studioml Documentation

studioml

Oct 24, 2018

Introduction

Github | pip |

Studio is a model management framework written in Python to help simplify and expedite your model building experience. It was developed to minimize the overhead involved with scheduling, running, monitoring and managing artifacts of your machine learning experiments. No one wants to spend their time configuring different machines, setting up dependencies, or playing archeologist to track down previous model artifacts.

Most of the features are compatible with any Python machine learning framework (Keras, TensorFlow, PyTorch, scikit-learn, etc) without invasion of your code; some extra features are available for Keras and TensorFlow.

Use Studio to:

- Capture experiment information- Python environment, files, dependencies and logs- without modifying the experiment code. Monitor and organize experiments using a web dashboard that integrates with TensorBoard.
- Run experiments locally, remotely, or in the cloud (Google Cloud or Amazon EC2)
- Manage artifacts
- Perform hyperparameter search
- Create customizable Python environments for remote workers.

Studio is a model management framework written in Python to help simplify and expedite your model building experience. It was developed to minimize the overhead involved with scheduling, running, monitoring and managing artifacts of your machine learning experiments. No one wants to spend their time configuring different machines, setting up dependencies, or playing archeologist to track down previous model artifacts.

Most of the features are compatible with any Python machine learning framework (Keras, TensorFlow, PyTorch, scikit-learn, etc); some extra features are available for Keras and TensorFlow.

Use Studio to:

- Capture experiment information- Python environment, files, dependencies and logs- without modifying the experiment code.
- Monitor and organize experiments using a web dashboard that integrates with TensorBoard.
- Run experiments locally, remotely, or in the cloud (Google Cloud or Amazon EC2)
- Manage artifacts
- Perform hyperparameter search
- Create customizable Python environments for remote workers.

NOTE: `studio` package is compatible with Python 2 and 3!

CHAPTER 1

Example usage

Start visualizer:

```
studio ui
```

Run your jobs:

```
studio run train_mnist_keras.py
```

You can see results of your job at http://localhost:5000. Run `studio {ui|run} --help` for a full list of ui / runner options. WARNING: because studio tries to create a reproducible environment for your experiment, if you run it in a large folder, it will take a while to archive and upload the folder.

CHAPTER 2

Installation

pip install studioml from the master pypi repositry:

```
pip install studioml
```

Find more details on installation methods and the release process.

CHAPTER 3

Authentication

Currently Studio supports 2 methods of authentication: email / password and using a Google account. To use studio runner and studio ui in guest mode, in studio/default_config.yaml, uncomment "guest: true" under the database section.

Alternatively, you can set up your own database and configure Studio to use it. See setting up database. This is a preferred option if you want to keep your models and artifacts private.

CHAPTER 4

Further reading and cool features

CHAPTER 5

Installation

5.1 Installation Packaging

pip install `studioml` from the master pypi repositry:

```
pip install studioml
```

or, install the source and development environment for Studio from the git project directory:

```
git clone https://github.com/studioml/studio && cd studio && pip install -e .
```

A setup.py is included in the top level of the git repository to allow the creation of tar archives for installation on runners and other systems where git is not the primary means of handling Python artifacts. To create the installable, use the following command from the top level directory of a cloned repository:

```
python setup.py sdist
```

This command will create a file dist/studio-x.x.tar.gz that can be used with pip as follows:

```
pip install studio-x.x.tar.gz
```

Certain types of runners can make use of the Studio software distribution to start projects without any intervention, i.e. devops-less runners. To include the software distribution, add the tar.gz file to your workspace directory under a dist subdirectory. Runners supporting software distribution will unroll the software and install it using virtualenv.

We recommend setting up a virtual environment.

5.2 CI/CD pipeline

The Studio project distributes official releases using a travis based build and deploy pipeline. The Travis project that builds the official github repository for Studio has associated encrypted user and password credentials that the Travis .yml file refers to. These secrets can be updated using the Travis configuration found at https://travis-ci.com/

SentientTechnologies/studio/settings. The PYPI_PASSWORD and PYPI_USER variables should point at an owner account for the project. To rotate these values, remove the old ones using the settings page and re-add the same variables with new values.

When code is pushed to the master branch in the github repository, a traditional build will be performed by Travis. To push a release after the build is complete, add a server compatible version number as a tag to the repository and do a 'git push –tags' to trigger the deployment to pypi. Non-tagged builds are never pushed to pypi. Any tag will result in a push to pypi, so care should be taken to manage the visible versions using the PYPI_USER account.

5.3 Release process

Studio is released as a binary or source distribution using a hosted package at pypi.python.org. To release Studio, you must have administrator role access to the Studio Package on the https://pypi.python.org/ web site. Releases are done using the setup packaging found inside the setup.py files.

When working with the pypi command line tooling you should create a ~/.pyirc file with your account details, for example:

```
[distutils]
index-servers=
    pypi
    testpypi

[testpypi]
repository = https://testpypi.python.org/pypi
username = {your pipy account}
password = {your password}

[pypi]
username = {your pipy account}
```

The command to push a release is as follows.

```
python setup.py sdist upload
```

If you wish to test releases and not pollute our pypi production release train and numbering, please use the '-r' option to specify the test pypi repository. pypi releases are idempotent.

