Schedy Documentation

Release 0.1.0a1

Incalia

May 07, 2018

Contents

Schedy is your machine learning assistant. It will help you record your experiments, your results, visualize them, and it will even suggest new parameters for your next experiments!

Schedy can do useful things for you:

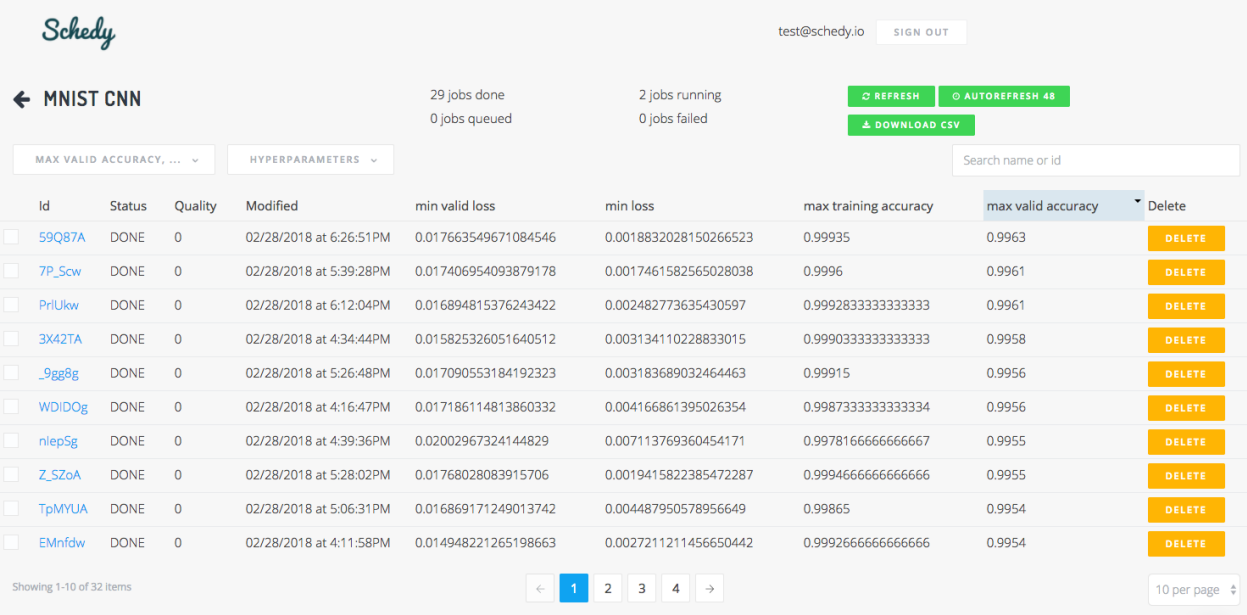
- Record the hyperparameters and results of all your past models.
- Suggest new hyperparameters for your next models.
- Coordinate a pool of workers (e.g. in a cluster), by making sure they stay busy trying to find the best combination of hyperparameters for your task.

And all of that in just a few lines of code! Coordinating a cluster of workers becomes as simple as this:

```
import schedy

db = schedy.SchedyDB()
experiment = db.get_experiment('My Task')
while True:
   with experiment.next_job() as job:
      my_train_function(job)
```

You can follow the evolution of your experiments thanks to our online dashboard.



We also provide a command line tool, that will help you with the most common tasks.

CHAPTER 1

Installation and setup

Sign up here, install Schedy & get your API token:

```
pip3 install schedy
schedy gen-token
```

You are now ready to *get started*!

1.1 Getting started

Before going through this section, make sure you have *installed Schedy*.

Schedy is the combination of three tools:

- A Python API.
- An online dashboard, that will help you visualize your experiments.
- A command line tool, `schedy`, that can help you with the basic setup and monitoring of your experiments.

Using the command line tool is totally optional. It simply wraps up the most common operations you would perform with the Python API.

For example, running `schedy add MyTask manual` in the command line tool is the same as running the following Python script:

```
import schedy

db = schedy.SchedyDB()
experiment = schedy.ManualSearch('MyTask')
db.add_experiment(experiment)
```

1.1.1 Database, experiments and jobs

When using Schedy, you store all your hyperparameters and results in the Schedy *database*, represented by the SchedyDB object. You can access these experiments and jobs from any workstation with your credentials (the ones you retrieved using `schedy gen-token` or on the website).

An *experiment* is a topic, a single task, for which you try to find the best configuration of hyperparameters. Examples of experiments are:

- Trying to fit a linear regression on your dataset
- Trying to train a Random Forest to predict tomorrow's temperature
- Trying to classify objects in a picture using an ResNet-50
- ...

A *job* is a trial for an experiment. A job tries to fulfill the task using a set of hyperparameters. Once it has completed its task, it reports how well it performed, by updating its results. (Actually, a job can also report results while it is running, e.g. the training loss associated to each epoch while training a neural network.)

1.2 Tutorials

If you want to get started right away, you can read this series of tutorial. You can also jump to the *reference* straight away, we've tried to make the API as clear as possible.

1.2.1 Using Schedy as a database

The scripts created in this tutorial can be found in our GitHub repository .

Schedy can be used as a simple database to store experiments. Schedy has two main concepts: experiments, and jobs. An experiment is a set of jobs, each job being a *trial* for its experiment.

For this example, let's say you are trying to find the values of `x` and `y` that minimize `x^2 * y^2`. First, you would create an experiment in Schedy:

```
schedy add MinimizeSimple manual
```

... or, in Python:

```
import schedy

db = schedy.SchedyDB()
experiment = schedy.ManualSearch('MinimizeSimple')
db.add_experiment(experiment)
```

This creates a new experiment called *MinimizeSimple*. The keyword *manual* tells Schedy that you are going to manage the jobs of this experiment yourself. More on that later.

