## Contents

1. Welcome!  
   - Page 3
2. Documentation Contents  
   - Page 5
Welcome!

SHERPA is a Python library for hyperparameter tuning of machine learning models. It provides:

- hyperparameter optimization for machine learning researchers
- a choice of hyperparameter optimization algorithms
- parallel computation that can be fitted to the user’s needs
- a live dashboard for the exploratory analysis of results.

Its goal is to provide a platform in which recent hyperparameter optimization algorithms can be used interchangeably while running on a laptop or a cluster.
2.1 Installation

2.1.1 Installation from PyPi

This is the most straightforward way to install Sherpa. The source code may however be slightly older than what is found on the GitHub.

```
pip install parameter-sherpa
```

2.1.2 Installation from GitHub

Clone from GitHub:

```
git clone https://github.com/LarsHH/sherpa.git
export PYTHONPATH=$PYTHONPATH:`pwd`/sherpa
```

Install dependencies:

```
pip install pandas
pip install numpy
pip install scipy
pip install scikit-learn
pip install flask
pip install enum34  # if on < Python 3.4
```

You can run an example to verify SHERPA is working:

```
cd sherpa/examples/
python simple.py
```

Note that to run hyperparameter optimizations in parallel with SHERPA requires the installation of Mongo DB. Further instructions can be found in the Parallel Installation section of the documentation.
2.2 From Keras to Sherpa in 30 seconds

This example will show how to adapt a minimal Keras script so it can be used with SHERPA. As starting point we use the “getting started in 30 seconds” tutorial from the Keras webpage.

We start out with this piece of Keras code:

```python
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(units=64, activation='relu', input_dim=100))
model.add(Dense(units=10, activation='softmax'))
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

The goal is to tune the number of hidden units via Random Search. To do that, we define one parameter of type `Discrete`. We also use the `RandomSearch` algorithm with maximum number of trials 50.

```python
import sherpa
parameters = [sherpa.Discrete('num_units', [50, 200])]
alg = sherpa.algorithms.RandomSearch(max_num_trials=50)
```

We use these objects to create a SHERPA Study:

```python
study = sherpa.Study(parameters=parameters,
                      algorithm=alg,
                      lower_is_better=True)
```

We obtain `trials` by iterating over the study. Each trial has a `parameter` attribute that contains the `num_units` parameter value. We can use that value to create our model.

```python
for trial in study:
    model = Sequential()
    model.add(Dense(units=trial.parameters['num_units'],
                     activation='relu', input_dim=100))
    model.add(Dense(units=10, activation='softmax'))
    model.compile(loss='categorical_crossentropy',
                  optimizer='sgd',
                  metrics=['accuracy'])

    model.fit(x_train, y_train, epochs=5, batch_size=32,
              callbacks=[study.keras_callback(trial, objective_name='val_loss')])

study.finalize(trial)
```

During training, objective values will be added to the SHERPA study via the callback. At the end of training `study.finalize` completes this trial. This means that no more observation will be added to this trial.

When the `Study` is created, SHERPA will display the dashboard address. If you put the address into your browser you will see the dashboard as shown below. As a next step you can take a look at this example of optimizing a Random Forest in `sherpa/examples/randomforest.py`.
2.3 A Guide to SHERPA

2.3.1 Parameters

Hyperparameters are defined via `sherpa.Parameter` objects. Available are

- `sherpa.Continuous`: Represents continuous parameters such as `weight-decay` multiplier. Can also be thought of as `float`.
- `sherpa.Discrete`: Represents discrete parameters such as `number of hidden units` in a neural network. Can also be thought of as `int`.
- `sherpa.Ordinal`: Represents categorical ordered parameters. For example `minibatch` size could be an ordinal parameter taking values 8, 16, 32, etc. Can also be thought of as `list`.
- `sherpa.Choice`: Represents unordered categorical parameters such as `activation` function in a neural network. Can also be thought of as a `set`.

Every parameter takes a `name` and `range` argument. The `name` argument is simply the name of the hyperparameter. The `range` is either the lower and upper bound of the range, or the possible values in the case of `sherpa.Ordinal` and `sherpa.Choice`. The `sherpa.Continuous` and `sherpa.Discrete` parameters also take a `scale` argument which can take values `linear` or `log`. This describes whether values are sampled uniformly on a linear or a log scale.

Hyperparameters are defined as a list to be passed to the `sherpa.Study` down the line. For example:

```python
parameters = [sherpa.Continuous(name='lr', range=[0.005, 0.1], scale='log'),
              sherpa.Continuous(name='dropout', range=[0., 0.4]),
              sherpa.Ordinal(name='batch_size', range=[16, 32, 64]),
              sherpa.Discrete(name='num_hidden_units', range=[100, 300]),
              sherpa.Choice(name='activation', range=['relu', 'elu', 'prelu'])]
```
SHERPA Documentation

Note that it is generally recommended not to represent continuous or discrete parameters as categorical. This is due to the fact that exploring a range of values rather than discrete options yields much more information to understand the relationship between the hyperparameter and the outcome.

2.3.2 The Algorithm

The algorithm refers to the procedure that determines how hyperparameter configurations are chosen and in some cases the resource they are assigned. All available algorithms can be found in sherpa.algorithms. The description in Available Algorithms gives an in-depth view of what algorithms are available, their arguments, and when one might chose one algorithm over another. The sherpa.algorithms module is also home to stopping rules. Those are procedures that define if a trial should be stopped before its completion. The initialization of the algorithm is simple. For example:

```
algorithm = sherpa.algorithms.RandomSearch(max_num_trials=150)
```

where max_num_trials stands for the number of trials after which the algorithm will finish.

2.3.3 The Study

In Sherpa a Study represents the hyperparameter optimization itself. It holds references to the parameter ranges, the algorithm, the results that have been gathered, and provides an interface to obtain a new trial, or add results from previously suggested trial. It also starts the dashboard in the background. When initializing the study it expects references to the parameter ranges, the algorithm, and at minimum a boolean variable on whether lower objective values are better. For a full list of the arguments see the Study-API reference.

```
study = sherpa.Study(parameters=parameters,
algorithm=algorithm,
lower_is_better=False)
```

In order to obtain a first trial one can either call Study.get_suggestion() or directly iterate over the Study object.

