Thermodynamic Analytics Toolkit (TATi)

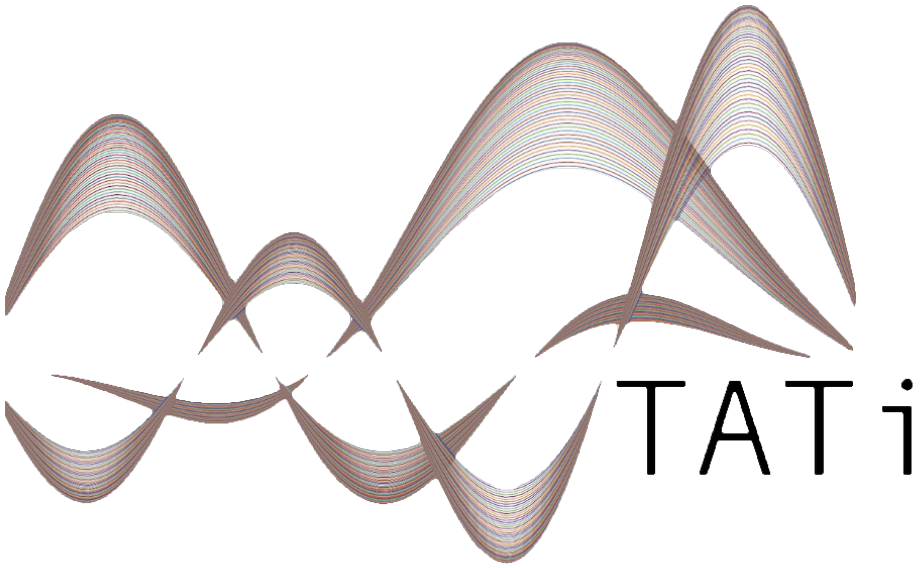
COLLABORATORS			
	TITLE : Thermodynamic Analytics Toolkit (TATi)		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Frederik Heber	2018-08-17	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
v0.9.1-0-g994aa50	2018-08-17		FH

Contents

List of Figures

List of Tables



2018-08-17 thermodynamicanalyticstoolkit: v0.9.1-0-g994aa50

TATi is a software suite written in Python based on tensorflow's Python API. It brings advanced sampling methods (GLA1 and GLA2, BAOAB, HMC) to *neural network training*. Its **tools** allow to assess the loss manifold's topology that depends on the employed neural network and the dataset. Moreover, its **simulation** module makes applying present sampling Python codes in the context of neural networks easy and straight-forward. The goal of the software is to enable the user to analyze and adapt the network employed for a specific classification problem to best fit her or his needs.

TATi has received financial support from a seed funding grant and through a Rutherford fellowship from the Alan Turing Institute in London (R-SIS-003, R-RUT-001) and EPSRC grant no. EP/P006175/1 (Data Driven Coarse Graining using Space-Time Diffusion Maps, B. Leimkuhler PI).

Frederik Heber

Chapter 1

Introduction

As an image says more than a thousand words, here is a mindmap showing all of TATi's main features.

Figure 1.1: Mindmap of all TATi features

If you know what all of this means, you might probably be interested. If not, then read on to the next section.

Note
TATi is growing, see the roadmap for things to come.

1.1 Before you start

In the following we assume that you, the reader, has a general familiarity with neural networks. You should know what a classification problem is, what an associated dataset for (supervised) learning needs to contain. You should know about what weights and biases in a neural network are and what the loss function does. You should also have a rough idea of what optimization is and that gradients with respect to the chosen loss function can be obtained through so-called backpropagation.

If you are *not* familiar with the above terminology, then please first read the section on Concepts.

1.2 What is ThermodynamicAnalyticsToolkit?

Typically, optimzation in neural network training employs methods such as Gradient Descent, Stochastic Gradient Descent or derived methods using momentum such as ADAM and so on. The loss function itself may be convex, however, the loss manifold given a large dataset is not convex in general. Hence, these methods will only find local minima. Naturally, multiple random starting positions are used and the winner takes it all. However, as there exponentially many minima this way we will never be able to glean any knowledge on how they are situated with respect to one another. We will never know how the network's topology, the number of nodes and so on influences it. Moreover, the eventually trained set of parameters of the neural network is never optimal. Not even when taking the generalization error into account. However, some simplicity must be hidden in those loss manifolds of neural networks: given enough data and processing power, they work marvelously even in spite of all these shortcomings and one may wonder why.

The essential idea behind this program package is that we use sampling instead of optimization: we are not interested in the local minimum closest to some random initial configuration and be done. Instead we aim at finding all of the minima and all possible barriers in between by treating the loss function as a high-dimensional potential and the weights and biases of the neural network as one-dimensional particles in a stochastic differential equation, namely Langevin Dynamics.

There is not need to panic: You do not need to know anything about these kinds of equations when using the program. However, rest assured that all statistical properties derived using trajectories obtained through these equations are meaningful.

The hope is to elucidate the marvel behind the wondrous performance of neural networks, maybe to obtain even better parametrizations or obtain the same in cheaper ways, and also to gather means of optimizing the neural network's topology given a specific dataset to train.

In essence, this program suite provides samplers using TensorFlow and analysis tools to extract specific statistical quantities from the computed particle trajectories.

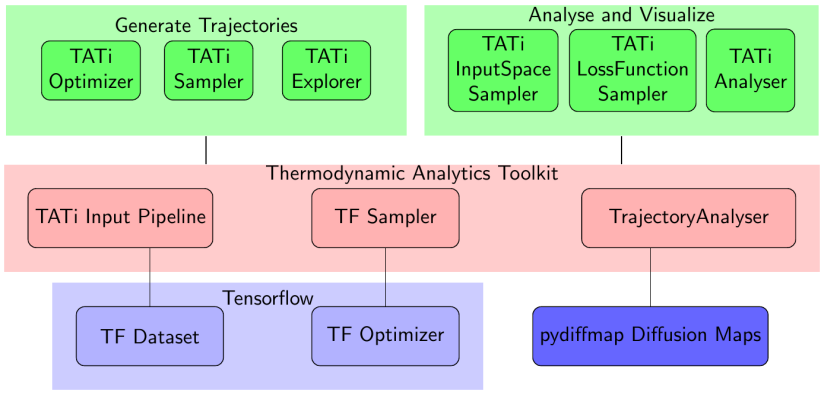


Figure 1.2: Architecture of TATi's tools

Moreover, it can also be used as a stand-alone Python module that allows to easily make use of present sampling Python code within the context of neural networks.

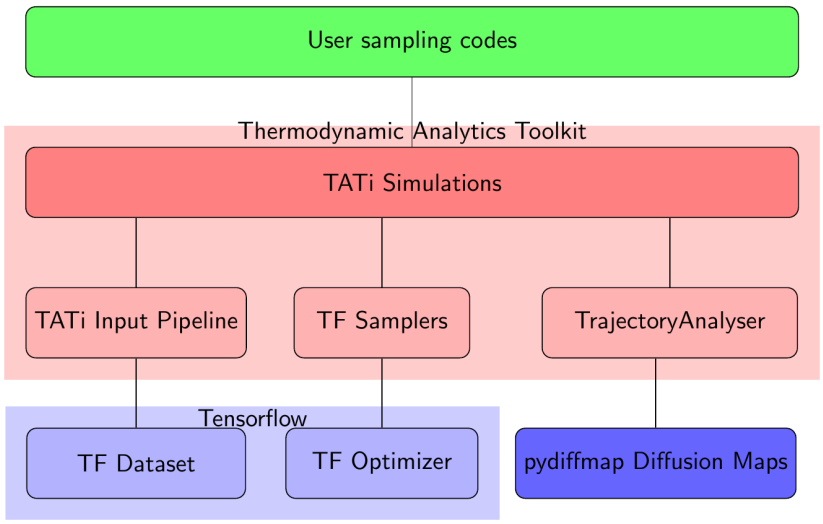


Figure 1.3: Architecture of the **simulation** module

It can be used both as python module and as stand-alone command-line tools. The former is called the **simulation** module and will probably be your first contact point with TATi.

1.3 Installation

In the following we explain the installation procedure to get ThermodynamicAnalyticsToolkit up and running.

1.3.1 Installation requirements

This program suite is implemented using python3 and the development mainly focused on Linux (development machine used Ubuntu 14.04 up to 18.04). At the moment other operation systems are not supported but may still work.

It has the following non-trivial dependencies:

- TensorFlow: version 1.4.1 till currently 1.6 supported
- Numpy:
- Pandas
- sklearn

Note that most of these packages can be easily installed using either the repository tool (using some linux derivate such as Ubuntu), e.g.

```
sudo apt install python3-numpy
```

or via **pip3**, i.e.

```
pip3 install numpy
```

Moreover, the following packages are not ultimately required but examples or tests may depend on them:

- matplotlib
- sqlite3

- gawk

The documentation is written in AsciiDoc and requires a suitable package to compile to HTML or create a PDF, e.g., using dblatex

- asciidoc
- dblatex

Finally, for the diffusion map analysis we recommend using the pydiffmap package, see https://github.com/DiffusionMapsAcademics/pyDiffMap.

In our setting what typically worked best was to use anaconda in the following manner:

```
conda create -n tensorflow python=3.5 -y
conda install -n tensorflow -y \
     tensorflow numpy scipy pandas scikit-learn matplotlib
```

In case your machine has GPU hardware for tensorflow, replace "tensorflow" by "tensorflow-gpu".

Note
Note that on systems with typical core i7 architecture recompiling tensorflow from source provided only very small runtime gains in our tests which in most cases do not support the extra effort. You may find it necessary for tackling really large networks and datasets and especially if you desire using Intel's MKL library for the CPU-based linear algebra computations.

Henceforth, we assume that there is a working tensorflow on your system, i.e. inside the python3 shell

```
import tensorflow as tf
```

does *not* throw an error.

Moreover,

```
a=tf.constant("Hello world")
sess=tf.Session()
sess.run(a)
```

should print "Hello world" or something similar.

Tip
You can check the version of your **tensorflow** installation at any time by inspecting `print(tf.__version__)`. Similarly, TATi's version can be obtained through

```
import tati

print(tati.__version__)
```

1.3.2 Installation procedure

Installation comes in two flavours: Either a tarball or a cloned repository.

The tarball releases are recommended if you only plan to use TATi and do not intend modifying its code. If, however, you need to use a development branch, then you have to clone from the repository.

In general, this package is distributed via autotools, "compiled" and installed via automake. If you are familiar with this set of tools, there should be no problem. If not, please refer to the text `INSTALL` file that is included in the tarball.

Note
Only the tarball contains a precompiled PDF. The cloned repository contains only the HTML pages.

1.3.3 From Tarball

Unpack the archive, assuming its suffix is `.bz2`.

```
tar -jxvf thermodynamicanalyticstoolkit-${revnumber}.tar.bz2
```

If the ending is `.gz`, you need to unpack by

```
tar -zxvf thermodynamicanalyticstoolkit-${revnumber}.tar.gz
```

Enter the directory

```
cd thermodynamicanalyticstoolkit
```

Continue then in section Configure, make, install.

1.3.4 From cloned repository

While the tarball does not require any autotools packages installed on your system, the cloned repository does. You need the following packages:

- autotools
- automake

To prepare code in the working directory, enter

```
./bootstrap.sh
```

1.3.5 Configure, make, make install

Next, we recommend to build the toolkit not in the source folder but in an extra folder, e.g., "build64". In the autotools lingo this is called an *out-of-source* build. It prevents cluttering of the source folder. Naturally, you may pick any name (and actually any location on your computer) as you see fit.

```
mkdir build64
cd build64
../configure --prefix="somepath" -C PYTHON="path to python3"
make
make install
```

More importantly, please replace "somepath" and "path to python3" by the desired installation path and the full path to the `python3` executable on your system.

Note
In case of having used *anaconda* for the installation of required packages, then you need to use

```
$HOME/.conda/envs/tensorflow/bin/python3
```

for the respective command, where `$HOME` is your home folder. This assumes that your anaconda environment is named **tensorflow** as in the example installation steps above.

Note
We recommend executing (after `make install` was run)

```
make -j4 check
```

additionally. This will execute every test on the extensive testsuite and report any errors. None should fail. If all fail, a possible cause might be a not working tensorflow installation. If some fail, please contact the author. The extra argument **-j4** instructs `make` to use four threads in parallel for testing. Use as many as you have cores on your machine.

1.4 License

As long as no other license statement is given, ThermodynamicAnalyticsToolkit is free for use under the GNU Public License (GPL) Version 3 (see https://www.gnu.org/licenses/gpl-3.0.en.html for full text).