Now, let's try many values of `x` and `y` to find which one works the best. If you do it by hand, you can record all your results in Schedy by creating new jobs. For each job, tell Schedy which parameters you tried, and what results you obtained.

```
schedy push MinimizeSimple --status DONE --hyperparameters x 1 y 2 --results result 5
# Or, for short:
schedy push MinimizeSimple -s DONE -p x 1 y 2 -r result 5
```

... or, in Python:

```
import schedy

db = schedy.SchedyDB()
experiment = db.get_experiment('MinimizeSimple')
job = experiment.add_job(
    status=schedy.Job.DONE,
    hyperparameters={
        'x': 1,
        'y': 2,
    },
    results={
        'result': 5
    },
)
```

You just added a new job to the experiment *MinimizeSimple*, which is *DONE* (finished). You tried `x = 1` and `y = 2`, and the *result* was 5 (`1^2 + 2^2`). Of course you could try to compute this expression for a bunch of values, then push the jobs to the database by hand, but a program would be much better at doing this.

Let's do this:

```
import schedy
import random

db = schedy.SchedyDB()
experiment = db.get_experiment('MinimizeSimple')
for i in range(20):
    # Test the problem for random values of x and y, 20 times
    x = random.uniform(-100, 100)
    y = random.uniform(-100, 100)
    result = x ** 2 + y ** 2
    # Tell Schedy about it!
    experiment.add_job(status=schedy.Job.DONE, hyperparameters={'x': x, 'y': y},␣
↪results={'result': result})
```

Not too difficult right? Now let's see how we performed, by listing the results of each job. The easiest way is to use your online dashboard.

However, if you want to do it using the command line, you can run:

```
# The -t flag indicates that we want the description of the jobs, not only
# their name, and that we want them in a table
schedy list -t MinimizeSimple
# You could also use the -p flag to display the jobs as a paragraph (this
# can be useful when you have lots of hyperparameters/results)
# schedy list -p MinimizeSimple
```

```
+--------+----------+----------+------------+-----------+
| id     | status   |        x |          y |    result |
|--------+----------+----------+------------+-----------|
| -bPmlQ | DONE     |  15.0542 |   3.27561  |   237.36  |
| 06wn6w | DONE     |  27.7519 |   0.301546 |   770.257 |
| 0jjY2Q | DONE     |  95.2792 |  36.0534   | 10378     |
| 5Jz0hA | DONE     | -60.2291 | -19.56     |  4010.13  |
| 8_7e5Q | DONE     |  24.3572 |  19.2384   |   963.389 |
| IOHsSw | DONE     | -82.2053 | -82.4315   | 13552.7   |
```

(continues on next page)

(continued from previous page)

```
| M4m6CA | DONE     | -66.6737 |  41.7379   |  6187.44  |
| MQmuTw | DONE     |  27.3775 | -31.1913   |  1722.43  |
| NavIrw | DONE     |   1      |   2        |     5     |
| NiHt6A | DONE     |  79.5122 | -74.5573   | 11881     |
| OP7aGw | DONE     | -12.5107 |  -0.683612 |   156.985 |
| Wjz2Wg | DONE     |  81.5054 | -66.08     | 11009.7   |
| ZM3nww | DONE     |  66.9189 | -52.3469   |  7218.33  |
| b6T0TA | DONE     |  70.9641 | -70.5859   | 10018.3   |
| csui0g | DONE     |  71.7953 |  49.0019   |  7555.74  |
| gRjRQA | DONE     | -47.0694 | -25.1969   |  2850.42  |
| gqfFQg | DONE     | -35.5846 | -46.4451   |  3423.41  |
| m0f9vA | DONE     | -80.614  | -72.4938   | 11754     |
| mL2NXw | DONE     |  18.0392 | -13.1687   |   498.828 |
| n8tNMQ | DONE     |  77.8921 |  80.532    | 12552.6   |
| yFvyFQ | DONE     | -41.0681 |  96.7539   | 11047.9   |
+--------+----------+----------+------------+-----------+
```

We are pretty far from the optimal result, but that's normal considering we tried only 20 combinations of hyperparameters.

Note that you can also access all these values using the Python API:

```
import schedy

db = schedy.SchedyDB()
experiment = db.get_experiment('MinimizeSimple')
for job in experiment.all_jobs():
    print('Id:', job.job_id)
    print('Status:', job.status)
    print('Hyperparameters:')
    for name, value in job.hyperparameters.items():
        print('- {}: {}'.format(name, value))
    print('Results:')
    for name, value in job.results.items():
        print('- {}: {}'.format(name, value))
    print()
```

You might be wondering:

> There are a lot of results. Can't we sort these jobs from the best to the worst?

