SHERPA Documentation

Lars Hertel, Peter Sadowski, and Julian Collado

Jul 31, 2020

Contents


SHERPA

CHAPTER 1

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.

See also:

If you are looking for the similarly named package "Sherpa" for modelling and fitting data go here: https://sherpa.readthedocs.io

CHAPTER 2

Documentation Contents

2.1 Installation

2.1.1 Installation from PyPi

This is the most straightforward way to install Sherpa.

```
pip install parameter-sherpa
```

However, since the source is regularly updated we **recommend to clone from GitHub** as described below.

2.1.2 Installation from GitHub

Clone from GitHub:

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

Here you might want to add *export PYTHONPATH=$PYTHONPATH:/home/packages/sherpa/* to your *.bash_profile* or *.bash_rc* so you won't have to run that line every time you re-open the terminal. Replace */home/packages/sherpa/* with the absolute path to

> the Sherpa folder on your system.

Install dependencies:

```
pip install pandas
pip install numpy
pip install scipy
pip install scikit-learn
pip install flask
pip install gpyopt
```

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:

```
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.

```
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:

```
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.

```
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:

```
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'])]
```

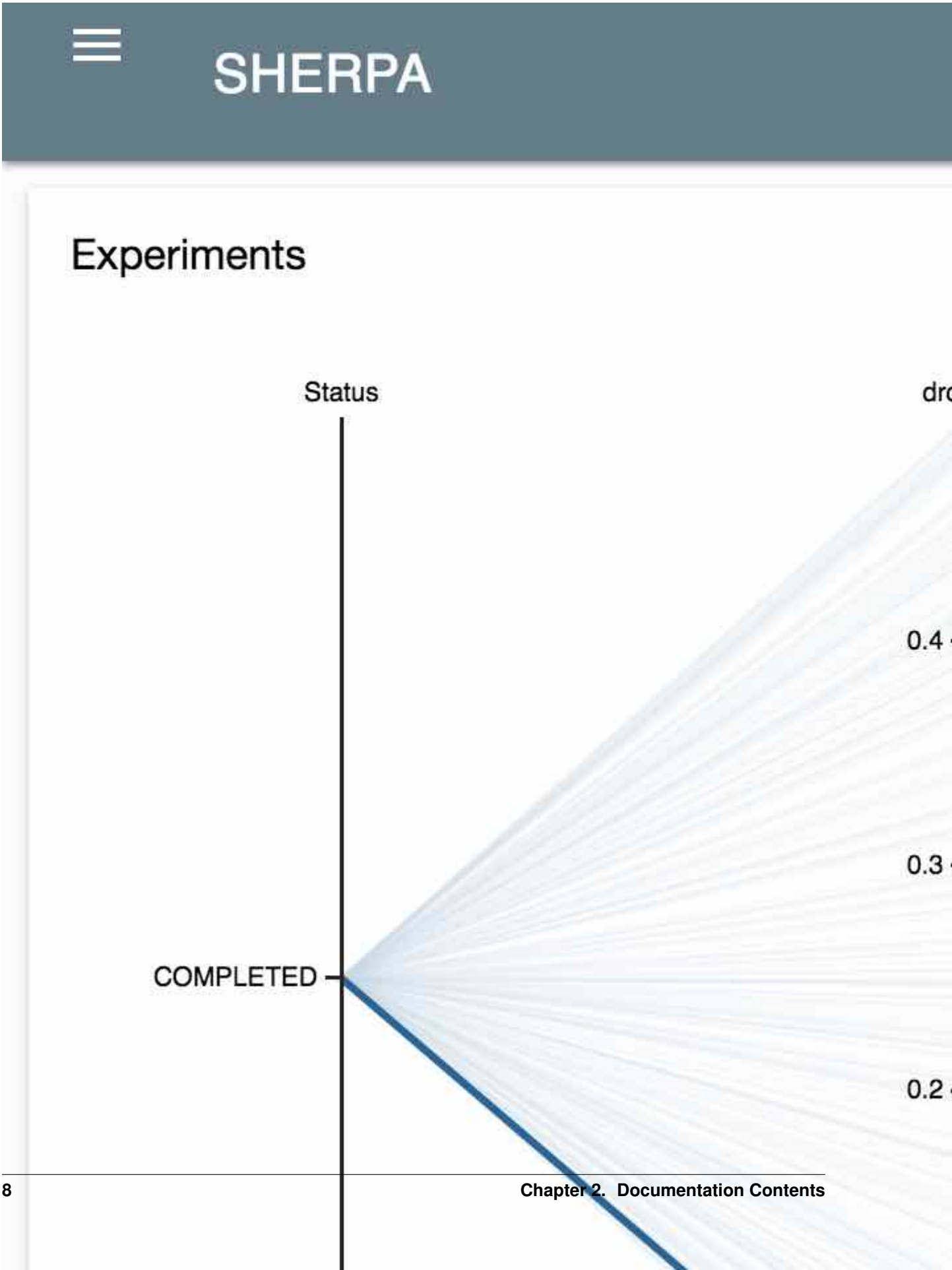
Note that it is generally recommended not to represent continuous or discrete parameters as categorical. This is due to the fact that exploring a range of values rather than discrete options yields much more information to understand the relationship between the hyperparameter and the outcome.

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.


SHERPA
Experiments
Status
COMPLETED
0.4
0.3
0.2

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.

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

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

```
for trial in study:
    model = init_model(trial.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

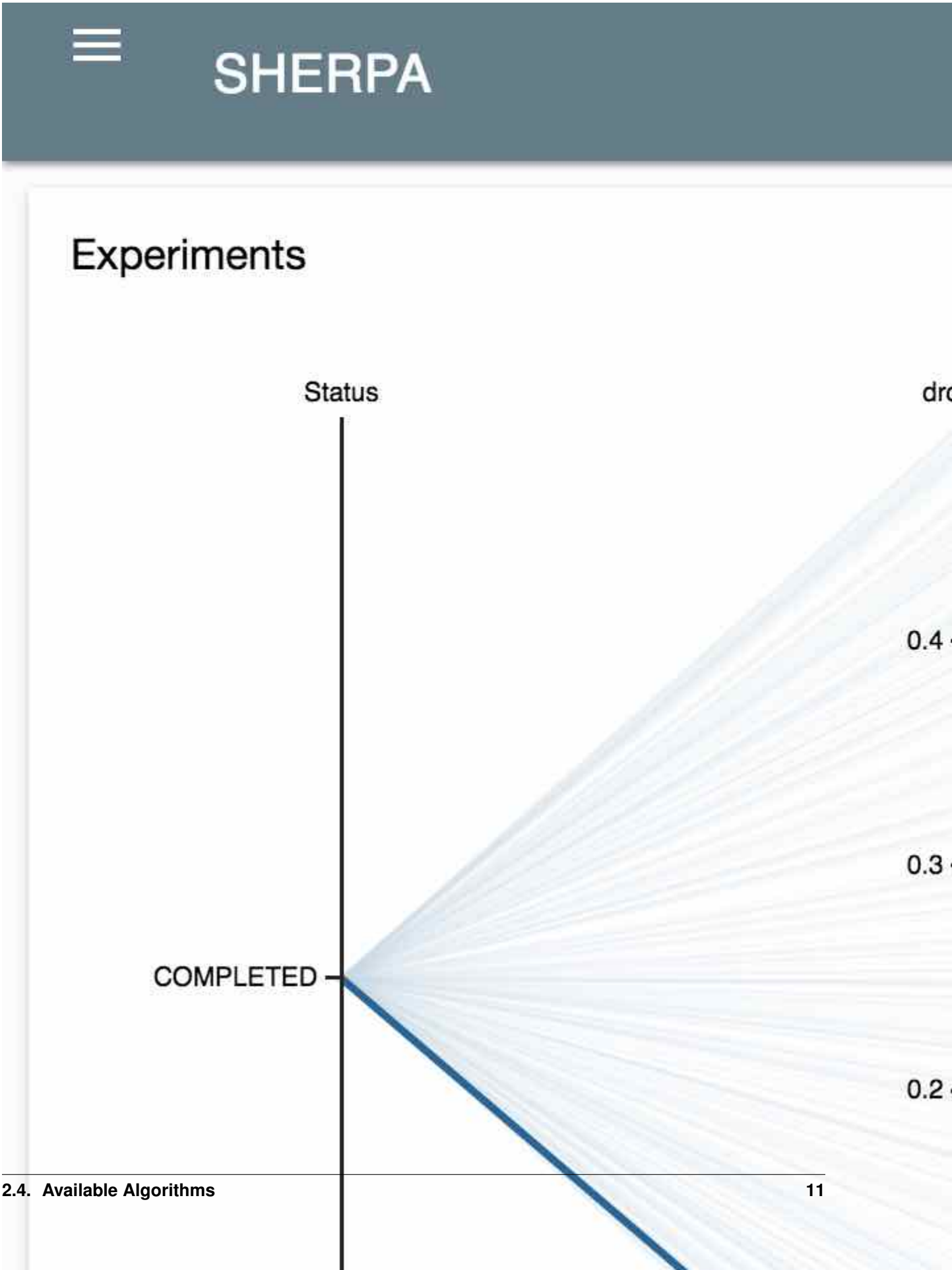
Once the `Study` object is initialized it will output the following:

```
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.

