Thermodynamic Analytics Toolkit (TATi) - Programmer's Guide

COLLABORATORS

	TITLE : Thermodynamic Analytics Toolkit (TATi) - Programmer's Guide	

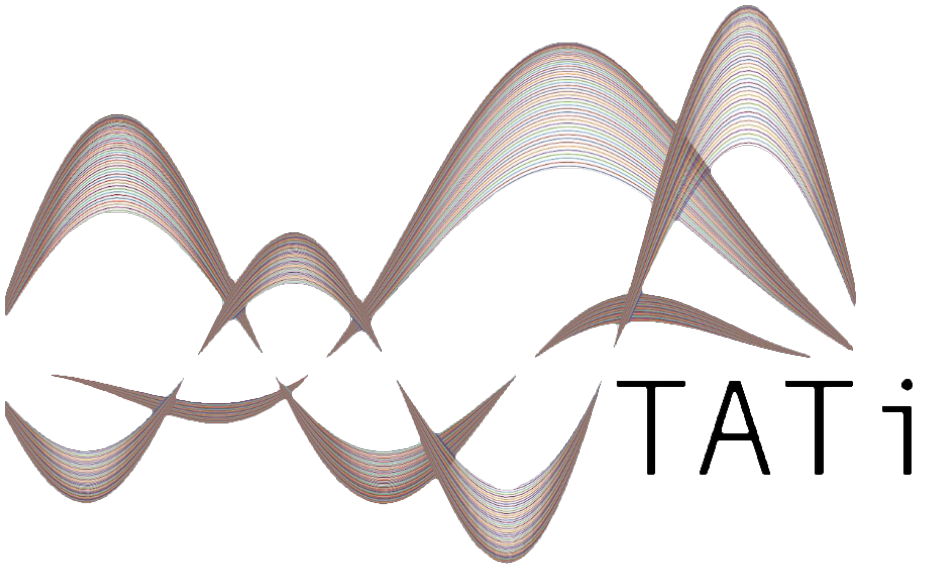
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Frederik Heber	2018-08-17	

REVISION HISTORY

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

Contents

List of Figures



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

Performing efficient neural network training or sampling requires many ingredients. In this programmer's guide we would like to equip you with the necessary knowledge of the abstract concepts of computational graphs, show how to generally use Tensorflow and moreover how to extend it to advanced sampling methods as it is done in TATi.

Note
In case you are generally unfamiliar with TATi, we would like to refer you to the userguide that is also contained in the documentation of this package.

In detail, this guide will give introductory details on the inner workings and major concepts of tensorflow. Moreover, we give extensive details on what is needed to perform neural network training using tensorflow including implementation examples. We conclude with providing details on how tensorflow was extended in order to allow for advanced sampling methods to be incorporated in the course of TATi. To this end, we show how general programming concepts like local and global variables, branching and so on can be executed. This will allow you to extend TATi with your own ideas taking full advantage of the Tensorflow performance.

Chapter 2

Concepts

2.1 Computational Graph

Underlying all of tensorflow and typically most of the Machine Learning (ML) frameworks is the concept of a *computational graph*.

A graph consists of a set of vertices V and a set of edges E. Edges $e_{ij} \in E$ connect two nodes v_i, v_j and may be directed.

In the context of a computational graph functions and variables represent the vertices and directed edges represent dependencies between. In Tensorflow's documentation and tutorials the vertices are referred to as *nodes*. Hence, we will use the term nodes in the following, too.

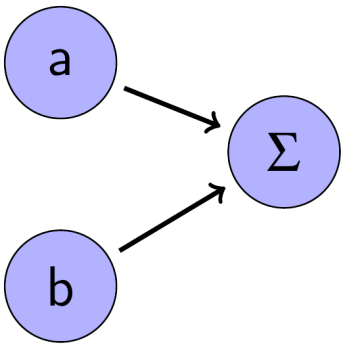


Figure 2.1: Computational Graph: Sum function node depending on the input of two variable nodes **a** and **b**.

Let us have a concrete example and take a look at Figure Computational Graph. There, we have two variable nodes **a** and **b** and one summation node Σ that depends on the two.

Assume we want to evaluate the sum function. The function node can be imagined as a callback whose parameters are supplied by **a** and **b**. Knowing the dependencies encoded in the edges of the computational graph, we know how to execute the callback and evaluate the sum.

Note
Graphs are standard concepts in computer science and enjoy a large variety of algorithms that discover their properties such a shortest paths, number of connected components, cliques, cycles, and so on. Standard algorithms such as Breadth-First Search (BFS) and Depth-First Search (DFS) allow to explore and enumerate all dependencies. For details, see Graph Theory, Reinhard Diestel and other textbooks.