5.4 Running tests

To run the unit and regression tests, run

```
python $(which nosetests) --processes=8 --process-timeout=600
```

Note that simply running `nosetests` tends to not use virtualenv correctly. If you have application credentials configured to work with distributed queues and cloud workers, those will be tested as well. Otherwise, such tests will be skipped. The total test runtime, when run in parallel as in the command above, should be no more than 10 minutes. Most of the tests are I/O limited, so parallel execution speeds up things quite a bit. The longest test is the gpu cloud worker test in EC2 cloud (takes about 500 seconds due to installation of the drivers / CUDA on the EC2 instance).

CHAPTER 6

Authentication

Currently, Studio uses GitHub auth for authentication. For command-line tools (studio ui, studio run etc) the authentication is done via personal access tokens. When no token is present (e.g. when you run studio for the first time), you will be asked to input your github username and password. Studio DOES NOT store your username or password, instead, those are being sent to GitHub API server in exchange to an access token. The access token is being saved and used from that point on. The personal access tokens do not expire, can be transferred from one machine to another, and, if necessary, can be revoked by going to GitHub -> Settings -> Developer Settings -> Personal Access Tokens

Authentication for the hosted UI server (https://zoo.studio.ml) follows the standard GitHub Auth flow for web apps.

CHAPTER 7

Command-line interface

In some cases, a (semi-)programmatic way of keeping track of experiments may be preferred. On top of the Python and HTTP API, we provide a command-line tool to get a quick overview of exising experiments and take actions on them. Commands available at the moment are:

- `studio runs list users` - lists all users
- `studio runs list projects` - lists all projects
- `studio runs list [user]` - lists your (default) or someone else's experiments
- `studio runs list project <project>` - lists all experiments in a project
- `studio runs list all` - lists all experiments
- `studio runs kill <experiment>` - deletes experiment
- `studio runs stop <experiment>` - stops experiment

Note that for now if the experiment is running, killing it will NOT automatically stop the runner. You should stop the experiment first, ensure its status has been changed to stopped, and then kill it. This is a known issue, and we are working on a solution.

CHAPTER 8

Artifact management

This page describes facilities that Studio provides for management of experiment artifacts. For now, artifact storage is backed by Google Cloud storage.

8.1 Basic usage

The goal of artifact management is three-fold:

1. With no coding overhead capture data the experiment depends on (e.g. dataset).
2. With no coding overhead save, and with minimal overhead visualize, the results of the experiment (neural network weights, etc).
3. With minimal coding overhead make experiments reproducible on any machine (without manual data download, path correction etc).

Below we provide the examples of each use case.

8.1.1 Capture data

Let's imagine that file `train_nn.py` in the current directory trains a neural network based on data located in `~/data/`. In order to capture the data, we need to invoke `studio run` as follows:

```
studio run --capture-once=~/data:data train_nn.py
```

Flag `--capture-once` (or `-co`) specifies that data at path ~/data needs to be captured once at experiment startup. Additionally, the tag `data` (provided as a value after `:`) allows the script to access data in a machine-independent way, and also distinguishes the dataset in the web-ui (the Web UI page of the experiment will contain download link for tar-gzipped folder `~/data`).

8.1.2 Save the result of the experiment

Let's now consider an example of a python script that periodically saves some intermediate data (e.g. weights of a neural network). The following example can be made more concise using the keras or tensorflow built-in checkpointers, but we'll leave that as an exercise for the reader. Consider the following contents of file `train_linreg.py` (also located in `studio/examples/general/` in repo):

```
import numpy as np
import pickle
import os

no_samples = 100
dim_samples = 5

learning_rate = 0.01
no_steps = 10

X = np.random.random((no_samples, dim_samples))
y = np.random.random((no_samples,))

w = np.random.random((dim_samples,))

for step in range(no_steps):
    yhat = X.dot(w)
    err = (yhat - y)
    dw = err.dot(X)
    w -= learning_rate * dw
    loss = 0.5 * err.dot(err)

    print("step = {}, loss = {}, L2 norm = {}".format(step, loss, w.dot(w)))

    with open(os.path.expanduser('~/weights/lr_w_{}_{}.pck'.format(step, loss)), 'w')␣
↪as f:
        f.write(pickle.dumps(w))
```

The reader can immediately see that we are solving a linear regression problem by gradient descent and saving weights at each step to ~/weights folder.

In order to simply save the weigths, we can run the following command:

```
studio run --capture=~/weights:weights train_linreg.py
```

Flag `--capture` (or `-c`) specifies that data from folder `~/weights` needs to be captured continously - every minute (the frequency can be changed in a config file), and at the end of the experiment. In the Web ui page of the experiment we now have a link to weights artifact. This simple script should finish almost immediately, but for longer running jobs upload happens every minute of runtime (the upload happens in a separate thread, so this should not slow down the actual experiment).

