Introducing Sacred: A Tool to Facilitate Reproducible Research

Klaus Greff KLAUS@IDSIA.CH
Jürgen Schmidhuber JUERGEN@IDSIA.CH

The Swiss AI Lab IDSIA
Università della Svizzera italiana (USI)
Scuola universitaria professionale della Svizzera italiana (SUPSI)

Abstract

We introduce Sacred, an open-source Python framework to help set up, run, and organize machine learning experiments. It focuses on convenience of use and features a powerful configuration system that can combine entries from within the code, from config-files, and from the command-line. Furthermore it helps to organize and keep the results reproducible by storing all relevant information in a database.

Keywords: Python, Reproducible Research, Hyperparameters

1. Introduction

The process of ML research still involves a lot of manual work apart from the actual innovation. This includes making the implementation configurable, tuning the hyperparamters, storing and organizing the parameters and results, and making sure they are reproducible. For many of these problems automatic solutions exist, but integrating them with the experiment still requires a lot of manual work. This is why under the pressure of deadlines important pillars of research like reproducibility often get neglected. What is missing is a framework that integrates solutions to all these parts together, such that dealing with these problems becomes effortless. This is the gap that Sacred tries to fill.

2. Overview

Sacred is an open-source Python library designed for easy of use and to be minimally intrusive to research code. The sourcecode was released under the MIT licence on Github (2015). Its main goal is to help with all the problems of day to day machine learning research, while requiring only minimal adjustments of the code.

The core abstraction in Sacred is the `Experiment` which has at least a name and a main method. Converting a python script into an experiment is very easy and involves only adding a handful of lines as demonstrated in Listing 1. Once that is done Sacred helps to making it configurable, run it from the command-line, and keep track of all runs by storing all related information like configuration, results, the source code, dependencies, random seeds, etc. in a database.

Listing 1 Simple code example for to training a SVM on the iris dataset (left) and a version of the same code as a Sacred `Experiment` (right). Notice that only seven lines had to be adjusted, and that this overhead is independent of the length of the code.

```python
from numpy.random import permutation
from sklearn import svm, datasets






C = 1.0
gamma = 0.7



iris = datasets.load_iris()
perm = permutation(iris.target.size)
iris.data = iris.data[perm]
iris.target = iris.target[perm]
clf = svm.SVC(C, 'rbf', gamma=gamma)
clf.fit(iris.data[:90],
        iris.target[:90])
print(clf.score(iris.data[90:],
                iris.target[90:]))
```

```python
from numpy.random import permutation
from sklearn import svm, datasets
from sacred import Experiment
ex = Experiment('iris_rbf_svm')

@ex.config
def cfg():
  C = 1.0
  gamma = 0.7

@ex.automain
def run(C, gamma):
  iris = datasets.load_iris()
  per = permutation(iris.target.size)
  iris.data = iris.data[per]
  iris.target = iris.target[per]
  clf = svm.SVC(C, 'rbf', gamma=gamma)
  clf.fit(iris.data[:90],
          iris.target[:90])
  return clf.score(iris.data[90:],
                   iris.target[90:])
```

3. Configuration

Each `Experiment` has a configuration whose entries represent its (hyper-)paramaters. An important goal of Sacred is to make it convenient to expose these parameters, such that they can be automatically optimized and kept track of.

3.1. Setting up

The preferred way to set up the configuration is by decorating a function with `@ex.config` which adds the variables from its local scope to the configuration. This is syntactically convenient and allows using the full expressiveness of python. For users that prefer plain dictionaries or external configuration files, those can also easily be added to the configuration. All the entries of the configuration are enforced to be JSON-serializable, such that they can easily be stored and queried (see Section 5.2).

3.2. Accessing

To make all configuration entries easily accessible, Sacred employs the mechanism of *dependency injection*. Every captured function can simply accept any configuration entry as a parameter. Whenever such a function is called Sacred will automatically fill in those parameters from the configuration. The main function is automatically considered a captured function, and so is any other function decorated with `@ex.capture`.

4. Running

Sacred interferes as little as possible with running an `Experiment`, thus leaving the user free to incorporate them in whatever workflow they are used to. Each `Experiment` automatically comes with a command-line interface, but they can just as easily be called directly from other Python code.

