SoftwareX 12 (2020) 100591

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

Sherpa: Robust hyperparameter optimization for machine learning

Lars Hertel [a,*], Julian Collado [b], Peter Sadowski [c], Jordan Ott [b], Pierre Baldi [b]

[a] Department of Statistics, Donald Bren School of Information and Computer Sciences, University of California, Irvine Bren Hall 2019 Irvine, CA 92697-1250, USA
[b] Department of Computer Science, Donald Bren School of Information and Computer Sciences, University of California, Irvine 3019 Donald Bren Hall Irvine, CA 92697-3435, USA
[c] Information and Computer Science, University of Hawai'i at Mãnoa, 1680 East-West Rd, Honolulu, HI 96822, USA

ARTICLE INFO

Article history:
Received 23 October 2019
Received in revised form 9 July 2020
Accepted 14 September 2020

Keywords:
Hyperparameter optimization
Machine learning
Deep neural networks

ABSTRACT

Sherpa is a hyperparameter optimization library for machine learning models. It is specifically designed for problems with computationally expensive, iterative function evaluations, such as the hyperparameter tuning of deep neural networks. With Sherpa, scientists can quickly optimize hyperparameters using a variety of powerful and interchangeable algorithms. Sherpa can be run on either a single machine or in parallel on a cluster. Finally, an interactive dashboard enables users to view the progress of models as they are trained, cancel trials, and explore which hyperparameter combinations are working best. Sherpa empowers machine learning practitioners by automating the more tedious aspects of model tuning. Its source code and documentation are available at https://github.com/sherpa-ai/sherpa.

Code metadata

Current code version	1.0.5
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX_2019_328
Code Ocean compute capsule	https://codeocean.com/capsule/cba6686e-f3d8-47fa-801f-1eb21b4c44c2/tree?ID=12baab1195e9445784ebed345f722b1f
Legal Code License	GPL v3
Code versioning system used	git
Software code languages, tools, and services used	Python, JavaScript, HTML, CSS
Compilation requirements, operating environments & dependencies	Python, pandas≥0.20.3, pymongo≥3.5.1, numpy≥1.8.2, scipy≥1.0.0, scikit-learn≥0.19.1, flask≥0.12.2, GPyOpt≥1.2.5, matplotlib, MongoDB for *parallel mode*
If available Link to developer documentation/manual	https://parameter-sherpa.readthedocs.io
Support email for questions	lhertel@uci.edu

1. Motivation and significance

Hyperparameters are tuning parameters of machine learning models. Hyperparameter optimization refers to the process of choosing optimal hyperparameters for a machine learning model. This optimization is crucial to obtain optimal performance from the machine learning model. Since hyperparameters cannot be directly learned from the training data, their optimization is often a process of trial and error conducted manually by the researcher. There are two problems with the trial and error approach. Firstly, it is time consuming and can take days or even weeks of the researcher's attention. Secondly, it is dependent on the researcher's ability to interpret results and choose good hyperparameter settings. These limitations lead to a large need to automate this process. Sherpa is a software that addresses this need.

Existing hyperparameter optimization software can be divided into bayesian optimization software, bandit and evolutionary algorithm software, framework specific software, and all-round software. Software that implements bayesian optimization

* Corresponding author.
E-mail addresses: lhertel@uci.edu (L. Hertel), colladou@uci.edu (J. Collado), peter.sadowski@hawaii.edu (P. Sadowski), jott1@uci.edu (J. Ott), pfbaldi@ics.uci.edu (P. Baldi).

https://doi.org/10.1016/j.softx.2020.100591

started with SMAC [1], Spearmint [2], and HyperOpt [3]. More recent software in this regime has been GPyOpt [4], RoBo [5], DragonFly [6], Cornell-MOE [7,8], and mlrMBO [9]. These software packages have high quality, stand-alone bayesian optimization implementations, often with unique twists. However, most of these do not provide infrastructure for parallel training.

As an alternative to bayesian optimization, multi-armed bandits and evolutionary algorithms have recently become popular. HpBandSter implements Hyperband [10] and BOHB [11], Pbt implements Population Based Training [12], PyCMA implements CMA-ES [13], and TPot [14,15] provides hyperparameter search via genetic programming.

A number of framework specific libraries have also been proposed. Auto-Weka [16] and Auto-Sklearn [17] focus on WEKA [18] and Scikit-learn [19], respectively. Furthermore, a number of packages have been proposed for the machine learning framework Keras [20]. Hyperas [21], Auto-Keras [22], Talos, Kopt, and HORD each provide hyperparameter optimization specifically for Keras. These libraries make it easy to get started due to their tight integration with the machine learning framework. However, researchers will inevitably run into limitations when a different machine learning framework is needed.

Lastly, a number of implementations aim at being framework agnostic and also support multiple optimization algorithms. Table 1 shows a detailed comparison of these "all-round" packages to Sherpa. Note that we excluded Google Vizier [23] and similar frameworks from other cloud computing providers since these are not free to use.

Sherpa is already being used in a wide variety of applications such as machine learning methods [29], solid state physics [30], particle physics [31], medical image analysis [32], and cyber security [33]. Due to the fact that the number of machine learning applications is growing rapidly we can expect there to be a growing need for hyperparameter optimization software such as Sherpa.

2. Software description

2.1. Hyperparameter optimization

We begin by laying out the components of a hyperparameter optimization. Consider the training of a machine learning model. A user has a *model* that is being trained with *data*. Before training there are hyperparameters that need to be set. At the end of the training we obtain an *objective* value.

This workflow can be illustrated via the training of a neural network. The *model* is a neural network. The *data* are images that the neural network is trained on. The *hyperparameter setting* is the number of hidden layers of the neural network. The *objective* is the prediction accuracy, prediction error, or loss on a hold-out dataset obtained at the end of training.

For automated hyperparameter optimization we also need hyperparameter *ranges*, a *results* table, and a hyperparameter optimization *algorithm*. The hyperparameter ranges define what values each hyperparameter is allowed to take. The results store hyperparameter settings and their associated objective value. Finally, the algorithm takes results and ranges and produces a new suggestion for a hyperparameter setting. We refer to this suggestion as a *trial*.