```
# To get a single trial
trial = study.get_suggestion()

# Or directly iterate over the study
for trial in study:
    ...
```

The Trial object has an id attribute and a parameters attribute. The latter contains a dictionary with a hyperparameter configuration from the previously specified ranges provides by the defined algorithm. The parameter configuration can be used to initialize, train, and evaluate a model.

```
model = init_model(train.parameters)
```

During training Study.add_observation can be used to add intermediate metric values from the model training.

```
for iteration in range(num_iterations):
    training_error = model.fit(epochs=1)
    validation_error = model.evaluate()
    study.add_observation(trial=trial,
        iteration=iteration,
        objective=validation_error,
        context={'training_error': training_error})
```
Once the model has completed training Sherpa expects a call to the `Study.finalize` function.

```python
study.finalize(trial)
```

This can be put together in a double for-loop of the form:

```python
for trial in study:
    model = init_model(train.parameters)
    for iteration in range(num_iterations):
        training_error = model.fit(epochs=1)
        validation_error = model.evaluate()
        study.add_observation(trial=trial,
                              iteration=iteration,
                              objective=validation_error,
                              context={'training_error': training_error})
    study.finalize(trial)
```

### 2.3.4 Visualization

Once the `Study` object is initialized it will output the following:

SHERPA Dashboard running on http://...  

Following that link brings up the dashboard. The figure at the top of the dashboard is a parallel coordinates plot. It allows the user to brush over axes and thereby restrict ranges of the trials she wants to see. This is useful to find what objective values correspond to hyperparameters of a certain range. Similarly, one can brush over the objective value axis to find the best performing configurations. The table in the bottom left of the dashboard is linked to the plot. Therefore, it is easy to see what exact hyperparameters the filtered trials correspond to. One can also sort the table by any of its columns. Lastly, on the bottom right is a line plot that shows the progression of objective values for each trial. This is useful in analyzing how and if the training converges. Below is a screenshot of the dashboard towards the end of a study.
2.4 Available Algorithms

This section provides an overview of the available hyperparameter optimization algorithms in Sherpa. Below is a table that discusses use cases for each algorithm. This is followed by a short comparison benchmark and the algorithms themselves.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Use cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid Search</td>
<td>Great for understanding the impact of one or two parameters.</td>
</tr>
<tr>
<td>Random Search</td>
<td>More efficient than grid search when used with many hyperparameters. Great for getting a full picture of the impact of many hyperparameters since hyperparameters are uniformly sampled from the whole space.</td>
</tr>
<tr>
<td>Local Search</td>
<td>Can quickly explore “tweaks” to a model that is already good while using less trials than Random search or Bayesian optimization.</td>
</tr>
<tr>
<td>GPyOpt Bayesian Optimization</td>
<td>More efficient than Random search when the number of trials is sufficiently large.</td>
</tr>
<tr>
<td>Population Based Training</td>
<td>Can discover <em>schedules</em> of training parameters and is therefore especially good for learning rate, momentum, batch size, etc.</td>
</tr>
</tbody>
</table>

2.4.1 Comparison on MNIST MLP

The figure below shows the mean, minimum, and maximum across five runs of Random Search against the same number of GPyOpt Bayesian optimization, and Local Search runs. For the Local Search the individual trials are shown since each run finished after a different number of trials.

The currently available algorithms in Sherpa are listed below:

2.4.2 Grid Search

```python
class sherpa.algorithms.GridSearch(num_grid_points=2, repeat=1)
```

Explores a grid across the hyperparameter space such that every pairing is evaluated.

For continuous and discrete parameters grid points are picked within the range. For example, a continuous parameter with range [1, 2] with two grid points would have points 1 1/3 and 1 2/3. For three points, 1 1/4, 1
Example:

```python
hp_space = {'act': ['tanh', 'relu'],
            'lrinit': [0.1, 0.01],
            }
parameters = sherpa.Parameter.grid(hp_space)
alg = sherpa.algorithms.GridSearch()
```

Parameters **num_grid_points** (*int*) – number of grid points for continuous / discrete.

### 2.4.3 Random Search

```python
class sherpa.algorithms.RandomSearch(max_num_trials=None, repeat=1)
```

Random Search with a repeat option.

Trials parameter configurations are uniformly sampled from their parameter ranges. The repeat option allows to re-run a trial `repeat` number of times. By default this is 1.

Parameters
- **max_num_trials** (*int*) – number of trials, otherwise runs indefinitely.
- **repeat** (*int*) – number of times to repeat a parameter configuration.
2.4.4 Local Search

```python
class sherpa.algorithms.LocalSearch(seed_configuration, perturbation_factors=(0.8, 1.2), repeat_trials=1)
```

Local Search Algorithm.

This algorithm expects to start with a very good hyperparameter configuration. It changes one hyperparameter at a time to see if better results can be obtained.

**Parameters**
- `seed_configuration (dict)` – hyperparameter configuration to start with.
- `perturbation_factors (Union[tuple, list])` – continuous parameters will be multiplied by these.
- `repeat_trials (int)` – number of times that identical configurations are repeated to test for random fluctuations.

2.4.5 Bayesian Optimization with GPyOpt

2.4.6 Population Based Training

```python
class sherpa.algorithms.PopulationBasedTraining(population_size=20, parameter_range={}, perturbation_factors=(0.8, 1.0, 1.2))
```


PBT trains a generation of `population_size` seed trials (randomly initialized) for a user specified number of iterations. After that the same number of trials are sampled from the top 33% of the seed generation. Those trials are perturbed in their hyperparameter configuration and continue training. After that trials are sampled from that generation etc.

**Parameters**
- `population_size (int)` – the number of randomly initialized trials at the beginning and number of concurrent trials after that.
- `parameter_range (dict[Union[list, tuple]])` – upper and lower bounds beyond which parameters cannot be perturbed.
- `perturbation_factors (tuple[float])` – the factors by which continuous parameters are multiplied upon perturbation; one is sampled randomly at a time.

2.5 Bayesian Optimization

2.5.1 Background

Bayesian optimization for hyperparameter tuning uses a flexible model to map from hyperparameter space to objective values. In many cases this model is a Gaussian Process (GP) or a Random Forest. This model is fitted to inputs of hyperparameter configurations and outputs of objective values. The model is used to make predictions about candidate hyperparameter configurations. Each candidate-prediction can be evaluated with respect to its utility via an acquisition function. The algorithm therefore consists of fitting the model, finding the hyperparameter configuration that maximize the acquisition function, evaluating that configuration, and repeating the process.
2.5.2 GPyOpt Wrapper

SHERPA uses a wrapper for the Bayesian optimization library GPyOpt (https://github.com/SheffieldML/GPyOpt/).

SHERPA's Discrete parameter is treated like a continuous variable that is discretized after a value is suggested. Choice parameters are treated as GPyOpt discrete variables.

The GPyOpt algorithm in SHERPA takes as arguments a number of GPyOpt arguments as well as the maximum number of trials. The argument max_concurrent refers to batch size that GPyOpt produces at each step and should be chosen equal to the number of concurrent parallel trials. The algorithm also accepts seed configurations via the initial_data_points argument.

2.5.3 Example

Using GPyOpt Bayesian Optimization in SHERPA is straightforward. The parameter ranges are defined as usual:

```python
parameters = [sherpa.Continuous('lrinit', [0.1, 0.01], 'log'),
              sherpa.Continuous('momentum', [0., 0.99]),
              sherpa.Continuous('lrdecay', [1e-2, 1e-7], 'log'),
              sherpa.Continuous('dropout', [0., 0.5])]
```

When defining the algorithm the GPyOpt class is used:

```python
algorithm = sherpa.algorithms.GPyOpt(max_num_trials=150)
```

The max_num_trials argument is optional and specifies the number of trials after which the algorithm will finish. If not specified the algorithm will keep running and has to be cancelled by the user.

The optimization is set up as usual. Assume we have a trial script `trial_script.py` in which we want to minimize the loss, then we run the optimization as

```python
sherpa.optimize(parameters=parameters,
                 algorithm=algorithm,
                 lower_is_better=True,
                 filename='trial_script.py')
```

A full example for MNIST can be found in `mnist_mlp.ipynb` from the SHERPA root.

2.6 Local Search

2.6.1 Background

The goal for the Local Search algorithm is to start with a good hyperparameter configuration and test if it can be improved. The starting configuration could have been obtained through one of the other algorithms or from hand-tuning. The algorithm starts by evaluating the seed_configuration. It then perturbs one parameter at a time. If a new configuration achieves a better objective value than the seed then the new configuration is made the new seed.

Perturbations are applied as multiplication by a factor in the case of Continuous or Discrete variables. The default values are $0.8$ and $1.2$. These can be modified via the perturbation_factors argument. In the case of Ordinal variables, the parameter is shifted one up or down in the provided values. For Choice variables, another choice is randomly sampled.

Due to the fact that the Local Search algorithm is meant to fine-tune a hyperparameter configuration, it also has an option to repeat trials. The repeat_trials argument takes an integer that indicates how often a specific hyperpa-
parameter configuration should be repeated. Since performance differences caused by local changes may be small, this can help to establish significance.

```python
class sherpa.algorithms.LocalSearch
    (seed_configuration, perturbation_factors=(0.8, 1.2), repeat_trials=1)
```

Local Search Algorithm.

This algorithm expects to start with a very good hyperparameter configuration. It changes one hyperparameter at a time to see if better results can be obtained.

**Parameters**

- **seed_configuration (dict)** – hyperparameter configuration to start with.
- **perturbation_factors (Union[tuple,list])** – continuous parameters will be multiplied by these.
- **repeat_trials (int)** – number of times that identical configurations are repeated to test for random fluctuations.

### 2.6.2 Example

In this example we will work with the MNIST fully connected neural network from the Bayesian Optimization tutorial. We had tuned initial learning rate, learning rate decay, momentum, and dropout rate. The top parameter configuration we obtained was:

- **initial learning rate**: 0.038
- **learning rate decay**: 1.2e-4
- **momentum**: 0.92
- **dropout**: 0.

rounded to two digits. We use this as seed_configuration in the Local Search. We set the perturbation_factors as (0.9, 1.1). The algorithm will multiply one parameter by 0.9 or 1.1 at a time and see if these local changes can improve performance. If all changes have been tried and none improves on the seed configuration the algorithm stops. The example can be run as

```bash
cd sherpa/examples/mnistmlp/
python runner.py --algorithm LocalSearch
```

After running, we can inspect the results in the dashboard:

We find that fluctuations in performance due to random initialization are larger than small changes to the hyperparameters.

### 2.7 Population Based Training

#### 2.7.1 Background

Population Based Training (PBT) as introduced by Jaderberg et al. 2017 is an evolutionary algorithm for hyperparameter search. The diagram below is taken from Jaderberg et al. 2017 and gives an intuition on the algorithm. It starts with a random population of hyperparameter configurations. Each population member is trained for a limited amount of time and evaluated. When every population member has been evaluated, the ones with low scores replace their own weights and hyperparameters with those from population members with high scores (exploit) and perturb the hyperparameters (explore). Then all population members are trained and evaluated again and the process repeats.
2.7. Population Based Training
Note that only parameters can be tuned that can be changed during training. The number of layers in a neural network for example is better tuned with Bayesian Optimization.

### 2.7.2 SHERPA Implementation

SHERPA implements this algorithm as follows. A population of hyperparameter configurations is trained and evaluated for an amount of time implicitly specified by the user through the Trial-script. Population members are then sampled from the top third of the population. Population members are always sampled from the generation previous to the current one. For each sampled member each hyperparameter is randomly increased, decreased, or held constant. In particular, the hyperparameter is multiplied by 0.8, 1.0, or 1.2. These numbers can be adjusted via the `perturbation_factors` argument. The user can also specify parameter boundaries for this perturbation process via the `parameter_ranges` argument. These ranges operate independently from the ranges specified when defining each parameter. Instead this argument limits the range in which each parameter can be perturbed to. Ordinal parameters are moved one up or one down by the perturbation. Choice parameters are randomly resampled.