Naturally, the variables **a** may themselves be functions depending on other variables. I.e. arbitrary function concatentations are possible. Moreover, even operations such as assignments are admissible. When the assignment is triggered, using another node as input, this value is written to the internal state of a variable node. Finally, nodes can also be combined into groups such that the execution of the group node triggers the execution of all contained nodes. This allows to program whole algorithms within the framework of computational graphs.

Note
The graph is usually never completely evaluated. When evaluating a certain node, then only dependent nodes must be evaluated, too. All other nodes are ignored.

Think of the computional graph of another way of writing a computer program. The program consists of many very tiny functions (nodes) and the edges encode which function relies/calls which other function. Using the program means executing certain functions that in turn trigger the execution of many other functions in order to deliver their output value.

This concept has even more advantages:

- *lazy evaluation*, i.e., only computes what is necessary and first when it is needed.
- the graph can be analysed for independent or only loosely dependent parts which therefore can be trivially or at least easily parallelized.
- as nodes consist of tiny functions, their computational complexity can be estimated depending on the size of their input arguments. This allows to queue the evaluation of nodes very efficiently.

Note
Tensorflow uses the graph analysis to automatically decide whether CPU or GPU will execute the necessary computations for evaluating a node. Although this automatic association can be overriden, ther is usually no need to. This is different in the dynamic graph setting where the user needs to decide.

2.2 Tensorflow

Historically, there were different approaches in setting up and using the computational graph. theano required to actually compile code that would form the graph. In tensorflow the graph needs to be static throughout the program: first the graph is constructed, then a session object is instantiated and nodes are evaluated. This session object contains the internal state of the graph and of every node. Note that while the graph is static, the session is not, i.e. the contents of variables, may naturally change. Pytorch on the other hand strictly believes in a dynamical graph, i.e. there is no session object containing temporary values but the internal state is directly encoded with each node.

Note
From Tensorflow version 1.6 so-called "eager execution" was introduced to allow for the same dynamical graph use as with PyTorch. Before that adding more nodes to the graph after the session had been created had a bad effect on performance. However, at the time of writing (tf1.9) the static graph is generally faster and generally adapts better to different hardwares and GPU setups, see tensorflow's roadmap on eager execution. In general, static graphs will always execute faster than dynamic graphs.

2.2.1 Constructing a graph

Let us directly see how the above graph, see Figure Computational Graph is constructed using tensorflow.

```
import tensorflow as tf

a = tf.Constant(2.)
b = tf.Constant(3.)
sum = tf.add(a,b)
```

We imported the tensorflow module and then created two nodes containing constant values. Afterwards, we construct a sum node that depends on the two constant nodes.

Note
The above does not perform any actual computation! All we do is construct objects in memory.

In order to actually evaluate the sum in this case, we need to create a `Session` object.

```
sess = tf.Session()
print(sess.run(sum))
```

This will print **5.0** as the result of the operation. The `Session` object contains all the temporary memory required for containing information in nodes.

Tensorflow internally has a default graph to which all created nodes are associated and of which the `Session` takes hold. The graph can be reset by calling `tf.reset_default_graph()`.

Tensorflow has a whole assortment of arithmetic, basic mathematical, linear algebra and similar functions, see Math Ops.

Tip
Functions for standard algebraic operations such as sum, difference, (scalar) multiplication the respective python operators have been overloaded. To give an example, `tf.add(a,b)` can also be written as `a+b`. Moreover, Tensorflow will automatically convert constant python variables into its tensors, e.g., `2. * a - b`. This allows to write mathematical operations in the same way as if manipulating `numpy` arrays.

2.2.2 Variables

Constants are given at the graph's construction and may not change. However, there also variables. These are constructed by giving an initial_value, a (derived) type and a name, i.e., in the same way as the constants. However, in contrast to constants variables allow *assignment*.

The shape is most important and for variables it is derived from the *initial value*. Tensorflow uses the shape to check the consistency when connecting nodes, i.e. chaining functions. For example, a matrix-vector multiplication of a (2,2) matrix with a (10,1) vector will not work because their shapes do not match.

```
a = tf.zeros((784))
b = tf.random_uniform((784,10), minval=-0.5, maxval=0.5)
v = tf.Variable(a, name="zero_vector")
W = tf.Variable(b, name="random_matrix")
```

We first construct a vector of 784 components, all zero. Next, we create a a randon matrix of 784 by 10 components, its values uniformly drawn from the interval [-0.5,0.5]. These serve as initial values to initialize the two variables **v** and **W**, we instantiate afterwards.