For the neural network example the hyperparameter range might be 1, 2, 3, or 4 hidden layers. We might have previous results that 1 corresponds to 80% accuracy and 3 to 90% accuracy. The algorithm might then produce a new trial with 4 hidden layers. After training the neural network with 4 hidden layers we find it achieves 88% accuracy and add this to the results. Then the next trial is suggested.

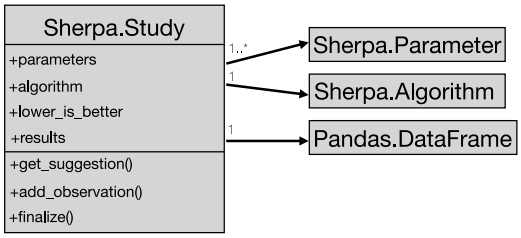


Fig. 1. Diagram showing Sherpa's `Study` class.

2.2. Components

We now describe how Sherpa implements the components described in Section 2.1. Sherpa implements hyperparameter ranges as `sherpa.Parameter` objects. The *algorithm* is implemented as a `sherpa.algorithms.Algorithm` object. A list of hyperparameter ranges and an algorithm are combined to create a `sherpa.Study` (Fig. 1). The study stores the *results*. Trials are implemented as `sherpa.Trial` objects.

Sherpa implements two user interfaces. We will refer to the two interfaces as *API mode* and *parallel mode*.

2.3. API mode

In API mode the user interacts with the `Study` object. Given a study *s*:

1. A new trial of name `t` is obtained by calling `s.get_suggestion()` or by iterating over the study (e.g. `for t in s`).
2. First, `t.parameters` is used to initialize and train a machine learning model. Then `s.add_observation(t, objective=o)` is called to add objective `o` for trial `t`. Invalid observations are automatically excluded from the results.
3. Finally, `s.finalize(t)` informs Sherpa that the model training is finished.

Interacting with the Study class is easy. It also requires minimal setup. The limitation in API mode is that it cannot evaluate trials in parallel.

2.4. Parallel mode

In *parallel-mode* multiple trials can be evaluated in parallel. The user provides two scripts: a *server script* and a *machine learning (ML) script*. The *server script* defines the hyperparameter ranges, the algorithm, the job scheduler, and the command to execute the machine learning script. The optimization starts by calling `sherpa.optimize`.

In the *machine learning script* the user trains the machine learning model given some hyperparameters and adds the resulting objective value to Sherpa. Using a `sherpa.Client` called `c` a trial `t` is obtained by calling `c.get_trial()`. To add observations `c.send_metrics(trial=t, objective=o)` is used.

Internally, `sherpa.optimize` runs a loop that uses the Study class. Fig. 2 illustrates the *parallel-mode* architecture.

1. The loop submits new trials if resources are available by submitting a job to the scheduler. Furthermore, the new trials are added to a database. From there they can be retrieved by the client.
2. The loop updates results by querying the database for new results.
3. Finally, the loop checks whether jobs have finished. This means resources are free again. In addition, the corresponding trials can be finalized.

Table 1
Feature comparison of hyperparameter optimization frameworks. *Bayesian optimization*, *evolutionary*, and *bandit/early-stopping* refer to the support of hyperparameter optimization algorithms based on these methods.

Software	Distributed	Visualizations	Bayesian-optimization	Evolutionary	Bandit/early-stopping
Sherpa	Yes	Yes	Yes	Yes	Yes
Advisor	Yes	No	Yes	Yes	Yes
Chocolate	Yes	No	Yes	Yes	No
Test-Tube[24]	Yes	No	No	No	No
Ray-Tune[25]	Yes	No	No	Yes	Yes
Optuna[26]	Yes	Yes	Yes	No	Yes
BTB [27]	No	No	Yes	No	Yes
Hyperas [21]	Yes	No	Yes	No	No
Keras-Tuner [28]	Yes	No	Yes	No	Yes

If the user's machine learning script does not submit an objective value such as when it crashed, Sherpa continues with the next trial.

3. Software functionalities

3.1. Available hyperparameter types

Sherpa supports four hyperparameter types:

- `sherpa.Continuous`
- `sherpa.Discrete`
- `sherpa.Choice`
- `sherpa.Ordinal`.

These correspond to a range of floats, a range of integers, an unordered categorical variable, and an ordered categorical variable, respectively. Each parameter has `name` and `range` arguments. The range expects a list defining lower and upper bound for continuous and discrete variables. For choice and ordinal variables the range expects the categories.

3.2. Diversity of algorithms

Sherpa aims to help researchers at various stages in their model development. For this reason, it provides a choice of hyperparameter tuning algorithms. The following optimization algorithms are currently supported.

- `sherpa.algorithms.RandomSearch`:
  Random Search [34] samples hyperparameter settings uniformly from the specified ranges. It is a robust algorithm because it explores the space uniformly. Furthermore, with the dashboard the user can make their own inference on the results.
- `sherpa.algorithms.GridSearch`:
  Grid Search follows a grid over the hyperparameter space and evaluates all combinations. It is useful to systematically explore one or two hyperparameters. It is not recommended for more than two hyperparameters.
- `sherpa.algorithms.bayesian_optimization.GPyOpt`:
  Bayesian optimization is a model-based search. For each trial it picks the most promising hyperparameter setting based on prior results. Sherpa's implementation wraps the package GPyOpt [4].
- `sherpa.algorithms.successive_halving.SuccessiveHalving`:
  Asynchronous Successive Halving (ASHA) [35] is a hyperparameter optimization algorithm based on multi-armed bandits. It allows the efficient exploration of a large hyperparameter space. This is accomplished by the early stopping of unpromising trials.
- `sherpa.algorithms.PopulationBasedTraining`:
  Population-based Training (PBT) [12] is an evolutionary algorithm. The algorithm jointly optimizes a population of models and their hyperparameters. This is achieved by adjusting hyperparameters during training. It is particularly suited for neural network training hyperparameters such as learning rate, weight decay, or batch size.
- `sherpa.algorithms.LocalSearch`:
  Local Search is a heuristic algorithm. It starts with a seed hyperparameter setting. During optimization it randomly perturbs one hyperparameter at a time. If a setting improves on the seed then it becomes the new seed. This algorithm is particularly useful if the user already has a well performing hyperparameter setting.