```python
class sherpa.algorithms.PopulationBasedTraining (population_size=20, parameter_range={}, perturbation_factors=(0.8, 1.0, 1.2))
```


PBT trains a generation of `population_size` seed trials (randomly initialized) for a user specified number of iterations. After that the same number of trials are sampled from the top 33% of the seed generation. Those trials are perturbed in their hyperparameter configuration and continue training. After that trials are sampled from that generation etc.

**Parameters**

- `population_size (int)` – the number of randomly initialized trials at the beginning and number of concurrent trials after that.
- `parameter_range (dict[Union[list,tuple])` – upper and lower bounds beyond which parameters cannot be perturbed.
- `perturbation_factors (tuple[float])` – the factors by which continuous parameters are multiplied upon perturbation; one is sampled randomly at a time.

### 2.7.3 Example

**Trial-Script**

In order to use PBT, the Trial-script needs to implement some additional functionality as compared to the regular Trial-script. The parameters are obtained as usual:

```python
client = sherpa.Client()
trial = client.get_trial()
```

**Load and Perturb**

The `trial.parameters` will now also contain the keys `load_from`, `save_to`, and `lineage`. The `lineage` indicates the heritage of this trial in terms of trial IDs and can be ignored at this point. The `load_from` key indicates whether weights need to be loaded. For example in Keras:
import keras.backend as K

if trial.parameters['load_from'] == '':
    model = create_model(trial.parameters)
else:
    model = load_model(os.path.join('./output', trial.parameters['load_from'] + '.h5'))
    K.set_value(model.optimizer.lr, trial.parameters['lr'])
    K.set_value(model.optimizer.momentum, trial.parameters['momentum'])

Note that when the model is loaded the user may have to change some hyperparameters manually like in this case the learning rate lr and momentum of the model.

Save

After the model is trained and evaluated it is crucial that it is saved using save_to. The user can choose where to save the models to and what exact name to give them so long each is identifiable by the number given in save_to and the path matches the path that the model is loaded from above.

model.save(os.path.join('./output', trial.parameters['save_to'] + '.h5'))

Runner Script

The runner script has the parameter ranges defined as usual.

parameters = [sherpa.Continuous('lrinit', [0.1, 0.01], 'log'),
              sherpa.Continuous('momentum', [0., 0.99]),
              sherpa.Ordinal(name='batch_size', range=[16, 32, 64])]

The parameters for the PBT algorithm are population_size, parameter_range, and perturbation_factors. Population size is the number of models that are randomly initialized at the beginning. The population_size is also the size of every generation thereafter. The parameter ranges correspond to ranges used by PBT for perturbation. The motivation for this parameter is that one may want the initial models to be sampled from the ranges provided in the regular way. The PBT parameter ranges may be larger allowing model parameters to drift in certain directions.

pbt_ranges = {'lr':[0.0000001, 1.], 'batch_size':[16, 32, 64, 128, 256, 512]}
algorithm = sherpa.algorithms.PopulationBasedTraining(population_size=50,
                                                      parameter_range=pbt_ranges)

And the optimization is called as before.

sherpa.optimize(algorithm=algorithm,
                scheduler=scheduler,
                parameters=parameters,
                lower_is_better=True,
                filename="mnist_cnn.py",
                output_dir='./output')

A full example can be found in examples/mnistcnnpbt/ from the SHERPA root and run with python runner.py. Below are some plots generated from the results of the example. The plot below shows seed trials as different colors with validation loss on the y-axis and epochs on the x-axis. It can be seen how for each trial at each epoch many different solutions are tried.

2.7. Population Based Training
For the trial with the best result one can plot the trajectory of the learning rate, batch size, and momentum up to that best performance (here achieved at epoch 15).

![PBT on MNIST CNN: Best Trial](image)


### 2.8 Writing Your Own Algorithm

Now we will take a look at how to create a new algorithm which will define the hyperparameters we will use to train the models. It defines the hyperparameters to use in the trials. It does not define the algorithm to train the model used in the trial, e.g. Stochastic Gradient Descent or Adam.

Every new algorithm inherits from the Algorithm Class and the main function we need to define is `get_suggestion()`. This function will receive information about the parameters it needs to define and returns a dictionary of hyperparameter values needed to train the next trial. The function `get_suggestion()` receives:

- **parameters**: List of `Parameter` objects.
- **results**: Dataframe storing the results of past trials.
- **lower_is_better**: Boolean specifying if lower is better in performance metric of trials.

With this information you are free to select the new hyperparameters in any way you want.

```python
import sherpa
class MyAlgorithm(sherpa.algorithms.Algorithm):
    def get_suggestion(self, parameters, results, lower_is_better):
```

(continues on next page)
For example let’s create a genetic-like algorithm which takes the trials from the top 1/3 of the trials and combines them to create the new set of hyperparameters. It will also randomly introduce a mutation 1/3 of the time.

The function `get_candidate()` will get the hyperparameters of a random trial among the top 1/3 and if there are very few trials, then it will generate them randomly. `get_suggestion()` is where the values for the hyperparameters of the new trial will be decided.

```python
import sherpa
import numpy as np
class MyAlgorithm(sherpa.algorithms.Algorithm):
    def get_suggestion(self, parameters, results, lower_is_better):
        
        Create a new parameter value as a random mixture of some of the best trials and sampling from original distribution.

        Returns:
        dict: parameter values dictionary

        # Choose 2 of the top trials and get their parameter values
        trial_1_params = self._get_candidate(parameters, results, lower_is_better)
        trial_2_params = self._get_candidate(parameters, results, lower_is_better)
        params_values_for_next_trial = {}
        for param_name in trial_1_params.keys():
```
param_origin = np.random.randint(3) # randomly choose where to get the value from
if param_origin == 1:
    params_values_for_next_trial[param_name] = trial_1_params[param_name]
elif param_origin == 2:
    params_values_for_next_trial[param_name] = trial_2_params[param_name]
else:
    for parameter_object in parameters:
        if param_name == parameter_object.name:
            params_values_for_next_trial[param_name] = parameter_object.sample()

return params_values_for_next_trial

def _get_candidate(self, parameters, results, lower_is_better, min_candidates=10):
    """
    Samples candidates parameters from the top 33% of population.
    ""
    Returns:
    dict: parameter dictionary.
    """
    if results.shape[0] > 0: # In case this is the first trial
        population = results.loc[results['Status'] != 'INTERMEDIATE', :]
        population = population.sort_values(by='Objective', ascending=lower_is_better)
        idx = numpy.random.randint(low=0, high=population.shape[0]//3)
        trial_all_values = population.iloc[idx].to_dict()
        trial_param_values = {param.name: d[param.name] for param in parameters}
        return trial_param_values
    else: # In case this is the first trial
        population = None
        if population is None or population.shape[0] < min_candidates: # Generate random values
            for parameter_object in parameters:
                trial_param_values[parameter_object.name] = parameter_object.sample()
            population = population.sort_values(by='Objective', ascending=lower_is_better)
            idx = numpy.random.randint(low=0, high=population.shape[0]//3) # pick randomly among top 33%
            trial_all_values = population.iloc[idx].to_dict() # extract the trial values
            on results table
            trial_param_values = {param.name: d[param.name] for param in parameters} # Select only parameter values
            return trial_param_values

2.9 Quickstart

Here we will show how to adapt a minimal Keras script so it can be used with Sherpa. As starting point we use the “getting started in 30 seconds” tutorial from the Keras webpage.

To run SHERPA you need a trial-script and a runner-script. The first specifies the machine learning model and will probably be very similar to the one you already have for Keras. The second one will specify information about SHERPA and the optimization.

2.9.1 Trial-script

For the trial.py we start by importing SHERPA and obtaining a trial. The trial will contain the hyperparameters that we are tuning.
import sherpa
client = sherpa.Client()
trial = client.get_trial()

Now we define the model, but for each tuning parameter we use trial.parameters[<name-of-parameter>]. For example the number of hidden units.

Before:
```python
from keras.models import Sequential
from keras.layers import Dense
model = Sequential()
model.add(Dense(units=64, activation='relu', input_dim=100))
model.add(Dense(units=10, activation='softmax'))
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