Well of course! He're how you would do it:

```
schedy list -t MinimizeSimple -s result
```

```
+--------+----------+----------+------------+-----------+
| id     | status   |        x |          y |    result |
|--------+----------+----------+------------+-----------|
| NavIrw | DONE     |   1      |   2        |     5     |
| OP7aGw | DONE     | -12.5107 |  -0.683612 |   156.985 |
| -bPmlQ | DONE     |  15.0542 |   3.27561  |   237.36  |
| mL2NXw | DONE     |  18.0392 | -13.1687   |   498.828 |
| 06wn6w | DONE     |  27.7519 |   0.301546 |   770.257 |
| 8_7e5Q | DONE     |  24.3572 |  19.2384   |   963.389 |
| MQmuTw | DONE     |  27.3775 | -31.1913   |  1722.43  |
| gRjRQA | DONE     | -47.0694 | -25.1969   |  2850.42  |
| gqfFQg | DONE     | -35.5846 | -46.4451   |  3423.41  |
```

(continues on next page)

(continued from previous page)

```
| 5Jz0hA | DONE     | -60.2291 | -19.56     |  4010.13  |
| M4m6CA | DONE     | -66.6737 |  41.7379   |  6187.44  |
| ZM3nww | DONE     |  66.9189 | -52.3469   |  7218.33  |
| csui0g | DONE     |  71.7953 |  49.0019   |  7555.74  |
| b6T0TA | DONE     |  70.9641 | -70.5859   | 10018.3   |
| 0jjY2Q | DONE     |  95.2792 |  36.0534   | 10378     |
| Wjz2Wg | DONE     |  81.5054 | -66.08     | 11009.7   |
| yFvyFQ | DONE     | -41.0681 |  96.7539   | 11047.9   |
| m0f9vA | DONE     | -80.614  | -72.4938   | 11754     |
| NiHt6A | DONE     |  79.5122 | -74.5573   | 11881     |
| n8tNMQ | DONE     |  77.8921 |  80.532    | 12552.6   |
| IOHsSw | DONE     | -82.2053 | -82.4315   | 13552.7   |
+--------+----------+----------+------------+-----------+
```

Once you are done, you can remove the experiment, so that it does not appear in your listings later, as this is just an experiment for the tutorial.

```
schedy rm MinimizeSimple
# You could also remove a single job using:
# schedy rm MinimizeSimple NavIrw
```

. . . or, in Python:

```
import schedy

db = schedy.SchedyDB()
db.get_experiment('MinimizeSimple').delete()
# Or, to delete a specific job:
# db.get_experiment('MinimizeSimple').get_job('NavIrw').delete()
```

However, do not hesitate to keep your real experiments in the database, if you want to keep track of them. You don't have to remove them if you don't want to!

1.2.2 Schedy as a scheduler

The scripts created in this tutorial can be found in our GitHub repository .

Schedy was primarily designed to be used as a scheduler, that is to say a service that orchestrates a cluster of workers by telling them which hyperparameters to try. The simplest way to do that is by creating a queue of jobs. Each of them will be pulled by a worker, which will try the set of hyperparameters, and report how it performed.

Let's use the same problem as before, that is to say the minimization of `x^2 + y^2`. First, we will create an experiment.

```
schedy add MinimizeManual manual
```

. . . or, in Python:

```
import schedy

db = schedy.SchedyDB()
experiment = schedy.ManualSearch('MinimizeManual')
db.add_experiment(experiment)
```

Let's create a worker using the Schedy Python API.

```
import schedy
import time

db = schedy.SchedyDB()
experiment = db.get_experiment('MinimizeManual')
while True:
    try:
        with experiment.next_job() as job:
            x = job.hyperparameters['x']
            y = job.hyperparameters['y']
            result = x ** 2 + y ** 2
            job.results['result'] = result
    except Exception as e:
        print(e)
        time.sleep(60)
```

As you can see, this is just a script that pulls jobs from Schedy, computes the results, and pushes the jobs back to Schedy. In case of crash it will just keep on trying. Here's a quick explanation of it in more details:

```
import schedy
import time

db = schedy.SchedyDB()
experiment = db.get_experiment('MinimizeManual')
while True:
```

We initialize Schedy and retrieve the experiment we just created, then start an infinite loop in which we'll handle incoming jobs.

```
try:
    with experiment.next_job() as job:
        x = job.hyperparameters['x']
        y = job.hyperparameters['y']
        result = x ** 2 + y ** 2
        job.results['result'] = result
```

We pull the next job, and start working on it. The `with` statement is there so that we always report to Schedy whether the job has crashed or succeeded. The results will only be pushed to Schedy at the end of the `with` statement. If you wanted to report intermediary results to Schedy before the end of the‘‘with‘‘ statement, you could call `job.put()`.

```
except Exception as e:
    print(e)
    time.sleep(60)
```

If something failed, print what went wrong and wait a minute before retrying. If everything was fine, pull the next job immediately.

You can run the worker (i.e. this script) in another terminal, in the background, on the nodes of your cluster... You might notice it starts by printing errors like this one:

```
HTTP Error None:
> No job left for experiment MinimizeManual.
```

This is fine, as you do not have enqueued any job to your experiment yet. You can keep the script running, as it will detect the new job as soon as we enqueue it (or in the worst-case scenario, 60 seconds after that).

Let's ask the worker to compute the result using `x = 1` and `y = 2`.

```
schedy push MinimizeManual -p x 1 y 2
```

... or, in Python:

```
import schedy

db = schedy.SchedyDB()
experiment = db.get_experiment('MinimizeManual')
job = experiment.add_job(hyperparameters={'x': 1, 'y': 2})
```

After at most 60 seconds, the worker should have computed the result and reported back. You can see the result using:

```
schedy list -t MinimizeManual
# Or, if you only want to see the results of the job you just pushed instead of the␣
↪whole list:
# schedy show MinimizeManual <job-id>
```

The id of the job was given to you when you pushed it. It is a sequence of random characters that should look like this: ExhnhQ.

You should see something like this:

```
+--------+----------+-----+-----+----------+
| id     | status   |   x |   y |   result |
|--------+----------+-----+-----+----------|
| ExhnhQ | DONE     |   1 |   2 |        5 |
+--------+----------+-----+-----+----------+
```

If you don't, and the status is still `QUEUED`, just wait a few seconds until the worker pulls the experiment.