1.5 Disclaimer

> Because the program is licensed free of charge, there is not warranty for the program, to the extent permitted by applicable law. Except when otherwise stated in writing in the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied. Including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair, or correction.
>
> — section 11 of the GPLv3 license *https://www.gnu.org/licenses/gpl-3.0.en.html*

1.6 Feedback

If you encounter any bugs, errors, or would like to submit feature request, please write to frederik.heber@gmail.com or open an issue at GitHub. The author is especially thankful for any description of all related events prior to occurrence of the error and auxiliary files. More explicitly, the **following information is crucial** in enabling assistance:

- **operating system** and version, e.g., Ubuntu 16.04
- **Tensorflow version**, e.g., TF 1.6
- **TATi version** (or respective branch on GitHub), e.g., TATi 0.8
- steps that lead to the error, possibly with **sample Python code**

Please mind sensible space restrictions of email attachments.

Chapter 2

Quickstart

Before we come to actually using TATi, we want to set the stage with a little trivial example: We will look at a very simple classification task and see how it is solved using neural networks.

2.1 Sampling in neural networks

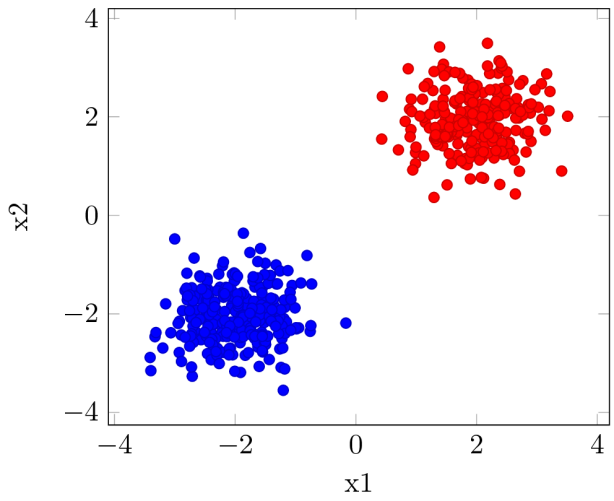


Figure 2.1: Dataset: "Two Clusters" dataset consisting of two normally distributed point clouds in two dimensions

Assume we are given a very simple data set as depicted in Dataset. The goal is to classify all red and blue dots into two different classes. This problem is quite simple to solve: a line in the two-dimensional space can easily separate the two classes.

A very simple neural network, a perceptron, is all we need: it uses two inputs nodes, namely each coordinate component, x_1 and x_2, and a single output node with an activation function f whose sign gives the class the input item belongs to. The network is given in Network.

Tip
A little while later we will see that an even simpler network suffices to classify the dataset well, which is intrinsically one-dimensional.

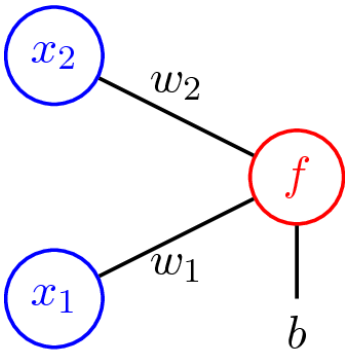


Figure 2.2: Network: Single-layer perceptron with weights and biases

2.1.1 Traing a Neural Network

In the following we want to use the mean square loss, i.e. the euclidean distance between the output from the network and the expected values per item, as the network's loss function. The loss depends implicitly on the dataset and explicitly on the weights and biases associated with the network. In our case, we have two weights for the two edges between input nodes, w_1 and w_2, and the output node and a single bias attached to the output node b.

The general goal of training is to find the set of parameters that achieve the lowest loss. To this end, we use the gradient that is readily obtained from the neural netwotk through backpropagation of the analytically known derivatives of the activation functions. The parameters are modified using Gradient Descent until the gradient becomes zero (or we stop before that).

2.1.2 What is Sampling?

Sampling typically has quite a different perspective: There, we look at a system of particles that have two internal properties: *location* and *momentum*. The location is simply their current value, that changes through its momentum over time. The momentum again changes because the particle are affected by a potential. The system is described by a so-called Hamilton operator that gives rise to its same named dynamics. If noise is additionally taken into account, then instead we look at Langevin Dynamics. *Noise is essential*: It is connected to the concept of *temperature* as the average squared momenta of each particle, the kinetic energy. High temperature therefore means large amounts of noise, while low temperature is associated with small amounts. Temperature is connected to the concept of heat bath that is a reservoir of kinetic energy allowing the particles to overcome barriers.

Returning to the neural networks, the role of the particles is taken up by the degrees of freedom of the system: weights and biases. The loss function is called the *potential* and it is accompanied by a *kinetic energy* that is simply the sum of all squared momenta. Adding Momentum to Optimizers in neural networks is a concept known already and inspired by physics. There, it counteracts areas of the loss function where it is essentially flat and the gradient therefore close to zero.

Sampling produces trajectories of particles moving along the manifold. Integrals along these trajectories, if they are long enough, are equivalent to integrating over the whole manifold, if the system is ergodic. And this is the key point! Essentially, it enables us to replace integrals over the whole (and possibly very high-dimensional domain) by a one-dimensional integral along the trajectory. This reduces dimensional complexity significantly if not all areas in the high-dimensional domain contribute equally to the integral, if the contribution from many areas is negligible (think: weight $\exp(-l(x))$ in integrand with a high value for $l(x)$). All we need to do is have the sampling produce trajectories only in the areas of interest.

By using sampling we mean to discover more of the loss manifold than just the closest local minimum, namely all minima. This excludes all regions with large loss function values. In other words, we would like to sample in such a way as only to stay in regions of the loss manifold associated with small values. Generating trajectories by dynamics where the negative of the gradient acts as a driving force onto each particle automatically brings them into regions where the loss' value is small. However, in general all possible minima locations will not form a connected region on the loss manifold. These minima regions may be separated by barriers which are needed to overcome. We distinguish two kinds,

- entropic barriers,
- and enthalpic barriers.

Both of which are conceptually very simple. The enthalpic barrier is simply a ridge that is very high where the particles would need a large momentum to overcome it. Entropic barriers on the other hand are passages very small in volume that are simply very difficult to find. In order to overcome barriers of the first kind, higher temperatures suffice. For the second type of barrier, this is not so easy. Metaphorically speaking, we are looking for a possibly very small door like Alice in Wonderland.

2.1.3 What Does the Landscape Look Like?

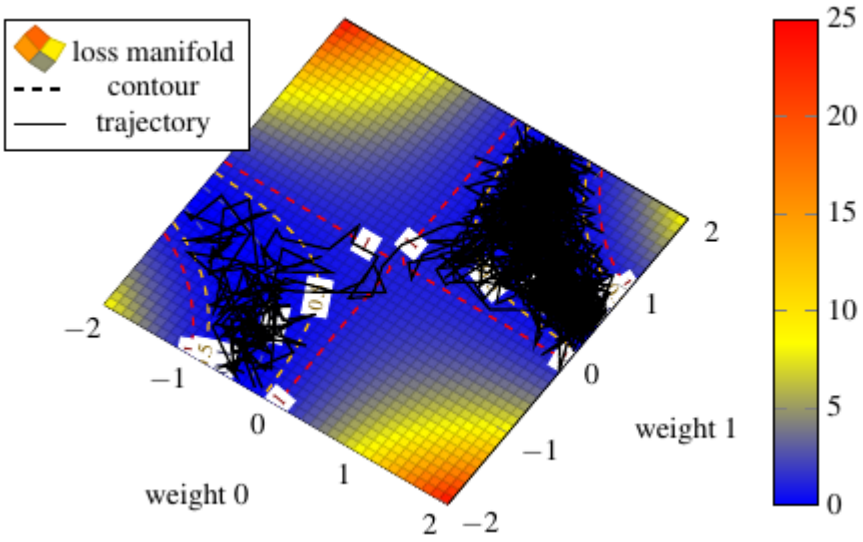
Let us have a closer look at a very simple loss landscape. In Figure Permutation symmetry we look at a very simple network of a single input node, with a single hidden layer containing just one node and a single output layer. Activation function is linear everywhere. We set the output node's and hidden node's bias to zero. The dataset contains two cluster of points, one (label -1) centered at -2, another (label +1) centered at 2 which is essentially the one given in Figure Dataset if projected onto x or y axis. Any product of the two degrees of freedom of the network, namely its two weights, equal to unity will classify the data well.

Permutation symmetry: Neural network with permutation symmetry to provoke multiple minima

In Figure Loss manifold we then turn to the loss landscape depending on either weight. We see two minima basins both of hyperbole or "banana" shape. There is a clear (enthalpic) potential barrier in between.

In the figure we also give a trajectory as squiggly black line. Here, we have chosen such an (inverse) temperature value such that it is able to pass the potential barrier and reach the other minima basin. As we have mentioned before, higher temperature helps to overcome enthalpic barriers.

Loss manifold: Loss landscape with an example trajectory



This quick description of the problem of sampling in the context of neural networks in data science should have acquainted you with some of the physical concepts underlying the idea of sampling. It hopefully has prepared you for the following quickstart tutorial on how to actually use ThermodynamicAnalyticsToolkit to perform sampling.

2.2 Using module simulation

As promised before, the first contact point in this quickstart tutorial with TATi is the `simulation` module. It has been designed explicitly for ease-of-use and to contain any functionality required for the sampling approach. However, it can to training as well as you will see. Typically, everything is achieved through two or three commands: One to setup TATi by handing it a dash of options, then calling a function to `fit()` or `sample()`. In the very end of this quickstart you will learn how to implement your first sampler using this interface.

If you have installed the package in the folder `/foo`, i.e. we have a folder `TATi` with a file `simulation.py` residing in there, then you probably need to add it to the `PYTHONPATH` as follows

```
PYTHONPATH=/foo python3
```

In this shell, you may import the sampling part of the package as follows

```
import TATi.simulation as tati
```

This will import the `Simulation` interface class as the shortcut `tati` from the file mentioned before. This class contains a set of convenience functions that hides all the complexity of setting up of input pipelines and networks. Accessing the loss function, gradients and alike or training and sampling can be done in just a few keystrokes.

In order to make your own python scripts executable and know about the correct (possibly non-standard) path to ThermodynamicAnalyticsToolkit, place the following two lines at the veryg beginning of your script:

```
import sys
sys.path.insert(1,"<path_to_TATi>/lib/python3.5/site-packages/")
```

where `<path_to_TATi>` needs to be replaced by your specific installation path and `python3.5` needs to be replaced if you are using a different python version. However, for the examples in this quickstart tutorial it is not necessary if you use `PYTHONPATH`.

2.2.1 Notation

In the following, we will use the following notation:

- dataset: $D = \{X, Y\}$ with features $X = \{x_d\}$ and labels $Y = \{y_d\}$
- batch of the dataset: $D_i = \{X_i, Y_i\}$
- network parameters: $w = \{w_1, \ldots, w_M\}$
- momenta of network parameters: $p = \{p_1, \ldots, p_M\}$
- neural network function: $F_w(x)$
- loss function: $L_D(w) = \sum_i l(F_w(x_i), y_i)$ with a loss $l(x, y)$
- gradients: $\nabla_w L_D(w)$
- Hessians: $H_{ij} = \partial_{w_i} \partial_{w_j} L_D(w)$

2.2.2 Instantiating TATi

The first thing in all the following example we will do is instantiate the `tati` class.

```
import TATi.simulation as tati

nn = tati(
  # comma-separated list of options
)
```

Although it is the `simulation` module, we "nickname" it `tati` in the following and hence will simply refer to this instance as `tati`.

This class takes a list of options in its construction or `__init__()` call. These options inform it about the dataset to use, the specific network topology, what sampler or optimizer to use and its parameters and so on.

To see how this works, we will first need a dataset to work on.

Note
All of the examples below can also be found in the folders `doc/userguide/python`, `doc/userguide/simulation`, and `doc/userguide/simulation/complex`.

2.2.2.1 Help on Options

`tati` has quite a number of options that control its behavior. You can request help to a specific option. Let us inspect the help for `batch_data_files`:

```
>>> from TATi.simulation as tati
>>> tati.help("batch_data_files")
Option name: batch_data_files
Description: set of files to read input from
Type       : list of <class 'str'>
Default    : []
```

This will print a description, give the default value and expected type.

Moreover, in case you have forgotten the name of one of the options.

```
>>> from TATi.simulation as tati
>>> tati.help()
averages_file:             CSV file name to write ensemble averages information such as ↩
    average kinetic, potential, virial
batch_data_file_type:      type of the files to read input from
 <remainder omitted>
```

This will print a general help listing all available options.

2.2.3 Setup

In the following we will first be creating a dataset to work on. This example code will be the most extensive one. All following ones are rather short and straight-forward.

2.2.4 Preparing a dataset

Therefore, let us prepare the dataset, see the Figure Dataset, for our following experiments.

At the moment, datasets are parsed from Comma Separated Values (CSV) or Tensorflow's own TFRecord files or can be provided in-memory from numpy arrays. In order for the following examples on optimization and sampling to work, we need such a data file containing features and labels.

TATi provides a few simple dataset generators contained in the class `ClassificationDatasets`.