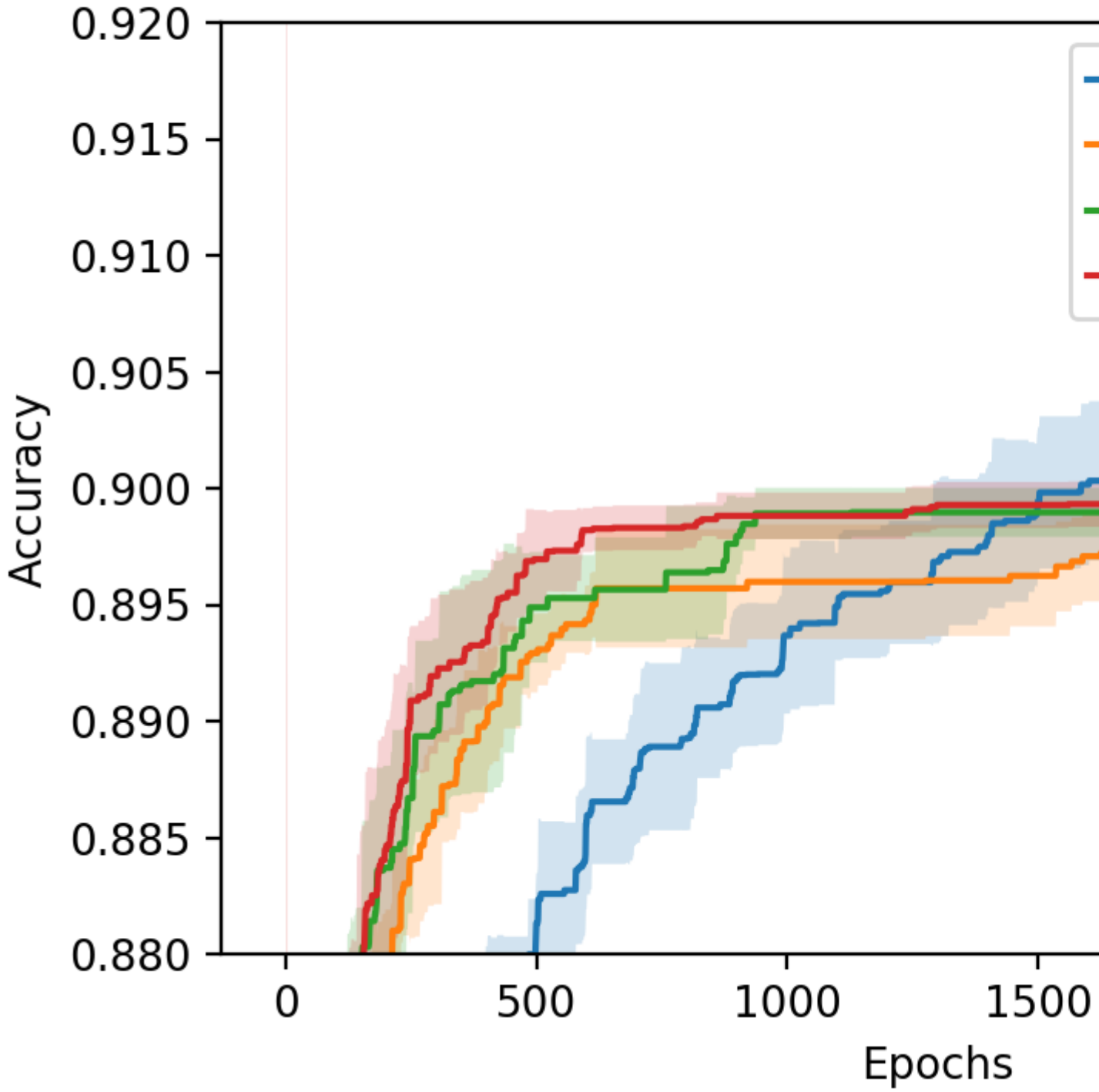

SHERPA
Experiments
Status
dro
0.4
0.3
COMPLETED
0.2

	Use cases
Grid Search	Great for understanding the impact of one or two parameters.
Random Search	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.
GPyOpt Bayesian Optimization	More efficient than Random search when the number of trials is sufficiently large.
Asynchronous Successive Halving	Due to its early stopping, especially useful when it would otherwise be infeasible to run a hyperparameter optimization because of the computational cost.
Local Search	Can quickly explore "tweaks" to a model that is already good while using less trials than Random search or Bayesian optimization.
Population Based Training	Can discover *schedules* of training parameters and is therefore especially good for learning rate, momentum, batch size, etc.

For the specification of each algorithm see below.

2.4.1 Comparison using Fashion MNIST MLP

We constructed a simple and fast to run benchmark to run these algorithms on. This uses a fully connected neural network trained on the Fashion MNIST dataset. The tuning parameters are the learning rate, the learning rate decay, the momentum, minibatch size, and the dropout rate. We compare Random Search, GPyOpt, Population Based Training (pbt), and Successive Halving. All algorithms are allowed an equal budget corresponding to 100 models trained for 26 epochs. The plot below shows the mean taken over five runs of each algorithm. The shaded regions correspond to two standard deviations. On the y-axis is the classification accuracy of the best model found. On the x-axis are the epochs spent. We run 20 evaluations in parallel. Note that the results of GPyOpt may be impacted by the high number of parallel evaluations. The code to reproduce this benchmark can be found at `sherpa/examples/parallel-examples/fashion_mnist_benchmark`.


0.920
0.915
0.910
0.905
0.900
0.895
0.890
0.885
0.880
Accuracy
0
500
1000
1500
Epochs

The currently available algorithms in Sherpa are listed below:

2.4.2 Grid Search

class sherpa.algorithms.**GridSearch**(*num_grid_points=2*)