Schedy will always make sure that only one worker will work on a given job (multiple workers will never pull the same job).

> But do I always have to push my jobs by hand? What if I want to do a systematic search (e.g. random search)?

Don't worry we've got you covered. Just go to the next tutorial!

1.2.3 Random search

The scripts created in this tutorial can be found in our GitHub repository .

A simple example

In the last tutorial, we learned how to create a manual scheduler. When using a manual scheduler, you have a worker (or a pool of workers) treating jobs that you submit to a queue manually. This is useful for manual finetuning, when you know which hyperparameters you would like to try.

However, in many cases, you would like to explore the space of hyperparameters automatically, using a black-box optimization algorithm for instance. This is where automatic schedulers become useful.

Random search is one of the most basic schedulers there are: it will simply create jobs, whose hyperparameters will be randomly picked. The distribution of these hyperparameters must be chosen by yourself. Using the previous example (finding the values of `x` and `y` that minimize `x^2 + y^2`), you could define these distributions using your instincts. For instance, we'll make `x` and `y` follow a normal distribution, centered around *0* with a standard deviation of *5* for `x`, and 2 for `y` (yes, this is totally arbitrary).

```
schedy add MinimizeRandom random x normal '{"mean": 0, "std": 5}' y normal '{"mean":␣
↪0, "std": 2}'
```

... or, in Python:

```
import schedy

db = schedy.SchedyDB()
distributions = {
    'x': schedy.random.Normal(0, 5),
    'y': schedy.random.Normal(0, 2),
}
experiment = schedy.RandomSearch('MinimizeRandom', distributions)
db.add_experiment(experiment)
```

As you can see, we're using a `random` scheduler this time, instead of a `manual` one. And we're defining the random variables immediately after this. The parameters of each distribution are supplied using JSON notation. We'll talk more about the supported distributions *later*.

The worker file does not change much:

```
import schedy
import time

db = schedy.SchedyDB()
experiment = db.get_experiment('MinimizeRandom')
for i in range(20):
    try:
        with experiment.next_job() as job:
            x = job.hyperparameters['x']
            y = job.hyperparameters['y']
            result = x ** 2 + y ** 2
            job.results['result'] = result
    except Exception as e:
        print(e)
        # Wait a minute before issuing the next request
        time.sleep(60)
```

As you can see all we changed was the name of the experiment. We also changed the infinite loop to a finite loop, because the random search scheduler will continuously send new jobs to us. Because we are able to perform a task in a few nanoseconds, an infinite loop would create several thousand jobs per second and spam the database (remember, we're not trying to optimize a neural network here, we're just minimizing `x^2 + y^2`).

Once again, you can start the worker, then list your results using:

```
schedy list -t MinimizeRandom
```

```
+--------+----------+------------+------------+-----------+
| id     | status   |          x |          y |    result |
|--------+----------+------------+------------+-----------|
| 2WDn_w | DONE     |  -0.836867 | -0.71981   |   1.21847 |
| m4hoTw | DONE     |  -1.15003  | -0.83331   |   2.01698 |
| l26a6g | DONE     |   0.862245 |  1.27614   |   2.372   |
| ZDmNqw | DONE     |  -2.52887  |  0.429102  |   6.57931 |
| LMEOaQ | DONE     |   2.86853  | -0.742761  |   8.78014 |
| iKCzuw | DONE     |   2.47215  |  1.95058   |   9.91631 |
| E6K6Ew | DONE     |  -2.90947  |  1.81924   |  11.7746  |
```

(continues on next page)

(continued from previous page)

```
| hRaPOQ | DONE     |   2.63032  |  3.00305   |  15.9369  |
| Tby5Og | DONE     |  -3.68871  |  1.66496   |  16.3787  |
| b0pp7g | DONE     |   1.76621  |  4.14727   |  20.3194  |
| NZQw7w | DONE     |   4.92685  |  0.71905   |  24.7909  |
| sMUVuA | DONE     |   5.58645  |  1.50509   |  33.4737  |
| zLxjYA | DONE     |   6.70355  |  0.0705488 |  44.9426  |
| hDi9uw | DONE     |  -6.75093  |  1.57475   |  48.0549  |
| oMcmeQ | DONE     |  -7.17896  |  0.100174  |  51.5475  |
| fF8NHQ | DONE     |   7.20394  |  0.692157  |  52.3758  |
| tKwlHw | DONE     |   9.02237  |  0.156419  |  81.4276  |
| m9G7GA | DONE     |   8.18227  |  3.95599   |  82.5994  |
| 7MgmuA | DONE     |  10.0929   | -2.78685   | 109.634   |
| l8L6xQ | DONE     | -10.6514   | -0.970788  | 114.395   |
+--------+----------+------------+------------+-----------+
```

Available distributions

Because this is a toy problem, using arbitrary normal distributions does not have a lot of impact. But in practice, the distributions you choose for your hyperparameters could change how fast you find a good solution.

In order to help you in this regard, Schedy offers several type of distributions. The following is a short description of these distributions (see also: *API reference*).

Uniform distribution

Values will be uniformly distributed in the interval [`low`, `high`).

Example:

```
# One hyperparameter (x) with values ranging from 2.1 (included) to 5 (excluded)
schedy add Test random x uniform '{"low": 2.1, "high": 5}'
```

Normal distribution

Values will be distributed following a normal distribution, centered around `mean` with a standard deviation of `std`.

Example:

```
schedy add Test random x normal '{"mean": 2.1, "std": 5}'
```

LogUniform distribution

Values will be distributed between `low` and `high`, such that log(value) is uniformly distributed between log(`low`) and log(`high`).