One option therefore is to use the TATiDatasetWriter that provides access to `ClassificationDatasets`, see Writing a dataset. However, we can do the same using python as well. This should give you an idea that you are not constrained to the `simulation` part of the Python interface, see the reference on the general Python interface where we go through the same examples without importing `simulation`.

```
from TATi.datasets.classificationdatasets \
    import ClassificationDatasets as DatasetGenerator

import csv
import numpy as np

np.random.seed(426)

dataset_generator = DatasetGenerator()
xs, ys = dataset_generator.generate(
    dimension=500,
    noise=0.01,
    data_type=dataset_generator.TWOCLUSTERS)

# always shuffle data set is good practice
```

```
randomize = np.arange(len(xs))
np.random.shuffle(randomize)
xs[:] = np.array(xs)[randomize]
ys[:] = np.array(ys)[randomize]

with open("dataset-twoclusters.csv", 'w', newline='') as data_file:
    csv_writer = csv.writer(data_file, delimiter=',', \
                            quotechar='"', \
                            quoting=csv.QUOTE_MINIMAL)
    header = ["x"+str(i+1) for i in range(len(xs[0]))]+["label"]
    csv_writer.writerow(header)
    for x, y in zip(xs, ys):
        csv_writer.writerow(
            ['{:{width}.{precision}e}'.format(val, width=8,
                                              precision=8)
             for val in list(x)] \
            + ['{}'.format(y[0], width=8,
                                                precision=8)])
    data_file.close()
```

Warning
The labels need to be integer values. Importing will fail if they are not.

After importing some modules we first fix the numpy seed to 426 in order to get the same items reproducibly. Then, we first create 500 items using the `ClassificationDatasets` class from the **TWOCLUSTERS** dataset with a random perturbation of relative 0.01 magnitude. We shuffle the dataset as the generators typically create first items of one label class and then items of the other label class. This is not needed here as our *batch_size* will equal the dataset size but it is good practice generally.

Note
The class `ClassificationDatasets` mimicks the dataset examples that can also be found on the Tensorflow playground.

Afterwards, we write the dataset to a simple CSV file with columns "x1", "x2", and "label".

Caution
The file `dataset-twoclusters.csv` is used in the following examples, so keep it around.

This is the very simple dataset we want to learn, sample from and exlore in the following.

2.2.5 Evaluating loss and gradients

Having created the dataset, we are good to go and the remaining examples are easy and straight-forward.

The main idea of the `simulation` module is to be used as a simplified interface to access the loss and the gradients of the neural network without having to know about the internal of the neural network. In other words, we want to treat it as an abstract high-dimensional function, depending implicitly on the weights and explicitly on the dataset. Moreover, we have another abstract high-dimensional function, the loss that depends explicitly on the weights and implicitly on the dataset, whose derivative (the gradients with respect to the parameters) is available as a numpy array, see also the section Notation.

See the following example which sets up a simple fully-connected hidden network and evaluates loss and then the associated gradients.

```
import TATi.simulation as tati

import numpy as np

# prepare parameters
nn = tati(
    batch_data_files=["dataset-twoclusters.csv"],
    batch_size=10,
    output_activation="linear"
)

# assign parameters of NN
nn.parameters = np.zeros([nn.num_parameters()])

# simply evaluate loss
print(nn.loss())

# also evaluate gradients (from same batch)
print(nn.gradients())
```

All we need to do is set some parameters — here, *batch_data_files* sets the dataset file to parse, batch size of 10 and we use a linear output activation function — assign all network parameters to zero and then evaluate first the loss and then the gradients of the network.

Note
Under the hood it is a bit more complicated: loss and gradients are inherently connected. If *batch_size* is chosen smaller than the dataset dimension, naive evaluation of first loss and then gradients in two separate function calls would cause them to be evaluated on different batches. Depending on the size of the batch, the gradients will then not belong the to the respective loss evaluation and vice versa. Therefore, loss, accuracy, gradients, and hessians (if *do_hessians* is True) are cached. Only when one of them is evaluated for the second time (e.g., inside the loop body on the next iteration), then the next batch is used. This makes sure that calling either `loss()` first and then `gradients()` or the other way round yields the same values connected to the same dataset batch. In essence, just don't worry about it!

As you see in the above example, `tati` forms the general interface class that contains the network along with the dataset and everything in its internal state.

This is basically all the access you need in order to use your own optimization, sampling, or exploration methods in the context of neural networks in a high-level, abstract way.

2.2.6 Optimizing the network

Let us then start with optimizing the network, i.e. learning the data.

```
import TATi.simulation as tati

import numpy as np

nn = tati(
    batch_data_files=["dataset-twoclusters.csv"],
    batch_size=500,
    learning_rate=3e-2,
    max_steps=1000,
    optimizer="GradientDescent",
    output_activation="linear",
    seed=426,
)
training_data = nn.fit()
```

```
print("Train results")
print(np.asarray(training_data.run_info[-10::]))
print(np.asarray(training_data.trajectory[-10:]))
print(np.asarray(training_data.averages[-10:]))
```

Again all options are set in the init call to the interface. These options control how the optimization is performed, what kind of network is created, how often values are stored, and so on.

Let us quickly go through each of the parameters:

- *batch_size*

  sets the subset size of the data set looked at per training step, if smaller than dimension, then we add stochasticity/noise to the training but for the advantage of smaller runtime.

- *learning_rate*

  defines the scaling of the gradients in each training step, i.e. the learning rate. Values too large may miss the minimum, values too small need longer to reach it.

- *max_steps*

  gives the amount of training steps to be performed.

- optimizer

  defines the method to use for training. Here, we use Gradient Descent (in case *batch_size* is smaller than dimension, then we actually have Stochastic Gradient Descent).

- *output_activation*

  defines the activation function of all output nodes, here it is linear. Other choices are: tanh, relu, relu6.

- *seed*

  sets the seed of the random number generator. We will still have full randomness but in a deterministic manner, i.e. calling the same procedure again will bring up the exactly same values.

In our case, the default option values are such that the network we use looks exactly as in the Figure Network, namely a single-layer perceptron whose number of input and output nodes are completely fixed by the dataset. See Setting up the network for more details on how to specify the network topology.

Tip
In case you need to change these options elsewhere in your python code, use `set_options()`.

Warning
`set_options()` may need to reinitialize certain parts of `tati` s internal state depending on what options you choose to reset. Keep in mind that modifying the network will reinitialize all its parameters and other possible side-effects. See `simulation._affected_map` in `src/TATi/simulation.py` for an up-to-date list of what options affects what part of the state.

For these small networks the option *do_hessians* might be useful which will compute the hessian matrix at the end of the trajectory and use the largest eigenvalue to compute the optimal step width. This will add nodes to the underlying computational graph for computing the components of the hessian matrix. However, we will not do so here.

Caution
The creation of these hessian evaluation nodes (not speaking of their evaluation) is a $O(N^2)$ process in the number of parameters of the network N. Hence, this should only be done for small networks and on purpose.

After the options have been provided, the network is initialized internally and automatically, we then call `fit()` which performs the training and returns a structure containing runtime info, trajectory, and averages as a pandas `DataFrame`.

In the following section on sampling we will explain what each of these three dataframes contains exactly.

Tip
You want more output of what is actually going on in each training step? Set `verbose=1` or even `verbose=2` in the options when constructing `tati()`.

Let us have a quick glance at the decrease of the loss function over the steps by using `matplotlib`. In other words, let us look at how effective the training has been.

```
import pandas as pd
import numpy as np
import matplotlib
# use agg as backend to allow command-line use as well
matplotlib.use("agg")
import matplotlib.pyplot as plt

df_run = pd.read_csv("run.csv", sep=',', header=0)
run=np.asarray(df_run.loc[:,\
   ['step','loss','kinetic_energy', 'total_energy']])

plt.scatter(run[:,0], run[:,1])
plt.savefig('loss-step.png',
            bbox_inches='tight')
#plt.show()
```

The graph should look similar to the one obtained with `pgfplots` here (see https://sourceforge.net/pgfplots).

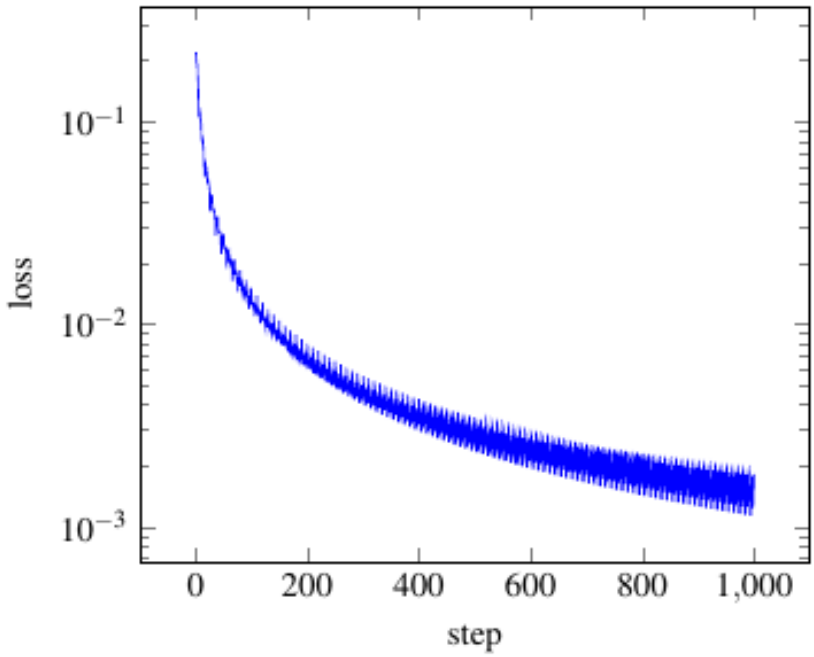


Figure 2.3: Loss history: Behavior of the loss over the optimization run

As you see the loss has decreased quite quickly down to 1e-3. Go and have a look at the other columns such as accuracy. Or try to visualize the change in the parameters (weights and biases) in the trajectories dataframe. See 10 Minutes to pandas if you are unfamiliar with the `pandas` module, yet.

Obviously, we did not use a different dataset set for testing the effectiveness of the training which should commonly be done. This way we cannot check whether we have overfitted or not. However, our example is trivial by design and the network too small to be prone to overfitting this dataset.

Nonetheless, we show how to supply a different dataset and evaluate loss and accuracy on it.

2.2.6.1 Provide your own dataset

You can directly supply your own dataset, e.g., from a numpy array residing in memory. See the following example where we do not generate the data but parse them from a CSV file instead of using the pandas module.

```
import TATi.simulation as tati

import numpy as np
import pandas as pd

nn = tati(
    output_activation="linear",
)
# e.g. parse dataset from CSV file into pandas frame
input_dimension = 2
output_dimension = 1
parsed_csv = pd.read_csv("dataset-twoclusters-test.csv", \
                         sep=',', header=0)
# extract feature and label columns as numpy arrays
features = np.asarray(\
    parsed_csv.iloc[:, 0:input_dimension])
labels = np.asarray(\
    parsed_csv.iloc[:, \
        input_dimension:input_dimension \
                        + output_dimension])

# supply dataset (this creates the input layer)
nn.dataset = [features, labels]

# this has created the network, now set
# parameters obtained from optimization run
nn.parameters = np.array([2.42835492e-01, 2.40057245e-01, \
    2.66429665e-03])

# evaluate loss and accuracy
print("Loss: "+str(nn.loss()))
print("Accuracy: "+str(nn.score()))
```

The major difference is that `batch_data_files` in `tati()` is now empty and instead we simply later assign `dataset` a numpy array to use. Note that we could also have supplied it directly with the filename `dataset-twoclusters.csv`, i.e. `nn.dataset = "dataset-twoclusters.csv"`. In this example we have parsed the same file as the in the previous section into a numpy array using the pandas module. Natually, this is just one way of creating a suitable numpy array.

Note
Input and output dimensions are directly deduced from the the tuple sizes.

Note
The nodes in the input layer can be modified using *input_columns*, e.g., `input_columns=["x1", "sin(x2)", "x1^2"]`.

2.2.7 Sampling the network

Optimization only steps down to the nearest local minimum from some initial random starting position. Only through sampling do we actually uncover the shape of the loss manifold and thereby are able to deduce whether our network is efficient at doing its job.

Nonetheless, optimization is always the initial step to sampling as we are still generally interested in all minimum regions.

Statistical background
In general, when sampling from a distribution (to compute empirical averages for example), one wants to start *close to equilibrium*, i.e. from states which are of high probability with respect to the target distribution (therefore the minima of the loss). The initial optimisation procedure is therefore a first guess to find such states, or at least to get close to them. In molecular dynamics, it is common to run sampling during an "equilibration period" in order to let the system relax to its equilibrium. During this equilibration time, the generated samples are not used for computing averages, as they will introduce large statistical error.