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:

```
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

class sherpa.algorithms.**RandomSearch**(*max_num_trials=None*)

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 Bayesian Optimization with GPyOpt

class sherpa.algorithms.**GPyOpt**(*model_type='GP'*, *num_initial_data_points='infer'*, *initial_data_points=[]*, *acquisition_type='EI'*, *max_concurrent=4*, *verbosity=False*, *max_num_trials=None*)

Sherpa wrapper around the GPyOpt package (https://github.com/SheffieldML/GPyOpt).

Parameters

- **model_type** (*str*) – The model used: - 'GP', standard Gaussian process. - 'GP_MCMC', Gaussian process with prior in the hyper-parameters. - 'sparseGP', sparse Gaussian process. - 'warperdGP', warped Gaussian process. - 'InputWarpedGP', input warped Gaussian process - 'RF', random forest (scikit-learn).
- **num_initial_data_points** (*int*) – Number of data points to collect before fitting model. Needs to be greater/equal to the number of hyper- parameters that are being optimized. Using default 'infer' corres- ponds to number of hyperparameters + 1 or 0 if results are not empty.
- **initial_data_points** (*list[dict] or pandas.Dataframe*) – Specifies initial data points. If len(initial_data_points)<num_initial_data_points then the rest is randomly sampled. Use this option to provide hyperparameter configurations that are known to be good.

- **acquisition_type** (*str*) – Type of acquisition function to use. - 'EI', expected improvement. - 'EI_MCMC', integrated expected improvement (requires GP_MCMC model). - 'MPI', maximum probability of improvement. - 'MPI_MCMC', maximum probability of improvement (requires GP_MCMC model). - 'LCB', GP-Lower confidence bound. - 'LCB_MCMC', integrated GP-Lower confidence bound (requires GP_MCMC model).
- **max_concurrent** (*int*) – The number of concurrent trials. This generates a batch of max_concurrent trials from GPyOpt to evaluate. If a new observation becomes available, the model is re-evaluated and a new batch is created regardless of whether the previous batch was used up. The used method is local penalization.
- **verbosity** (*bool*) – Print models and other options during the optimization.
- **max_num_trials** (*int*) – maximum number of trials to run for.

2.4.5 Asynchronous Successive Halving aka Hyperband

class sherpa.algorithms.**SuccessiveHalving**(*r=1*, *R=9*, *eta=3*, *s=0*, *max_finished_configs=50*)

Asynchronous Successive Halving as described in:

> @article{li2018massively, title={Massively parallel hyperparameter tuning}, author={Li, Liam and Jamieson, Kevin and Rostamizadeh, Afshin and Gonina, Ekaterina and Hardt, Moritz and Recht, Benjamin and Talwalkar, Ameet}, journal={arXiv preprint arXiv:1810.05934}, year={2018} }

Asynchronous successive halving operates based on the multi-armed bandit algorithm Successive Halving (SHA) and performs principled early stopping for random search.

Parameters

- **r** (*int*) – minimum resource that each configuration will be trained for.
- **R** (*int*) – maximum resource.
- **eta** (*int*) – elimination rate.
- **s** (*int*) – minimum early-stopping rate.
- **max_finished_configs** (*int*) – stop once max_finished_configs models have been trained to completion.

2.4.6 Local Search

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.

2.4.7 Population Based Training

class sherpa.algorithms.**PopulationBasedTraining**(*num_generations*, *population_size=20*, *parameter_range={}*, *perturbation_factors=(0.8, 1.2)*)

Population based training (PBT) as introduced by Jaderberg et al. 2017.

PBT trains a generation of `population_size` seed trials (randomly initialized) for a user specified number of iterations e.g. one epoch. The top 80% then move on unchanged into the second generation. The bottom 20% are re-sampled from the top 20% and perturbed. The second generation again trains for the same number of iterations and the same procedure is repeated to move into the third generation etc.

Parameters

- **num_generations** (*int*) – the number of generations to run for.
- **population_size** (*int*) – the number of randomly intialized 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.

2.4.8 Repeat

class sherpa.algorithms.**Repeat**(*algorithm*, *num_times=5*, *wait_for_completion=False*, *agg=False*)

Takes another algorithm and repeats every hyperparameter configuration a given number of times. The wrapped algorithm will be passed the mean objective values of the repeated experiments.

Parameters

- **algorithm** (*sherpa.algorithms.Algorithm*) – the algorithm to produce hyperparameter configurations.
- **num_times** (*int*) – the number of times to repeat each configuration.
- **wait_for_completion** (*bool*) – only relevant when running in parallel with max_concurrent > 1. Means that the algorithm won't issue the next suggestion until all repetitions are completed. This can be useful when the repeats have impact on sequential decision making in the wrapped algorithm.
- **agg** (*bool*) – whether to aggregate repetitions before passing them to the parameter generating algorithm.

2.4.9 Iterate

class sherpa.algorithms.**Iterate**(*hp_iter*)

Iterate over a set of fully-specified hyperparameter combinations.

Parameters **hp_iter** (*list*) – list of fully-specified hyperparameter dicts.

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. The model is fitted to inputs of hyperparameter configurations and outputs of objective values. It is then used to make predictions about candidate hyperparameter configurations. Each candidate-prediction can be evaluated with respect to its utility via an acquisiton function - trading off exploration and exploitation. 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 implements Bayesian optimization via a wrapper for the popular Bayesian optimization library GPyOpt (https://github.com/SheffieldML/GPyOpt/). The `GPyOpt` algorithm in SHERPA has a number of arguments that specify the Bayesian optimization in GPyOpt. The argument `max_concurrent` refers to the 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. This would be parameter configurations that you know to be reasonably good and that can be used as starting points for the Bayesian optimization. For the full specification see below. Note that as of right now *sherpa.algorithms.GPyOpt* does not accept *Discrete* variables with the option *scale='log'*.

class sherpa.algorithms.**GPyOpt**(*model_type='GP'*, *num_initial_data_points='infer'*, *initial_data_points=[]*, *acquisition_type='EI'*, *max_concurrent=4*, *verbosity=False*, *max_num_trials=None*)

Sherpa wrapper around the GPyOpt package (https://github.com/SheffieldML/GPyOpt).

Parameters

- **model_type** (*str*) – The model used: - 'GP', standard Gaussian process. - 'GP_MCMC', Gaussian process with prior in the hyper-parameters. - 'sparseGP', sparse Gaussian process. - 'warperdGP', warped Gaussian process. - 'InputWarpedGP', input warped Gaussian process - 'RF', random forest (scikit-learn).
- **num_initial_data_points** (*int*) – Number of data points to collect before fitting model. Needs to be greater/equal to the number of hyper- parameters that are being optimized. Using default 'infer' corres- ponds to number of hyperparameters + 1 or 0 if results are not empty.
- **initial_data_points** (*list[dict] or pandas.Dataframe*) – Specifies initial data points. If len(initial_data_points)<num_initial_data_points then the rest is randomly sampled. Use this option to provide hyperparameter configurations that are known to be good.
- **acquisition_type** (*str*) – Type of acquisition function to use. - 'EI', expected improvement. - 'EI_MCMC', integrated expected improvement (requires GP_MCMC model). - 'MPI', maximum probability of improvement. - 'MPI_MCMC', maximum probability of improvement (requires GP_MCMC model). - 'LCB', GP-Lower confidence bound. - 'LCB_MCMC', integrated GP-Lower confidence bound (requires GP_MCMC model).
- **max_concurrent** (*int*) – The number of concurrent trials. This generates a batch of max_concurrent trials from GPyOpt to evaluate. If a new observation becomes available, the model is re-evaluated and a new batch is created regardless of whether the previous batch was used up. The used method is local penalization.
- **verbosity** (*bool*) – Print models and other options during the optimization.

- **max_num_trials** (*int*) – maximum number of trials to run for.

2.5.3 Example

Using GPyOpt Bayesian Optimization in SHERPA is straight forward. The parameter ranges are defined as usual, for example:

```
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:

```
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 shown in the *Guide*. For example

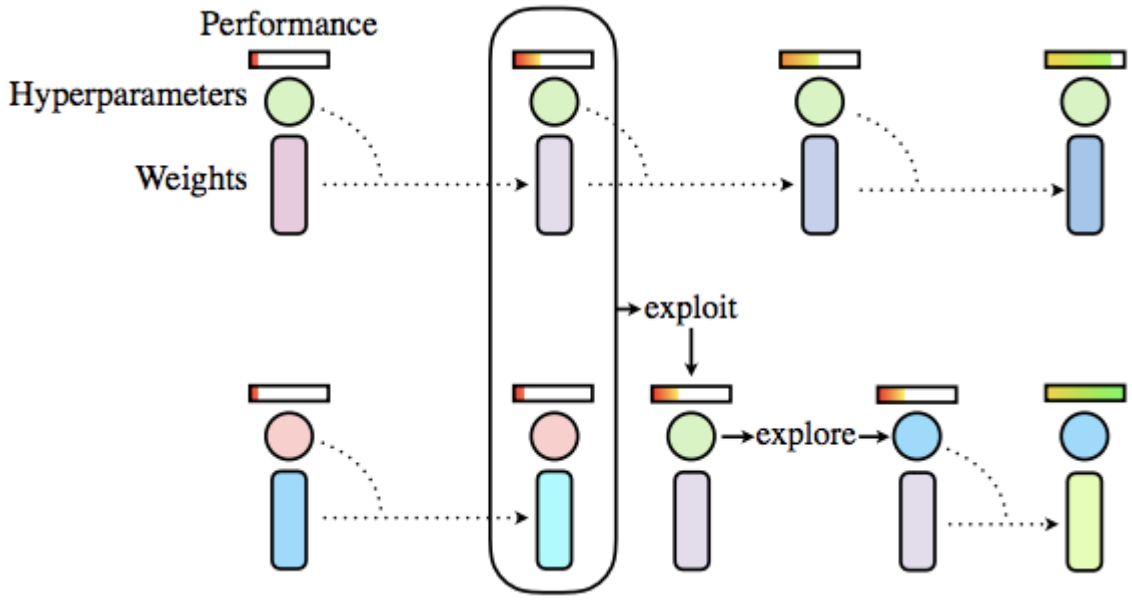
```
for trial in study:
    model = init_model(train.parameters)
    for iteration in range(num_iterations):
        training_error = model.fit(epochs=1)
        validation_error = model.evaluate()
        study.add_observation(trial=trial,
                              iteration=iteration,
        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

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

2.6 Population Based Training

2.6.1 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 hyperparamater 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. This process achieves a joint optimization of the model parameters and training hyperparameters.



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 e.g. the GPyOpt Algorithm.

[119]:
```
import sherpa
import keras
from keras.models import Sequential, load_model
from keras.layers import Dense, Flatten
from keras.datasets import mnist
from keras.optimizers import Adam
import tempfile
import os
import shutil
import keras.backend as K
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import cm
%matplotlib inline
```

2.6.2 Dataset Preparation

Training data is normalized to the [0, 1] range.