This might be useful for hyperparameters that only have an influence when they change their order of magnitude (e.g. learning rates for neural networks).

Example:

```
schedy add Test random x loguniform '{"low": 0.000001, "high": 0.1}'
```

Choice distribution

Values will be picked randomly in a set of `values`. You can optionally provide `weights` for these values, to make some of them more likely to be suggested by Schedy than others. The values can be numbers, strings, booleans, strings, arrays or objects, and you can mix those.

Simple example:

```
schedy add Test random x choice '{"values": [2, 4, 8, 10]}'
```

Advanced example:

```
schedy add Test random x choice '{"values": [false, 1, "two", {"key": "three", "key2
↪": 3}, [4, "four"]], "weights": [0.1, 0.2, 0.3, 0.3, 0.1]}'
```

Constant distribution

The value will always be the same. The value can be a number, a string, a boolean, an array or an object. This can be useful to pass configuration parameters to the workers, for instance.

```
schedy add Test random x const 0 config const '{"log_dir": "/var/log", "schedy_rocks
↪": true}'
```

1.2.4 Population based training

Population Based Training (PBT) allows you to train your models in a smarter way. It takes care of finding not only the best set of hyperparameters, but it also able to find the best hyperparameters schedule during training. For instance, having a fixed learning rate during training is often suboptimal, so PBT helps you find out when and how you should change your learning rate.

If you want to use Population Based Training with Schedy, you only need to know the following:

PBT is an improvement over random search: it is able to focus on the most promising jobs using to strategies:

- An *exploit* strategy, in which the least promising jobs are thrown away, and replaced by copies of the most promising ones. This allows you not to waste resources on the wrong jobs.
- An *explore* strategy, that tries new values for the hyperparameters of the most promising jobs during training. For instance, this is what allows you to find the optimal learning rate schedule of a neural network.

An example using PBT to finetune an Image Recognition neural network can be found on our GitHub repository.

Creating an experiment

An experiment using Population Based Training can be created this way:

```
import schedy
db = schedy.SchedyDB()
experiment = schedy.PopulationBasedTraining(
    'MNIST with PBT',
    schedy.pbt.MAXIMIZE,
    'max_accuracy',
    exploit=schedy.pbt.Truncate(),
    explore={
```

(continues on next page)

(continued from previous page)

```
        'learning_rate': schedy.pbt.Perturb(),
        'dropout_rate': schedy.pbt.Perturb(),
    },
    initial_distributions={
        'num_layers': schedy.random.Choice(range(1, 10)),
        'activations': schedy.random.Choice(['relu', 'tanh']),
        'kernel_size': schedy.random.Choice([3, 5, 7]),
        'num_filters': schedy.random.Choice([2, 4, 8, 16, 32, 64, 128, 256, 512]),
        'learning_rate': schedy.random.LogUniform(1e-6, 1e-1),
        'dropout_rate': schedy.random.Uniform(0.0, 0.8),
    },
    population_size=20,
)
db.add_experiment(experiment)
```

The first argument (`MNIST with PBT`) is the name of the experiment.

The second argument tells Schedy that we are trying to maximize (*`schedy.pbt.MAXIMIZE`*) the result specified in the third argument, the `max_accuracy` obtained by the network.

The argument called `exploit` tells us that we are using the *Truncate* strategy to exploit results (i.e. if we are working on a job that scored in the bottom 20%, explore a job from the top 20% instead, see *`schedy.pbt.Truncate`*).

The argument called `explore` tells us that we are using the *Perturb* strategy to explore the learning rate and the dropout rate. This strategy multiplies the values of these hyperparameters by a random number (see *`schedy.pbt.Perturb`*).

Remember the exploration modifies the value of your hyperparameters *during training* so you should only use it when it makes sense. For instance, it is possible to change the value of the learning rate while training (it does not change the model in itself), but it is not possible to change the number of layers (it usually does not make sense to create/remove weights while training).

Using `schedy.pbt.Truncate` as your exploit strategy, and `schedy.pbt.Perturb` as your explore strategy is usually a sensible default.

The argument called `initial_distributions` tells Schedy how to pick values for the initial jobs, as those are basically created using random search. The available distributions are the same as *the ones used for random search*. The next argument, `population_size`, specifies the number of initial jobs that should be created before starting to exploit/explore.

Note: Specifying the population size and the initial distributions is optional. You can also create the initial jobs by hand, using `schedy push` *in the command line or `schedy.Experiment.add_job()` . This allows you to choose the initial value of your hyperparameters by hand, instead of using random search.*

Creating the worker

Creating a worker that will work efficiently with PBT requires a few more steps than other experiment types (e.g. Random Search).

Let's have a little reminder. When using random search, the basic worker followed these steps:

```
import schedy

db = schedy.SchedyDB()
experiment = db.get_experiment('MyExperiment')
with experiment.next_job() as job:
    model = create_model(job)
```

(continues on next page)

(continued from previous page)

```
    train(model) # Full training until convergence
    job.results = evaluate(model)
```

When using PBT, you should be doing something along those lines instead:

```
import schedy

db = schedy.SchedyDB()
experiment = db.get_experiment('MyExperiment')
with experiment.next_job() as job:
    model = create_model(job)
    if 'model_path' in job.results:
        model.load(job.results['model_path'])
    partial_train(model) # Partial training for a limited amount of time
    job.results = evaluate(model)
    model_save_path = 'dump_dir/' + job.id + '.mdl'
    model.save(model_save_path)
    job.results['model_path'] = model_save_path
```

For every job it receives, the worker follows these three simple steps:

- Try to reload the model if it exists
- Train the model a bit more
- Save the model

As you can see, instead of training the model until convergence, you should only train it for a limited amount of time (e.g. five epochs, 30 minutes...). You should then save your model to a location that can be accessed by all workers (here we suppose that all workers have accessed to the `dump_dir` directory, and we save the model as `dump_dir/<job_id>.mdl`). You should also record the location of your model into the job's results.