8.1.3 Machine-independent access to the artifacts

So far we have been assuming that all experiments are being run on a local machine, and the only interaction with artifacts has been to save them for posterity's sake. But what if our experiments are growing a bit too big to be run locally? Fortunately, Studio comes with a dockerized worker that can run your jobs on a beefy gpu server, or on a cloud instance. But how do we make local data available to such a worker? Clearly, a local path along the lines of `/Users/john.doe/weights/` will not always be reproducible on a remote worker. Studio provides a way to access files in a machine-independent way, as follows. Let us replace the last three lines in the script above by:

```
from studio import fs_tracker
with open(os.path.join(fs_tracker.get_artifact('weights'),
                       'lr_w_{}_{}.pck'.format(step, loss)),
          'w') as f:
    f.write(pickle.dumps(w))
```

We can now run the script locally, the exact same way as before:

```
studio run --capture=~/weights:weights train_linreg.py
```

Or, if we have a worker listening to the queue `work_queue`:

```
studio run --capture=~/weights:weights --queue work_queue train_linreg.py
```

In the former case, the call `fs_tracker.get_artifact('weights')` will simply return `os.path.expanduser('~/weights')`. In the latter case, a remote worker will set up a cache directory that corresponds to the artifact tagged as `weights` and copy existing data from storage into it (so that data can be read from that directory as well). The call `fs_tracker.get_artifact('weights')` will return the path to that directory. In both cases, the `--experiment` flag is not mandatory; if you don't specify a name, a random uuid will be generated.

8.1.4 Re-using artifacts from other experiments

A neat side-benefit of using machine-independent access to the artifacts is the ability to plug different datasets into an experiment without touching the script at all - simply provide different paths for the same tag in the `--capture(-once)` flags. More importantly, one can reuse datasets (or any artifacts) from another experiment using the `--reuse` flag. First, let's imagine we've run the `train_linreg.py` script, this time giving the experiment a name:

```
studio run --capture=~/weights:weights --experiment linear_regression train_linreg.py
```

Say we now want to print the L2 norm of the last set of weights. Let's consider the following script (`print_norm_linreg.py`):

```
import glob
import os
from studio import fs_tracker
import pickle


weights_list = glob.glob(os.path.join(fs_tracker.get_artifact('w'),'*.pck'))
weights_list.sort()

print('*****')
print(weights_list[-1])
with open(weights_list[-1], 'r') as f:
    w = pickle.load(f)

print w.dot(w)
print('*****')
```

We can run it via

```
studio run --reuse=linear_regression/weights:w print_norm_linreg.py
```

The flag `reuse` tells `studio run` that artifact `weights` from experiment `linear_regression` will be used in the current experiment with a tag `w`. There is a bit of a catch - for download optimization, all artifacts from other

experiments are considered immutable, and cached as such. If you re-run the experiment with the same name and would like to use new artifacts from it, clean the cache folder `~/.studioml/blobcache/`.

8.2 Default artifacts

Each experiment gets default artifacts that it can use via `fs_tracker.get_artifact()` even without the `--reuse` or `--capture(-once)` flags. Those are:

1. `workspace`- this artifact always gets cached to/from `.` folder, thus creating a copy of the working directory on a remote machine and saving the state of the scripts
2. `output`- this artifact is a file with the stdout and stderr produced by running the script
3. `modeldir`- it is recommended to save weights to this directory because Studio will try to do some analysis on it, such as count the number of checkpoints etc.
4. `tb`- it is recommended to save Tensorboard logs to this directory, this way Studio will be able to automatically feed them into Tensorboard

All of the default artifacts are considered mutable (i.e. are stored continously). The default artifacts can be overwritten by –capture(-once) flags.

8.3 Ignoring Files

By placing an .studioml_ignore file inside the directory of the script invoked by studio run, you can specify certain directories or files to avoid being uploaded. These files will not exist in the workspace directory when the script is running remotely.

8.4 Custom storage

The Firebase API is great for small projects, but it is easy to grow beyond its free storage limits (5 Gb as of 08/02/2017), after which it becomes very expensive. Studio can utilize Google Cloud storage directly for artifact storage if your projects don't fit into Firebase (support for Amazon S3 is on the way).

For now, the downside of using Google Cloud storage is that Google service account credentials are used, which means that all users in possession of the credential's file have read/write access to the objects in the storage, so in principle one user can delete the experiments of another. See here for instructions on how to generate service account credentials. Once you have generated a credentials file, uncomment the "storage" section in your config.yaml file, set the type of storage to `gcloud`, and specify a storage bucket. Note that the bucket name needs to be unique, and an error will be thrown if a bucket with that name cannot be created. The safest approach is to create a bucket manually from the Google Cloud console, and then specify it in config.yaml. Folder/file structure within the bucket is the same as for Firebase storage, so if you want to migrate all your Firebase experiments to the new storage you can copy the Firebase storage bucket and point config.yaml to the copy (you could point config.yaml to the original, but then you'll be paying the same Firebase prices).

CHAPTER 9

Hyperparameter search

This page describes facilities that Studio provides for hyperparameter search and optimization.

9.1 Basics

For now, Studio can launch a batch of experiments using regex substitution of variables inside the script. These experiments are launched in a separate project, and can be compared in tensorboard or by the value of some scalar metrics reported in tensorboard logs.

Consider the following code snippet (code in here):

```
lr=0.01
print('learning rate = {}'.format(lr))
model.compile(loss='categorical_crossentropy', optimizer=optimizers.SGD(lr=lr))


# some data preparation code


tbcallback = TensorBoard(log_dir=fs_tracker.get_tensorboard_dir(),
                 histogram_freq=0,
                 write_graph=True,
                 write_images=False)

model.fit(
    x_train, y_train, validation_data=(
        x_test,
        y_test),
    epochs=int(sys.argv[1]),
    callbacks=[tbcallback])
```

We compile a keras model with the specified learning rate for stochastic gradient descent. What if we want to search a range of learning rates to determine the best value? (as a side note, in the `train_mnist_keras.py` you can simply

use an adaptive learning rate optimizer such as adam to get better results, but let's forget about that for demonstration purposes)

We can add the following argument to `studio run` call:

```
studio run --hyperparam=lr=0.01:0.01:0.1 train_mnist_keras.py 30
```

This will create a new project with 10 experiments. For each experiment, a copy of the working directory will be put in the `studioml` cache, and within each copy of the script `train_mnist_keras.py` regex substitution of `lr` not followed by = (i.e. located in right-hand side of an expression) will be performed to for values from 0.01 to 0.1 with a step size of 0.01. Those experiments will then be submitted to the queue (to the local queue in the version of the call above) and executed. The progress of the experiments can be seen in the Studio WebUI. The last argument `30` refers to number of training epochs, as can be seen in the code snippet above.