[12]:
```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train/255.0, x_test/255.0
```

2.6.3 Sherpa Setup

We define one hyperparameter `learning_rate`. The algorithms uses a `population_size` of 5. This means the first 5 trials returned by the algorithm are randomly sampled. Each is trained by the user for say one epoch. This is the first generation. After that the 4 best out of these five are returned one after another (the top 80%). The fifth one (bottom 20%) is resampled from the top 20% (here just the best trial from generation 1) and its parameters are perturbed. For perturbation the parameter, here the `learning_rate` is randomly multiplied by 0.8 or 1.2 (as defined

by the `perturbation_factors`). After that the next generation evolves in a similar way. The algorithm stops after `num_generations` generations.

[102]:
```
parameters = [sherpa.Continuous('learning_rate', [1e-4, 1e-2], 'log')]
algorithm = sherpa.algorithms.PopulationBasedTraining(population_size=5,
                                                      num_generations=5,
                                                      perturbation_factors=(0.8, 1.2),
                                                      parameter_range={'learning_rate
↪': [1e-6, 1e-1]})
study = sherpa.Study(parameters=parameters,
                     algorithm=algorithm,
                     lower_is_better=False,
                     dashboard_port=8997)
```

```
INFO:sherpa.core:
------------------------------------------------------
SHERPA Dashboard running. Access via
http://128.195.75.106:8997 if on a cluster or
http://localhost:8997 if running locally.
------------------------------------------------------
```

Make a temporary directory to store model files in. Population based training jointly optimizes a population of models and their hyperparameters. To train all models *at the same time* we train each model for one epoch (or more if you like), save it (using *trial.parameters['save_to']*), and load it again at a later time when needed (using *trial.parameters['load_from']*). For this reason, we need a directory to save these models in.

[104]:
```
model_dir = tempfile.mkdtemp()
```

2.6.4 Hyperparameter Optimization

Technically Population Based Training could go on forever, training ever more generations. In reality however we would like to stop at some point. For this reason we set a *max_num_generations*. You can set this to the number of epochs that you would normally train the model for. Here, we choose something small to speed up the example.

[105]:
```
for trial in study:
    generation = trial.parameters['generation']
    load_from = trial.parameters['load_from']
    training_lr = trial.parameters['learning_rate']

    print("-"*100)
    print("Generation {}".format(generation))

    if load_from == "":
        print("Creating new model with learning rate {}\n".format(training_lr))

        # Create model
        model = Sequential([Flatten(input_shape=(28, 28)),
                            Dense(64, activation='relu'),
                            Dense(10, activation='softmax')])

        # Use learning rate parameter for optimizer
        optimizer = Adam(lr=training_lr)

        model.compile(loss='sparse_categorical_crossentropy',
                      optimizer=optimizer,
                      metrics=['accuracy'])
```

(continues on next page)

(continued from previous page)

```
    else:
        print(f"Loading model from ", os.path.join(model_dir, load_from), "\n")

        # Loading model
        model = load_model(os.path.join(model_dir, load_from))

        if not np.isclose(K.get_value(model.optimizer.lr), training_lr):
            print("Perturbing learning rate from {} to {}".format(K.get_value(model.
↪optimizer.lr), training_lr))
            K.set_value(model.optimizer.lr, training_lr)
        else:
            print("Continuing training with learning rate {}".format(training_lr))

    # Train model for one epoch
    model.fit(x_train, y_train)
    loss, accuracy = model.evaluate(x_test, y_test)

    print("Validation accuracy: ", accuracy)
    study.add_observation(trial=trial, iteration=generation,
                          objective=accuracy,
                          context={'loss': loss})
    study.finalize(trial=trial)

    print(f"Saving model at: ", os.path.join(model_dir, trial.parameters['save_to']))
    model.save(os.path.join(model_dir, trial.parameters['save_to']))

    study.save(model_dir)
```

```
--------------------------------------------------------------------------------
↪--------------
Generation 1
Creating new model with learning rate 0.0033908174916255636

Epoch 1/1
60000/60000 [==============================] - 7s 120us/step - loss: 0.2279 - acc: 0.
↪9324
10000/10000 [==============================] - 1s 117us/step
Validation accuracy:  0.962
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/1
--------------------------------------------------------------------------------
↪--------------
Generation 1
Creating new model with learning rate 0.00024918992981810167

Epoch 1/1
60000/60000 [==============================] - 8s 129us/step - loss: 0.4917 - acc: 0.
↪8711
10000/10000 [==============================] - 1s 124us/step
Validation accuracy:  0.9203
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/2
--------------------------------------------------------------------------------
↪--------------
Generation 1
Creating new model with learning rate 0.006140980350498516

Epoch 1/1
60000/60000 [==============================] - 8s 125us/step - loss: 0.2292 - acc: 0.
↪9309
```

(continues on next page)

(continued from previous page)

```
10000/10000 [==============================] - 1s 124us/step
Validation accuracy:  0.9548
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/3
------------------------------------------------------------------------------------
↪--------------
Generation 1
Creating new model with learning rate 0.0010303357974986642

Epoch 1/1
60000/60000 [==============================] - 7s 122us/step - loss: 0.3119 - acc: 0.
↪9096
10000/10000 [==============================] - 1s 124us/step
Validation accuracy:  0.948
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/4
------------------------------------------------------------------------------------
↪--------------
Generation 1
Creating new model with learning rate 0.0010189099949300035

Epoch 1/1
60000/60000 [==============================] - 8s 125us/step - loss: 0.2994 - acc: 0.
↪9150
10000/10000 [==============================] - 1s 123us/step
Validation accuracy:  0.9517
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/5
------------------------------------------------------------------------------------
↪--------------
Generation 2
Loading model from  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/1 ...

Continuing training with learning rate 0.0033908174916255636
Epoch 1/1
60000/60000 [==============================] - 6s 93us/step - loss: 0.1107 - acc: 0.
↪9669
10000/10000 [==============================] - 1s 125us/step
Validation accuracy:  0.9678
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/6
------------------------------------------------------------------------------------
↪--------------
Generation 2
Loading model from  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/3 ...

Continuing training with learning rate 0.006140980350498516
Epoch 1/1
60000/60000 [==============================] - 6s 94us/step - loss: 0.1347 - acc: 0.
↪9596
10000/10000 [==============================] - 1s 127us/step
Validation accuracy:  0.9507
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/7
------------------------------------------------------------------------------------
↪--------------
Generation 2
Loading model from  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/5 ...

Continuing training with learning rate 0.0010189099949300035
Epoch 1/1
60000/60000 [==============================] - 6s 98us/step - loss: 0.1454 - acc: 0.
↪9574
```

(continues on next page)

(continued from previous page)

```
10000/10000 [==============================] - ETA:  - 1s 135us/step
Validation accuracy:  0.9634
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/8
--------------------------------------------------------------------------------
--------------
Generation 2
Loading model from  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/4 ...

Continuing training with learning rate 0.0010303357974986642
Epoch 1/1
60000/60000 [==============================] - 6s 100us/step - loss: 0.1527 - acc: 0.
9558
10000/10000 [==============================] - 1s 131us/step
Validation accuracy:  0.9591
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/9
--------------------------------------------------------------------------------
--------------
Generation 2
Loading model from  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/1 ...

Perturbing learning rate from 0.0033908174373209476 to 0.004068980989950676
Epoch 1/1
60000/60000 [==============================] - 6s 98us/step - loss: 0.1178 - acc: 0.
9644
10000/10000 [==============================] - 1s 134us/step
Validation accuracy:  0.9687
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/10
--------------------------------------------------------------------------------
--------------
Generation 3
Loading model from  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/10␣
...

Continuing training with learning rate 0.004068980989950676
Epoch 1/1
60000/60000 [==============================] - 6s 99us/step - loss: 0.0897 - acc: 0.
9720
10000/10000 [==============================] - 1s 137us/step
Validation accuracy:  0.9711
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/11
--------------------------------------------------------------------------------
--------------
Generation 3
Loading model from  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/6 ...

Continuing training with learning rate 0.0033908174916255636
Epoch 1/1
60000/60000 [==============================] - 6s 101us/step - loss: 0.0837 - acc: 0.
9742
10000/10000 [==============================] - 2s 150us/step
Validation accuracy:  0.9697
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/12
--------------------------------------------------------------------------------
--------------
Generation 3
Loading model from  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/8 ...
```

(continues on next page)