However, let us first ignore this good practice for a moment and simply look at sampling from a random initial place on the loss manifold. We will come back to it later on.

```
import TATi.simulation as tati

import numpy as np

nn = tati(
    batch_data_files=["dataset-twoclusters.csv"],
    batch_size=500,
    max_steps=1000,
    output_activation="linear",
    sampler="GeometricLangevinAlgorithm_2ndOrder",
    seed=426,
    step_width=1e-2
)
sampling_data = nn.sample()

print("Sample results")
print(np.asarray(sampling_data.run_info[0:10]))
print(np.asarray(sampling_data.trajectory[0:10]))
print(np.asarray(sampling_data.averages[0:10]))
```

Here, the *sampler* setting takes the place of the *optimizer* before as it states which sampling scheme to use. See Section 3.3 for a complete list and their parameter names. Apart from that the example code is very much the same as in the example involving `fit()`.

Note
In the context of sampling we use *step_width* in place of *learning_rate*.

Again, we produce a single data structure that contains three data frames: run info, trajectory, and averages. Trajectories contains among others all parameter degrees of freedom for each step (or *every_nth* step). Run info contains loss, accuracy, norm of gradient, norm of noise and others, again for each step. Finally, in averages we compute running averages over the trajectory such as average (ensemble) loss, average kinetic energy, average virial, see general concepts.

Take a peep at `sampling_data.run_info.columns` to see all columns in the run info dataframe (and similarly for the others.)

For the running averages it is advisable to skip some initial ateps (*burn_in_steps*) to allow for some burn in time, i.e. for kinetic energies to adjust from initially zero momenta.

Some columns in averages and in run info depend on whether the sampler provides the specific quantity, e.g. SGLD does not have momentum, hence there will be no average kinetic energy.

2.2.7.1 Using a prior

You may add a prior to the sampling. At the current state two kinds of priors are available: wall-repelling and tethering.

The options *prior_upper_boundary* and *prior_lower_boundary* give the admitted interval per parameter. Within a relative distance of 0.01 (with respect to length of domain and only in that small region next to the specified boundary) an additional force acts upon the particles to drives them back into the desired domain. Its magnitude increases with distance to the covered inside the boundary region. The distance is taken to the power of *prior_power*. The force is scaled by *prior_factor*.

In detail, the prior consists of an extra force added to the time integration within each sampler. We compute its magnitude as

$$\Theta(\frac{||x-\pi||}{\tau} - 1.) \cdot a||x-\pi||^n$$

where **x** is the position of the particle, **a** is the *prior_factor*, π is the position of the boundary (*prior_upper_boundary* π_{ub} or *prior_lower_boundary* π_{lb}), and **n** is the `prior_power`. Finally, the force is only in effect within a distance of $\tau = 0.01 \cdot ||\pi_{ub} - \pi_{lb}||$ to either boundary by virtue of the Heaviside function $\Theta()$. Note that the direction of the force is such that it always points back into the desired domain.

If upper and lower boundary coincide, then we have the case of tethering, where all parameters are pulled inward to the same point.

At the moment applying prior on just a subset of particles is not supported.

Note

The prior force is acting directly on the variables. It does not modify momentum. Moreover, it is a force! In other words, it depends on step width. If the step width is too large and if the repelling force increases too steeply close to the walls with respect to the normal dynamics of the system, it may blow up. On the other hand, if it is too weak, then particles may even escape.

2.2.7.2 First optimize, then sample

As we have already alluded to before, optimizing before sampling is the **recommended** procedure. In the following example, we concatenate the two. To this end, we might need to modify some of the options in between. Let us have a look, however with a slight twist.

The dataset shown in Figure Dataset can be even learned by a simpler network: only one of the input nodes is actually needed because of the symmetry.

Hence, we look at such a network by using *input_columns* to only use input column "x1" although the dataset contains both "x1" and "x2".

Moreover, we will add a hidden layer with a single node and thus obtain a network as depicted in Figure Permutation symmetry. We add this hidden node to make the loss manifold a little bit more interesting.

Additionally, we fix the biases to **0.** for both the hidden layer bias and the output bias. Effectively, we have two degrees of freedom left. This is not strictly necessary but allows to plot all degrees of freedom at once.

Finally, we add a prior.

```
import TATi.simulation as tati

import numpy as np

nn = tati(
    batch_data_files=["dataset-twoclusters.csv"],
    batch_size=500,
    every_nth=100,
    fix_parameters="layer1/biases/Variable:0=0.;output/biases/Variable:0=0.",
    hidden_dimension=[1],
    input_columns=["x1"],
    learning_rate=1e-2,
```

```
    max_steps=100,
    optimizer="GradientDescent",
    output_activation="linear",
    sampler = "BAOAB",
    prior_factor=2.,
    prior_lower_boundary=-2.,
    prior_power=2.,
    prior_upper_boundary=2.,
    seed=428,
    step_width=1e-2,
    trajectory_file="trajectory.csv",
)
training_data = nn.fit()

nn.set_options(
    friction_constant = 10.,
    inverse_temperature = .2,
    max_steps = 5000,
)

sampling_data = nn.sample()

print("Sample results")
print(np.asarray(sampling_data.run_info[0:10]))
print(np.asarray(sampling_data.trajectory[0:10]))
```

Note
Setting *every_nth* large enough is essential when playing around with small networks and datsets as otherwise time spent writing files and adding values to arrays will dominate the actual neural network computations by far.

As you see, some more options have popped up in the `__init__()` of the simulation interface: *hidden_dimension* which is a list of the number of hidden nodes per layer, *input_columns* which contains a list of strings, each giving the name of an input dimension (indexing starts at 1), and all sorts of *prior_...* that define a wall-repelling prior, again see for details. This will keep parameter values within the interval of [-2,2]. Last but not least, *trajectory_file* writes all parameters per *every_nth* step to this file.

Moreover, we needed to change the number of steps, set a sampling step width and add the sampler (which might depend on additional parameters, see Section 3.3). At the very end we again obtain the data structure containing the `pandas` DataFrame containing runtime information, trajectory, and averages as its member variables.

Warning
This time we need the trajectory file for the upcoming analysis. Hence, we write it to a file using the *trajectory_file* option. Keep the file around as it is needed in the following.

Let us take a look at the two degrees of freedom of the network, namely the two weights, where we plot one against the other similarly to the Sampled weights before.

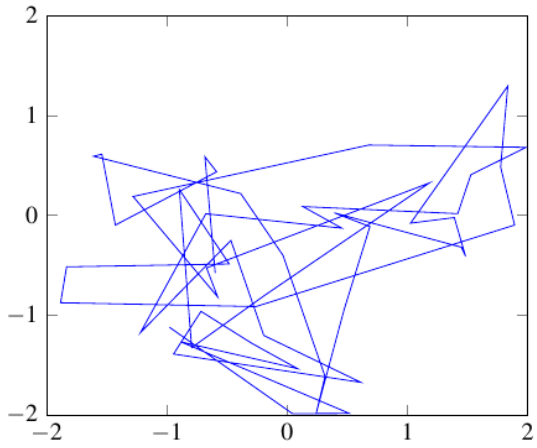


Figure 2.4: Sampled weights: Plot of first against second weight.

First of all, take note that the prior (given *prior_force* is strong enough with respect to the chosen *inverse_temperature*) indeed retains both parameters within the interval [-2,2] as requested.

Compare this to the Figure Loss manifold. You will notice that this trajectory (due to the large enough temperature) has also jumped over the ridge around the origin.

Note

To bound the runtime of this example, we have set the parameters such that we obtain a good example of a barrier-jumping trajectory. The original values from the introduction are obtained when you reduce the *inverse_temperature* to **4.** and increase *max_steps* to **20000** (or even more) if you do not mind waiting a minute or two for the sampling to execute.

2.2.8 Analysing trajectories

Analysis involves parsing in run and trajectory files that you would write through optimization and sampling runs. Naturally, you could also perform this on the `pandas` dataframes directly. However, for completeness we will read from files in the examples of this section.

To this end, specify `FLAGS.run_file` and `FLAGS.trajectory_file` with some valid file names.

2.2.8.1 Averages

Subsequently, these may be easily parsed as follows, see also `tools/TATiAnalyser.in` in the repository.

```
import pandas as pd
import numpy as np
import matplotlib
# use agg as backend to allow command-line use as well
matplotlib.use("agg")
import matplotlib.pyplot as plt

df_trajectory = pd.read_csv("trajectory.csv", sep=',', \
    header=0)
traj=np.asarray(df_trajectory)

conv=np.zeros(traj.shape)

# then we plot the running averages of the parameters
# inside weights
```

```
for i in range(1,traj.shape[0]):
    for d in range(traj.shape[1]):

        conv[i,d]=np.mean(traj[:i,d])

[plt.scatter(range(len(traj)), conv[:,i]) \
    for i in range(traj.shape[1])]
plt.savefig('step-parameters.png', bbox_inches='tight')
#plt.show()

print(conv[-1,:])
```

This would give a plot of the running average for each parameter in the trajectory file. In a similar, the run file can be loaded and its average quantities such as loss or kinetic energy be analysed and plotted.

2.2.8.2 Diffusion Map

Note
The former text was not agreed upon by the whole team and is therefore withdrawn at the moment.

2.2.9 Exploring the loss manifold

Exploration of the loss manifold is a bit more involved and hence uses a different part of the Python interface. We do not use the simulation interface anymore but the general Python interface as we require greater access to its internals.

In general, the procedure has the following stages:

1. We sample a few starting trajectory.
2. For the current set of trajectory points we perform a diffusion map analysis. Using the first eigenvector as the dominant diffusion mode, we pick the first corner points at its maximal component.
3. If more corner points are needed, then we look at the diffusion distance with respect to already picked corner points over all eigenvectors of the diffusion map and pick the next point always such that it maximizes the diffusion distance to the present ones.
4. Finally, we sample further trajectories, one starting at each of the picked corner points.
5. This is repeated (go to second step) for as many exploration steps as we want to do.

Note that each single trajectory is sampled in a special way:

- First, three legs of sampling are performed
- Then, we analyse the resulting diffusion map.
- If the eigenvalues have not yet converged with respect to some relative threshold, we continue for one more leg and analyse again after that
- If they have converged, we stop.
- Finally, we look at the norm of the gradients along the trajectory. If it is below a certain threshold, then within this section of the trajectory (with gradient norms beneath the threshold) we pick the smallest gradient value as the trajectory step being a possible minimum candidate.
- For all minimum candidates (if any) we run additional optimization trajectories, e.g. using GradientDescent, to find a local minima.

Have a look at the following example.

Warning
This example code is very much involved. Do not worry if you do not understand what is going on right away. This may become a lot simpler in future versions of TATi.

```
from TATi.models.model import model
from TATi.exploration.explorer import Explorer

import numpy as np

FLAGS = model.setup_parameters(
    batch_data_files=["dataset-twoclusters.csv"],
    batch_size=500,
    diffusion_map_method="vanilla",
    learning_rate=3e-2,
    max_steps=10,
    number_of_eigenvalues=1,
    optimizer="GradientDescent",
    output_activation="linear",
    sampler="BAOAB",
    seed=426,
    step_width=1e-2,
    use_reweighting=False
)
nn = model(FLAGS)
# init both sample and train right away
nn.init_network(None, setup="sample")
nn.init_network(None, setup="train")
nn.init_input_pipeline()
nn.reset_dataset()

explorer = Explorer(parameters=FLAGS, max_legs=5, number_pruning=0)

print("Creating starting trajectory.")
# a. add three legs to queue
explorer.spawn_starting_trajectory(nn)
# b. continue until queue has run dry
explorer.run_all_jobs(nn, FLAGS)

print("Starting multiple explorations from starting trajectory.")
# 2. with the initial trajectory done and analyzed,
#    find maximally separate points and sample from these
max_exploration_steps = 2
exploration_step = 1
while exploration_step < max_exploration_steps:
    # a. combine all trajectories
    steps, parameters, losses = \
        explorer.combine_sampled_trajectories()
    # b. perform diffusion map analysis for eigenvectors
    idx_corner = \
        explorer.get_corner_points(parameters, losses, \
                                   FLAGS, \
                                   number_of_corner_points=1)
    # d. spawn new trajectories from these points
    explorer.spawn_corner_trajectories(steps, parameters, losses,
                                       idx_corner, nn)
    # d. run all trajectories till terminated
    explorer.run_all_jobs(nn, FLAGS)
```

```
    exploration_step += 1


run_info, trajectory = explorer.get_run_info_and_trajectory()

print("Exploration results")
print(np.asarray(run_info[0:10]))
print(np.asarray(trajectory[0:10]))
```

This performs exactly the procedure described before using very, very short trajectories (*max_steps*), only a few legs (*max_legs*) and only a very limited number of exploration steps (*exploration_steps*). This is simple for the purpose of illustration. Naturally, larger values for all these parameters are required in order to explore complex manifolds and eventually find the global minimum.