All implemented algorithms allow parallel evaluation and can be used with all available parameter types. An empirical comparison of the algorithms can be found in the documentation.[1]

3.3. Accounting for random variation

Sherpa can account for variation via the `Repeat` algorithm. The objective value of a model may vary between training runs. Reasons for this can be random initialization or stochastic training. The `Repeat` algorithm runs each hyperparameter setting multiple times. Thus variation can be taken into account when analyzing results.

3.4. Visualization dashboard

Sherpa provides an interactive web-based dashboard. It allows the user to monitor progress of the hyperparameter optimization in real time. Fig. 3 shows a screenshot of the dashboard.

At the top of the dashboard is a parallel coordinates plot [36, 37]. It allows exploration of relationships between hyperparameter settings and objective values (Fig. 3 top). Each vertical axis corresponds to a hyperparameter or the objective. The axes can be brushed over to select subsets of trials. The plot is implemented using the D3.js parallel-coordinates library by [38]. At the bottom right is a line chart. It shows objective values against training iteration (Fig. 3 bottom right). This chart allows to monitor training progress of each trial. It is also useful to analyze whether a trial's training converged. At the bottom left is a table of all completed trials (Fig. 3 bottom left). Hovering over trials in the table highlights the corresponding lines in the plots. Finally, the dashboard has a stopping button (Fig. 3 top right corner). This allows the user to cancel the training for unpromising trials.

The dashboard runs automatically during a hyperparameter optimization. It can be accessed in a web-browser via a link provided by Sherpa. The dashboard is useful to quickly evaluate questions such as:

[1] https://parameter-sherpa.readthedocs.io/en/latest/algorithms/algorithms.html

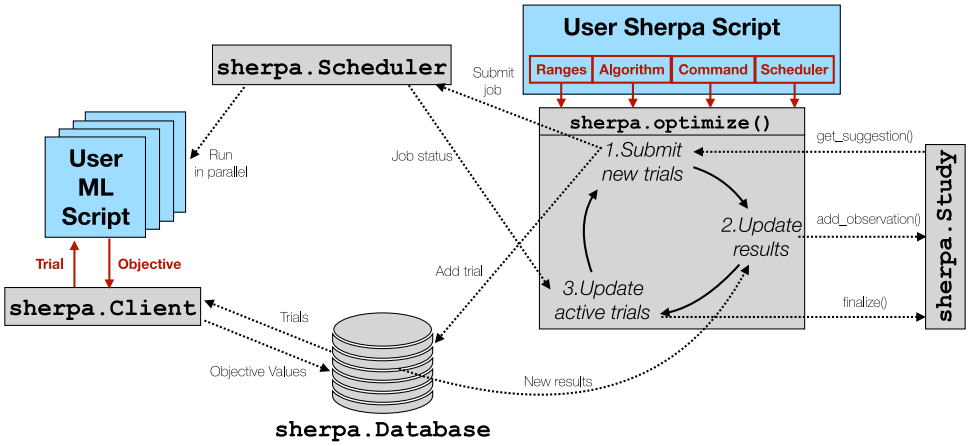
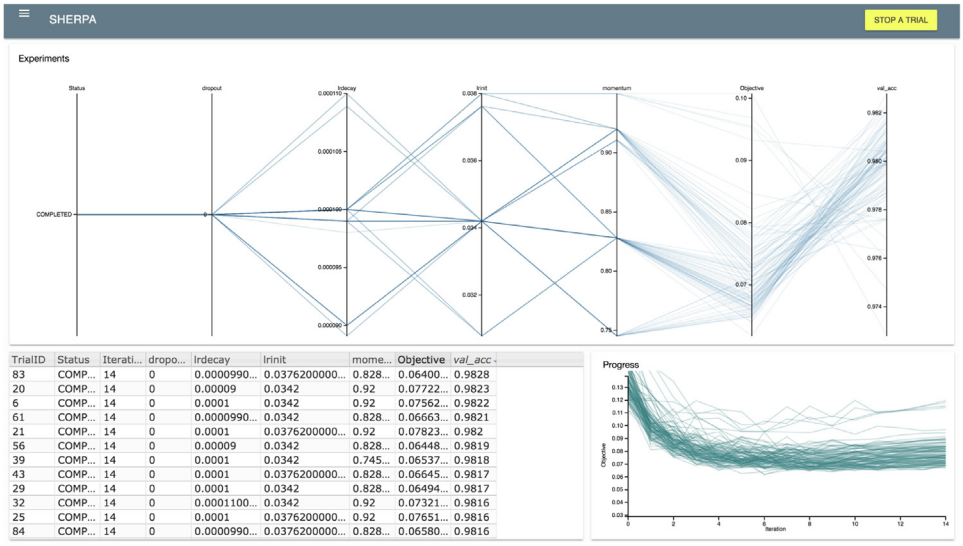


Fig. 2. Architecture diagram for parallel hyperparameter optimization in Sherpa. The user only interacts with Sherpa via the solid red arrows, everything else happens internally.