(continued from previous page)

```
Continuing training with learning rate 0.0010189099949300035
Epoch 1/1
60000/60000 [==============================] - 6s 108us/step - loss: 0.1054 - acc: 0.
↪9687
10000/10000 [==============================] - 1s 148us/step
Validation accuracy:  0.9646
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/13
--------------------------------------------------------------------------------
↪--------------
Generation 3
Loading model from  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/9 ...

Continuing training with learning rate 0.0010303357974986642
Epoch 1/1
60000/60000 [==============================] - 7s 111us/step - loss: 0.1112 - acc: 0.
↪9675
10000/10000 [==============================] - 2s 155us/step
Validation accuracy:  0.9687
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/14
--------------------------------------------------------------------------------
↪--------------
Generation 3
Loading model from  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/10␣
↪...

Perturbing learning rate from 0.004068980924785137 to 0.004882777187940811
Epoch 1/1
60000/60000 [==============================] - 7s 114us/step - loss: 0.1022 - acc: 0.
↪9686
10000/10000 [==============================] - 2s 153us/step
Validation accuracy:  0.9679
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/15
--------------------------------------------------------------------------------
↪--------------
Generation 4
Loading model from  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/11␣
↪...

Continuing training with learning rate 0.004068980989950676
Epoch 1/1
60000/60000 [==============================] - 7s 117us/step - loss: 0.0772 - acc: 0.
↪9759
10000/10000 [==============================] - 1s 149us/step
Validation accuracy:  0.9669
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/16
--------------------------------------------------------------------------------
↪--------------
Generation 4
Loading model from  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/12␣
↪...

Continuing training with learning rate 0.0033908174916255636
Epoch 1/1
60000/60000 [==============================] - 7s 109us/step - loss: 0.0728 - acc: 0.
↪9775
10000/10000 [==============================] - 1s 148us/step
Validation accuracy:  0.9659
```

(continues on next page)

(continued from previous page)

```
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/17
------------------------------------------------------------------------------------
↪--------------
Generation 4
Loading model from  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/14␣
↪...

Continuing training with learning rate 0.0010303357974986642
Epoch 1/1
60000/60000 [==============================] - 7s 112us/step - loss: 0.0885 - acc: 0.
↪9735
10000/10000 [==============================] - 2s 185us/step
Validation accuracy:  0.9711
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/18
------------------------------------------------------------------------------------
↪--------------
Generation 4
Loading model from  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/15␣
↪...

Continuing training with learning rate 0.004882777187940811
Epoch 1/1
60000/60000 [==============================] - 7s 118us/step - loss: 0.0862 - acc: 0.
↪9740
10000/10000 [==============================] - 2s 151us/step
Validation accuracy:  0.968
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/19
------------------------------------------------------------------------------------
↪--------------
Generation 4
Loading model from  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/12␣
↪...

Perturbing learning rate from 0.0033908174373209476 to 0.002712653993300451
Epoch 1/1
60000/60000 [==============================] - 7s 112us/step - loss: 0.0625 - acc: 0.
↪9806
10000/10000 [==============================] - 2s 151us/step
Validation accuracy:  0.9721
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/20
------------------------------------------------------------------------------------
↪--------------
Generation 5
Loading model from  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/20␣
↪...

Continuing training with learning rate 0.002712653993300451
Epoch 1/1
60000/60000 [==============================] - 7s 114us/step - loss: 0.0528 - acc: 0.
↪9835
10000/10000 [==============================] - 2s 151us/step
Validation accuracy:  0.9734
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/21
------------------------------------------------------------------------------------
↪--------------
Generation 5
Loading model from  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/18␣
↪...
```

(continues on next page)

(continued from previous page)

```
Continuing training with learning rate 0.0010303357974986642
Epoch 1/1
60000/60000 [==============================] - 7s 116us/step - loss: 0.0725 - acc: 0.
↪9778
10000/10000 [==============================] - 2s 187us/step
Validation accuracy:  0.9718
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/22
------------------------------------------------------------------------------------
↪--------------
Generation 5
Loading model from  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/19␣
↪...

Continuing training with learning rate 0.004882777187940811
Epoch 1/1
60000/60000 [==============================] - 7s 114us/step - loss: 0.0784 - acc: 0.
↪9762
10000/10000 [==============================] - 2s 166us/step
Validation accuracy:  0.9716
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/23
------------------------------------------------------------------------------------
↪--------------
Generation 5
Loading model from  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/16␣
↪...

Continuing training with learning rate 0.004068980989950676
Epoch 1/1
60000/60000 [==============================] - 7s 122us/step - loss: 0.0653 - acc: 0.
↪9800
10000/10000 [==============================] - 2s 177us/step
Validation accuracy:  0.9679
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/24
------------------------------------------------------------------------------------
↪--------------
Generation 5
Loading model from  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/11␣
↪...

Perturbing learning rate from 0.004068980924785137 to 0.004882777187940811
Epoch 1/1
60000/60000 [==============================] - 7s 117us/step - loss: 0.0881 - acc: 0.
↪9734
10000/10000 [==============================] - 2s 175us/step
Validation accuracy:  0.9663
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/25
```

The best found hyperparameter configuration is:

[126]:
```
study.get_best_result()
```

[126]:
```
{'Iteration': 5,
 'Objective': 0.9734,
 'Trial-ID': 21,
 'generation': 5,
 'learning_rate': 0.002712653993300451,
```

(continues on next page)

(continued from previous page)

```
 'lineage': '1,6,12,20,',
 'load_from': '20',
 'loss': 0.09256360807713354,
 'save_to': '21'}
```

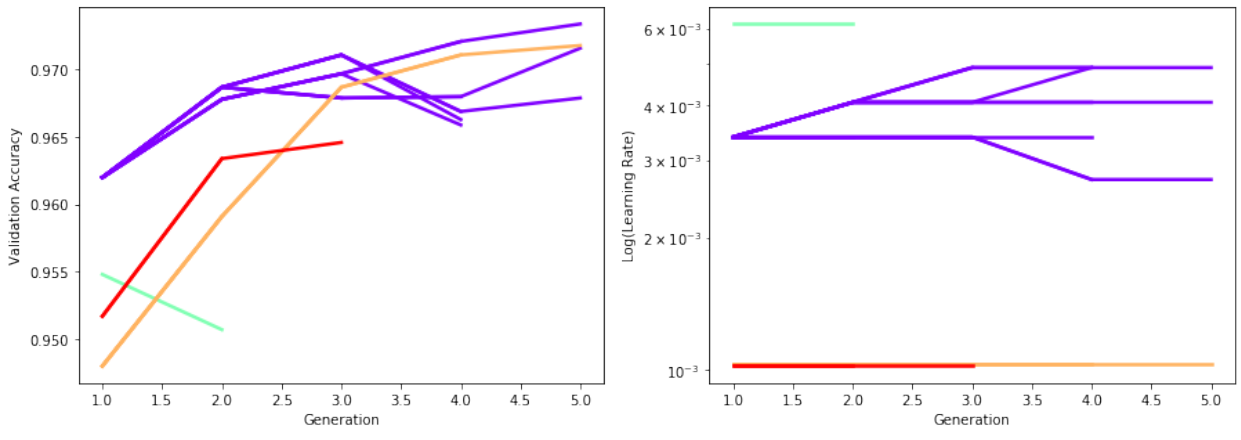
This model is stored at:

[107]:
```
print(os.path.join(model_dir, study.get_best_result()['save_to']))
```

```
/var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpt_1z3upl/21
```

These are plots of the evolution of validation accuracy and log learning rates. Lines of equal color indicate that these belong to equal seed trials, that is they stem from the same population member from the first generation.

[125]:
```
completed = study.results.query("Status == 'COMPLETED'")
fig, axis = plt.subplots(ncols=2, figsize=(15, 5))
n = 5
color=cm.rainbow(np.linspace(0,1,n))
for j in range(1, n+1):
    descendents = completed[(completed['lineage'].str.startswith('{},'.format(j)).
↪fillna(False))]
    for i, row in descendents.iterrows():
        x = list(range(1, len(row['lineage'].split(","))+1))
        obj = []
        lr = []
        for tid in row['lineage'].split(",")[:-1]:
            obj.append(completed.loc[completed['Trial-ID']==int(tid)]['Objective'].
↪values[0])
            lr.append(completed.loc[completed['Trial-ID']==int(tid)]['learning_rate'])
        obj.append(row['Objective'])
        lr.append(row['learning_rate'])
        axis[0].plot(x, obj, '-', color=color[j-1], linewidth=2.5)
        axis[1].plot(x, lr, '-', color=color[j-1], linewidth=2.5)
axis[0].set_xlabel("Generation")
axis[0].set_ylabel("Validation Accuracy")
axis[1].set_xlabel("Generation")
axis[1].set_ylabel("Log(Learning Rate)")
axis[1].set_yscale('log')
```



To remove the model directory:

[103]:
```
# Remove model_dir
shutil.rmtree(model_dir)
```

2.7 Asynchronous Successive Halving (ASHA)

Successive halving is an algorithm based on the multi-armed bandit methodology. The ASHA algorithm is a way to combine random search with principled early stopping in an asynchronous way. We highly recommend this blog post by the authors of this method: https://blog.ml.cmu.edu/2018/12/12/massively-parallel-hyperparameter-optimization/ .