Note the calls to `run_all_jobs()`: What is happening behind the scenes is that the class `Explorer` contains a queue. Sampling a single leg of a trajectory is encoded as a single job, similarly performing a diffusion map analysis and so on. All these jobs are placed in the queue. `run_all_jobs()` will launch the jobs one after the other, or even in parallel if *number_processes* is larger than **1**.

2.2.10 Conclusion

This has been the quickstart introduction to the `simulation` interface.

If you want to take this further, we recommend reading how to implement a GLA2 sampler using this module.

If you still want to take it further, then you need to look at the programmer's guide that should accompany your installation.

2.3 Using command-line interface

All the tests use the command-line interface and for performing rigorous scientific experiments, we recommend using this interface as well. Here, it is to do parameter studies and have extensive runs using different seeds.

2.3.1 Creating the dataset

As data is read from file, this file needs to be created beforehand.

For a certain set of simple classification problems, namely those that can be found in the tensorflow playground, we have added a TATiDatasetWriter` that spills out the dataset in CSV format.

```
TATiDatasetWriter \
        --data_type 2 \
        --dimension 500 \
        --noise 0.1 \
        --seed 426 \
        --train_test_ratio 0 \
        --test_data_file testset-twoclusters.csv
```

This will write 500 datums of the dataset type 2 ("two clusters") to a file "testset-twoclusters.csv" using all of the points as we have set the test/train ratio to 0. Note that we also perturb the points by 0.1 relative noise.

2.3.2 Parsing the dataset

Similarly, for testing the dataset can be parsed using the same tensorflow machinery as is done for sampling and optimizing, using

```
TATiDatasetParser \
    --batch_data_files dataset-twoclusters.csv \
    --batch_size 20 \
    --seed 426
```

where the *seed* is used for shuffling the dataset.

2.3.3 Optimizing the network

As weights (and biases) are usually uniformly random initialized and the potential may therefore start with large values, we first have to optimize the network, using (Stochastic) Gradient Descent (GD).

```
TATiOptimizer \
    --batch_data_files dataset-twoclusters.csv \
    --batch_size 50 \
    --loss mean_squared \
        --learning_rate 1e-2 \
    --max_steps 1000 \
    --optimizer GradientDescent \
    --run_file run.csv \
    --save_model `pwd`/model.ckpt.meta \
    --seed 426 \
    -v
```

This call will parse the dataset from the file "dataset-twoclusters.csv". It will then perform a (Stochastic) Gradient Descent optimization in batches of 50 (10% of the dataset) of the parameters of the network using a step width/learning rate of 0.01 and do this for 1000 steps after which it stops and writes the resulting neural network in a TensorFlow-specific format to a set of files, one of which is called `model.ckpt.meta` (and the other filenames are derived from this).

We have also created a file `run.csv` which contains among others the loss at each ("every_nth", respectively) step of the optimization run. Plotting the loss over the step column from the run file will result in a figure similar to in Loss history.

Note
Since Tensorflow 1.4 an absolute path is required for the storing the model. In the example we use the current directory returned by the unix command `pwd`.

If you need to compute the optimal step width, which is possible for smaller networks from the largest eigenvalue of the hessian matrix, then use the option "do_hessians 1" to activate it.

Note
The creation of the nodes is costly, $O(N^2)$ in the number of parameters of the network N. Hence, may not work for anything but small networks and should be done on purpose.

In case you have read the quickstart tutorial on the Python interface before, then the names of the command-line option will probably remind you of the variables in the FLAGS structure.

2.3.4 Sampling trajectories on the loss manifold

We continue from this optimized or equilibrated state with sampling. It is called equilibrated as the network's parameter should now be close to a (local) minimum of the potential function and hence in equilibrium. This means that small changes to the parameters will result in gradients that force it back into the minimum.

Let us call the sampler.

```
TATiSampler \
    --averages_file averages.csv \
    --batch_data_files dataset-twoclusters.csv \
    --batch_size 50 \
    --friction_constant 10 \
    --inverse_temperature 10 \
    --loss mean_squared \
    --max_steps 1000 \
    --sampler GeometricLangevinAlgorithm_2ndOrder \
    --run_file run.csv \
    --seed 426 \
    --step_width 1e-2 \
    --trajectory_file trajectory.csv
```

This will cause the sampler to parse the same dataset as before. Moreover, the sampler will load the neural network from the model, i.e. using the optimized parameters right from the start. Afterwards it will use the GLA in 2nd order discetization using again *step_width* of 0.01 and running for 1000 steps in total. The GLA is a descretized variant of Langevin Dynamics whose accuracy scales with the inverse square of the *step_width* (hence, 2nd order).

The seed is needed as we sample using Langevin Dynamics where a noise term is present. The term basically ascertains a specific temperature which is proportional to the average momentum of each particle.

After it has finished, it will create three files; a run file `run.csv` containing run time information such as the step, the potential, kinetic and total energy at each step, a trajectory file `trajectory.csv` with each parameter of the neural network at each step, and an averages file `averages.csv` containing averages accumulated along the trajectory such as average kinetic energy, average virial (connected to the kinetic energy through the virial theorem, valid if a prior keeps parameters bound to finite values), and the average (ensemble) loss. Moreover, for the HMC sampler the average rejection rate is stored there. The first two files we need in the next stage.

2.3.5 Analysing trajectories

Eventually, we now perform the diffusion map analysis on the obtained trajectories. The trajectory file written in the last step is simply a matrix of dimension (number of parameters) times (number of trajectory steps). The eigenvector to the largest (but one) eigenvalue will give the dominant direction in which the trajectory is moving.

Note
The largest eigenvalue is usually unity and its eigenvector is constant. Therefore, it is omitted. That's why indexing for the diffusion maps eigenvectors starts at 1 (omitted the constant eigenvector 0).

The analysis can perform three different tasks:

- Calculating averages.
- Calculating the diffusion map's largest eigenvalues and eigenvectors.
- Calculating landmarks and level sets to obtain an approximation to the free energy.

2.3.5.1 Averages

Averages are calculated by specifying two options as follows:

```
TATiAnalyser \
    --average_run_file average_run.csv \
    --average_trajectory_file average_trajectory.csv \
    --drop_burnin 100 \
    --every_nth 10 \
    --run_file run.csv \
    --steps 10 \
    --trajectory_file trajectory.csv
```

This will load both the run file `run.csv` and the trajectory file `trajectory.csv`and average over them using only every 10th data point (*every_nth*) and also dropping the first steps below 100 (*drop_burnin*). It will produce then ten averages (*steps*) for each of energies in the run file and each of the parameters in the trajectories file (along with the variance) from the first non-dropped step till one of the ten end steps. These end steps are obtained by equidistantly splitting up the whole step interval.

Eventually, we have two output file. The averages over the run information such as total, kinetic, and potential energy in `average_run.csv`. Also, we have the averages over the degrees of freedom in `average_trajectories.csv`.

Note
Averages depend crucially on the number of steps we average over. I.e. the more points we throw away, the less accurate it becomes. In other words, if large accuracy is required, the averages file (if it contains the value of interest) is a better place to look for.

2.3.5.2 Diffusion map

The eigenvalues and eigenvectors can be written as well to two output files.

```
TATiAnalyser \
    --diffusion_map_file diffusion_map_values.csv \
    --diffusion_map_method vanilla \
    --diffusion_matrix_file diffusion_map_vectors.csv \
    --drop_burnin 100 \
    --every_nth 10 \
    --inverse_temperatur 1e4 \
    --number_of_eigenvalues 4 \
    --steps 10 \
    --trajectory_file trajectory.csv
```

The files ending in `..values.csv` contains the eigenvalues in two columns, the first is the eigenvalue index, the second is the eigenvalue.

The other file ending in `..vectors.csv` is simply a matrix of the eigenvector components in one direction and the trajectory steps in the other. Additionally, it contains the parameters at the steps and also the loss and the kernel matrix entry.

Note that again the all values up till step 100 are dropped and only every 10th trajectory point is considered afterwards.

There are two methods available. Here, we have used the simpler (and less accurate) (plain old) vanilla method. The other is called TMDMap.

If you have installed the *pydiffmap* python package, this mal also be specified as diffusion map method. It has the benefit of an interal optimal parameter choice. Hence, it should behave more robustly than the other two methods. TMDMap is different only in reweighting tre samples according to the specific temperature.

2.3.5.3 Free energy

Last but not least, the free energy is calculated.

```
TATiAnalyser \
    --diffusion_map_method TMDMap \
    --drop_burnin 100 \
    --every_nth 10 \
    --inverse_temperature 10 \
    --landmarks 5 \
    --landmark_file landmarks-ev_1.csv \
    --number_of_eigenvalues 2 \
    --steps 10 \
    --trajectory_file trajectory.csv
```

This will extract landmark points from the trajectory. Basically, the loss manifold is discretized using these landmarks where all configurations close to a landmark step are combined onto a so-called level-set, i.e. all these configurations have a similar loss function value. By knowing the number of configurations in each level set and knowing the level sets loss value, an approximation of the free energy is computed.

This is computed for every step of the trajectory and it is insightful to look at the free energy over the course of the trajectory represented by the first eigenvalue. If in this graph clear minima with maxima in between can be seen, then there are enthalpic barriers between two local minima. If on the other hand there are flat areas, then we found entropic barriers.

Both these types of barriers obstruct trajectories and keep the optimization trapped in so-called meta-stable states. Each type of barrier requires a different type of remedy to overcome.

2.3.5.4 Exploring the loss manifold

Eventually, we are not interested in obtaining trajectories on the loss manifold. Instead we would like to find the global minima. Or at least have a good idea about whether the minimas we have found so far are reasonable.

To this end, a command-line tool called `TATiExplorer` is provided. The idea is to make use of the diffusion map with its diffusion distance to assess what part of the loss manifold has been explored already. Moreover, we use multiple trajectories that are spawned from a specific number of places that are maximally separate with respect to their diffusion distance. This will ensure that we cover the most ground possible.

In the end, the eigenvectors obtained through a run using the `TATiExplorer` will return the dominant diffusion directions and therefore those pointing in the direction along the minima, i.e. where the sampling usually gets stuck and remains for a while, hence diffusion is slow.

```
TATiExplorer \
        --batch_data_files dataset-twoclusters.csv \
    --batch_size 50 \
    --diffusion_map_method vanilla \
    --friction_constant 10 \
    --inverse_temperature 10 \
        --learning_rate 3e-2 \
    --loss mean_squared \
    --max_exploration_steps 2 \
    --max_legs 10 \
    --max_steps 10 \
    --number_of_eigenvalues 1 \
    --number_of_parallel_trajectories 1 \
    --number_pruning 0 \
    --optimizer GradientDescent \
    --sampler GeometricLangevinAlgorithm_2ndOrder \
    --run_file run.csv \
    --seed 426 \
    --step_width 1e-2 \
    --trajectory_file trajectory.csv \
    --use_reweighting 0
```

In the example we call the explorer utility in much the same way as we have called the sampler. There are some additional options that give the number of eigenvalues to calculate and which diffusion map method to use. Note that `max_steps` now gives the number of steps of a single leg. Further down you find what a lag actually is.

Furthermore, there are two options unique to the explorer. This is `max_legs` which gives the maximum number of legs to look at. Each leg goes over max_steps. After that a diffusion map analysis is performed that checks whether the eigenvalues have converged already. If yes, the trajectory is ended, if not we continue with a new leg (of max_steps steps). If no convergence should occur, max_legs gives the maximum number of legs after which the trajectory is terminated regardlessly.

Finally, we run multiple trajectories in parallel from starting points that are maximally apart from each other in the sense of the diffusion distances. This is controlled by `number_of_parallel_trajectories`.

2.4 A note on parallelization

Internally, Tensorflow uses a computational graph to represent all operations. Nodes in the graph represent computations and their results and edges represent dependencies between these values, i.e. some may act as input to operations resulting in certain output.

Because of this internal representation Tensorflow has two kind of parallelisms:

- inter ops
- intra ops

Each is connected to its its own thread pool. Both the command-line and the Python interface let you pick the number of threads per pool. If 0 is stated (default), then the number of threads is picked automatically.

In general, “inter_ops_threads”refers to multiple cores performing matrix multiplication or reduction operations together. ``intra_ops_threads” seems to be connected to executing multiple nodes in parallel that are independent of each other but this is guessing at the moment.

Warning
When setting `inter_ops_threads` *unequal* to 1, then subsequent runs may produce different results, i.e. results are no longer strictly reproducible. According to Tensorflow this is because reduction operations such as `reduce_sum` run non-deterministically on multiple cores for sake of speed.

2.5 Conclusion

This has been the very quick introduction into samping done on neural network’s loss function manifolds. You have to take it from here.

Chapter 3

The reference

3.1 General concepts

Before we dive into the internals of this program suite, let us first introduce some general underlying concepts assuming that the reader is only roughly familiar with them. This is not meant as a replacement for the study of more in-depth material but should rather be seen as a reminder of the terms and notation that will appear later on.