The command-line interface allows changing arbitrary configuration entries, using the standard python syntax[1] like this:

```
> python example.py run with C=3.4e-2 gamma=0.5
```

Apart from running the main function (here: `run`) it also provides commands to inspect the configuration (`print_config`) and to display the dependencies (`print_dependencies`). It also provides flags to get help, control the log-level, add a MongoDB observer (see Section 5), and for debugging. The command-line interface also allows adding custom commands, by just decorating a function with `@ex.command`.

All of the above can just as easily accomplished directly from python:

```
from example import ex
# runs the default configuration
r = ex.run()
# run with updated configuration
r = ex.run(config_updates={'C': 3.4e2, 'gamma': 0.5})
# run the print_config command
r = ex.run_command('print_config', config_updates={'gamma': 7})
```

After each of these calls `r` will contain a `Run` object with all kinds of details about the run including the result and the (modified) configuration.

5. Bookkeeping

`Experiments` implement the observer pattern (Gamma et al., 1994) by publishing all kinds of information in the form of events and allowing observers to subscribe to them. These events are fired when a run is started, every couple of seconds while it is running and once it stops (either successfully or by failing). Sacred ships with an observer that stores all the information about the run in a MongoDB database, but the interface also supports adding custom observers.

5.1. Collected Information

The MongoObserver collects a lot of information about the experiment and the run. Most importantly of course it will save the configuration and the result. But it will also among others save a snapshot of the source-code, a list of auto-detected package dependencies and the stdout of the experiment. Below is a summary of all the collected data:

Configuration configuration values used for this run

Source Code source code of all used source files

Dependencies version of all detected package dependencies

[1] Except for an extra set of quotes needed sometimes to satisfy the argument parsing of the bash.

Host information about the host that is running the experiment

Metadata start and stop times, status, result or fail-trace if needed

Custom Info a dictionary of custom information

stdout captured console output of the run

Resources and Artifacts extra files needed or created by the run that should be saved

5.2. MongoDB

MongoDB (2015) is a noSQL database, or more precisely a *Document Database*: It allows the storage of arbitrary JSON documents without the need for a schema like in a SQL database. These database entries can be queried based on their content and structure. This flexibility makes it a good fit for Sacred, because it permits arbitrary configuration for each experiment that can still be queried and filtered later on. In particular this feature has been very useful to perform large scale studies like the one in Greff et al. (2015).

6. Reproducibility

Maybe the most important goal of Sacred is to collect all the necessary information to make all the runs reproducible. To ensure that it features a simple integrated version control system that guarantees that for each run all the required files are stored in the same database. Notice that the database entry in **??** contains the name and MD5 hash of the `example.py` file (line 12). Sacred actually also saves the contents of that file in a separate collection[2]. The same mechanism can also be used to save additional resources or files created by the run (called artifacts).

There is one major obstacle of reproducibility left: randomness. Randomization is an important part of many machine learning algorithms, but it inherently conflicts with the goal of reproducibility. The solution of course is to use pseudo random number generators (PRNG) that take a seed and generate seemingly random numbers from that in a deterministic fashion. But this is only effective if the seed of the PRNG is not manually set and kept track of. Also if the seed is set to a fixed value as part of the code, then all runs will share the same randomness, which can be an undesired effect.

Sacred solves these problems by always generating a seed for each experiment that is stored as part of the configuration. It can be accessed from the code in the same way as every other config entry, but Sacred can also automatically generate seeds and PRNGs that deterministically depend on that root seed for you. Furthermore, Sacred automatically seeds the global PRNGs of the `random` and `numpy` modules, thus making most applications of randomization reproducible without any intervention of the user.

7. Related Work

There are only a few projects that we are aware of that have a focus similar to Sacred with the closest one being Sumatra (Davison, 2012). It comes as a command-line tool that can operate also with non-python experiments, and helps to do all the bookkeeping. Under the hood it uses a SQL database to store all the runs and comes with a versatile web-interface

[2] Collections in MongoDB are like tables in SQL databases.

to view and edit the stored information. The main drawback of Sumatra, and indeed the main reason why we opted for our own library is its workflow. It requires initializing a project directory, the parameters need to be in a separate file and the experiment must be an executable that takes the name of a config-file as a command-line parameter.