9.2 Metrics

But wait, do you have to go and check each experiment individually to figure out which one has done best? Wouldn't it be nice if we could look at the project and immediately figure out which experiments have done better than the others? There is indeed such a feature. We can specify an option `--metric` to `studio run` to specify which tensorflow / tensorboard variable to report as a metric, and how to accumulate it throughout experiment. For keras experiments, that would most often be `val_loss`, but in principle any scalar reported in tensorboard can be used. Note that tensorboard logs need to be written for this feature to work. Let's modify the command line above a bit:

```
studio run --hyperparam=lr=0.01:0.01:0.1 --metric=val_loss:min train_mnist_keras.py
```

This will report the smallest value of `val_loss` so far in the projects page or in the WebUI dashboard. Together with the column sorting feature of the dashboard you can immediately figure out the best experiment. The reason why this option is given to the runner and not in the WebUI after the run is because we are planning to incorporate more complicated hyperparameter search where new experiments actually depend on previously seen values of metric. Other allowed values for the `--metric` parameter suffix are ":max" for maximum value seen throughout experiment, or empty for the last value.

9.3 Specifying hyperparameter ranges for grid search

Scanning learning rate in constant steps is not always the best idea, especially if we want to cover several orders of magnitude. We can specify range with a log step as follows:

```
--hyperparam=lr=1e-5:l5:0.1
```

which will make 10 steps spaced logarithmically between 1e-5 and 0.1 (that is, 1e-5, 1e-4, 1e-3, 0.01, 0.1 - matlab style rather than numpy) Other options are:

1. `lr=1e-5:10:0.1` or `lr=1e-5:u10:0.1` will generate a uniformly spaced grid from 1e-5 to 0.1 (bad idea - the smaller end of the range will be spaced very coarsely)
2. `no_layers=0:3` or `nolayers=:3` will generate uniformly spaced grid with a step 1 (0,1,2,3 - endpoints are handled in matlab style, not numpy style)
3. `lr=0.1` will simply substitute lr with 0.1
4. `no_layers=2,5,6` will generate three values - 2,5 and 6

Note that option `--hyperparam/-hp` can be used several times for different hyperparameters; however, keep in mind that grid size grows exponentially with number of hyperparameters to try.

9.4 Specifying hyperparameter ranges for plugin optimizers

Plugin optimizers are also supported. They can be enabled with the `-opt/--optimizer` flag. Either the name of the optimizer (or its file path) can be specified as argument. The format is slightly different for plugin optimizers. Below are some examples:

```
--hyperparam=lr=0:1:10:urla
```

```
--hyperparam=lr=0:5:l
```

```
--hp=lr=5:5:10:alu
```

The general format is [min range]:[max range]:{array length}:{flags}, where {array length} and {flags} are optional arguments. The following flags are supported:

1. {u}: whether or not to constrain hyperparameters to [min range]:[max range] (default is constrained).
2. {r}: whether to initialize hyperparameters with random value between [min range]:[max range] or right in the middle (default is nonrandom).
3. {l}: whether to use log scaling for the hyperparameter (default is nonlog).
4. {a}: whether the hyperparameter is a numpy array or a scalar. If the hyperparameter is a numpy array, then the {array length} field must be present as well (default is scalar).

In addition, the python script whose hyperparameters are being optimized must contain a line with the fitness printed to stdout as shown below. For hyperparameters whose contents are numpy arrays, they must be loaded using the `fs_tracker.get_artifact` function call as shown below:

```
from studio import fs_tracker

lr = np.load(fs_tracker.get_artifact('lr'))

print "fitness: %s" % np.sum(lr)
```

9.5 Cloud workers

Waiting till your local machine runs all experiments one after another can be time consuming. Fortunately, we can outsource the compute to Google Cloud or Amazon EC2. Please refer to this page for setup instructions; all the custom hardware configuration options can be applied to the hyperparameter search as well.

```
studio run --hyperparam=lr=0.01:0.01:0.1 --metric=val_loss:min --cloud=gcloud --num-
↪workers=4 train_mnist_keras.py
```

will spin up 4 cloud workers, connect the to the queue and run experiments in parallel. Beware of spinning up too many workers - if a worker starts up and finds that everything in the queue is done, it will (for now) listen to the queue indefinitely waiting for the work, and won't shut down automatically.

CHAPTER 10

Trained model pipelines

Both Keras and TensorFlow handle data that has been packaged into tensors gracefully. However, real-world data is often not properly formatted, especially at the time of prediction. It may come as a collection (a list, set, or generator) of urls, have non-numeric metadata that needs to be converted into tensor format, etc. Some of the data may actually be missing or cause exceptions in preprocessing / prediction stages.