After:
```python
from keras.models import Sequential
from keras.layers import Dense
model = Sequential()
model.add(Dense(units=trial.parameters['num_units'], activation='relu', input_dim=100))
model.add(Dense(units=10, activation='softmax'))
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

For the training of the model, we include a callback to send the information back to SHERPA at the end of each epoch so it can update the state of it and decide if it should continue training. Here you can include all the usual Keras callbacks as well.

Before:
```python
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

After:
```python
callbacks = [client.keras_send_metrics(trial, objective_name='val_loss',
                                        context_names=['val_acc'])]
model.fit(x_train, y_train, epochs=5, batch_size=32, callbacks=callbacks)
```

### 2.9.2 Runner-script

Now we are going to create the runner-script in a file called runner.py and specify our hyperparameter `num_units` along with information for the hyperparameter algorithm, in this case Random Search.

```python
import sherpa
parameters = [sherpa.Choice('num_units', [100, 200, 300]),]
alg = sherpa.algorithms.RandomSearch(max_num_trials=150)
rval = sherpa.optimize(parameters=parameters,
                        algorithm=alg,
                        lower_is_better=True,  # Minimize objective
                        filename='./trial.py',  # Python script to run, where the model...
                        )
```
And that’s it! Now to run your hyperparameter optimization you just have to do:

```
python runner.py
```

### 2.10 Setup for Parallel Computation

Install dependencies:

```
pip install pymongo
```

#### 2.10.1 Mongo DB

Training models in parallel with SHERPA requires MongoDB. If you are using a cluster, chances are that it is already installed, so check for that. Otherwise the [installation guide for Linux](https://docs.mongodb.com/manual/administration/install-on-linux/) is straightforward. For MacOS, MongoDB can either be installed via Homebrew

```
brew update
brew install mongodb
```

or via the [instructions](https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/).

#### 2.10.2 Example

To verify SHERPA with MongoDB is working:

```
cd sherpa/examples/parallel-examples/
python simple.py
```

### 2.11 Parallel Guide

This section expands on the Keras-to-Sherpa tutorial in that it goes more into detail about the configuration options. An optimization in SHERPA consists of a trial-script and a runner-script.

#### 2.11.1 Trial-script

The trial-script trains your machine learning model with a given parameter-configuration and sends metrics to SHERPA. To get a trial, use the `Client`:

```
import sherpa

client = sherpa.Client()
trial = client.get_trial()
```
The client will connect to the MongoDB instance created by the Runner-script (more below). From that it obtains a hyperparameter configuration i.e. a trial. The trial contains the parameter configuration for your training:

```python
# Model training
num_iterations = 10
for i in range(num_iterations):
    pseudo_objective = trial.parameters['param_a'] / float(i + 1) * trial.parameters['param_b']
    client.send_metrics(trial=trial, iteration=i+1, objective=pseudo_objective)
```

During training `send_metrics` is used every iteration to return objective values to SHERPA i.e. send them to the MongoDB instance. When using Keras the client also has a callback `Client.keras_send_metrics` that can be used directly.

### 2.11.2 Runner-script

The runner-script defines the optimization and runs SHERPA. Parameters are defined as a list of Parameter-objects. For a list of the available parameters see [here](#).

```python
import sherpa
parameters = [sherpa.Choice(name="param_a", range=[1, 2, 3]),
              sherpa.Continuous(name="param_b", range=[0, 1])]
```

Once you decided on the parameters and their ranges you can choose an optimization algorithm.

```python
algorithm = sherpa.algorithms.RandomSearch(max_num_trials=10)
```

Schedulers allow to run an optimization on one machine or a cluster:

```python
scheduler = sherpa.schedulers.LocalScheduler()
```

The optimization is run via :ref:`sherpa.optimize <optimize-api>`:

```python
results = sherpa.optimize(parameters=parameters,
                          algorithm=algorithm, lower_is_better=True,
                          filename=filename, output_dir=tempdir,
                          scheduler=scheduler, max_concurrent=2,
                          verbose=1)
```

The code for this example can be run as `python ./examples/runner_mode.py` from the SHERPA root.

### 2.12 SGE

The SGEScheduler class allows SHERPA to run hyperparameter optimizations via the Sun Grid Engine. This works just like you would use a grid. While SHERPA is running it calls `qsub` with a temporary bash script that loads your environment, sets any SHERPA specific environment variables, and runs your trial-script.
Using the `SGEScheduler`, optimizations can easily be scheduled to run a large number of concurrent instances of the trial-script. Below is the `SGEScheduler` class. Keep reading for more information on the environment and submit options.

```python
class sherpa.schedulers.SGEScheduler(submit_options, environment, output_dir="")
```

Submits jobs to SGE, can check on their status, and kill jobs.

Uses `drmaa` Python library. Due to the way SGE works it cannot distinguish between a failed and a completed job.

**Parameters**

- `submit_options` (*str*)  
  Command line options such as `queue -q`, or `-P` for project, all written in one string.