The CDE project (Guo, 2012) takes a completely different and much more general approach to facilitate reproducible research. It uses the linux kernel to track *all* files, including data, programs and libraries that were used for an experiment. These files are then bundled together and because it also includes system libraries the resulting package can be run on virtually any other linux machine. It doesn't help organization or bookkeeping, but, given that the user takes care of parameters and randomness, provides a very thorough solution to the problem of reproducibility.

Jobman (2012) is a python library that grew out of the need for scheduling lots of machine learning experiments. It helps with organizing hyperparameter searches and as a side-effect it also keeps track of hyperparameters and results. It requires the experiment to take the form a python function with a certain signature.

Experiment databases (Vanschoren et al., 2012; Smith et al., 2014) make an effort to unify the storage of machine learning problems and experiments by expressing them in a common language. By standardizing that language they improve comparability and communicability of the results. The most wellknown example of might be the OpenML project Vanschoren et al. (2014). Expressing experiments in a common language implies certain restrictions on the performed experiments. For this reason we chose not to build Sacred ontop of an experiment database, to keep it applicable to as many usecases as possible. That being said, we believe there is a lot of value in adding (optional) interfaces to experiment databases to Sacred.

8. Roadmap

Sacred is a framework that mainly integrates different solutions to data-science research problems. Because of that, there are many useful ways in which it could be extended. Apart from the above mentioned interface to OpenML the following points are high up our list:

Hyperparameter optimization has become a common and very important part of machine learning research, and with the powerful configuration system of Sacred in place this an obvious next step. So with the next release (0.7) of Sacred we plan to ease integration of tools like `spearmint` (Snoek et al., 2012) and `hyperopt` (Bergstra et al., 2013) into the workflow. In the same vein it is necessary to include tools for analysing the importance of hyperparameters like the FANOVA framework of Hutter et al. (2014).

The next important step will be to also provide a graphical interface to help inspecting and edit past and current runs. Ideally this will take the form of a web-interface that connects directly to the database.

Another popular request is to have a bookkeeping backend that supports local storage. That could be in the form of flat files in a directory or a SQLite database. These backends are particularly easy to add so we also hope for contributions from the users for more specialized usecases.

References

James Bergstra, Dan Yamins, and David D Cox. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *Proceedings of the 12th Python in Science Conference*, pages 13–20, 2013.

Andrew Davison. Automated capture of experiment context for easier reproducibility in computational research. *Computing in Science & Engineering*, 14(4):48–56, 2012.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

Github. Sacred 0.6.3, 2015. URL https://github.com/IDSIA/sacred/releases/tag/0.6.3. [Online; accessed 8-June-2015].

Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *arXiv preprint arXiv:1503.04069*, 2015.

Philip Guo. Cde: A tool for creating portable experimental software packages. *Computing in Science & Engineering*, 14(4):32–35, 2012.

Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. An efficient approach for assessing hyperparameter importance. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 754–762, 2014.

Jobman. Welcome – jobman 0.1 documentation, 2012. URL http://deeplearning.net/software/jobman/. [Online; accessed 8-June-2015].

MongoDB. Mongodb, 2015. URL https://www.mongodb.org/. [Online; accessed 8-June-2015].

Michael R. Smith, Andrew White, Christophe Giraud-Carrier, and Tony Martinez. An easy to use repository for comparing and improving machine learning algorithm usage. *arXiv:1405.7292 [cs, stat]*, May 2014. URL http://arxiv.org/abs/1405.7292.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

Joaquin Vanschoren, Hendrik Blockeel, Bernhard Pfahringer, and Geoffrey Holmes. Experiment databases. *Machine Learning*, 87(2):127–158, January 2012. ISSN 0885-6125, 1573-0565. doi: 10.1007/s10994-011-5277-0. URL http://link.springer.com/article/10.1007/s10994-011-5277-0.

Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. OpenML: networked science in machine learning. *ACM SIGKDD Explorations Newsletter*, 15(2):49–60, 2014. URL http://dl.acm.org/citation.cfm?id=2641198.