TrialID	Status	Iterati...	dropo...	lrdecay	lrinit	mome...	Objective	val_acc
83	COMP...	14	0	0.0000990...	0.0376200000...	0.828...	0.06400...	0.9828
20	COMP...	14	0	0.00009	0.0342	0.92	0.07722...	0.9823
6	COMP...	14	0	0.0001	0.0342	0.92	0.07562...	0.9822
61	COMP...	14	0	0.0000990...	0.0342	0.828...	0.06663...	0.9821
21	COMP...	14	0	0.0001	0.0376200000...	0.92	0.07823...	0.982
56	COMP...	14	0	0.00009	0.0342	0.828...	0.06448...	0.9819
39	COMP...	14	0	0.0001	0.0342	0.745...	0.06537...	0.9818
43	COMP...	14	0	0.0001	0.0376200000...	0.828...	0.06645...	0.9817
29	COMP...	14	0	0.0001	0.0342	0.828...	0.06494...	0.9817
32	COMP...	14	0	0.0001100...	0.0342	0.92	0.07321...	0.9816
25	COMP...	14	0	0.0001	0.0376200000...	0.92	0.07651...	0.9816
84	COMP...	14	0	0.0000990...	0.0376200000...	0.828...	0.06580...	0.9816

Fig. 3. The dashboard provides a parallel coordinates plot (top) and a table of finished trials (bottom left). Trials in progress are shown via a progress line chart (bottom right). Figure recommended to be viewed as PDF and via zooming in.

- Are the selected hyperparameter ranges appropriate?
- Is training unstable for some hyperparameter settings?
- Does a particular hyperparameter have little impact on the performance of the machine learning algorithm?
- Are the best observed hyperparameter settings consistent?

Based on these observations the user can refine the hyperparameter ranges or choose a different algorithm, if appropriate.

3.5. Scaling up with a cluster

In *parallel mode* Sherpa can run parallel evaluations. A job scheduler is responsible for running the user's machine learning script. The following job schedulers are implemented.

- The `LocalScheduler` evaluates parallel trials on the same computation node. This scheduler is useful for running on multiple local CPU cores or GPUs. It has a simple resource handler for GPU allocation (see Fig. 5 for an example).
- The `SGEScheduler` uses Sun Grid Engine (SGE) [39]. Submission arguments and an environment profile can be specified via arguments to the scheduler.
- The `SLURMScheduler` is based on SLURM [40]. Its interface is similar to the `SGEScheduler`.

Concurrency between workers is handled via MongoDB, a NoSQL database program. Parallel mode expects that MongoDB is installed on the system.

4. Illustrative examples

4.1. Handwritten digits classification with a neural network

The following is an example of a Sherpa hyperparameter optimization. It uses the MNIST handwritten digits dataset [41]. A Keras neural network is used to classify the digits. The neural network has one hidden layer and a softmax output. The hyperparameters are the learning rate of the Adam [42] optimizer, the number of hidden units, and the hidden layer activation function. The search is first conducted using Sherpa's API mode. After that we show the same example using Sherpa's parallel mode.

4.1.1. API mode

Fig. 4 shows the hyperparameter optimization in Sherpa's API mode. The script starts with imports and loading of the MNIST dataset. Next, the hyperparameters *learning_rate*, *num_units*, and *activation* are defined. These refer to the Adam learning rate,

```
import sherpa
import sherpa.algorithms.bayesian_optimization as
   bayesian_optimization
import keras
from keras.models import Sequential
from keras.layers import Dense, Flatten
from keras.datasets import mnist
from keras.optimizers import Adam
epochs = 15
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train/255.0, x_test/255.0

# Sherpa setup
parameters = [sherpa.Continuous('learning_rate', [1e-4, 1e-2]),
              sherpa.Discrete('num_units', [32, 128]),
              sherpa.Choice('activation',
                            ['relu', 'tanh', 'sigmoid'])]
algorithm = bayesian_optimization.GPyOpt(max_num_trials=50)
study = sherpa.Study(parameters=parameters,
                     algorithm=algorithm,
                     lower_is_better=False)

for trial in study:
    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'])
    my_callbacks = [ModelCheckpoint(filepath='myModel.h5')]
    # Train model
    for i in range(epochs):
        model.fit(x_train, y_train,
                  callbacks=my_callbacks)
        loss, accuracy = model.evaluate(x_test, y_test)
        study.add_observation(trial=trial, iteration=i,
                              objective=accuracy,
                              context={'loss': loss})
    study.finalize(trial=trial)
```

Fig. 4. An example showing how to tune the hyperparameters of a neural network on the MNIST dataset using Sherpa in API mode.

number of hidden layer units, and hidden layer activation function, respectively. As optimization algorithm the *GPyOpt* algorithm is chosen. Hyperparameter ranges and algorithm are combined via the *Study*. The `lower_is_better` flag indicates that lower objective values are not better. This is because we will be maximizing the classification accuracy. After that a for-loop iterates over the study. The for-loop yields a trial at each iteration. A Keras model is instantiated using the hyperparameter settings. Any Keras callbacks can be added to the model, here we add a model checkpointer. The Keras model is iteratively trained and evaluated via an inner for-loop. We add an observation for each iteration and use `finalize` after the training is finished. Note that we pass the loss as context to `add_observation`. The context accepts a dictionary with any additional metrics that the user wants to record. Code to replicate this example is available as a Jupyter notebook.[2] and on Google Colab[3] A video tutorial is also available on YouTube.[4] Tutorials using the Successive Halving and Population Based Training algorithms are also available.[5,6]

4.1.2. Parallel mode

We now show the same hyperparameter optimization using Sherpa's parallel mode. Fig. 5 (top) shows the server script. First, the hyperparameters and search algorithm are defined. This time

2 https://github.com/sherpa-ai/sherpa/blob/master/examples/keras_mnist_mlp.ipynb

3 https://colab.research.google.com/drive/1I19R1GfKPjlgNdHlxJwNC4PitvySsdon

4 https://youtu.be/-exnF3uv0Ws

5 https://github.com/sherpa-ai/sherpa/blob/master/examples/keras_mnist_mlp_successive_halving.ipynb