[ ]:
```
import sherpa
import sherpa.algorithms.bayesian_optimization as bayesian_optimization
import keras
from keras.models import Sequential, load_model
from keras.layers import Dense, Flatten
from keras.datasets import mnist
from keras.optimizers import Adam
import tempfile
import os
import shutil
```

2.7.1 Dataset Preparation

[2]:
```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train/255.0, x_test/255.0
```

2.7.2 Sherpa Setup

In this example we use $R = 9$ and $\eta = 3$. That means to obtain one finished configuration we will train 9 configurations for 1 epochs, pick 3 configurations of those and train for 3 more epochs, then pick one out of those and train for another 9 epochs. You can increase the *max_finished_configs* argument to do a larger search.

[20]:
```
parameters = [sherpa.Continuous('learning_rate', [1e-4, 1e-2], 'log'),
              sherpa.Discrete('num_units', [32, 128]),
              sherpa.Choice('activation', ['relu', 'tanh', 'sigmoid'])]
algorithm = alg = sherpa.algorithms.SuccessiveHalving(r=1, R=9, eta=3, s=0, max_
↪finished_configs=1)
study = sherpa.Study(parameters=parameters,
                     algorithm=algorithm,
                     lower_is_better=False,
                     dashboard_port=8995)
```

```
INFO:sherpa.core:
------------------------------------------------------
SHERPA Dashboard running. Access via
http://128.195.75.106:8995 if on a cluster or
http://localhost:8995 if running locally.
------------------------------------------------------
```

Make a temporary directory to store model files in. Successive Halving tries hyperparameter configurations for bigger and bigger budgets (training epochs). Therefore, intermediate models have to be saved.

[21]:
```
model_dir = tempfile.mkdtemp()
```

2.7.3 Hyperparameter Optimization

Note: we manually infer the number of epochs that the model has trained for so we can give this information to Keras.

[22]:
```
for trial in study:
    # Getting number of training epochs
    initial_epoch = {1: 0, 3: 1, 9: 4}[trial.parameters['resource']]
    epochs = trial.parameters['resource'] + initial_epoch

    print("-"*100)
    print(f"Trial:\t{trial.id}\nEpochs:\t{initial_epoch} to {epochs}\nParameters:
↪{trial.parameters}\n")

    if trial.parameters['load_from'] == "":
        print(f"Creating new model for trial {trial.id}...\n")

        # Get hyperparameters
        lr = trial.parameters['learning_rate']
        num_units = trial.parameters['num_units']
        act = trial.parameters['activation']

        # Create model
        model = Sequential([Flatten(input_shape=(28, 28)),
                            Dense(num_units, activation=act),
                            Dense(10, activation='softmax')])
        optimizer = Adam(lr=lr)
        model.compile(loss='sparse_categorical_crossentropy',
                      optimizer=optimizer,
                      metrics=['accuracy'])
    else:
        print(f"Loading model from: ", os.path.join(model_dir, trial.parameters['load_
↪from']), "...\n")

        # Loading model
        model = load_model(os.path.join(model_dir, trial.parameters['load_from']))


    # Train model
    for i in range(initial_epoch, epochs):
        model.fit(x_train, y_train, initial_epoch=i, epochs=i+1)
        loss, accuracy = model.evaluate(x_test, y_test)

        print("Validation accuracy: ", accuracy)
        study.add_observation(trial=trial, iteration=i,
                              objective=accuracy,
                              context={'loss': loss})

    study.finalize(trial=trial)
    print(f"Saving model at: ", os.path.join(model_dir, trial.parameters['save_to']))
    model.save(os.path.join(model_dir, trial.parameters['save_to']))

    study.save(model_dir)
```

```
--------------------------------------------------------------------------------
↪--------------
Trial:  1
Epochs: 0 to 1
Parameters:{'learning_rate': 0.0006779922111149317, 'num_units': 67, 'activation':
↪'tanh', 'resource': 1, 'rung': 0, 'load_from': '', 'save_to': '1'}
Creating new model for trial 1...

Epoch 1/1
60000/60000 [==============================] - 4s 72us/step - loss: 0.3451 - acc: 0.
↪9059
10000/10000 [==============================] - 1s 53us/step
Validation accuracy:  0.9426
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpa7vbw5xz/1
--------------------------------------------------------------------------------
↪--------------
Trial:  2
Epochs: 0 to 1
Parameters:{'learning_rate': 0.0007322493943507595, 'num_units': 53, 'activation':
↪'sigmoid', 'resource': 1, 'rung': 0, 'load_from': '', 'save_to': '2'}
Creating new model for trial 2...

Epoch 1/1
60000/60000 [==============================] - 4s 71us/step - loss: 0.5720 - acc: 0.
↪8661
10000/10000 [==============================] - 0s 47us/step
Validation accuracy:  0.9213
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpa7vbw5xz/2
--------------------------------------------------------------------------------
↪--------------
Trial:  3
Epochs: 0 to 1
Parameters:{'learning_rate': 0.00013292608500661002, 'num_units': 115, 'activation':
↪'tanh', 'resource': 1, 'rung': 0, 'load_from': '', 'save_to': '3'}
Creating new model for trial 3...

Epoch 1/1
60000/60000 [==============================] - 5s 80us/step - loss: 0.5496 - acc: 0.
↪8571
10000/10000 [==============================] - 0s 48us/step
Validation accuracy:  0.9112
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpa7vbw5xz/3
--------------------------------------------------------------------------------
↪--------------
Trial:  4
Epochs: 1 to 4
Parameters:{'learning_rate': 0.0006779922111149317, 'num_units': 67, 'activation':
↪'tanh', 'save_to': '4', 'resource': 3, 'rung': 1, 'load_from': '1'}
Loading model from:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpa7vbw5xz/1␣
↪...

Epoch 2/2
60000/60000 [==============================] - 3s 50us/step - loss: 0.1818 - acc: 0.
↪9473
10000/10000 [==============================] - 0s 46us/step
Validation accuracy:  0.9559
Epoch 3/3
```

(continues on next page)

(continued from previous page)

```
60000/60000 [==============================] - 3s 51us/step - loss: 0.1353 - acc: 0.
↪9617
10000/10000 [==============================] - 0s 39us/step
Validation accuracy:  0.9629
Epoch 4/4
60000/60000 [==============================] - 3s 52us/step - loss: 0.1074 - acc: 0.
↪9687
10000/10000 [==============================] - 0s 22us/step
Validation accuracy:  0.9659
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpa7vbw5xz/4
--------------------------------------------------------------------------------
↪--------------
Trial:  5
Epochs: 0 to 1
Parameters:{'learning_rate': 0.0003139094199248622, 'num_units': 88, 'activation':
↪'sigmoid', 'resource': 1, 'rung': 0, 'load_from': '', 'save_to': '5'}
Creating new model for trial 5...

Epoch 1/1
60000/60000 [==============================] - 4s 68us/step - loss: 0.7136 - acc: 0.
↪8431
10000/10000 [==============================] - 0s 49us/step
Validation accuracy:  0.9098
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpa7vbw5xz/5
--------------------------------------------------------------------------------
↪--------------
Trial:  6
Epochs: 0 to 1
Parameters:{'learning_rate': 0.0008001577665974275, 'num_units': 36, 'activation':
↪'sigmoid', 'resource': 1, 'rung': 0, 'load_from': '', 'save_to': '6'}
Creating new model for trial 6...

Epoch 1/1
60000/60000 [==============================] - 4s 59us/step - loss: 0.6274 - acc: 0.
↪8588
10000/10000 [==============================] - 0s 48us/step
Validation accuracy:  0.9169
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpa7vbw5xz/6
--------------------------------------------------------------------------------
↪--------------
Trial:  7
Epochs: 0 to 1
Parameters:{'learning_rate': 0.003299640159323735, 'num_units': 63, 'activation':
↪'tanh', 'resource': 1, 'rung': 0, 'load_from': '', 'save_to': '7'}
Creating new model for trial 7...

Epoch 1/1
60000/60000 [==============================] - 4s 68us/step - loss: 0.2387 - acc: 0.
↪9294
10000/10000 [==============================] - 1s 52us/step
Validation accuracy:  0.9521
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpa7vbw5xz/7
--------------------------------------------------------------------------------
↪--------------
Trial:  8
Epochs: 1 to 4
Parameters:{'learning_rate': 0.003299640159323735, 'num_units': 63, 'activation':
↪'tanh', 'save_to': '8', 'resource': 3, 'rung': 1, 'load_from': '7'}
```

(continues on next page)