As you notice immediately, tensorflow has functions in likeness very similar to `numpy`. First, `a` is a vector with 784 zero components. Then, a random (constant) matrix `b` is constructed. Finally, both are used as initial values for variables. Of course, even higher order tensor can be constructed.

Note
As these tensors "flow" through the computational graph, this gave rise to the name "tensorflow".

The *type* can be `tf.int`, `tf.float32`, `tf.float64`, and so on. Tensorflow will admonish operations where the types are not used consistently. Tensors must be explicitly converted to another type using `tf.cast()`.

The *name* identifies the node - in general, each node has a name and this eases debugging and allows for readable errors messages.

Caution
In contrast to constants and the up-coming placeholders, variables *used* in evaluation need to be initialized once at the beginning of the session. Use `session.run(tf.global_variables_initializer()).`

All in all there are actually three different types of variables: `tf.constant`, `tf.Variable`, and `tf.placeholder`. The last of which we come to now.

2.2.3 Placeholders

Placeholders represent a value of a predefined shape that is supplied by the user lateron. In other words, while constants have to be given at "compile-time" (when writing the python program), placeholder values are supplied at "run-time" (when the program is executed).

This allows for great flexibility. For example, one could extend the above example to a small program that would sum two arbitrary values given by the user in the following way.

```
import tensorflow as tf
import sys

a = tf.placeholder(shape=(), dtype=tf.float32)
b = tf.placeholder(shape=(), dtype=tf.float32)
sum = a + b
sess = tf.Session()
print(sess.run(sum, feed_dict={a: float(sys.argv[1]), b: float(sys.argv[2])}))
```

As you see the two `tf.constant()` nodes **a** and **b** have been replaced by `tf.placeholder()` where we set the shape to `()`, signifying a scalar value. Next, we again create the sum node and instantiate a `Session` object as before. In the last line, execute `run()` on the session. However, there we needed to supply an additional parameter, the `feed_dict`.

This is because a placeholder is nothing but a promise to tensorflow that we will provide a value of the designated shape lateron. The means of providing the value is the feed_dict.

This `feed_dict` is simply a python `dict` with keys and values. The keys are are simply the node references themselves and the values are whatever the user decides to feed in.

Here, we use `sys.argv[..]` to read the first and second command-line argument if you call this script in a file `sum.py` as `python3 sum.py 2. 3.`. For simplicity of the example we do not catch any errors such as missing arguments.

Of course, this is a silly example! However, it serves a point. Through the feed_dict all values may enter that the user needs to specify outside of your algorithm and outside of the tensorflow graph. All parameters that control how a method executes typically should be placeholders.

Note
Not all values for each placeholder need to be fed each time `run()` is executed, but only those which the evaluated node(s) depend on.

2.2.4 Summary

This has been a very brief introduction to tensorflow. In case you need more information, then head over to the tensorflow website where there are plenty of well-written tutorials on light-weight examples such as seen above. Moreover, there you find the Application Programmer's Interface (API) documentation explaining each and every function.

2.3 Neural Networks

Having briefly explained how computations work in general with tensorflow, we would like to come to the concrete example of how neural networks are implemented in such a framework.

Let us make a list of what we need to perform training of neural networks:

- neural network with weights, biases, activation functions and so on
- input pipeline that feeds in the dataset
- elements of training: loss function, gradients, optimizer

2.3.1 Network topology

The neural network is itself a graph and also consists of nodes and edges. Each node has a bias value **b** acting as data-independent shifts and an activation function **f**. Edges connect nodes by stating which output of one node acts as input to another node. They have weights **w** that act as scaling factors, i.e. they are data-dependent.

Note
Tensorflow comes with the `keras` module that is a high-level API providing convenience functions to setup arbitrary network topologies in a layer-wise fashion. In order to maintain complete control over our network we will build it up from the bottom ourselves.

To directly work on a concrete example, we will be using the "Hello, world" pendant of the machine learning world, namely the MNIST dataset: it consists of 55.000 grey-scale images, each 28x28 pixels. These contain hand-written digits, i.e. there are ten different classes to learn for the digits 0 to 9. Each pixel in the image has a single integer value in [0,255] which is usually converted to a floating point number in [0,1]. We will build a simple single-layer perceptron having one input and one output layer.

Warning
Although we let this example be guided by the dimensions of the MNIST dataset, we will not actually show how to use MNIST directly here for simplicity. We catch up on this in a supplement to this section on Section 2.3.