6 https://github.com/sherpa-ai/sherpa/blob/master/examples/keras_mnist_mlp_population_based_training.ipynb

```
import sherpa
import sherpa.algorithms.bayesian_optimization as
   bayesian_optimization
from sherpa.schedulers import LocalScheduler
params = [sherpa.Continuous('learning_rate', [1e-4, 1e-2]),
              sherpa.Discrete('num_units', [32, 128]),
              sherpa.Choice('activation',
                            ['relu', 'tanh', 'sigmoid'])]
alg = bayesian_optimization.GPyOpt(max_num_trials=50)
sched = LocalScheduler(resources=[0,1])
sherpa.optimize(parameters=params, algorithm=alg,
                scheduler=sched, lower_is_better=False,
                command='python trial.py', max_concurrent=2)
```

```
import sherpa
import os
GPU_ID = os.environ['SHERPA_RESOURCE']
os.environ['CUDA_VISIBLE_DEVICES'] = GPU_ID
import keras
from keras.models import Sequential
from keras.layers import Dense, Flatten
from keras.datasets import mnist
from keras.optimizers import Adam
epochs = 15
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train/255.0, x_test/255.0
# Sherpa client
client = sherpa.Client()
trial = client.get_trial()
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'])
# Train model
for i in range(epochs):
    model.fit(x_train, y_train)
    loss, accuracy = model.evaluate(x_test, y_test)
    client.send_metrics(trial=trial, iteration=i,
                        objective=accuracy,
                        context={'loss': loss})
```

Fig. 5. A code listing showing how to use Sherpa in parallel mode to tune the hyperparameters of a neural network trained on the handwritten digits dataset MNIST. The top code listing shows the server-script. The bottom listing shows the trial-script.

we also define a `LocalScheduler` instance. Hyperparameters, algorithm, and scheduler are passed to the `sherpa.optimize` function. We also pass a command "python trial.py". The command indicates how to execute the user's machine learning script. Furthermore, the argument `max_concurrent=2` indicates that two evaluations will be running at a time. Fig. 5 (bottom) shows the machine learning script. First, we set environment variables for GPU configuration. Next we create a *Client*. To obtain hyperparameters we call the client's `get_trial` method. Furthermore, during training we call the client's `send_metrics` method. This replaces `add_observation` in parallel mode. Also, in parallel mode no *finalize* call is needed.

4.2. Deep learning for cloud resolving models

4.2.1. Introduction

The following illustrates an example of a Sherpa hyperparameter optimization in the field of climate modeling, specifically cloud resolving models (CRM). We apply Sherpa to optimize the deep neural network (DNN) proposed by [43].

The input to the model is a 94-dimensional vector. Features include temperature, humidity, meridional wind, surface pressure, incoming solar radiation, sensible heat flux, and latent heat flux. The output of the DNN is a 65-dimensional vector. It is composed of the sum of the CRM and radiative heating rates, the CRM moistening rate, the net radiative fluxes at the top

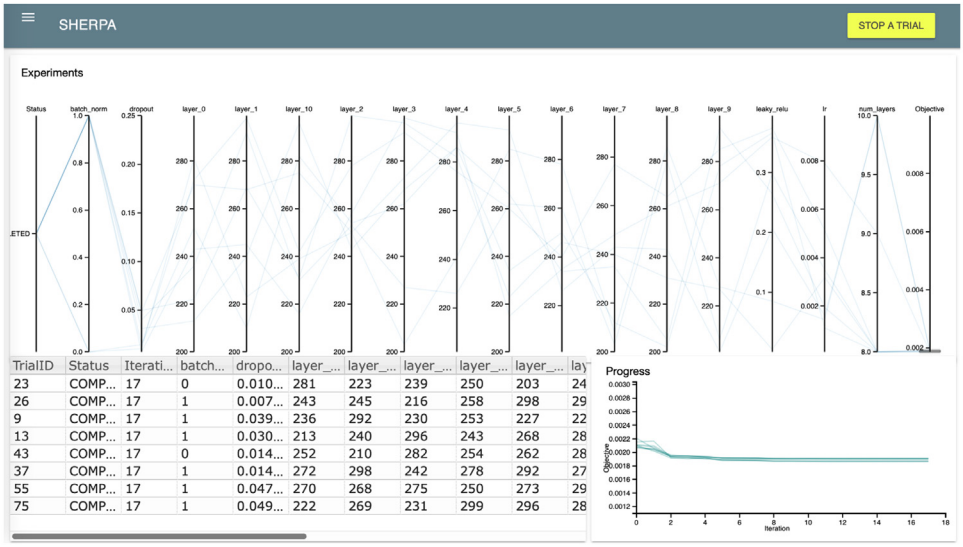


Fig. A.6. Screenshot of the dashboard at the end of the initial random search. The 8 best trials were selected by brushing of the *Objective* axis in the parallel coordinates plot.

Table A.2
DNN hyperparameter search space.

Name	Options	Parameter type
Batch normalization [44]	[yes, no]	Choice
Dropout [45,46]	[0, 0.25]	Continuous
Leaky ReLU coefficient [47]	[0–0.4]	Continuous
Learning rate	[0.0001–0.01]	Continuous (log)
Nodes per layer	[200–300]	Discrete
Number of layers	[8–10]	Discrete

Table A.3
Best hyperparameter configuration found by Sherpa.

Batch Normalization	No
Dropout	0.0
Leaky ReLU coefficient	0.3957
Learning Rate	0.001301
Learning Rate Decay	0.843784
Nodes per Layer	[299, 269, 248, 293, 251, 281, 258, 277, 209, 270]
Number of layers	10

of the atmosphere and surface of the earth, and the observed precipitation.

4.2.2. General hyperparameter optimization

Initially a random search was conducted on the following hyperparameters: batch normalization [44], dropout [45,46], Leaky ReLU coefficient [47], learning rate, nodes per hidden layer, number of hidden layers. The parameter ranges were chosen to encompass the parameters specified in [43]. From the dashboard (Fig. A.6) we identify that the best performing configurations have low dropout, leaky ReLU coefficients mostly around 0.3 or larger, and learning rates mostly near 0.002. The majority of good models have 8 layers and batch normalization. However, the number of units does not seem to have a large impact. The hyperparameter ranges and best configuration are provided in Tables A.2 and A.3 in the Appendix.

