge start --server
```

The worker can be started by itself using

```
deepforge start --worker http://154.95.87.1:7543
```

where *http://154.95.87.1:7543* is the url of the deepforge server.

Installing and Upgrading Torch7

Torch7 is lazily installed when starting a worker (if torch isn't already installed) with the rnn package. This installation can be manually updated as described in the update and installation section.

Update/Uninstall DeepForge

DeepForge can be updated or uninstalled using

```
deepforge update
```

The torch installation can be updated using

```
deepforge update --torch
```

DeepForge can be uninstalled using `deepforge uninstall`

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
```

CHAPTER 11

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:
{
  "torch": {
    "dir": "/home/irishninja/.deepforge/torch"
  },
  "blob": {
    "dir": "/home/irishninja/.deepforge/blob"
  },
  "worker": {
    "cache": {
      "useBlob": true,
      "dir": "~/.deepforge/worker/cache"
    },
    "dir": "~/.deepforge/worker"
  },
  "mongo": {
    "dir": "~/.deepforge/data"
  }
}
```

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

```
deepforge config worker.cache.dir /tmp
```

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 a worker with a

different cache than set in *worker.cache.dir* can be done with:

```
DEEPFORGE_WORKER_CACHE=/tmp deepforge start -w
```

The complete list of the environment variable overrides for the configuration options can be found here.

Settings

torch.dir

The path to the local installation of torch to be used by the deepforge worker. This is used when installing, upgrading and removing the local torch installation

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.

worker.dir

The path to the directory used for worker executions. The workers will run the executions from this directory.

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

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.

worker.cache.dir

The path to the worker cache directory.

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

worker.cache.useBlob

When running the worker on the same machine as the server, this allows the worker to use the blob as a cache and simply create symbolic links to the data (eg, training data, models) to prevent having to even perform a copy of the data on the given machine.

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

CHAPTER 12

Operation Feedback

DeepForge provides the *deepforge* global object in operation implementations for providing feedback during the execution. The various types of metadata are provided and discussed below.

Graphs

Real-time graphs can be created using the graph constructor:

```
local graph = deepforge.Graph('My Graph')  -- created a new graph called "My Graph"
```

After creating a graph, lines can be added similarly.

```
local line1 = graph:line('first line')  -- created a new line called "first line"
local line2 = graph:line('second line')  -- created a second line called "second line"
```

Finally, points can be added to the lines by calling the *:add* method on the line and passing the x and y values for the given point.

```
line1:add(1, 3)  -- adding point (1, 3) to line1
line2:add(1, 4)  -- adding point (1, 4) to line2

line1:add(2, 5)  -- adding point (2, 5) to line1
line2:add(2, 6)  -- adding point (2, 6) to line2
```

Graphs can then label their axis as follows:

```
graph:xlabel('x axis')  -- label the x axis "x axis"
graph:ylabel('y axis')  -- label the y axis "y axis"
```

Images

Images can be created using:

```
local image = deepforge.Image('My Example Image', imageTensor)
```

The first argument is the title of the image and the second argument is the tensor for the image (optional). Both the title and the tensor can be updated during execution as follows.

```
image:title('My New Title')  -- updating the image title
image:update(newTensor)  -- updating the displayed image
```