(continued from previous page)

```
Loading model from:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpa7vbw5xz/7...

Epoch 2/2
60000/60000 [==============================] - 3s 52us/step - loss: 0.1209 - acc: 0.9641
10000/10000 [==============================] - 1s 52us/step
Validation accuracy:  0.961
Epoch 3/3
60000/60000 [==============================] - 3s 53us/step - loss: 0.0953 - acc: 0.9704
10000/10000 [==============================] - 0s 24us/step
Validation accuracy:  0.9667
Epoch 4/4
60000/60000 [==============================] - 3s 52us/step - loss: 0.0800 - acc: 0.9756
10000/10000 [==============================] - 0s 23us/step
Validation accuracy:  0.9679
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpa7vbw5xz/8
--------------------------------------------------------------------------------
Trial:  9
Epochs: 0 to 1
Parameters:{'learning_rate': 0.0025750610635902832, 'num_units': 48, 'activation': 'sigmoid', 'resource': 1, 'rung': 0, 'load_from': '', 'save_to': '9'}
Creating new model for trial 9...

Epoch 1/1
60000/60000 [==============================] - 4s 62us/step - loss: 0.3477 - acc: 0.9065
10000/10000 [==============================] - 1s 54us/step
Validation accuracy:  0.9421
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpa7vbw5xz/9
--------------------------------------------------------------------------------
Trial:  10
Epochs: 0 to 1
Parameters:{'learning_rate': 0.0025240507488864423, 'num_units': 124, 'activation': 'tanh', 'resource': 1, 'rung': 0, 'load_from': '', 'save_to': '10'}
Creating new model for trial 10...

Epoch 1/1
60000/60000 [==============================] - 5s 85us/step - loss: 0.2297 - acc: 0.9303
10000/10000 [==============================] - 1s 58us/step
Validation accuracy:  0.9644
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpa7vbw5xz/10
--------------------------------------------------------------------------------
Trial:  11
Epochs: 1 to 4
Parameters:{'learning_rate': 0.0025240507488864423, 'num_units': 124, 'activation': 'tanh', 'save_to': '11', 'resource': 3, 'rung': 1, 'load_from': '10'}
Loading model from:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpa7vbw5xz/10...

Epoch 2/2
```

(continues on next page)

(continued from previous page)

```
60000/60000 [==============================] - 5s 78us/step - loss: 0.1079 - acc: 0.
↪9670
10000/10000 [==============================] - 1s 63us/step
Validation accuracy:  0.971
Epoch 3/3
60000/60000 [==============================] - 5s 77us/step - loss: 0.0761 - acc: 0.
↪9764
10000/10000 [==============================] - 0s 28us/step
Validation accuracy:  0.9731
Epoch 4/4
60000/60000 [==============================] - 4s 73us/step - loss: 0.0599 - acc: 0.
↪9811
10000/10000 [==============================] - 0s 30us/step
Validation accuracy:  0.9692
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpa7vbw5xz/11
--------------------------------------------------------------------------------
↪--------------
Trial:  12
Epochs: 4 to 13
Parameters:{'learning_rate': 0.0025240507488864423, 'num_units': 124, 'activation':
↪'tanh', 'save_to': '12', 'resource': 9, 'rung': 2, 'load_from': '11'}
Loading model from:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpa7vbw5xz/11␣
↪...

Epoch 5/5
60000/60000 [==============================] - 5s 77us/step - loss: 0.0466 - acc: 0.
↪9850
10000/10000 [==============================] - 1s 60us/step
Validation accuracy:  0.973
Epoch 6/6
60000/60000 [==============================] - 4s 74us/step - loss: 0.0416 - acc: 0.
↪9866
10000/10000 [==============================] - 0s 27us/step
Validation accuracy:  0.9726
Epoch 7/7
60000/60000 [==============================] - 5s 75us/step - loss: 0.0354 - acc: 0.
↪9884
10000/10000 [==============================] - 0s 27us/step
Validation accuracy:  0.9744
Epoch 8/8
60000/60000 [==============================] - 4s 72us/step - loss: 0.0292 - acc: 0.
↪9908
10000/10000 [==============================] - 0s 28us/step
Validation accuracy:  0.9739
Epoch 9/9
60000/60000 [==============================] - 4s 73us/step - loss: 0.0286 - acc: 0.
↪9905
10000/10000 [==============================] - 0s 27us/step
Validation accuracy:  0.974
Epoch 10/10
60000/60000 [==============================] - 4s 72us/step - loss: 0.0245 - acc: 0.
↪9919
10000/10000 [==============================] - 0s 27us/step
Validation accuracy:  0.9725
Epoch 11/11
60000/60000 [==============================] - 4s 72us/step - loss: 0.0233 - acc: 0.
↪9916
```

(continues on next page)

(continued from previous page)

```
10000/10000 [==============================] - 0s 31us/step
Validation accuracy:  0.9714
Epoch 12/12
60000/60000 [==============================] - 4s 72us/step - loss: 0.0203 - acc: 0.
↪9935
10000/10000 [==============================] - 0s 28us/step
Validation accuracy:  0.972
Epoch 13/13
60000/60000 [==============================] - 4s 74us/step - loss: 0.0195 - acc: 0.
↪9934
10000/10000 [==============================] - 0s 27us/step
Validation accuracy:  0.9727
Saving model at:  /var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpa7vbw5xz/12
```

The best found hyperparameter configuration is:

[23]:
```
study.get_best_result()
```

[23]:
```
{'Iteration': 6,
 'Objective': 0.9744,
 'Trial-ID': 12,
 'activation': 'tanh',
 'learning_rate': 0.0025240507488864423,
 'load_from': '11',
 'loss': 0.08811961327217287,
 'num_units': 124,
 'resource': 9,
 'rung': 2,
 'save_to': '12'}
```

This model is stored at:

[24]:
```
print(os.path.join(model_dir, study.get_best_result()['save_to']))
```

```
/var/folders/5v/l788ch2j7tg0q0y1rt04c08w0000gn/T/tmpa7vbw5xz/12
```

To remove the model directory:

[25]:
```
# Remove model_dir
shutil.rmtree(model_dir)
```

2.8 Local Search

2.8.1 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 hyperparameter configuration should be repeated. Since performance differences caused by local changes may be small, this can help to establish significance.

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.

2.8.2 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

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

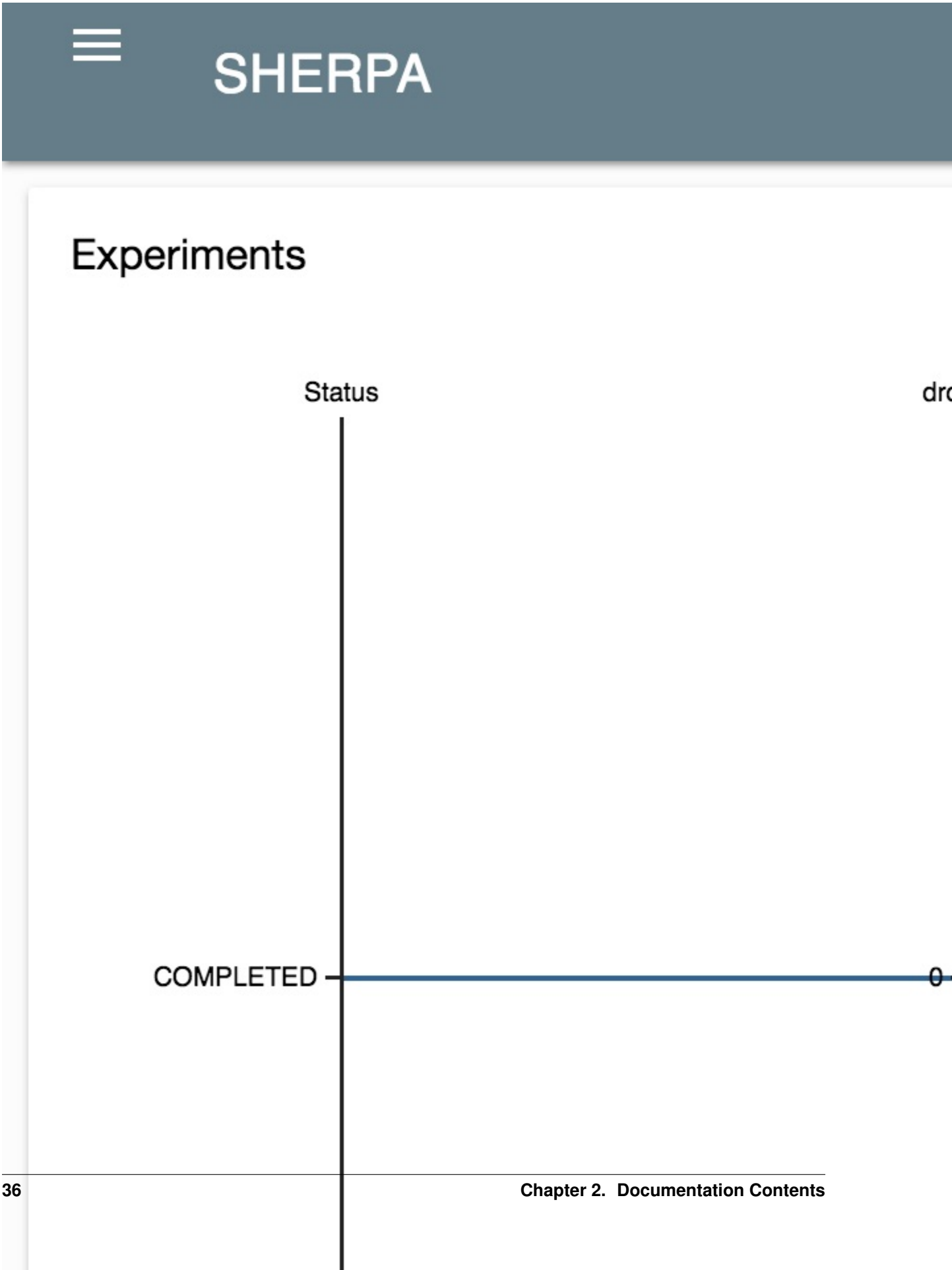
After running, we can inspect the results in the dashboard:

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

2.9 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. Stochasting 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:


SHERPA
Experiments
Status
COMPLETED
0

- `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.

```
import sherpa
class MyAlgorithm(sherpa.algorithms.Algorithm):
    def get_suggestion(self, parameters, results, lower_is_better):
        # your code here
        return params_values_for_next_trial
```

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.

```
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', :]  #␣
↪select only completed trials
```

(continues on next page)

(continued from previous page)

```
        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()
                    return trial_param_values
        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
```

2.10 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.

2.10.1 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:

```
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:

```
from keras.models import Sequential
from keras.layers import Dense
model = Sequential()
```

(continues on next page)

(continued from previous page)

```
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:

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

After:

```
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)
```

2.10.2 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.

```
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␣
↪was defined
                       scheduler=sherpa.schedulers.LocalScheduler(), # Run on local␣
↪machine
                       )
```

And that's it! Now to run your hyperparameter optimization you just have to do:

```
python runner.py
```

2.11 Setup for Parallel Computation

Install dependencies:

```
pip install pymongo
```

2.11.1 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/ .

2.11.2 Example

To verify SHERPA *with* MongoDB is working:

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

2.12 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.

2.12.1 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:

```
# 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.

2.12.2 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*.

```
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*.

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

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

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

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

```
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.

2.13 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.

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.

2.13.1 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.

2.13.2 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.

2.13.3 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.

2.14 Core

2.14.1 Setting up the Optimization

Parameters

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.

static from_dict(*config*)

Returns a parameter object according to the given dictionary config.

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

Example:

```
{'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

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

```
{'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.
- **random_seed** (*int*) – seed to use for NumPy random number generators throughout.

add_observation(*trial*, *objective*, *iteration=1*, *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*, *command=None*, *filename=None*, *output_dir='./output_20200731-053050'*, *max_concurrent=1*, *db_port=None*, *stopping_rule=None*, *dashboard_port=None*, *resubmit_failed_trials=False*, *verbose=1*, *load=False*, *mongodb_args={}*, *disable_dashboard=False*)
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.
- **command** (*str*) – the command to run for the trial script.
- **filename** (*str*) – the filename of the script to run. Will be run as "python <filename>".
- **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 "–".

2.14.2 Setting up the Trial

Client

class sherpa.database.**Client**(*host=None*, *port=None*, *test_mode=False*, ***mongo_client_args*)
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.

2.15 Schedulers

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.

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 SHERPA_RESOURCE environment variable. If no resource is available '' will be passed.

2.16 Algorithms

2.16.1 Optimization Algorithms

class sherpa.algorithms.**Algorithm**

Abstract algorithm that generates new set of parameters.

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

load(*num_trials*)

Reinstantiates the algorithm when loaded.

Parameters **num_trials** (*int*) – number of trials in study so far.

class sherpa.algorithms.**RandomSearch**(*max_num_trials=None*)

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.

class sherpa.algorithms.**GridSearch**(*num_grid_points=2*)

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:

```
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.

class sherpa.algorithms.**GPyOpt**(*model_type='GP'*, *num_initial_data_points='infer'*, *initial_data_points=[]*, *acquisition_type='EI'*, *max_concurrent=4*, *verbosity=False*, *max_num_trials=None*)

Sherpa wrapper around the GPyOpt package (https://github.com/SheffieldML/GPyOpt).

Parameters

- **model_type** (*str*) – The model used: - 'GP', standard Gaussian process. - 'GP_MCMC', Gaussian process with prior in the hyper-parameters. - 'sparseGP', sparse Gaussian process. - 'warperdGP', warped Gaussian process. - 'InputWarpedGP', input warped Gaussian process - 'RF', random forest (scikit-learn).
- **num_initial_data_points** (*int*) – Number of data points to collect before fitting model. Needs to be greater/equal to the number of hyper- parameters that are being optimized. Using default 'infer' corres- ponds to number of hyperparameters + 1 or 0 if results are not empty.
- **initial_data_points** (*list[dict] or pandas.Dataframe*) – Specifies initial data points. If len(initial_data_points)<num_initial_data_points then the rest is randomly sampled. Use this option to provide hyperparameter configurations that are known to be good.
- **acquisition_type** (*str*) – Type of acquisition function to use. - 'EI', expected improvement. - 'EI_MCMC', integrated expected improvement (requires GP_MCMC model). - 'MPI', maximum probability of improvement. - 'MPI_MCMC', maximum probability of improvement (requires GP_MCMC model). - 'LCB', GP-Lower confidence bound. - 'LCB_MCMC', integrated GP-Lower confidence bound (requires GP_MCMC model).
- **max_concurrent** (*int*) – The number of concurrent trials. This generates a batch of max_concurrent trials from GPyOpt to evaluate. If a new observation becomes available, the model is re-evaluated and a new batch is created regardless of whether the previous batch was used up. The used method is local penalization.
- **verbosity** (*bool*) – Print models and other options during the optimization.
- **max_num_trials** (*int*) – maximum number of trials to run for.

class sherpa.algorithms.**SuccessiveHalving**(*r=1*, *R=9*, *eta=3*, *s=0*, *max_finished_configs=50*)

Asynchronous Successive Halving as described in:

> @article{li2018massively, title={Massively parallel hyperparameter tuning}, author={Li, Liam and Jamieson, Kevin and Rostamizadeh, Afshin and Gonina, Ekaterina and Hardt, Moritz and Recht, Benjamin and Talwalkar, Ameet}, journal={arXiv preprint arXiv:1810.05934}, year={2018} }

Asynchronous successive halving operates based on the multi-armed bandit algorithm Successive Halving (SHA) and performs principled early stopping for random search.

Parameters

- **r** (*int*) – minimum resource that each configuration will be trained for.
- **R** (*int*) – maximum resource.
- **eta** (*int*) – elimination rate.

- **s** (*int*) – minimum early-stopping rate.
- **max_finished_configs** (*int*) – stop once max_finished_configs models have been trained to completion.

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.

class sherpa.algorithms.**PopulationBasedTraining**(*num_generations*, *population_size=20*, *parameter_range={}*, *perturbation_factors=(0.8, 1.2)*)

Population based training (PBT) as introduced by Jaderberg et al. 2017.

PBT trains a generation of `population_size` seed trials (randomly initialized) for a user specified number of iterations e.g. one epoch. The top 80% then move on unchanged into the second generation. The bottom 20% are re-sampled from the top 20% and perturbed. The second generation again trains for the same number of iterations and the same procedure is repeated to move into the third generation etc.

Parameters

- **num_generations** (*int*) – the number of generations to run for.
- **population_size** (*int*) – the number of randomly intialized 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.

class sherpa.algorithms.**Repeat**(*algorithm*, *num_times=5*, *wait_for_completion=False*, *agg=False*)

Takes another algorithm and repeats every hyperparameter configuration a given number of times. The wrapped algorithm will be passed the mean objective values of the repeated experiments.

Parameters

- **algorithm** (*sherpa.algorithms.Algorithm*) – the algorithm to produce hyperparameter configurations.
- **num_times** (*int*) – the number of times to repeat each configuration.
- **wait_for_completion** (*bool*) – only relevant when running in parallel with max_concurrent > 1. Means that the algorithm won't issue the next suggestion until all repetitions are completed. This can be useful when the repeats have impact on sequential decision making in the wrapped algorithm.
- **agg** (*bool*) – whether to aggregate repetitions before passing them to the parameter generating algorithm.

class sherpa.algorithms.**Iterate**(*hp_iter*)
Iterate over a set of fully-specified hyperparameter combinations.

Parameters **hp_iter** (*list*) – list of fully-specified hyperparameter dicts.

2.16.2 Stopping Rules

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

2.17 Development

2.17.1 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.

2.17.2 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.

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`).

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.
- **random_seed** (*int*) – seed to use for NumPy random number generators throughout.

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.

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.

2.18 Writing Schedulers

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

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