Keras provides some features for image preprocessing to address this issue, and TensorFlow has functions to include non-tensor values into the computational graph. Both approaches have limitations - neither of them can handle missing data while retaining performance. As a concrete example, let us consider the following (which is a basis of the unit test `ModelPipeTest.test_model_pipe_mnist_urls` in studio/tests/model_util_test.py):

We are training a network to classify mnist digits, and then trying to predict images from urls. The simplest way to achieve this would be:

```
from PIL import Image
from io import BytesIO
import urlllib
import numpy as np

from studio import model_util

# setup and train keras model
#
# urls is a list of urls

labels = []
for url in urls:
    data = urllib.urlopen(url).read()
    img = Image.open(BytesIO(data))

    # resize image to model input, and convert to tensor:
    img_t = model_util.resize_to_model_input(model)(img)

    labels.append(np.argmax(model.predict(img_t)))
```

The function `model_util.resize_to_model_input(model)` is used for brevity and performes conversion

of an image into a tensor with values between 0 and 1, and shaped accoring to model input size. `np.argmax` in the last line is used to convert output class probablilities into the class label.

Ok, so why do we need anything else? The code above has two problems: 1) it does not handle exceptions (though that is easy to fix) and 2) it processes the data sequentially. To address 2, we could have spawned a bunch of child processes, and if the model is evaluated on a CPU that probably would have been ok. But modern neural networks get a substantial speed boost from using GPUs that don't do well with hundreds of processes trying to run different instructions. To leverage GPU speedup, we'll need to create batches of data and feed them to the GPU (preferrably in parallel with urls being fetched).

Keras offers a built in mechanism to do this: `keras.models.predict_generator`. The relevant part of the code above can then become:

```
labels = []
batch_size = 32
tensor_generator = (model_util.resize_to_model_input(model)(Image.open(BytesIO(urllib.
↪urlopen(url).read())))  for url in urls)
output = model.predict_generator(tensor_generator, batch_size = batch_size, num_
↪workers=4, no_batches = len(urls) / batch_size)

for out_t in output:
    labels = np.stack(labels, np.argmax(out_t, axis=1))
```

We are handling de-batching implicitly when doing stacking of the labels from different batches. This code will spin up 4 workers that will read from the `tensor_generator` and prepare the next batch in the background at the same time as the heavy lifting of prediction is handled by GPU. So is that good enough? Not really. Rememeber our problem 1) - what if there is an exception in the pre-processing / url is missing etc? By default the entire script will come to a halt at that point. We could filter out the missing urls by passing an exception-handling function into the generator, but filtering out bad values will ruin the mapping from url to label, rendering values after exception just as useless as if the script were to stop.

The least ugly solution using Keras is to add another input to the model, so that model applies to a key:tensor value; and then after prediction sort out which ones were successfull. But this process really doesn't have to be that complicated.

Studio provides primitives that make this job (that is conceptually very simple) simple in code; and similar in usage to Keras. The code above becomes (see unit test `ModelPipeTest.test_model_pipe_mnist_urls` in `studio/tests/model_util_test.py`)

```
pipe = model_util.ModelPipe()
pipe.add(lambda url: urllib.urlopen(url).read(), num_workers=4, timeout=5)
pipe.add(lambda img: Image.open(BytesIO(img)))
pipe.add(model_util.resize_to_model_input(model))
pipe.add(lambda x: 1-x)
pipe.add(model, num_workers=1, batch_size=32, batcher=np.vstack)
pipe.add(lambda x: np.argmax(x, axis=1))

output_dict = pipe({url:url for url in urls})
```

This runs the preprocessing logic (getting the url, converting it to an image, resizing the image, and convering to a tensor) using 4 workers that populate the queue. The prediction is run using 1 worker with batch size 32 using the same queue as input. Conversion of class probabilities to class labels is also now a part of the model pipeline. In this example the input is a dictionary mapping url to url. The functions will be applied only to the values, so the output will become url: label. The pipeline can also be applied to lists, generators and sets, in which case it returns the same type of collection (if the input was list, it returns list etc) with tuples (index, label) If any step of the preprocessing raises an exception, it is caught, and corresponding output is filtered out. We are using additional function `lambda x: 1-x` because in the mnist dataset the digits are white on black background, whereas in the test urls digits are black on white background.

The timeout parameter controls how long the workers wait if the queue is empty (e.g. when urls take too long to fetch). Note that this also means that the call `pipe()` will not return for a number of seconds specified by the last (closest to output) timeout value.

Note that `pipe.add()` calls that don't specify a number of workers, timeout, batch_size, or batcher (function to assemble list of values into a batch digestable by a function that operates on batches) are composed with the function in previous calls to `pipe.add()` directly, so that there is no unnecessary queues / buffers / workers.

10.1 Benchmark

For a benchmark, we use StyleNet inference on a dataset of 7k urls, some of which are missing / broken. The benchmark is being run using EC2 p2.xlarge instances (with nVidia Tesla K80 gpus). The one-by-one experiment is running inference one image at a time, pipe is using model pipe primitives as described above. Batch size is number of images being processed as a single call to `model.predict`, and `workers` is number of prefetching workers

Experiment	Time (s)	Time per url (s)
One-by-one	6994	~ 0.98
Pipe (batch 64, workers 4)	1581	~ 0.22
Pipe (batch 128, workers 32)	157	~ 0.02

CHAPTER 11

Setting up a database and API server

This page describes the process of setting up your own database / storage for the models. This puts you in full control of who has access to the experiment data. For the moment, Studio only supports Firebase (https://firebase.google.com/) as a database backend, and firebase / google cloud storage (GCS) / Amazon S3 as storage backends.