4.2.3. Optimization of the learning rate schedule

An additional search was conducted to fine-tune the DNN training hyperparameters. Specifically, the initial learning rate and the learning rate decay were optimized. The range of initial learning rate values was $\pm 10^{-4}$ of the best value from Section 4.2.2. The range of learning rate decay factors was 0.5 to 1. The learning rate gets multiplied by this factor after every epoch to produce a new learning rate. In comparison, the model in [43] uses a decay factor of approximately 0.58. The remaining hyperparameters were set to the best configuration from Section 4.2.2. A total of 50 trials were evaluated via random search. The best learning rate was found to be 0.001196. The best decay value was found as 0.843784. The overall optimal hyperparameter setting is shown in Table A.3 of the Appendix.

4.2.4. Results

We compare the model found by Sherpa to the model from [43] via R^2 plots (Fig. A.7). The R^2 plots show the coefficient of determination at different pressures and latitudes. We find that the Sherpa model consistently outperforms the comparison model. In particular, it is able to perform for latitudes for which the prior model fails. Fig. A.7(f) shows that the Sherpa model's loss reduces further after the [43] model has converged. This is the result of the learning rate fine-tuning from Section 4.2.3.

5. Impact

Machine learning is used to ever larger extends in the scientific community. Nearly every machine learning application can benefit from hyperparameter optimization. The issue is that researchers often do not have a practical tool at hand. Therefore, they usually resort to manually tuning parameters. Sherpa aims to be this tool. Its goal is to require minimal learning from the user to get started. It also aims to support the user as their needs for parallel evaluation or exotic optimization algorithms grow. As shown by references in Section 1, Sherpa is already being used by researchers to achieve improvements in a variety of domains. In addition to that, the software has been downloaded more than 20000 times from the PyPi Python package manager.[7] It also has over 260 stars on the software hosting website GitHub. A GitHub star means that another user has added the software to a personal list for later reference.

[7] https://pepy.tech/project/parameter-sherpa

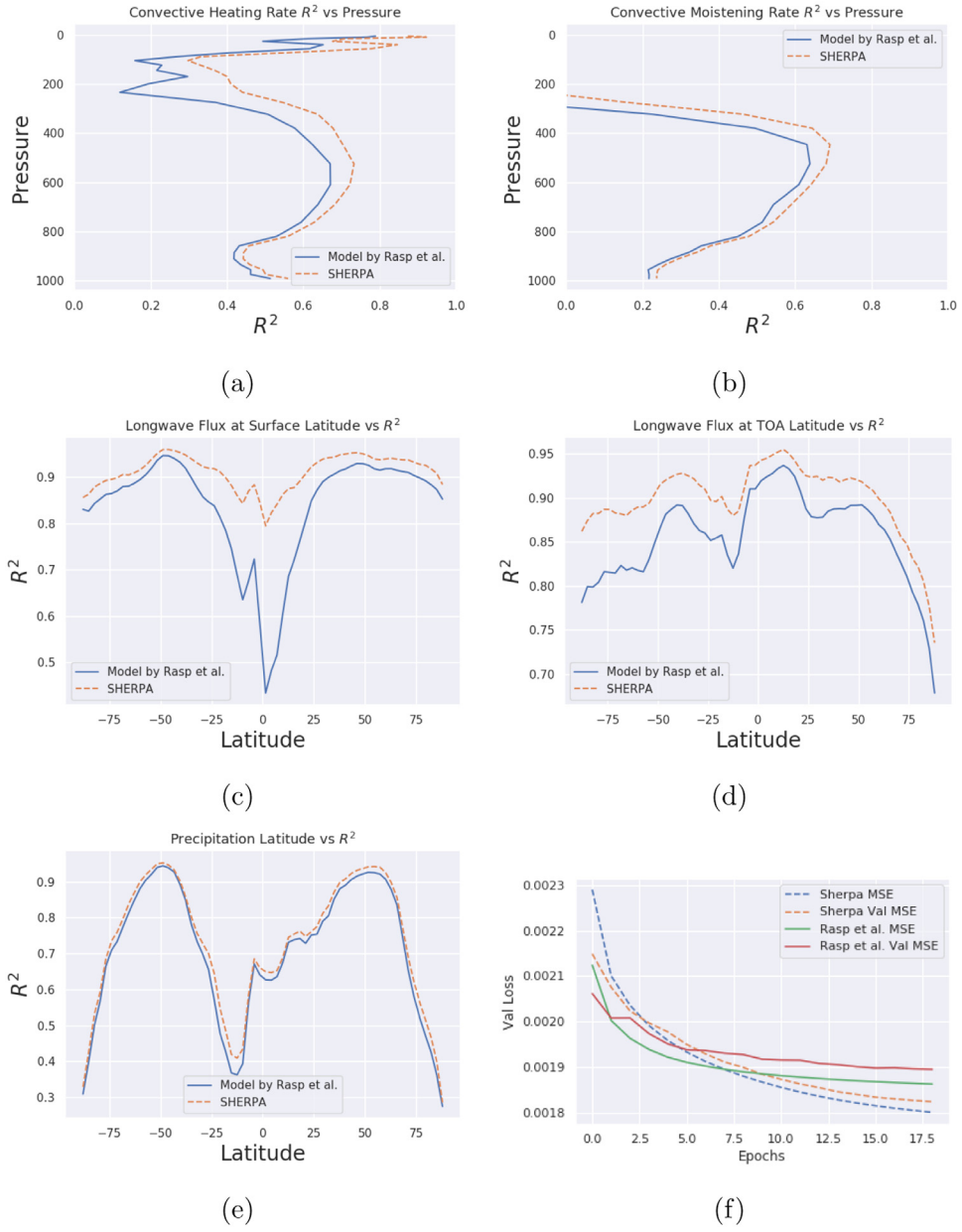


Fig. A.7. Case study results for an optimized deep neural network applied to cloud resolving models. Figs. A.7(a) and A.7(b) show the coefficient of determination R^2 vs. pressure for convective heating rate and convective moistening rate, respectively. Figs. A.7(c), A.7(d), and A.7(e) show R^2 values against latitude, and A.7(f) shows loss trajectories. All figures compare the optimized Sherpa model against the model developed by [43].