- `environment` (*str*)  
  The path to a file that contains environment variables; will be sourced before job is run.

- `output_dir` (*str*)  
  Path to directory in which `stdout` and `stderr` will be written to. If not specified this will use the same as defined for the study.

### 2.12.1 Your environment profile

In order to use SHERPA with a grid scheduler you will have to set up a profile with environment variables. This will be loaded every time a job is submitted. An SGE job will not load your `.bashrc` so all necessary settings need to be in your profile.

For example, in the case of training machine learning models on a GPU, the profile might contain environment variables relating to CUDA or activating a container that contains the requirements. If you installed SHERPA via Git, then the profile also might have to add the SHERPA folder to the `PYTHONPATH`. Finally, your environment might load a virtual environment that contains your personal Python packages.

### 2.12.2 SGE submit options

SGE requires submit options. In Sherpa, those are defined as a string via the `submit_options` argument in the scheduler. The string is attached after the `qsub` command that SHERPA issues. To figure out what submit options are needed for your setup you might want to refer to the cluster documentation, group-wiki, or system administrator. In general, you will need

- `-N`: the job name
- `-P`: the project name
- `-q`: the queue name.

### 2.12.3 Running it

Note that while SHERPA is running in your runner-script it will repeatedly submit your trial-script to SGE using `qsub`. It is preferable to run the runner-script itself in an interactive session since it is useful to be able to monitor the output as it is running.

### 2.13 Core

#### 2.13.1 Setting up the Optimization
Parameters

```python
class sherpa.core.Continuous(name, range, scale='linear')
Continuous parameter class.

class sherpa.core.Discrete(name, range, scale='linear')
Discrete parameter class.

class sherpa.core.Choice(name, range)
Choice parameter class.

class sherpa.core.Ordinal(name, range)
Ordinal parameter class. Categorical, ordered variable.

class sherpa.core.Parameter(name, range)
Defines a hyperparameter with a name, type and associated range.
```

Parameters

- **name** (*str*) – the parameter name.
- **range** (*list*) – either [low, high] or [value1, value2, value3].
- **scale** (*str*) – linear or log, defines sampling from linear or log-scale. Not defined for all parameter types.

```python
static from_dict(config)
Returns a parameter object according to the given dictionary config.
```

Parameters

- **config** (*dict*) – parameter config.

Example:

```python
{'name': '<name>',
 'type': '<continuous/discrete/choice>',
 'range': [<value1>, <value2>, ...],
 'scale': '<'log' to sample continuous/discrete from log-scale>'
}
```

Returns the parameter range object.

Return type sherpa.core.Parameter

```python
static grid(parameter_grid)
Creates a list of parameters given a parameter grid.
```

Parameters

- **parameter_grid** (*dict*) – Dictionary mapping hyperparameter names lists of possible values.

Example

```python
{'parameter_a': [aValue1, aValue2, ...],
 'parameter_b': [bValue1, bValue2, ...],
 ...}
```

Returns list of parameter ranges for SHERPA.

Return type list[sherpa.core.Parameter]
Study

class sherpa.core.Study(parameters, algorithm, lower_is_better, stopping_rule=None, dashboard_port=None, disable_dashboard=False, output_dir=None)

The core of an optimization.

Includes functionality to get new suggested trials and add observations for those. Used internally but can also be used directly by the user.

Parameters

• parameters (list[sherpa.core.Parameter]) – a list of parameter ranges.
• algorithm (sherpa.algorithms.Algorithm) – the optimization algorithm.
• lower_is_better (bool) – whether to minimize or maximize the objective.
• stopping_rule (sherpa.algorithms.StoppingRule) – algorithm to stop badly performing trials.
• dashboard_port (int) – the port for the dashboard web-server, if None the first free port in the range 8880 to 9999 is found and used.
• disable_dashboard (bool) – option to not run the dashboard.
• output_dir (str) – directory path for CSV results.

add_observation(trial, iteration, objective, context={})

Add a single observation of the objective value for a given trial.

Parameters

• trial (sherpa.core.Trial) – trial for which an observation is to be added.
• iteration (int) – iteration number e.g. epoch.
• objective (float) – objective value.
• context (dict) – other metrics or values to record.

add_trial(trial)

Adds a trial into queue for next suggestion.

Trials added via this method forego the suggestions made by the algorithm and are returned by the get_suggestion method on a first in first out basis.

Parameters trial (sherpa.core.Trial) – the trial to be enqueued.

finalize(trial, status='COMPLETED')

Once a trial will not add any more observations it must be finalized with this function.

Parameters

• trial (sherpa.core.Trial) – trial that is completed.
• status (str) – one of ‘COMPLETED’, ‘FAILED’, ‘STOPPED’.

get_best_result()

Retrieve the best result so far.

Returns row of the best result.

Return type pandas.DataFrame

get_suggestion()

Obtain a new suggested trial.

This function wraps the algorithm that was passed to the study.
Returns a parameter suggestion.

Return type dict

**keras_callback** *(trial, objective_name, context_names=[])*

Keras Callbacks to add observations to study

Parameters

- **trial** *(sherpa.core.Trial)* – trial to send metrics for.
- **objective_name** *(str)* – the name of the objective e.g. *loss*, *val_loss*, or any of the submitted metrics.
- **context_names** *(list[str])* – names of all other metrics to be monitored.

**static load_dashboard** *(path)*

Loads a study from an output dir without the algorithm.

Parameters **path** *(str)* – the path to the output dir.

Returns

the study running the dashboard, note that currently this study cannot be used to continue the optimization.

Return type sherpa.core.Study

**save** *(output_dir=None)*

Stores results to CSV and attributes to config file.

Parameters **output_dir** *(str)* – directory to store CSV to, only needed if Study output_dir is not defined.

**should_trial_stop** *(trial)*

Determines whether given trial should stop.

This function wraps the stopping rule provided to the study.

Parameters **trial** *(sherpa.core.Trial)* – trial to be evaluated.

Returns decision.

Return type bool

### Running the Optimization in Parallel

**sherpa.core.optimize** *(parameters, algorithm, lower_is_better, scheduler, filename, output_dir='./output_20190208-144730', max_concurrent=1, db_port=None, stopping_rule=None, dashboard_port=None, resubmit_failed_trials=False, verbose=1, load=False, mongodb_args={})*

Runs a Study with a scheduler and automatically runs a database in the background.