11.1 Introduction

Firebase and Firebase Storage provide fairly simple rules to control use access to experiments. Additionally, GCS and S3 don't work directly with Firebase authentication tokens, so one cannot create access rules for the storage. In order to provide more rigorous rules, we are employing an API server that proxies database requests and can provide arbitrarily complex access rules (at the moment the access rules are still very simple - anyone can read any experiment, and only user who created the experiment can delete / overwrite it). Also, API server should allow one to swap database backends (i.e. from Firebase to DynamoDB) completely seamlessly for the users, without even updating the users' config files. Yet another reason to use the API server is that GCS and S3 are much cheaper than Firebase Storage for large amounts of data.

Generally, the outline of the API server / database / storage interaction is as follows:

1. API server has read/write access to database and storage
2. When getting/writing the data about experiment, user signs the HTTP request with firebase authentication token. The API server then validates that user indeed has permissions to do so, and either returns data about experiment (for /api/get_experiment method) or writes the expeirment data
3. Artifacts are being read and written via communicating with storage directly using signed urls, generated by API server

The detailed instructions on setting up the API server (we'll use google app engine, GAE, but these steps can be trivially adapted for heroku or just running API server on a dedicated instance)

11.1.1 Prerequisites

If deploying onto google app engine, you'll need to have Google Cloud SDK installed (https://cloud.google.com/sdk/downloads)

In what follows, deployment machine means either the local machine (when deploying on GAE) or the instance on which you are planning to run the API server

11.1.2 Deploying the API server

1. Create a new Firebase project: go to https://firebase.google.com, sign in, click add project, specify project name
2. Enable authentication using google accounts (in the left-hand pane select Authentication, then tab "Sign-in method", click on "Google", select "Enabled")
3. Go to project settings (little cogwheel next to "Overview" on the left-hand pane), tab "General"
4. Copy the Web API key and paste it in apiKey of the database section of studio/apiserver_config.yaml
5. Copy the project ID and paste it in projectId of the database section of config yaml file.
6. Go to Service Accounts tab and generate a new key for the firebase service account. This key is a json file that will give API server admin access to the database. Save it to the deployment machine.
7. Modify other entries of the apiserver_config.yaml file to your specs (e.g. storage type and bucket)
8. On the deployment machine in the folder studio/studio, run

   ```
   ./deploy_apiserver.sh gae
   ```

   for GAE and

   ```
   ./deploy_apiserver.sh local <port>
   ```

   when running on a dedicated instance (where port is the port on which the server will be listening). When prompted, input path to the firebase admin credentials json file generated in step 6.

11.1.3 Configuring studio to work with the API server

For clients to work with the API server, you'll need to modify their config.yaml files as follows:

1. Remove storage section
2. In the database section, set type: http, serverUrl: <url of your deployed server>. When deploying to GAE, the url will have format https://<project_name>.appspot.com. When deploying on a dedicated instance, don't forget to specify the port.

CHAPTER 12

Setting up a remote worker

This page describes a procedure for setting up a remote worker for Studio. Remote workers listen to the queue; once a worker receives a message from the queue, it starts the experiments.

12.1 Getting credentials

1. Remote workers work by listening to a distributed queue. Right now the distributed queue is backed by Google PubSub, so to access it you'll need application credentials from Google (in the future, it may be implemented via Firebase itself, in which case this step should become obsolete). If you've made it this far, you are likely to have a Google Cloud Compute account set up, but if not, go to http://cloud.google.com and either set up an account or sign in.
2. Next, create a project if you don't have a project corresponding to Studio just yet.
3. Then go to API Manager -> Credentials, and click "Create credentials" -> "Service account key"
4. Choose "New service account" from the "Select accout" dropdown, and keep key type as JSON.
5. Enter a name of your liking for the account (Google will convert it to a unique name), and choose "PubSub Editor" for a role (technically, you can create 2 keys, and keep the publisher on a machine that submits work, and subscriber key on a machine that implements the work). If you are planning to use cloud workers, it is also recommended to add Compute Engine / Compute Engine Admin (v1).
6. Save a json credentials file. It is recommended that the credential file be saved in a safe location such as your ~/.ssh directory and that you use the 'chmod 0600 file.json' command to help secure the file within your Linux user account.
7. Add the `GOOGLE_APPLICATION_CREDENTIALS` variable to the environment that points to the saved json credentials file both on the work submitter and work implementer.

12.2 Enabling Google PubSub for the Google Application

In order to use Google queues for your own remote workers, as opposed to the Google Cloud Platform workers, PubSub API services will need to be enabled for the project. To do this go to the Google API Manager Dashboard within the Google Cloud Platform console and select the Enable API drop down, which is located at the top of the Dashboard with a '+' icon beside it. From here you will see a panel of API services that can be enabled, choose the PubSub API. In the PubSub Dashboard there is an option to enable the API at the top of the Dashboard.

12.3 Setting up remote worker

If you don't have your own docker container to run jobs in, follow the instructions below. Otherwise, jump to the next section.