The reason for this is that Schedy might choose to ask another worker to resume the work on your job later, by copying the job's hyperparameters and results to a new job, and sending it to a new worker. This is why this worker starts by checking whether there is a result called `model_path`, and if there is, it reloads the weights from this location.

Everything else is handled by Schedy. All you need to do is to reload the model if it exists, to train it a bit more, then to save it.

We provide examples here, and a more detailed description of the PBT experiments in the API reference, *here* and *here*.

1.3 API Reference

1.3.1 SchedyDB

class schedy.**SchedyDB**(*config_path=None*, *config_override=None*)

SchedyDB is the central component of Schedy. It represents your connection the the Schedy service.

Parameters

- **config_path** (*str or file-object*) – Path to the client configuration file. This file contains your credentials (email, API token). By default, ~/.schedy/client.json is used. See *Installation and setup* for instructions about how to use this file.
- **config_override** (*dict*) – Content of the configuration. You can use this to if you do not want to use a configuration file.

add_experiment(*exp*)
Adds an experiment to the Schedy service. Use this function to create new experiments.

Parameters **exp** (schedy.Experiment) – The experiment to add.

Example

```
>>> db = schedy.SchedyDB()
>>> exp = schedy.ManualSearch('TestExperiment')
>>> db.add_experiment(exp)
```

get_experiment(*name*)
Retrieves an experiment from the Schedy service by name.

Parameters **name** (*str*) – Name of the experiment.

Returns An experiment of the appropriate type.

Return type *schedy.Experiment*

Example

```
>>> db = schedy.SchedyDB()
>>> exp = db.get_experiment('TestExperiment')
>>> print(type(exp))
<class 'schedy.experiments.ManualSearch'>
```

get_experiments()
Retrieves all the experiments from the Schedy service.

Returns Iterator over all the experiments.

Return type iterator of *schedy.Experiment*

1.3.2 Experiments

Base class

class schedy.**Experiment**(*name*, *status='RUNNING'*)
Base-class for all experiments.

Parameters

- **name** (*str*) – Name of the experiment. An experiment is uniquely identified by its name.
- **status** (*str*) – Status of the experiment. See *Experiment status*.

add_job(***kwargs*)
Adds a new job to this experiment.

Parameters

- **hyperparameters** (*dict*) – A dictionnary of hyperparameters values.
- **status** (*str*) – Job status. See *Job status*. Default: QUEUED.
- **results** (*dict*) – A dictionnary of result values. Default: No results (empty dictionary).

Returns The instance of the new job.

Return type *schedy.Job*

next_job()
Returns a new job to be worked on. This job will be set in the RUNNING state. This function handles everything so that two workers never start working on the same job.

Returns The instance of the requested job.

Return type *schedy.Job*

all_jobs()
Retrieves all the jobs belonging to this experiment.

Returns An iterator over all the jobs of this experiment.

Return type iterator of *schedy.Job*

get_job(*job_id*)
Retrieves a job by id.

Parameters **job_id** (*str*) – Id of the job to retrieve.

Returns Instance of the requested job.

Return type *schedy.Job*

push_updates()
Push all the updates made to this experiment to the service.

delete(*ensure=True*)
Deletes this experiment.

Parameters **ensure** (*bool*) – If true, an exception will be raised if the experiment was deleted before this call.

Experiment status

Experiment.**RUNNING = 'RUNNING'**
Status of a running experiment.

Experiment.**DONE = 'DONE'**
Status of a completed (or paused) experiment.

Manual search

class schedy.**ManualSearch**(*name*, *status='RUNNING'*)
Bases: schedy.experiments.Experiment

Represents a manual search, that is to say an experiment for which the only jobs returned by *schedy.Experiment.next_job()* are jobs that were queued beforehand (by using *schedy.Experiment.add_job()* for example).

Base-class for all experiments.

Parameters

- **name** (*str*) – Name of the experiment. An experiment is uniquely identified by its name.
- **status** (*str*) – Status of the experiment. See *Experiment status*.

Random search

class schedy.**RandomSearch**(*name*, *distributions*, *status='RUNNING'*)

Bases: schedy.experiments.Experiment

Represents a random search, that is to say en experiment that returns jobs with random hyperparameters when no job was queued manually using *schedy.Experiment.add_job()*.

If you create a job manually for this experiment, it must have only and all the hyperparameters specified in the distributions parameter.

Parameters

- **name** (*str*) – Name of the experiment. An experiment is uniquely identified by its name.
- **distributions** (*dict*) – A dictionary of distributions (see *schedy.random*), whose keys are the names of the hyperparameters.
- **status** (*str*) – Status of the experiment. See *Experiment status*.

Population Based Training

class schedy.**PopulationBasedTraining**(*name*, *objective*, *result_name*, *exploit*, *explore={}*, *initial_distributions={}*, *population_size=None*, *status='RUNNING'*, *max_generations=None*)

Bases: schedy.experiments.Experiment

Implements Population Based Training (see paper).

You have two ways to specify the initial jobs for Population Based training. You can create them manually using *schedy.Experiment.add_job()*, or you can specify the initial_distributions and population_size parameters.

If you create a job manually for this experiment, it must have at least the hyperparameters specified in the explore parameter.