- Dataset

  The dataset contains a fixed number of datums of input tuples and output tuples. They are typically referred to as *features* and *labels* in the machine learning community. Basically, they are samples taken from the unknown function which we wish to approximate using the neural network. If the output tuples are binary in each component, the approximation problem is called a *classification* problem. Otherwise, it is a *regression* problem.

- Neural network

  The neural network is a black-box representing a certain set of general functions that are efficient in solving classification problems (among others). They are parametrized explicitly using weights and biases and implicitly through the topology of the network (connections of nodes residing in layers) and the activation functions used. Moreover, the loss function determines the best set of parameters for a given task.

- Loss

  The default is *mean_squared*.

The loss function determines for a given (labeled) dataset what set of neural network's parameters are best. Note that there are losses that do not require labels though. Different losses result in different set of parameters. It is a high-dimensional manifold that we want to learn and capture using the neural network. It implicitly depends on the given dataset and explicitly on the parameters of the neural network, namely weights and biases. Dual to the loss function is the network's output that explicitly depends on the dataset's current datum (fed into the network) and implicitly on the parameters.

+ Most important to understand about the loss is that it is a *non-convex* function and therefore in general does not just have a single minimum. This makes the task of finding a good set of parameters that (globally) minimize the loss difficult as one would have to find each and every minima in this high-dimensional manifold and check whether it is actually the global one.

- Momenta and kinetic energy

  Momenta is a concept taken over from physics where the parameters are considered as particles each in a one-dimensional space where the loss is a potential function whose (negative) gradient acts as a force onto the particle driving them down-hill (towards the local minimum). This force is integrated in a classical Newton's mechanic style, i.e. Newton's equation of motion is discretized with small time steps (similar to the learning rate in Gradient Descent). This gives first rise to/velocity and second to momenta, i.e. second order ordinary differential equation (ODE) split up into a system of two one-dimensional ODEs. There are numerous stable time integrators, i.e. velocity Verlet/leapfrog, that are employed to propagate both particle position (i.e. the parameter value) and its momentum through time. Note that momentum and velocity are actually equivalent as usually the mass is set to unity.

  The kinetic energy is computed as sum over kinetic energies of each parameter.

- Virials

  Virials are defined as one half of the sum over the scalar product of gradients with parameters, see https://en.wikipedia.org/-wiki/Virial_theorem.

- Optimizers

  Optimizers are used to drive the parameters to the local minimum from a given (random) starting position. [GD] is best known, but there are more elaborate Optimizers that use the concept of momentum as well. This helps in overcoming flat parts of the manifold where the gradient is effectively zero but momentum still drives the particles towards the minimum.

- Samplers

  The goal of samplers is different than the goal of optimizers. Samplers such as [GLA] aim at discovering a great deal of the manifold, not constraint to the local minimum. Usually, they are started from the local minimum and drive the particles further and further out until new minima are found between which potential barriers had to be overcome.

3.1.1 Neural Networks

A neural network (NN) is a tool used in the context of machine learning. Formally, it is a graph with nodes and edges, where nodes represent (simple) functions. The edges represent scalar values by which the output of one node is scaled as input to another node. The scalar value is called *weight* and each node also has a constant value, the *bias*, that does not depend on the input of other nodes. Nodes are organized in layers and nodes are (mostly) only connected between adjacent layer. Special are the very first layer with input nodes that simply accept input from the user and the very last layer whose output is eventually all that matters.

Typically, a NN might be used for the task of classification: Data is fed into the network's input layer and its output layer has nodes equal to the number of classes to be distinguished. This can for example be used for image classification.

The essential task at hand is to determine a good set of parameters, i.e. values for the weights and biases, such that the task is performed best with respect to some measure.

3.1.2 The loss function

At the moment, there are two little utility programs that help in evaluating the loss function given a certain dataset, namely the `TATiLossFunctionSampler`. Let us give an example call right away.

```
TATiLossFunctionSampler \
        --batch_data_files dataset-twoclusters.csv \
        --batch_size 20 \
        --csv_file TATiLossFunctionSampler-output-SGLD.csv \
        --parse_parameters_file trajectory.csv
```

It takes as input the dataset file `dataset-twoclusters.csv` and either a parameter file `trajectory.csv`. This will cause the program the re-evaluate the loss function at the trajectory points which should hopefully give the same values as already stored in the trajectory file itself.

However, this may be used with a different dataset file, e.g. the testing or validation dataset, in order to evaluate the generalization error in terms of the overall accuracy or the loss at the points along the given trajectory.

Interesting is also the second case, where instead of giving a parameters file, we sample the parameter space equidistantly as follows:

```
TATiLossFunctionSampler \
    --batch_data_files dataset-twoclusters.csv \
    --batch_size 20 \
    --csv_file TATiLossFunctionSampler-output-SGLD.csv \
    --interval_weights -5 5 \
    --interval_biases -1 1 \
    --samples_weights 10 \
    --samples_biases 4
```

Here, sample for each weight in the interval [-5,5] at 11 points (10 endpoint), and similarly for the weights in the interval [-1,1] at 5 points.

Note
For anything but trivial networks the computational cost quickly becomes prohibitively large. However, you may use `fix_parameter` to lower the computational cost by choosing a certain subsets of weights and biases to sample.

```
TATiLossFunctionSampler \
  --batch_data_files dataset-twoclusters.csv \
  --batch_size 20 \
  --csv_file TATiLossFunctionSampler-output-SGLD.csv \
  --fix_parameters "output/weights/Variable:0=2.,2." \
  --interval_weights -5 5 \
  --interval_biases -1 1 \
  --samples_weights 10 \
  --samples_biases 4
```

Moreover, using `exclude_parameters` can be used to exclude parameters from the variation, i.e. this subset is kept at fixed values read from the file given by `parse_parameters_file` where the row designated by the value in `parse_steps` is taken.

This can be used to assess the shape of the loss manifold around a found minimum.

```
TATiLossFunctionSampler \
  --batch_data_files dataset-twoclusters.csv \
  --batch_size 20 \
  --csv_file TATiLossFunctionSampler-output-SGLD.csv \
  --exclude_parameters "w1" \
  --interval_center_step 1 \
  --interval_weights -5 5 \
  --interval_biases -1 1 \
  --parse_parameters_file centers.csv \
  --parse_steps 1 \
  --samples_weights 10 \
  --samples_biases 4
```

Here, we have excluded the second weight, named **w1**, from the sampling. Note that all weight and all bias degrees of freedom are simply enumerated one after the other when going from the input layer till the output layer.

Furthermore, we have specified a file containing center points for all excluded parameters. This file is of CSV style having a column **step** to identify which row is to be used and moreover a column for every (excluded) parameter that is fixed at a value unequal to 0. Note that the minima file written by `TATiExplorer` can be used as this centers file. Moreover, also the trajectory files have the same structure.

3.1.3 The learned function

The second little utility programs does not evaluate the loss function itself but the unknown function learned by the neural network depending on the loss function, called the `TATiInputSpaceSampler`. In other words, it gives the classification result for data point sampled from an equidistant grid. Let us give an example call right away.

```
TATiInputSpaceSampler \
        --batch_data_files grid.csv \
        --csv_file TATiInputSpaceSampler-output.csv \
        --input_dimension 2 \
        --interval_input -4 4 \
        --parse_steps 1 \
        --parse_parameters_file trajectory.csv \
        --samples_input 10 \
        --seed 426
```

Here, `batch_data_files` is an input file but it does not need to be present. (Sorry about that abuse of the parameter as usually `batch_data_files` is read-only. Here, it is overwritten!). Namely, it is generated by the utility in that it equidistantly samples the input space, using the interval [-4,4] for each input dimension and 10+1 samples (points on -4 and 4 included). The parameters file `trajectory.csv` now contains the values of the parameters (weights and biases) to use on which the learned function depends or by, in other words, by which it is parametrized. As the trajectory contains a whole flock of these, the `parse_steps` parameter tells it which steps to use for evaluating each point on the equidistant input space grid, simply referring to rows in said file.

Note
For anything but trivial input spaces the computational cost quickly becomes prohibitively large. But again `fix_parameters` is heeded and can be used to fix certain parameters. This is even necessary if parsing a trajectory that was created using some parameters fixed as they then will *not* appear in the set of parameters written to file. This will raise an error as the file will contain too few values.

3.2 Examples

We show how to set up some basic models, which can be a useful tool to study and test methods.

3.2.1 Harmonic oscillator

It can be sometimes useful to use TATi to simulate a simple harmonic oscillator. This model can be obtained by training the neural network with one parameter on one point $X = 1$ in dimension one with a label equal to zero, i.e. $Y = 0$, and using the mean square loss function and linear activation function. More precisely, the cost function becomes in this setting:

$L(\omega|X,Y) = |\omega X + b - Y|^2$,

where we fix the bias $b = 0$.

- Dataset:

```
from TATi.common import data_numpy_to_csv
import numpy as np

X = np.asarray([[1]])
Y = np.asarray([[0]])

# prepare and save the trivial data set for later
datasetName = 'dataset_ho.csv'
data_numpy_to_csv(X, Y, datasetName)
numberOfPoints = 1
```

- Setup and train neural network:

```
import TATi.simulation as tati

import numpy as np
import pandas as pd

nn = tati(
    batch_data_files=["dataset_ho.csv"],
    batch_size=1,
    fix_parameters="output/biases/Variable:0=0.",
    friction_constant=10.0,
    input_dimension=1,
    inverse_temperature=1.0,
    loss="mean_squared",
    max_steps=1000,
```

```
    output_activation = "linear",
    output_dimension=1,
    run_file="run_ho.csv",
    sampler="BAOAB",
    seed=426,
    step_width=0.5,
    trajectory_file="trajectory_ho.csv",
    verbose=1
)

data = nn.sample()

df_trajectories = pd.DataFrame(data.trajectory)
# sampled trajectory
weight0 = np.asarray(df_trajectories['weight0'])
```

- Sampled trajectory:

  The output trajectory in `trajectory_ho.csv` or `weight0` is distributed w.r.t. a Gaussian, i.e. the density of X:=weight0 is $exp(-X^2)$, see Figure Gaussian distribution.

Note
The figure was obtained with setting `max_steps` to 10000.

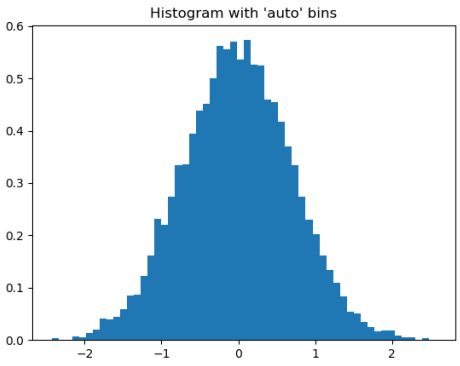


Figure 3.1: Gaussian distribution: Histogram of the trajectories obtained by simulating 1D harmonic oscillator with BAOAB sampler.

3.3 Samplers

Samplers are discretizations of a given dynamical system, set by an ODE or SDE. In our case, it is mostly Langevin dynamics. Given that the sampler is ergodic we can replace integrals over the whole domain by integrals along sufficiently long trajectories.

Certain asymptotical properties are inherently connected to the respective dynamics, for example, the average kinetic energy. It can evaluated as average by integrating along the trajecories. However, as it is not possible to integrate the continuous dynamics directly, we rely on the respective discretizen, the sampler, which naturally introduces an error.

This discretization error depends on the chosen *finite step width* by which the trajectories are produced. It shows as a finite error to the asymptotical value regardless of the length of the trajectory. Choosing a smaller step width will produce an error.

This can be observed for the average kinetic energy by looking at a small example network and dataset and producing sufficiently long trajectories. If one plots the difference against the known asymptotical value (namely $\frac{N^{\text{dof}}}{2} k_B T$ with temperature **T**, and

degrees of freedom N^{dof}) over different step widths in double logarithmic fashion, one obtains straight lines whose slope depends on the discretization order.

Different samplers have different discretization orders and the order also depends on the observed quantity.

This makes picking the right sampler a critical choice: From a statistical point of view the most accurate sampler is best, i.e. the one having the highest discretization order. However, as all of them have roughly the same computational cost, it is generally recommended to pick BAOAB which has second order convergence and even fourth order in the high-friction limit, see [Leimkuhler2012] for details.

Note
At the beginning of each the following subsections we give the name of the respective sampler in order to activate it using the **sampler** keyword, see Section 2.2.7 and Section 2.3.4.

3.3.1 Stochastic Gradient Langevin Dynamics

sampler: `StochasticGradientLangevinDynamics`

The Stochastic Gradient Langevin Dynamics (SGLD) was proposed by [Welling2011] based on the [SGD][Stochastic Gradient Descent], which is a variant of the [GD][Gradient Descent] using only a subset of the dataset for computing gradients. The central idea behind SGLD was to add an additional noise term whose magnitude then controls the noise induced by the approximate gradients.