1. Install docker, and nvidia-docker to use gpus
2. Clone the repo

   ```
   git clone https://github.com/ilblackdragon/studio && cd studio && pip install -e .
   ```

   To check the success of the installation, you can run `python $(which nosetests) --processes=10 --process-timeout=600` to run the tests (may take about 10 min to finish)
3. Start the worker (queue name is a name of the queue that will define where submit work to)

   ```
   studio start remote worker --queue=<queue-name>
   ```

12.4 Setting up a remote worker with exising docker image

This section applies when you already have a docker image/container and would like the Studio remote worker to run inside it.

1. Make sure that the image has python-dev, python-pip, and git installed, as well as Studio. The easiest way is to make your Dockerfile inherit from from the Studio Dockerfile (located in the Studio root directory). Otherwise, copy relevant contents of Studio Dockerfile into yours.
2. Bake the credentials into your image. Run

   ```
   studio add credentials [--base_image=<image>] [--tag=<tag>] [--check-gpu]
   ```

   where `<image>` is the name of your image (default is peterzhokhoff/studioml); `<tag>` is the tag of the image with credentials (default is `<image>_creds`). Add option check-gpu if you are planning to use image on the same machine you are running the script from. This will check for presence of the CUDA toolbox and uninstall tensorflow-gpu if not found.
3. Start the remote worker passing `--image=<tag>`:

   ```
   studio start remote worker --image=<tag> --queue=<queue-name>
   ```

   You can also start the container and remote worker within it manually, by running:

   ```
   studio remote worker --queue=<queue-name>
   ```

   within the container - this is essentially what the `studio-start-remote-worker` script does, plus mounting cache directories `~/.studioml/experiments` and `~/.studioml/blobcache`

12.5 Submitting work

On a submitting machine (usually local):

```
studio run --queue <queue-name> <any_other_args> script.py <script_args>
```

This script should quit promptly, but you'll be able to see experiment progress in the Studio WebUI.

CHAPTER 13

Custom environments

13.1 Using custom environment variables at runtime

You can add an env section to your yaml configuration file in order to send environment variables into your runner environment variables table. Variables can be prefixed with a $ sign if you wish to substitute local environment variables into your run configuration. Be aware that all values are stored in clear text. If you wish to exchange secrets you will need to encrypt them into your configuration file and then decrypt your secrets within your python code used during the experiment.

13.2 Customization of python environment for the workers

Sometimes your experiment relies on an older / custom version of some python package. For example, the Keras API has changed quite a bit between versions 1 and 2. What if you are using a new environment locally, but would like to re-run old experiments that needed older version of packages? Or, for example, you'd like to see if your code would work with the latest version of a package. Studio gives you this opportunity.

```
studio run --python-pkg=<package_name>==<package_version> <script.py>
```

allows you to run `<script.py>` on a remote / cloud worker with a specific version of a package. You can also omit `==<package_version>` to install the latest version of the package (which may not be equal to the version in your environment). Note that if a package with a custom version has dependencies conflicting with the current version, the situation gets tricky. For now, it is up to pip to resolve conflicts. In some cases it may fail and you'll have to manually specify dependencies versions by adding more `--python-pkg` arguments.

CHAPTER 14

Cloud computing

Studio can be configured to submit jobs to the cloud. Right now, only Google Cloud is supported (CPU only), as well as Amazon EC2 (CPU and GPU). Once configured (see configuration instructions for Google Cloud, and Amazon AWS) the command

```
studio run --cloud={gcloud|ec2|gcspot|ec2spot} my_script.py
```

will create an instance, set up the python environment, run `my_script.py`, and shutdown the instance. You'll be able to see the progress of the job in `studio ui`. Different experiments might require different hardware. Fortunately, Google Cloud offers flexibility of instance configuration, and Amazon EC2 offers a variety of instances to select from; Studio can leverage either. To specify the number of cpus or gpus needed, use flags `--cpus` and `--gpus` respectively. That is, the command:

```
studio run --cloud={gcloud|ec2|gcspot|ec2spot} --cpus=8 --gpus=1 my_script.py
```

will create an instance with 8 cpus and 1 gpu. The top of the line gpu in Amazon EC2 is Tesla K80 at the moment, and that's the only one available through Studio; we might provide some gpu selection flags in the future as well.

The amount of ram and hard drive space can be configured via the `--ram` / `--hdd` flags (using standard suffixes like g(G,Gb,GiB), m(M,MiB)). Note that the amount of RAM will be rounded up to the next factor of 256 Mb. Also note that for now extended RAM for Google Cloud is not supported, which means the amount of RAM per CPU should be between 1 and 6 Gb. For Amazon EC2, Studio will find the cheapest instances with higher specs than required, or throw an exception for too extravagant of a request.

14.1 Running on EC2 spot instances

14.1.1 Basics

Amazon EC2 offers so-called spot instances that are provided with a substantial discount with the assumption that they can be taken from the user at any moment. Google Compute Engine has a similar product called preemptible instances, but Studio does not support it just yet. In short, for spot instances the user specifies the max price to pay per instance-hour. As long as the instance-hour price is below the specified limit (bid), the user is pays the current

price and uses the instance. Otherwise, the instance shuts down and is given to the higher bidder. For a more detailed explanation, refer to the spot instances user guide https://aws.amazon.com/ec2/spot/.

As you might have guessed, when running with the `--cloud=ec2spot` option the job is submitted to spot instances. You can additionally specify how much are you willing to pay for these instances via `--bid=<bid_in_usd>` or `--bid=<percent_of_ondemand_price>%`. The latter format specifies bid in percent of on-demand price. Unless you feel very generous towards Amazon there is no reason to specify a price above 100% the on-demand price (in fact, the spot instance user guide discourages users from doing so).