Parameters

- **name** (*str*) – Name of the experiment. An experiment is uniquely identified by its name.
- **objective** (*str*) – The objective of the training, either *schedy.pbt.MINIMIZE* (to minimize a result) or *schedy.pbt.MAXIMIZE* (to maximize a result).
- **result_name** (*str*) – The name of the result to optimize. This result must be present in the results of all *RUNNING* jobs of this experiment.
- **exploit** (*schedy.pbt.ExploitStrategy*) – Strategy to use to exploit the results (i.e. to focus on the most promising jobs).
- **explore** (*dict*) – Strategy to use to explore new hyperparameter values. The keys of the dictionary are the name of the hyperparameters (str), and the values are the strategy associated with the hyperparameter (schedy.pbt.ExploreStrategy). Values for the omitted hyperparameters will not be explored. This parameter is optional: if you do not specify any explore strategy, only exploitation will be used.
- **initial_distributions** (*dict*) – The initial distributions for the hyperparameters, as dictionary of distributions (see *schedy.random*) whose keys are the names of the hyperparameters. This parameter optional, you can also create the initial jobs manually. If you use this parameter, make sure to use population_size as well.

- **population_size** (*int*) – Number of initial jobs to create, before starting to exploit/explore (i.e. size of the population). It does **not** have to be the number of jobs you can process in parallel. The original paper used values between 10 and 80.
- **status** (*str*) – Status of the experiment. See *Experiment status*.
- **max_generations** (*int*) – Maximum number of generations to run before marking the experiment the experiments as done (*Experiment status*). When the maximum number of generations is reached, subsequent calls to *schedy.Experiment.next_job()* will raise *schedy.errors.NoJobError*, to indicate that the job queue is empty.

1.3.3 Jobs

Job class

class schedy.**Job**(*job_id*, *experiment*, *hyperparameters*, *status='QUEUED'*, *results={}*, *etag=None*)

Represents a job instance belonging to an experiment. You should not need to create it by hand. Use *schedy.Experiment.add_job()*, *schedy.Experiment.get_job()*, *schedy.Experiment.all_jobs()* or *schedy.Experiment.next_job()* instead.

Jobs object are context managers, that it to say they can be used with a `with` statement. They will be put in the RUNNING state at the start of the with statement, and in the DONE or CRASHED state at the end (depending on whether an uncaught exception is raised within the `with` block). See *schedy.Job.__enter__()* for an example of how to use this feature.

Parameters

- **job_id** (*str*) – Unique id of the job.
- **experiment** (schedy.Experiment) – Experiment containing this job.
- **hyperparameters** (*dict*) – A dictionnary of hyperparameters values.
- **status** (*str*) – Job status. See *Job status*.
- **results** (*dict*) – A dictionnary of results values.
- **etag** (*str*) – Value of the entity tag sent by the backend.

PRUNED = 'PRUNED'

Status of a job that was abandonned because it was not worth working on.

put(*safe=True*)

Puts a job in the database, either by creating it or by updating it.

This function is always called at the end of a `with` block.

Parameters **safe** (*bool*) – If true, this operation will make sure not to erase any content that would have been put by another Schedy call in the meantime. For example, this ensures that no two workers overwrite each other's work on this job because they are working in parallel.

try_run()

Try to set the status of the job as `RUNNING`, or raise an exception if another worker tried to do so before this one.

delete(*ensure=True*)

Deletes this job from the Schedy service.

Parameters **ensure** (*bool*) – If true, an exception will be raised if the job was deleted before this call.

__enter__()
Context manager __enter__ method. Will try to set the job as CRASHED if the job has not been modified by another worker concurrently.

Example:

```
>>> db = schedy.SchedyDB()
>>> exp = db.get_experiment('Test')
>>> with exp.next_job() as job:
>>>     my_train_function(job)
```

If my_train_function raises an exception, the job will be marked as CRASHED. Otherwise it will be marked as DONE. (See py:meth:*Job.__exit__*.)

Note that since *schedy.Experiment.next_job()* will always return a RUNNING job, this method will never raise *schedy.errors.UnsafeUpdateError* in this case.

__exit__(*exc_type*, *exc_value*, *traceback*)
Context manager __exit__ method. Will try to set the job status as CRASHED if an exception was raised in the with block. Otherwise, it will try to set the job status as DONE. It will also push all the updates that were made locally to the Schedy service (by calling *Job.put()* for you).

Job status

Job.**QUEUED = 'QUEUED'**
Status of a queued job. Queued jobs are returned when calling *schedy.Experiment.next_job()*.

Job.**RUNNING = 'RUNNING'**
Status of a job that is currently running on a worker.

Job.**CRASHED = 'CRASHED'**
Status of job that was being processed by a worker, but the worker crashed before completing the job.

Job.**DONE = 'DONE'**
Status of a completed job.

1.3.4 Random distributions

class schedy.random.**LogUniform**(*low*, *high*)
LogUniform distribution. Values are sampled betweend low and high, such that log(value) is uniformly distributed between log(low) and log(high).

Parameters

- **low** (*float*) – Minimal value (inclusive).
- **high** (*float*) – Maximum value (exclusive).

class schedy.random.**Uniform**(*low*, *high*)
Uniform distribution. Values will be uniformly distributed in the interval [low, high).

Parameters

- **low** (*float*) – Minimal value (inclusive).
- **high** (*float*) – Maximum value (exclusive).

class schedy.random.**Choice**(*values*, *weights=None*)
Choice distribution. Values will be picked randomly in a set of values. You can optionally provide weights for these values, to make some of them more likely to be suggested by Schedy than others.

Parameters

- **values** (*list*) – Possible values that can be picked. They can be numbers, strings, booleans, strings, lists or dictionaries, and you can mix those.
- **weights** (*list*) – Weight associated with each value. If provided, the length of `weights` must be the same as that of `values`.

class schedy.random.**Normal**(*mean*, *std*)
Normal distribution.