Parameters

- **algorithm** *(sherpa.algorithms.Algorithm)* – takes results table and returns parameter set.
- **parameters** *(list[sherpa.core.Parameter])* – parameters being optimized.
- **lower_is_better** *(bool)* – whether lower objective values are better.
- **filename** *(str)* – the name of the file which is called to evaluate configurations
- **output_dir** *(str)* – where scheduler and database files will be stored.
- **scheduler** *(sherpa.schedulers.Scheduler)* – a scheduler.
• **max_concurrent** (*int*) – the number of trials that will be evaluated in parallel.
• **db_port** (*int*) – port to run the database on.
• **stopping_rule** (*sherpa.algorithms.StoppingRule*) – rule for stopping trials prematurely.
• **dashboard_port** (*int*) – port to run the dashboard web-server on.
• **resubmit_failed_trials** (*bool*) – whether to resubmit a trial if it failed.
• **verbose** (*int*, default=1) – whether to print submit messages (0=no, 1=yes).
• **load** (*bool*) – option to load study, currently not fully implemented.
• **mongodb_args** (*dict[str, any]*) – arguments to MongoDB beyond port, dir, and log-path. Keys are the argument name without “-”.

### 2.13.2 Setting up the Trial

**Client**

```python
class sherpa.database.Client:

Registers a session with a Sherpa Study via the port of the database.
```

This function is called from trial-scripts only.

**Variables**

- **host** (*str*) – the host that runs the database. Passed host, host set via environment variable or ‘localhost’ in that order.
- **port** (*int*) – port that database is running on. Passed port, port set via environment variable or 27010 in that order.

**get_trial()**

Returns the next trial from a Sherpa Study.

- **Returns** The trial to run.
- **Return type** sherpa.core.Trial

**keras_send_metrics(trial, objective_name, context_names=[])**

Keras Callbacks to send metrics to SHERPA.

**Parameters**

- **trial** (*sherpa.core.Trial*) – trial to send metrics for.
- **objective_name** (*str*) – the name of the objective e.g. *loss*, *val_loss*, or any of the submitted metrics.
- **context_names** (*list[str]*) – names of all other metrics to be monitored.

**send_metrics(trial, iteration, objective, context={})**

Sends metrics for a trial to database.

**Parameters**

- **trial** (*sherpa.core.Trial*) – trial to send metrics for.
- **iteration** (*int*) – the iteration e.g. epoch the metrics are for.
- **objective** (*float*) – the objective value.
- **context** (*dict*) – other metric-values.
2.14 Schedulers

```python
class sherpa.schedulers.SGEScheduler(submit_options, environment, output_dir="")
```

Submits jobs to SGE, can check on their status, and kill jobs.

Uses `drmaa` Python library. Due to the way SGE works it cannot distinguish between a failed and a completed job.

**Parameters**

- `submit_options (str)` – command line options such as queue `-q`, or `-P` for project, all written in one string.
- `environment (str)` – the path to a file that contains environment variables; will be sourced before job is run.
- `output_dir (str)` – path to directory in which `stdout` and `stderr` will be written to. If not specified this will use the same as defined for the study.

```python
class sherpa.schedulers.LocalScheduler(submit_options=", output_dir=", resources=None)
```

Runs jobs locally as a subprocess.

**Parameters**

- `submit_options (str)` – options appended before the command.
- `resources (list[str])` – list of resources that will be passed as SHERPARESOURCE environment variable. If no resource is available `''` will be passed.

2.15 Algorithms

2.15.1 Optimization Algorithms

```python
class sherpa.algorithms.Algorithm
```

Abstract algorithm that generates new set of parameters.

```python
def get_suggestion(parameters, results, lower_is_better)
```

Returns a suggestion for parameter values.

**Parameters**

- `parameters (list[sherpa.Parameter])` – the parameters.
- `results (pandas.DataFrame)` – all results so far.
- `lower_is_better (bool)` – whether lower objective values are better.

**Returns**

Parameter values.

**Return type** dict

```python
def load(num_trials)
```

Reinstantiates the algorithm when loaded.

**Parameters**

- `num_trials (int)` – number of trials in study so far.

```python
class sherpa.algorithms.RandomSearch(max_num_trials=None, repeat=1)
```

Random Search with a repeat option.

Trials parameter configurations are uniformly sampled from their parameter ranges. The repeat option allows to re-run a trial `repeat` number of times. By default this is 1.
Parameters

- `max_num_trials (int)` – number of trials, otherwise runs indefinitely.
- `repeat (int)` – number of times to repeat a parameter configuration.

```python
class sherpa.algorithms.GridSearch(num_grid_points=2, repeat=1)
```

Explores a grid across the hyperparameter space such that every pairing is evaluated.

For continuous and discrete parameters grid points are picked within the range. For example, a continuous parameter with range [1, 2] with two grid points would have points 1 1/3 and 1 2/3. For three points, 1 1/4, 1 1/2, and 1 3/4.

Example:

```python
hp_space = {'act': ['tanh', 'relu'],
            'lrinit': [0.1, 0.01],
            }
parameters = sherpa.Parameter.grid(hp_space)
alg = sherpa.algorithms.GridSearch()
```

- `num_grid_points (int)` – number of grid points for continuous / discrete.

```python
class sherpa.algorithms.LocalSearch(seed_configuration, perturbation_factors=(0.8, 1.2), repeat_trials=1)
```

Local Search Algorithm.

This algorithm expects to start with a very good hyperparameter configuration. It changes one hyperparameter at a time to see if better results can be obtained.

Parameters

- `seed_configuration (dict)` – hyperparameter configuration to start with.
- `perturbation_factors (Union[tuple,list])` – continuous parameters will be multiplied by these.
- `repeat_trials (int)` – number of times that identical configurations are repeated to test for random fluctuations.

```python
class sherpa.algorithms.PopulationBasedTraining(population_size=20, parameter_range={}, perturbation_factors=(0.8, 1.0, 1.2))
```


PBT trains a generation of `population_size` seed trials (randomly initialized) for a user specified number of iterations. After that the same number of trials are sampled from the top 33% of the seed generation. Those trials are perturbed in their hyperparameter configuration and continue training. After that trials are sampled from that generation etc.