Note that bid is the max price for *one* instance; number of instances will vary (see below).

14.1.2 Autoscaling and number of instances

Given the ephemeral nature of spot workers, we need an additional mechanism controlling / balancing number of such instances. This mechanism is called auto-scaling, and in the simplest setting it tries to keep number of running instances constant. Studio handles downsizing of the auto-scaling groups when some workers are done and there is no work left in the queue. You can specify this behaviour by setting the `--num-workers` flag.

Autoscaling allows more complex behaviour, such as spinning up extra machines if there are too many messages in the queue. The default behaviour of Studio is as follows - start start with one spot worker, and scale up when the number of outstanding work messages in the queue is above 0.

14.2 Running on Google Cloud spot (preemptible) instances

Google Cloud's analog of EC2 spot instances are called preemptible instances. Preemptible instances are similar to EC2 spot instances in that they are much cheaper than regular (on-demand) instances and that they can be taken away at any moment with very little or no notice. They are different from EC2 spot instances in the bidding / market system - the prices on preemptible instances are fixed and depend only on hardware configuration. Thus, `--bid` has no effect when running with `--cloud=gcspot`.

Also, autoscaling on a queue for Google Cloud is in an alpha state and has some serious limitations; as such, we do not support it just yet. The required number of workers has to be specified via `--num-workers` (the default is 1), and Google group will try to keep it constant (that is, if the instances are taken away, it will try to spin up their replacements). When instances run out of work, they automatically spin down and eventually the instance group is deleted.

CHAPTER 15

Setting up Amazon EC2

This page describes the process of configuring Studio to work with Amazon EC2. We assume that you already have AWS credentials and an AWS account set up.

15.1 Install boto3

Studio interacts with AWS via the boto3 API. Thus, in order to use EC2 cloud you'll need to install boto3:

```
pip install boto3
```

15.2 Set up credentials

Add credentials to a location where boto3 can access them. The recommended way is to install the AWS CLI:

```
pip install awscli
```

and then run

```
aws configure
```

and enter your AWS credentials and region. The output format cam be left as None. Alternatively, use any method of letting boto3 know the credentials described here: http://boto3.readthedocs.io/en/latest/guide/configuration.html

CHAPTER 16

Setting up Google Cloud Compute

This page describes the process of setting up Google Cloud and configuring Studio to integrate with it.

16.1 Configuring Google Cloud Compute

16.1.1 Create and select a new Google Cloud project

Go to the Google Cloud console (https://console.cloud.google.com), and either choose a project that you will use to back cloud computing or create a new one. If you have not used the Google console before and there are no projects, there will be a big button "create project" in the dashboard. Otherwise, you can create a new project by selecting the drop-down arrow next to current project name in the top panel, and then clicking the "+" button.

16.1.2 Enable billing for the project

Google Cloud computing actually bills you for the compute time you use, so you must have billing enabled. On the bright side, when you sign up with Google Cloud they provide $300 of promotional credit, so really in the beginning you are still using it for free. On the not so bright side, to use machines with gpus you'll need to show that you are a legitimate customer and add $35 to your billing account. In order to enable billing, go to the left-hand pane in the Google Cloud console, select billing, and follow the instructions to set up your payment method.

16.1.3 Generate service credentials

The machines that submit cloud jobs will need to be authorized with service credentials. Go to the left-hand pane in the Google Cloud console and select API Manager -> Credentials. Then click the "Create credentials" button, choose service account key, leave key type as JSON, and in the "Service account" drop-down select "New service account". Enter a service account name (the name can be virtually anything and won't matter for the rest of the instructions). The important part is selecting a role. Click the "Select a role" dropdown menu, in "Project" select "Service Account Actor", and then scroll down to "Compute Engine" and select "Compute Engine Admin (v1)". Then scroll down to "Pub/Sub", and add a role "Pub/Sub editor" (this is required to create queues, publish and read messages from

them). If you are planning to use Google Cloud storage (directly, without the Firebase layer) for artifact storage, select the Storage Admin role as well. You can also add other roles if you are planning to use these credentials in other applications. When done, click "Create". Google Cloud console should generate a json credentials file and save it to your computer.

16.2 Configuring Studio

16.2.1 Adding credentials

Copy the json file credentials to the machine where Studio will be run, and create the environment variable `GOOGLE_APPLICATION_CREDENTIALS` that points to it. That is, run

```
export GOOGLE_APPLICATION_CREDENTIALS=/path/to/credentials.json
```

Note that this variable will be gone when you restart the terminal, so if you want to reuse it, add it to `~/.bashrc` (linux) or `~/.bash_profile` (OS X)

16.2.2 Modifying the configuration file

In the config file (the one that you use with the `--config` flag, or, if you use the default, in the `studio/default_config.yaml`), go to the `cloud` section. Change projectId to the project id of the Google project for which you enabled cloud computing. You can also modify the default instance parameters (see Cloud computing for studio for limitations though).

16.2.3 Test

To test if things are set up correctly, go to `studio/examples/general` and run

```
studio run --cloud=gcloud report_system_info.py
```

Then run `studio` locally, and watch the new experiment. In a little while, it should change its status to "finished" and show the system information (number of cpus, amount of ram / hdd) of a default instance. See Cloud computing for studio for more instructions on using an instance with specific hardware parameters.