Parameters

- **mean** (*float*) – Desired mean of the distribution.
- **std** (*float*) – Desired standard deviation of the distribution.

class schedy.random.**Constant**(*value*)
"Constant" distribution. Will always yield the same value.

Parameters **value** – The value of the samples that will be returned by this distribution. Can be a number, string, boolean, string, list or dictionary.

1.3.5 Population Based Training

See *schedy.PopulationBasedTraining* for a description of the experiment type.

schedy.pbt.**MINIMIZE = 'min'**
Minimize the objective

schedy.pbt.**MAXIMIZE = 'max'**
Maximize the objective

class schedy.pbt.**Truncate**(*proportion=0.2*)
Truncate exploit strategy: if the selected candidate job is in the worst n%, use a candidate job in the top n% instead.

Parameters **proportion** (*float*) – Proportion of jobs that are considered to be "best" jobs, and "worst" jobs. For example, if `proportion = 0.2`, if the selected candidate job is in the bottom 20%, it will be replaced by a job in the top 20%. Must satisfy `0 < proportion <= 0.5`.

class schedy.pbt.**Perturb**(*min_factor=0.8*, *max_factor=1.2*)
Perturb explore strategy: multiply the designated hyperparameter by a random factor, sampled from a uniform distribution.

Parameters

- **min_factor** (*float*) – Minimum value for the factor (inclusive).
- **max_factor** (*float*) – Maximum value for the factor (exclusive).

1.3.6 Errors

exception schedy.errors.**SchedyError**
Base class for all Schedy exceptions.

exception schedy.errors.**HTTPError**(*body*, *code*, **args*)
Base class for exceptions caused by a transaction with the service.

Parameters

- **body** (*str*) – Error message.
- **code** (*int or None*) – HTTP status code.
- **args** (*list*) – Other arguments passed to Exception.

exception schedy.errors.**ClientError**(*body*, *code*, **args*)
Exception caused by the client side.

Parameters

- **body** (*str*) – Error message.
- **code** (*int or None*) – HTTP status code.
- **args** (*list*) – Other arguments passed to Exception.

exception schedy.errors.**ClientRequestError**(*body*, *code*, **args*)
Exception caused by the content of the request.

Parameters

- **body** (*str*) – Error message.
- **code** (*int or None*) – HTTP status code.
- **args** (*list*) – Other arguments passed to Exception.

exception schedy.errors.**AuthenticationError**(*body*, *code*, **args*)
Authentication error, access to the resource is forbidden.

Parameters

- **body** (*str*) – Error message.
- **code** (*int or None*) – HTTP status code.
- **args** (*list*) – Other arguments passed to Exception.

exception schedy.errors.**ReauthenticateError**(*body*, *code*, **args*)
Authentication error, the client should retry after authenticating again.

Parameters

- **body** (*str*) – Error message.
- **code** (*int or None*) – HTTP status code.
- **args** (*list*) – Other arguments passed to Exception.

exception schedy.errors.**ResourceExistsError**(*body*, *code*, **args*)
The resource cannot be created because it exists already.

Parameters

- **body** (*str*) – Error message.
- **code** (*int or None*) – HTTP status code.
- **args** (*list*) – Other arguments passed to Exception.

exception schedy.errors.**UnsafeUpdateError**(*body*, *code*, **args*)
The resource cannot be updated safely because it has been modified by another client since its state was retrieved, so updating it could overwrite these modifications.

Parameters

- **body** (*str*) – Error message.

- **code** (*int or None*) – HTTP status code.
- **args** (*list*) – Other arguments passed to Exception.

exception schedy.errors.**NoJobError**(*body*, *code*, **args*)
The request could not return any job.

Parameters

- **body** (*str*) – Error message.
- **code** (*int or None*) – HTTP status code.
- **args** (*list*) – Other arguments passed to Exception.

exception schedy.errors.**UnhandledResponseError**(*body*, *code*, **args*)
The response could not be parsed or handled.

Parameters

- **body** (*str*) – Error message.
- **code** (*int or None*) – HTTP status code.
- **args** (*list*) – Other arguments passed to Exception.

exception schedy.errors.**ServerError**(*body*, *code*, **args*)
Server-side exception.

Parameters

- **body** (*str*) – Error message.
- **code** (*int or None*) – HTTP status code.
- **args** (*list*) – Other arguments passed to Exception.

1.3.7 Advanced

schedy.core.**NUM_AUTH_RETRIES = 2**
Number of retries if the authentication fails.

You can also set schedy.core.Retry.BACKOFF_MAX to set the maximum backoff time for a failed request.

1.4 Frequently Asked Questions

1.4.1 There's something I want to ask you, how can I contact you?

Feel free to contact us using the chat integrated in our website!

1.4.2 Can I use string/array/dictionaries as hyperparameters?

Yes you can. However, when using the command-line, you most likely want to put single quotes around the value, because JSON notation uses reserved Bash caracters.

For instance:

```
# Notice that you have to surround strings with double-quotes, because
# that's how JSON strings work
schedy push MyExperiment -p my_string_param '"string value"'
schedy push MyExperiment -p my_array_param '["stuff", true, 6]'
schedy push MyExperiment -p my_dict_param '{"key0": "value0", "key1": 42}'
```

1.4.3 I think I found a bug. How can I report it?

First of all, we want to thank you for contributing! You can report bugs on our GitHub tracker.

CHAPTER 2

Other

- Alphabetical index
- Modules index

Python Module Index

S

Index

U