Parameters

- `population_size (int)` – the number of randomly initialized trials at the beginning and number of concurrent trials after that.
- `parameter_range (dict[Union[list,tuple]])` – upper and lower bounds beyond which parameters cannot be perturbed.
- `perturbation_factors (tuple[float])` – the factors by which continuous parameters are multiplied upon perturbation; one is sampled randomly at a time.
2.15.2 Stopping Rules

```python
class sherpa.algorithms.MedianStoppingRule(min_iterations=0, min_trials=1)
Median Stopping-Rule similar to Golovin et al. “Google Vizier: A Service for Black-Box Optimization”.

- For a Trial t, the best objective value is found.
- Then the best objective value for every other trial is found.
- Finally, the best-objective for the trial is compared to the median of the best-objectives of all other trials.

If trial t's best objective is worse than that median, it is stopped.
If t has not reached the minimum iterations or there are not yet the requested number of comparison trials, t is not stopped. If t is all nan’s it is stopped by default.

Parameters
- **min_iterations (int)** – the minimum number of iterations a trial runs for before it is considered for stopping.
- **min_trials (int)** – the minimum number of comparison trials needed for a trial to be stopped.

```should_trial_stop(trial, results, lower_is_better)```

Parameters
- **trial (sherpa.Trial)** – trial to be stopped.
- **results (pandas.DataFrame)** – all results so far.
- **lower_is_better (bool)** – whether lower objective values are better.

Returns decision.
Return type bool

2.16 Development

2.16.1 How to contribute

The easiest way to contribute to SHERPA is to implement new algorithms or new schedulers.

Style Guide

SHERPA uses Google style Python doc-strings (e.g. here).

Unit Testing

Unit tests are organized in scripts under /tests/ from the SHERPA root: test_sherpa.py tests core features of SHERPA, test_algorithms.py tests implemented algorithms, and test_schedulers.py tests schedulers. The file long_tests.py does high level testing of SHERPA and takes longer to run. All testing makes use of pytest, especially pytest.fixtures. The mock module is also used.


### 2.16.2 SHERPA Code Structure

#### Study and Trials

In Sherpa a parameter configuration corresponds to a Trial object and a parameter optimization corresponds to a Study object. A trial has an ID attribute and a dict of parameter name-value pairs.

```python
class sherpa.core.Trial(id, parameters)
    Represents one parameter-configuration here referred to as one trial.
```

**Parameters**

- `id` (int) – the Trial ID.
- `parameters` (dict) – parameter-name, parameter-value pairs.

A study comprises the results of a number of trials. It also provides methods for adding a new observation for a trial to the study (`add_observation`), finalizing a trial (`finalize`), getting a new trial (`get_suggestion`), and deciding whether a trial is performing worse than other trials and should be stopped (`should_trial_stop`).

```python
class sherpa.core.Study(parameters, algorithm, lower_is_better, stopping_rule=None, dashboard_port=None, disable_dashboard=False, output_dir=None)
    The core of an optimization.
    Includes functionality to get new suggested trials and add observations for those. Used internally but can also be used directly by the user.
```

**Parameters**

- `parameters` (list[sherpa.core.Parameter]) – a list of parameter ranges.
- `algorithm` (sherpa.algorithms.Algorithm) – the optimization algorithm.
- `lower_is_better` (bool) – whether to minimize or maximize the objective.
- `stopping_rule` (sherpa.algorithms.StoppingRule) – algorithm to stop badly performing trials.
- `dashboard_port` (int) – the port for the dashboard web-server, if `None` the first free port in the range 8880 to 9999 is found and used.
- `disable_dashboard` (bool) – option to not run the dashboard.
- `output_dir` (str) – directory path for CSV results.

In order to propose new trials or decide whether a trial should stop, the study holds an sherpa.algorithms.Algorithm instance that yields new trials and a sherpa.algorithms.StoppingRule that yields decisions about performance. When using Sherpa in API-mode the user directly interacts with the study.

#### Runner

The _Runner class automates the process of interacting with the study. It consists of a loop that updates results, updates currently running jobs, stops trials if necessary and submits new trials if necessary. In order to achieve this it interacts with a sherpa.database._Database object and a sherpa.schedulers.Scheduler object.

```python
class sherpa.core._Runner(study, scheduler, database, max_concurrent, command, resubmit_failed_trials=False)
    Encapsulates all functionality needed to run a Study in parallel.
```

**Responsibilities:**

- Get rows from database and check if any new observations need to be added to Study.
• Update active trials, finalize any completed/stopped/failed trials.

• **Check what trials should be stopped and call scheduler kill_job method.**

• Check if new trials need to be submitted, get parameters and submit as a job.

**Parameters**

• **study** (*sherpa.core.Study*) – the study that is run.

• **scheduler** (*sherpa.schedulers.Scheduler*) – a scheduler object.

• **database** (*sherpa.database._Database*) – the database.

• **max_concurrent** (*int*) – how many trials to run in parallel.

• **command** (*list[str]*) – components of the command that runs a trial script e.g. [“python”, “train_nn.py”].

• **resubmit_failed_trials** (*bool*) – whether a failed trial should be resubmitted.

**Putting it all together**

The user does not directly interact with the `_Runner` class. Instead it is wrapped by the function `sherpa.optimize` that sets up the database and takes algorithm and scheduler as arguments from the user.

### 2.17 Writing Schedulers

A new scheduler inherits from the `sherpa.schedulers.Scheduler` class and re-implements its methods `submit_job`, `get_status`, and `kill_job`.

```python
class sherpa.schedulers.Scheduler
    The job scheduler gives an API to submit jobs, retrieve statuses of specific jobs, and kill a job.

    get_status (job_id)
    Obtains the current status of the job.
    Parameters job_id (str) – identifier returned when submitting the job.
    Returns the job-status.
    Return type sherpa.schedulers._JobStatus

    kill_job (job_id)
    Kills a given job.
    Parameters job_id (str) – identifier returned when submitting the job.

    submit_job (command, env={}, job_name=")
    Submits a job to the scheduler.
    Parameters
    • command (list[str]) – components to the command to run by the scheduler e.g. ["python", "train.py"]
    • env (dict) – environment variables to pass to the job.
    • job_name (str) – this specifies a name for the job and its output directory.
    Returns a job ID, used for getting the status or killing the job.
    Return type str
```