Implements a Stochastic Gradient Langevin Dynamics Sampler in the form of a TensorFlow Optimizer, overriding tensorflow.python.training.Optimizer.

The step update is: $\theta^{n+1} = \theta^n - \beta\nabla U(\theta)\Delta t + \sqrt{\Delta t}G^n$, where β is the inverse temperature coefficient, Δt is the (discretization) step width and θ is the parameter vector and $U(\theta)$ the energy or loss function.

Option name	Description
`step_width`	time integration step width δt

Table 3.1: Table of parameters for SGLD

Note
SGLD is very much like **SGD** and **GD** in terms that the `step_width` needs to be small enough with respect to the gradient sizes of your problem.

3.3.2 Covariance Controlled Adaptive Langevin

sampler: `CovarianceControlledAdaptiveLangevin`

This is an extension of Stochastic Gradient Descent proposed by [Shang2015]. The key idea is to dissipate the extra heat caused by the approximate gradients through a suitable thermostat. However, the discretisation used here is not based on the (first-order) Euler-Maruyama as [SGLD] but on [GLA] 2nd order.

Note
`sigma` and `sigmaA` are two additional parameters that control the action of the thermostat. Moreover, we require the same parameters as for [GLA] 2nd order.

Option name	Description
`friction_constant`	friction constant γ that controls how much momentum is replaced by random noise each step
`inverse_temperature`	inverse temperature factor scaling the noise, β
`sigma`	controlling the thermostat
`sigmaA`	controlling the thermostat's adaptivity
`step_width`	time integration step width δt

Table 3.2: Table of parameters for CCAdL

3.3.3 Geometric Langevin Algorithms

sampler: `Geometric Langevin Algorithms_1stOrder`, `Geometric Langevin Algorithms_2ndOrder`

GLA results from a first-order splitting between the Hamiltonian and the Ornstein-Uhlenbeck parts, see [Leimkuhler2015][section 2.2.3, Leimkuhler 2015] and also [Leimkuhler2012]. If the Hamiltonian part is discretized with a scheme of second order as in **GLA2**, it provides second order accuracy at basically no extra cost.

The update step of the parameters for second order GLA is BABO:

$B: p^{n+1/2} = p^n - \nabla U(q^n)\frac{\Delta t}{2}$

$A: q^{n+1} = q^n + M^{-1} p^{n+1/2}\Delta t$

$B: \tilde{p}^{n+1} = p^{n+1/2} - \nabla U(q^{n+1})\frac{\Delta t}{2}$

$O: p^{n+1} = \alpha_{\Delta t}\tilde{p}^{n+1} + \sqrt{\frac{1-\alpha_{\Delta t}^2}{\beta}M}G^n$ where $\alpha_{\Delta t} = exp(-\gamma\Delta t)$ and $G \sim N(0,1)$.

The first order GLA is BAO.

Option name	Description
`friction_constant`	friction constant γ that controls how much momentum is replaced by random noise each step
`inverse_temperature`	inverse temperature factor scaling the noise, β
`step_width`	time integration step width δt

Table 3.3: Table of parameters for GLA1 and GLA2

Note

All GLA samplers have two more parameters: `inverse_temperature` (usually denoted as β) and `friction_constant` (usually denoted as γ). Inverse temperature controls the average momentum of each parameter while the friction constant decides over how much of the momentum is replaced by random noise, i.e. the random walker character of the trajectory.

Good values for beta depend on the loss manifold and its barriers and need to be find by try&error at the moment.

As a rough guide, $\gamma = 10$ is a good start for the friction constant. Moreover, when choosing such a friction constant, with $\beta = 1000$ sampling will remain in the starting minimum basin, while $\beta = 1$ exists basins very soon. Note that large noise can be hidden by a too small friction constant, i.e. they both depend on each other.

3.3.4 BAOAB

sampler: `BAOAB`

BAOAB derives from the basic building blocks A (position update), B (momentum update), and O (noise update) into which the Langevin system is split up. Each step is solved in a separate step. Hence, we perform a B step, then an A step, ... and so on. This scheme has second-order accuracy and superb overall accuracy with respect to positions. See [Leimkuhler2012] for more details.

The update step of the parameters is:

$B: p^{n+1/2} = p^n - \nabla U(q^n)\frac{\Delta t}{2}$

$A: q^{n+1} = q^n + M^{-1} p^{n+1/2}\frac{\Delta t}{2}$

$O: p^{n+1} = \alpha_{\Delta t}\tilde{p}^{n+1} + \sqrt{\frac{1-\alpha_{\Delta t}^2}{\beta}} M G^n$

$A: q^{n+1} = q^n + M^{-1} p^{n+1/2}\frac{\Delta t}{2}$

$B: p^{n+1/2} = p^n - \nabla U(q^n)\frac{\Delta t}{2}$

where $\alpha_{\Delta t} = exp(-\gamma\Delta t)$ and $G \sim N(0,1)$.

Option name	Description
`friction_constant`	friction constant γ that controls how much momentum is replaced by random noise each step
`inverse_temperature`	inverse temperature factor scaling the noise, β
`step_width`	time integration step width δt

Table 3.4: Table of parameters for BAOAB

3.3.5 Hamiltonian Monte Carlo

sampler: `HamiltonianMonteCarlo`

HMC is based on Hamiltonian dynamics instead of Langevin Dynamics. Noise only enters when, after the evaluation of an acceptance criterion, the momenta are redrawn randomly. It has first been proposed by [Duane1987]. The Metropolisation ensures that the sampled distribution is unbiased. One virtue of HMC is that when using longer Verlet time integration legs (`hamiltonian_dynamics_time` larger than 1) that the walker will progress much further than in the case of Langevin Dynamics. However, this comes at the price of extra computational work in terms of rejected legs. See also [Neal2011] for a very readable introduction.

Important
The exact amount of Hamiltonian dynamics time integration steps is slightly varied (uniformly randomly scaled in [0.9,1.1]) such that not all walkers evaluate at the same step and are correlated thereby.

NOTE:

3.3.6 Ensemble of Walkers

Ensemble of Walkers uses a collection of walkers that exchange gradient and parameter information in each step in order to calculate a preconditioning matrix. This preconditioning allows to explore elongated minimum basins faster than independent walkers would do alone, see [Matthews2018].

This is activated by setting the `number_walkers` to a value larger than 1. Note that `covariance_blending` controls the magnitude of the covariance matrix approximation and `collapse_after_steps` controls after how many steps the walkers are restarted at the parameter configuration of the first walker to ensure that the harmonic approximation still holds.

This works for all of the aforementioned samplers as simply the gradient of each walker is rescaled.

Option name	Description
`hamiltonian_dynamics_time`	Time used for integrating Hamiltonian dynamics. This is relative to `step_width`. For example, if `step_width` is **0.05** and this chose as **0.05** as well, it will evaluate every other step, i.e. perform only a single time integration step before evaluating the acceptance criterion. If you want to evaluate every **n** steps, then use $n \cdot \Delta t$
`inverse_temperature`	inverse temperature factor scaling the initially randomly drawn momenta, β
`step_width`	time integration step width δt

Table 3.5: Table of parameters for HMC

3.4 Simulation module: Implementing a sampler

We would like to demonstrate how to implement the [GLA] sampler of 2nd order using the simulation module.

Let us look at the four integration steps.

1. $p_{n+\frac{1}{2}} = p_n - \frac{\lambda}{2}\nabla_x L(x_n)$
2. $x_{n+1} = x_n + \lambda p_{n+\frac{1}{2}}$
3. $\widehat{p}_{n+1} = p_{n+\frac{1}{2}} - \frac{\lambda}{2}\nabla_x L(x_{n+1})$
4. $p_{n+1} = \alpha \widehat{p}_{n+1} + \sqrt{\frac{1-\alpha^2}{\beta}} \cdot \eta_n$

If you are familar with the **A** (position integration), **B** (momentum integration), **O** (noise integration) notation, see [Leimkuhler2012] then you will notice that we have the steps: **BABO**.

3.4.1 Simple update implementation

Therefore, let us write a python function that works on several `numpy` arrays producing a single GLA2 update step by performing the four integration steps above. To this end, we make use `tati.gradients()` for computing the gradients $\nabla_x L(x_{n+1})$.

A first GLA2 update implementation

```
def gla2_update_step(nn, momenta, step_width, beta, gamma): # ❶
  # 1. p_{n+\tfrac 1 2} = p_n - \tfrac {\lambda}{2} \nabla_x L(x_n)
  momenta -= .5*step_width * nn.gradients() # ❷

  # 2. x_{n+1} = x_n + \lambda p_{n+\tfrac 1 2}
  nn.parameters = nn.parameters + step_width * momenta  # ❸

  # 3. \widehat{p}_{n+1} = p_{n+\tfrac 1 2} - \tfrac {\lambda}{2} \nabla_x L(x_{n+1})
  momenta -= .5*step_width * nn.gradients()  # ❹

  # 4. p_{n+1} = \alpha \widehat{p}_{n+1} + \sqrt{\frac{1-\alpha^2}{\beta}} \cdot \eta_n
  alpha = math.exp(-gamma*step_width)
  momenta = alpha * momenta + \
```

```
            math.sqrt((1.-math.pow(alpha,2.))/beta) * np.random.standard_normal(momenta. ↩
                shape) # ❺

    return momenta
```

❶ In the function header we need access to the `tati` reference, to the `numpy` array containing the `momenta`. Moreover, we need a few parameters, namely the step width `step_width`, the inverse temperature factor `beta` and the friction constant `gamma`.

❷ First, we perform the **B** step integrating the momenta.

❸ Next comes the **A** step, integrating positions with the updated momenta.

❹ Then, we integrate momenta again, **B**.

❺ Last, we perform the noise integration **O**. First, we compute the value of α_n and the the momenta are partially reset by the noise.

As the source of noise we have simply used `numpy`s standard normal distribution.

Tip
It is advisable to fix the seed using `numpy.random.seed(426)` (or any other value) to allow for reproducible runs.

We could have used `nn.momenta` for storing momenta. However, this needs some extra computations for assigning the momenta inside the tensorflow computational graph. As they are not needed in the graph anyway, we can store them outside directly.

Note that we have been a bit wasteful in the above implementation but very close to the formulas in Section 3.4.

3.4.2 Saving a gradient evaluation

We evaluate the gradient twice but actually only one evaluation would have been needed: The update gradients in step 3. are the same as the gradients in step 1. on the next iteration.

Hence, let us refine the function with respect to this.

GLa2 update implementation with just one gradient evaluation

```
def gla2_update_step(nn, momenta, old_gradients, step_width, beta, gamma):
    # 1. p_{n+\tfrac 1 2} = p_n - \tfrac {\lambda}{2} \nabla_x L(x_n)
    momenta -= .5*step_width * old_gradients

    # 2. x_{n+1} = x_n + \lambda p_{n+\tfrac 1 2}
    nn.parameters = nn.parameters + step_width * momenta

    # \nabla_x L(x_{n+1})
    gradients = nn.gradients()

    # 3. \widehat{p}_{n+1} = p_{n+\tfrac 1 2} - \tfrac {\lambda}{2} \nabla_x L(x_{n+1})
    momenta -= .5*step_width * gradients

    # 4. p_{n+1} = \alpha \widehat{p}_{n+1} + \sqrt{\frac{1-\alpha^2}{\beta}} \cdot \eta_n
    alpha = math.exp(-gamma*step_width)
    momenta = alpha * momenta + \
              math.sqrt((1.-math.pow(alpha,2.))/beta) * np.random.standard_normal(momenta. ↩
                  shape)

    return gradients, momenta
```

Now, we use `old_gradients` in step 1. and return the updated gradients such that it can be given as old gradients in the next call.

3.4.3 The loop

Now, we ad the loop body.

```
import math
import numpy as np
import TATi.simulation as tati

np.random.seed(426)

nn = tati( # ❶
    batch_data_files=["dataset-twoclusters.csv"],
)

momenta = np.zeros((nn.num_parameters())) # ❷
old_gradients = np.zeros((nn.num_parameters())) # ❸

for i in range(100):  # ❹
    print("Current step #"+str(i)) # ❺
    old_gradients, momenta = gla2_update_step(
        nn, momenta, old_gradients, step_width=1e-2, beta=1e3, gamma=10) # ❻
    print(nn.loss()) # ❼
```

❶ We instantiate a `tati` instance as usual, giving it the dataset and using its default single-layer perceptron topology.

❷, ❸ We create two numpy arrays to contain the momenta and the old gradients.

❹, ❺ We iterate for 100 steps, printing the current step.

❻ We use the `gla2_update_step()` function written to perform a single update step. We store the returned gradients and momenta.

❼ Finally, we print the loss per step.

3.5 A Note on Reproducibility

In many of the examples in the quickstart tutorials we have set a *seed* value to enforce reproducible runs.