6. Conclusions

Sherpa is a flexible open-source software for robust hyperparameter optimization of machine learning models. It provides the user with several interchangeable hyperparameter optimization algorithms, each of which may be useful at different stages of model development. Its interactive dashboard allows the user to monitor and analyze the results of multiple hyperparameter optimization runs in real-time. It also allows the user to see patterns in the performance of hyperparameters to judge the robustness of individual settings. Sherpa can be used on a laptop or in a distributed fashion on a cluster. In summary, rather than a black-box that spits out one hyperparameter setting, Sherpa provides the tools that a researcher needs when doing hyperparameter exploration and optimization for the development of machine learning models.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We would like to thank Amin Tavakoli, Christine Lee, Gregor Urban, and Siwei Chen for helping test the software and providing useful feedback, and Yuzo Kanomata for computing support. This work was in part supported by the National Science Foundation, USA under grant number 1633631 to JC and 1839429 to PB. We also wish to acknowledge a hardware grant from NVIDIA, USA.

Appendix. Deep learning for cloud resolving models

Initially a random search was conducted on the hyperparameters listed in Table A.2.

A screenshot of the Sherpa dashboard at the end of the hyperparameter optimization is shown in Fig. A.6 (recommended to be viewed as PDF and via zooming in). On the dashboard layer_x refers to the number of nodes in layer x. From Fig. A.6 one can see that the best performing configurations have low dropout, leaky ReLU coefficients mostly around 0.3 or larger, and learning rates mostly near 0.002. The majority of good models have 8 layers and batch normalization. However, the number of units does not seem have a large impact.

Following the secondary search for an optimal learning rate schedule (Section 4.2.3) the hyperparameters in Table A.3) were found to be overall optimal. The optimized learning rate and schedule found by Sherpa is of considerable importance. Referencing the loss curves in Fig. A.7(f) one can see the learning rate schedule used in [43] forces the learning rate to decay rapidly causing an early plateau of the loss. The learning rate schedule discovered by Sherpa on the other hand allows the DNN to keep learning, further reducing the loss.

Fig. A.7 displays results of the optimized model as they pertain to climate modeling metrics. These plots denote R^2 values at corresponding pressures and latitudes. Larger values of the R^2 indicate that the DNN is able to explain more variance in the corresponding variable. Of particular importance, are areas where Sherpa is able to perform well in regions where the previously published model fails (e.g. latitudes between -25 and 25 in Fig. A.7(c)). At all pressures and latitudes the Sherpa model outperforms the previously published model and thereby achieves a new state of the art for this dataset.

References

[1] Hutter F, Hoos HH, Leyton-Brown K. Sequential model-based optimization for general algorithm configuration. In: International conference on learning and intelligent optimization. Springer; 2011, p. 507–23.

[2] Snoek J, Larochelle H, Adams RP. Practical bayesian optimization of machine learning algorithms. In: Advances in neural information processing systems. 2012, p. 2951–9.

[3] Bergstra J, Yamins D, Cox DD. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In: Proceedings of the 12th Python in science conference. Citeseer; 2013, p. 13–20.

[4] Authors TG. Gpyopt: A Bayesian optimization framework in python. 2016, http://github.com/SheffieldML/GPyOpt.

[5] Klein A, Falkner S, Mansur N, Hutter F. RoBO: A flexible and robust Bayesian optimization framework in Python. In: NIPS 2017 Bayesian optimization workshop; 2017.

[6] Kandasamy K, Vysyaraju KR, Neiswanger W, Paria B, Collins CR, Schneider J, Poczos B, Xing EP. Tuning hyperparameters without grad students: Scalable and robust Bayesian optimisation with dragonfly. J Mach Learn Res 2020;21(81):1–27, http://jmlr.org/papers/v21/18-223.html.

[7] Wu J, Frazier P. The parallel knowledge gradient method for batch bayesian optimization. In: Advances in neural information processing systems. 2016, p. 3126–34.

[8] Wu J, Poloczek M, Wilson AG, Frazier PI. Bayesian optimization with gradients. In: Advances in neural information processing systems. 2017, p. 5267–78.

[9] Bischla B, Richterb J, Bossekc J, Hornb D, Thomasa J, Langb M. mlrMBO: A Modular framework for model-based optimization of expensive black-box functions. stat 2017;1050:9.

[10] Li L, Jamieson K, DeSalvo G, Rostamizadeh A, Talwalkar A. Hyperband: A novel bandit-based approach to hyperparameter optimization. J Mach Learn Res 2017;18(1):6765–816.

[11] Falkner S, Klein A, Hutter F. BOHB: Robust and efficient hyperparameter optimization at scale. In: Dy J, Krause A, editors. Proceedings of the 35th international conference on machine learning. Proceedings of machine learning research, vol. 80, Stockholmsmässan, Stockholm Sweden: PMLR; 2018, p. 1437–46, http://proceedings.mlr.press/v80/falkner18a.html.

[12] Jaderberg M, Dalibard V, Osindero S, Czarnecki WM, Donahue J, Razavi A, Vinyals O, Green T, Dunning I, Simonyan K, et al. Population based training of neural networks. 2017, arXiv preprint arXiv:1711.09846.

[13] Igel C, Suttorp T, Hansen N. A computational efficient covariance matrix update and a (1+ 1)-CMA for evolution strategies. In: Proceedings of the 8th annual conference on genetic and evolutionary computation. ACM; 2006, p. 453–60.

[14] Olson RS, Urbanowicz RJ, Andrews PC, Lavender NA, Kidd LC, Moore JH. Applications of evolutionary computation: 19th european conference, evoapplications 2016, porto, Portugal, march 30 – april 1, 2016, proceedings, part I. Springer International Publishing; 2016, p. 123–37. http://dx.doi.org/10.1007/978-3-319-31204-0_9.