We have gone through great lengths to make sure that runs using the same set of options yield the same output on every evocation.

Tensorflow is not fully reproducible per se. Its internal random number seeds change when the computational graph changes. Its reduction operations are non-deterministic. The latter can be overcome by setting *inter_ops_threads* to *1*, which take away some of the parallelization for the sake of reproducibility. The former is taken care of by TATi itself. We make sure to set the random number seeds deterministically to ensure that values are unchanged even if the graph is slighlty changed.

If you find that this should not be the case, please file an issue, see Section 1.6.

3.6 A Note on Performance

Performance is everything in the world of neural network training. Codes and machines are measured by how fast they perform in images/second when training AlexNet or other networks on the ImageNet dataset, see Tensorflow Benchmarks.

We worked hard to ensure that whatever Tensorflow offers in performance is also seen when using TATi. In order to guide the user in what to expect and what to do when these expectations are not met, we invite to go through this section.

In general, performance hinges **critically** on the input pipeline. In other words, it depends very much on how fast a specific machine setup can feed the dataset into the input layer of the neural network.

Note
In our examples both datasets and networks are very small. This causes the sequential parts of tensorflow to overwhelm any kind of parallel execution.

Typically, these datasets are stored as a set of files residing on disk. Note that reading from disk is very slow compared to reading from memory. Hence, the first step is to read the dataset from disk and this will completely dominate the computational load at the beginning.

If the dataset is small enough to completely fit in memory, TATi will uses Tensorflow's *caching* to speed up the operations. This will become noticeable after the first epoch, i.e. when all batches of the dataset have been processed exactly once. Caching delivers at least a tenfold increase in learning speed, depending on your hard drive setup.

In memory pipeline
If your dataset fits in memory, it is advised to use the `InMemoryPipeline` by setting the appropriate options in `tati` instantiation, see Section 2.2.

```
nn = tati(
        # ...
        in_memory_pipeline = True,
        # ...
)
```

When using the command-line interface, add the respective option, see Section 2.3.

```
...
  --in_memory_pipeline 1 \
...
```

Furthermore, TATi uses Tensorflow's prefetching to interleave feeding and training operations. This will take effect roughly after the second epoch. Prefetching will show an increase by another factor of 2.

A typical runtime profile is given in Figure Figure 3.2 where we show the time spent for every 10 steps over the whole history. This is done by simply plotting the *time_per_nth_step* column from the run file against the *step* column. There, we have used the [BAOAB] sampler. Initially, there is a large peak caused by the necessary parsing of the dataset from disk. This is followed by a period where the caching is effective and runtime per nth step has dropped dramatically. From this time on, Tensorflow will be able to make use of parallel threads for training. Then, we see another drop when prefetching kicks in.

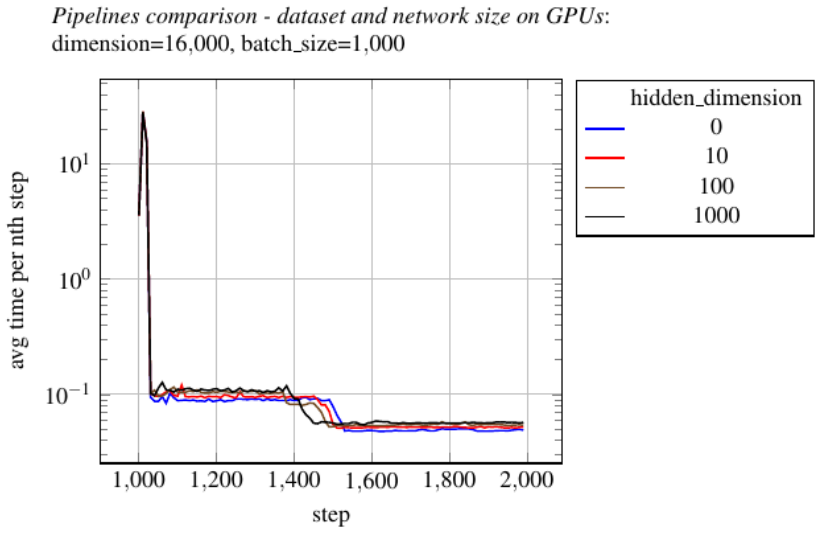


Figure 3.2: Runtime comparison, CPU: Core i7, network with a single hidden layer and various numbers of nodes on a random MNIST dataset

Note that Tensorflow has been designed to use GPGPU cards such as offered by NVIDIA (and also Google's own domain-specific chip called Tensor Proccessing Unit). If such a GPGPU card is employed, the actual linear algebra operations necessary for the gradient calculation and weight and bias updates during training will become negligible except for very large networks (1e6 dof and beyond).

In Figure 3.3 we give the same runtime profile as before. In contrast to before, the simulation is now done on a system with 2 NVIDIA V100 cards. Comparing this to figure Figure 3.2 we notice that now all curves associated to different number of nodes in the hidden layer (**hidden_dimension**) basically lie on top of each other. In the runtime profile on CPUs alone there is a clear trend for networks with more degrees of freedom to significantly require more time per training step. We conclude that with these networks (784 input nodes, 10 output nodes, **hidden_dimension** hidden nodes, i.e. ~1e6 dof) the V100s do not see full load, yet.

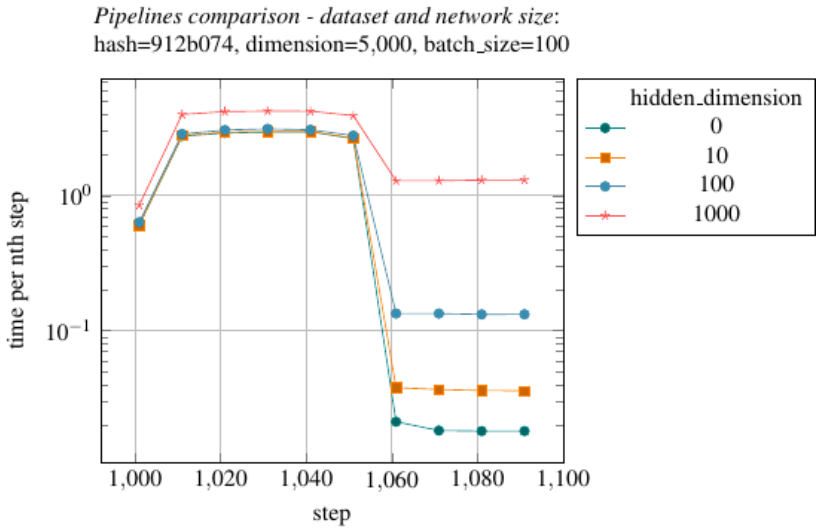


Figure 3.3: Runtime comparison, GPU: 2x V100 cards, network with a single hidden layer and various numbers of nodes on a random MNIST dataset

3.7 Miscellaneous

3.7.1 Freezing parameters

Sometimes it might be desirable to freeze parameters during training or sampling. This can be done as follows:

```
from TATi.models.model import model

import numpy as np

FLAGS = model.setup_parameters(
    batch_data_files=["dataset-twoclusters-small.csv"],
    fix_parameters="output/biases/Variable:0=2.",
    max_steps=5,
    seed=426,
)

nn = model(FLAGS)
nn.init_input_pipeline()
nn.init_network(None, setup="train")
nn.reset_dataset()
run_info, trajectory, _ = nn.train(return_run_info=True, \
  return_trajectories=True)

print("Train results")
```

```
print(np.asarray(trajectory[0:5]))
```

Note that you need to initialize the network without adding training or sampling methods, i.e. `setup` is None. Then, we fix the parameter where we give its name in full tensorflow parlance. Afterwards, we may add sample or training nodes and start training/sampling.

Note
Single values cannot be frozen but only entire weight matrices or bias vectors per layer at the moment.

3.7.2 Displaying a progress bar

For longer simulation runs it is desirable to obtain an estimate after a few steps of the time required for the entire run.

This is possible using the `progress` option. Specified to 1 or True it will produce a progress bar showing the total number of steps, the iterations per second, the elapsed time since start and the estimated time till finish.

This features requires the tqdm package.

Note
On the debug verbosity level per output step also an estimate of the remaining run time is given.

3.7.3 Tensorflow summaries

Tensorflow delivers a powerful instrument for inspecting the inner workings of its computational graph: TensorBoard.

This tool allows also to inspect values such as the activation histogram, the loss and accuracy and many other parameters and values internal to TATi.

Supplying a path `/foo/bar` present in the file system using the `summaries_path` variable, summaries are automatically written to the path and can be inspected with the following call to tensorboard.

```
tensorboard --logdir /foo/bar
```

The tensorboard essentially comprises a web server for rendering the nodes of the graph and figures of the inspected values inside a web page. On execution it provides a URL that needs to be entered in any web browser to access the web page.

Note
The accumulation and writing of the summaries has quite an impact on TATi's overall performance and is therefore switched off by default.

Chapter 4

Acknowledgements

Thanks to all users of the code!

Chapter 5

Literature

- Coifman, R. R., & Lafon, S. (2006). Diffusion maps. Applied and Computational Harmonic Analysis, 21(1), 5–30. https://doi.org/-10.1016/j.acha.2006.04.006
- Duane, S., Kennedy, A. D., Pendleton, B. J., & Roweth, D. (1987). Hybrid Monte Carlo. Physics Letters B, 195(2), 216–222. http://doi.org/10.1016/0370-2693(87)91197-X
- Leimkuhler, B., & Matthews, C. (2012). Rational Construction of Stochastic Numerical Methods for Molecular Sampling. Applied Mathematics Research EXpress, 2013(1), 34–56. http://doi.org/10.1093/amrx/abs010
- Leimkuhler, B., Matthews, C., & Stoltz, G. (2015). The computation of averages from equilibrium and nonequilibrium Langevin molecular dynamics. IMA Journal of Numerical Analysis, 1–55. http://doi.org/10.1093/imanum/dru056
- Matthews, C., Weare, J., & Leimkuhler, B. (2018). Ensemble preconditioning for Markov chain Monte Carlo simulation. Statistics and Computing, 28(2), 277–290. http://doi.org/10.1007/s11222-017-9730-1
- Neal, R. M. (2011). MCMC Using Hamiltonian Dynamics. In Handbook of Markov Chain Monte Carlo (pp. 113–162).
- Shang, X., Zhu, Z., Leimkuhler, B., & Storkey, A. J. (2015). Covariance-Controlled Adaptive Langevin Thermostat for Large-Scale Bayesian Sampling. In C. Cortes and N. D. Lawrence and D. D. Lee and M. Sugiyama and R. Garnett (Ed.), Advances in Neural Information Processing Systems 28 (pp. 37–45). Curran Associates, Inc. http://doi.org/10.1515/jip-2012-0071
- Trstanova, Z. (2016). Mathematical and Algorithmic Analysis of Modified Langevin Dynamics.
- Welling, M., & Teh, Y.-W. (2011). Bayesian Learning via Stochastic Gradient Langevin Dynamics. Proceedings of the 28th International Conference on Machine Learning, 681–688. http://doi.org/10.1515/jip-2012-0071

Chapter 6

Glossary

- **BAOAB**

  BAOAB is the short-form for the order of the exact solution steps in the splitting of the Langevin Dynamics SDE: B means momentum update, A is the position update, and O is the random noise update. It has 2nd order convergence properties, showing even 4th order super-convergence in the context of high friction, see [Leimkuhler2012].

- **Covariance Controlled Adaptive Langevin** (CCAdL)

  This is an extension of [SGD] that uses a thermostat to dissipate the extra noise through approximate gradients from the system.

- **Gradient Descent** (GD)

  An iterative, first-order optimization that use the negative gradient times a step width to converge towards the minimum.

- **Geometric Langevin Algorithm** (GLA)

  This family of samplers results from a first-order splitting between the Hamiltonian and the Ornstein-Uhlenbeck parts. It provides up to second-order accuracy. In the package we have implemented both the 1st and 2nd order variant. GLA2nd is among the most accurate samplers, especially when it comes to accuracy of momenta. It is surpassed by BAOAB, particularly for positions.

- **Hamiltonian Monte Carlo** (HMC)

  Instead of Langevin Dynamics this sampler relies on Hamiltonian Dynamics. After a specific number of trajectory steps an acceptance criterion is evaluated. Afterwards momenta are drawn randomly. Hence, here noise comes into play at distinct intervals while for the other samplers noise enters gradually in every step.

- **Stochastic Gradient Descent** (SGD)

  A variant of [GD] where not the whole dataset is used for the gradient computation but only a smaller part. This lightens the computational complexity and adds some noise to the iteration as gradients are only approximate. However, given redundancy in the dataset this noise is often welcome and helps in overcoming barriers in the non-convex minimization problem.

  See also [GD].

- **Stochastic Gradient Langevin Dynamics** (SGLD)

  A variant of SGD where the approximate gradients are not only source of noise but an additional noise term is added whose magnitude controls the noise from the gradients.

  See also [SGD].