[15] Olson RS, Bartley N, Urbanowicz RJ, Moore JH. Evaluation of a tree-based pipeline optimization tool for automating data science. In: Proceedings of the genetic and evolutionary computation conference 2016. GECCO '16, New York, NY, USA: ACM; 2016, p. 485–92, http://doi.acm.org/10.1145/2908812.2908918.

[16] Kotthoff L, Thornton C, Hoos HH, Hutter F, Leyton-Brown K. Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA. J Mach Learn Res 2017;18(1):826–30.

[17] Feurer M, Klein A, Eggensperger K, Springenberg J, Blum M, Hutter F. Efficient and robust automated machine learning. In: Advances in neural information processing systems. 2015, p. 2962–70.

[18] Holmes G, Donkin A, Witten IH. Weka: A machine learning workbench. In: Proceedings of ANZIIS'94-Australian New Zealnd intelligent information systems conference. IEEE; 1994, p. 357–61.

[19] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, et al. Scikit-learn: Machine learning in python. J Mach Learn Res 2011;12(Oct):2825–30.

[20] Chollet F, et al. Keras. 2015, https://keras.io.

[21] Pumperla M. Hyperas. Github; 2019.

[22] Jin H, Song Q, Hu X. Auto-keras: An efficient neural architecture search system. In: Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining. ACM; 2019, p. 1946–56.

[23] Golovin D, Solnik B, Moitra S, Kochanski G, Karro J, Sculley D. Google vizier: A service for black-box optimization. In: Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining. ACM; 2017, p. 1487–95.

[24] Falcon W. Test tube. 2017, https://github.com/williamfalcon/test-tube.

[25] Liaw R, Liang E, Nishihara R, Moritz P, Gonzalez JE, Stoica I. Tune: A research platform for distributed model selection and training. 2018, arXiv preprint arXiv:1807.05118.

[26] Akiba T, Sano S, Yanase T, Ohta T, Koyama M. Optuna: A next-generation hyperparameter optimization framework. In: Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining. ACM; 2019, p. 2623–31.

[27] Gustafson L. Bayesian tuning and bandits: an extensible, open source library for automl. [M. eng thesis], Cambridge, MA: Massachusetts Institute of Technology; 2018, https://dai.lids.mit.edu/wp-content/uploads/2018/05/Laura_MEng_Final.pdf.

[28] O'Malley T, Bursztein E, Long J, Chollet F, Jin H, Invernizzi L, et al. Keras tuner. 2019, https://github.com/keras-team/keras-tuner.

[29] Sadowski P, Baldi P. Neural network regression with beta, Dirichlet, and Dirichlet-multinomial outputs [Unpublished results].

[30] Cao Z, Dan Y, Xiong Z, Niu C, Li X, Qian S, Hu J. Convolutional neural networks for crystal material property prediction using hybrid orbital-field matrix and magpie descriptors. Crystals 2019;9(4):191.

[31] Baldi P, Bian J, Hertel L, Li L. Improved energy reconstruction in NOvA with regression convolutional neural networks. Phys Rev D 2019;99(1):012011.

[32] Ritter C, Wollmann T, Bernhard P, Gunkel M, Braun DM, Lee J-Y, Meiners J, Simon R, Sauter G, Erfle H, et al. Hyperparameter optimization for image analysis: application to prostate tissue images and live cell data of virus-infected cells. Int J Comput Assist Radiol Surg 2019;1–11.

[33] Langford Z, Eisenbeiser L, Vondal M. Robust signal classification using siamese networks. In: Proceedings of the ACM workshop on wireless security and machine learning. ACM; 2019, p. 1–5.

[34] Bergstra J, Bengio Y. Random search for hyper-parameter optimization. J Mach Learn Res 2012;13(Feb):281–305.

[35] Li L, Jamieson K, Rostamizadeh A, Talwalkar A. Parallelizing hyperband for large-scale tuning. In: SysML. 2018.

[36] Inselberg A, Dimsdale B. Parallel coordinates for visualizing multi-dimensional geometry. In: Computer graphics 1987. Springer; 1987, p. 25–44.

[37] Hauser H, Ledermann F, Doleisch H. Angular brushing of extended parallel coordinates. In: Information visualization, 2002. INFOVIS 2002. IEEE symposium on. IEEE; 2002, p. 127–30.

[38] Chang K. Parallel coordinates. 2019, https://github.com/syntagmatic/parallel-coordinates.

[39] Gentzsch W. Sun grid engine: Towards creating a compute power grid. In: Cluster computing and the grid, 2001. Proceedings. First IEEE/ACM international symposium on. IEEE; 2001, p. 35–6.

[40] Yoo AB, Jette MA, Grondona M. Slurm: Simple linux utility for resource management. In: Workshop on job scheduling strategies for parallel processing. Springer; 2003, p. 44–60.

[41] Deng L. The MNIST database of handwritten digit images for machine learning research [best of the web]. IEEE Signal Process Mag 2012;29(6):141–2.
[42] Kingma DP, Ba JL. Adam: A method for stochastic gradient descent. In: ICLR: international conference on learning representations; 2015.
[43] Rasp S, Pritchard M, Gentine P. Deep learning to represent subgrid processes in climate models. Proc Natl Acad Sci 2018;115(39):9684–9.
[44] Ioffe S, Szegedy C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: Bach F, Blei D, editors. Proceedings of the 32nd international conference on machine learning. Proceedings of machine learning research, vol. 37, Lille, France: PMLR; 2015, p. 448–56, http://proceedings.mlr.press/v37/ioffe15.html.
[45] Srivastava N, Hinton G, Krizhevsky A, Sutskever I, Salakhutdinov R. Dropout: a simple way to prevent neural networks from overfitting. J Mach Learn Res 2014;15(1):1929–58.
[46] Baldi P, Sadowski PJ. Understanding dropout. In: Advances in neural information processing systems. 2013, p. 2814–22.
[47] Agostinelli F, Hoffman M, Sadowski P, Baldi P. Learning activation functions to improve deep neural networks. 2014, arXiv preprint arXiv:1412.6830.