2.3.1.1 Input layer

So, let us build the first layer: the input layer. Inputs to the nodes of the input layer come from the outside, namely the user feeding in a dataset or a batch thereof. Having read about placeholders before, otherwise see Placeholders, this is the first place where we need them.

```
x = tf.Placeholder(float32, [None, 784], name="x")
```

Here, the dimension is 28x28=784. Note that we need to specify a type here. This is used to check that the value later fed into the network in place of this placeholders matches with the type specified. Moreover, the type is used in checking consistency when chaining nodes together. Some of the frequent types are float16, float64, even int16. Moreover, we have set a name for the node to ease debugging and readability of error messages.

What's the purpose of `None`? Remember that we do not have just one image but 55.000 images. Typically, they are fed in batches into the network. As the `batch_size` is not known a-priori but depends on the choice of the user and tensorflow does not allow a placeholder in the definition of the shape, it makes up for it by allowing to specify `None` for all dimensions not known a-priori.

In TATi an additional layer is place in between where a subselection of features can be used and further allowing for transformation with trigonometric functions, i.e. **sin(x1)** would use the sine of the first input dimension as one input to the network.

This is possible as placeholders (and also variables) can be spliced in a pythonic fashion. Let us take a look at how this is done.

```
list_of_nodes = [tf.sin(x[0])]
```

Here, we simply give a list containing only the first component of the placeholder `x` we have constructed before and of which we take the sine. Naturally, this list could contain more values. Next, all these are stacked together to form a single node.

```
x = tf.transpose(tf.stack(list_of_nodes))
```

We need to `tf.transpose()` the result additionally because of the choice of the shape of our input layer `x` as `[None, 784]`. If it were the other way round, there would be no need for it.

Note

In TATi we have chosen the dimension such that they match when writing them down from left to right. For example, matrix vector multiplication of a vector of `[None, 784]` with a matrix of `[784,10]` matches.

Naturally, transforming the input like this may change the input dimension which is important for the subsequent layer. `x.get_shape()` returns an array of all its dimensions.

2.3.1.2 Output layer

Then we continue with the next layer, the output layer. In essence, our network computes the function $f(Wx+b)$, where $\mathbf{W}$ is the weight matrix, $\mathbf{x}$ is the input vector and $\mathbf{b}$ is a bias vector.

Weights and biases are represented by variables `tf.Variable()`, an activation function could be `tf.nn.relu()`. Edges are obtained through algebraic operations such as addition and multiplication and of course concatenation of functions. Naturally, tensorflow offers a whole flock of different activation functions.

In contrast to placeholders, variables and constants need to be given their initial value right from the start. So, how do we initialize the weights and biases? Tensorflow has of course *random numbers* and we simply generate a matrix containing random numbers. The bias vector is usually set to a small, non-zero value, here **0.1**.

Caution

Taking control of the seed of the random number generator will be a critical step and is the main reason for building networks up from the bottom. Although tensorflow offers setting `tf.set_global_seed()` which derives seeds for all internal random number sequences in a deterministic fashion, this fashion changes when a single node (even non-dependent) is added to the computational graph. This is therefore **unusable in a scientific context** where reproducible experiments are a top concern.

All together, we construct the output layer.

```
seed=426
w_init = tf.random_uniform((784,10), minval=-0.5, maxval=0.5, seed=seed,
  dtype=tf.float32)
b_init = tf.constant(0.1, shape=shape, dtype=tf.float32)
W = tf.Variabel(w_init, type=tf.float32)
b = tf.Variable(b_init, type=tf.float32)
pre_activation = W*x+b
output = tf.nn.linear(pre_activation)
```

We fix an arbitrary seed which is used for the random number generator producing the components of the initial weight matrix uniformly in the range [-0.5, 0.5]. Moreover, we have a vector with constant components of 0.1. From these two we create two variables. Next, we have created a node `pre_act` containing the input to the activation function obtained from algebraic operations on these variables. Then, we obtain the `output` as the output of the activation function with this input.

2.3.1.3 Hidden layers

Adding a *hidden layer* is then as simple as adding another weight matrix where we need to use a **different** seed for its random numbers and setting up the function $f_o(W_o f_h(W_h x + b_h) + b_o)$, where we used indices **h** and **o** for the hidden and output layer.

This way we can construct arbitrary feed-forward networks.

Note
Variables are internally marked as *trainable* by default. This means that they go into a special internal collection and we will later see where this is needed when we get to the training part. One can exclude a variable by either stating `...,trainable=False,...` in its construction or by removing it from the collection lateron. A reference to it can be obtained by `tf.get_collection_ref(...)`. Note that `tf.get_collection()` only returns a copy.

2.3.2 Input pipeline

The next step is a bit more involved: Datasets are usually stored on hard drives. We need to parse the files and prepare them. In the case of tensorflow, we need to convert them to `numpy` arrays or directly to tensorflow tensors.

Note
This input pipeline crucially determines the performance of the network overall. If the data is not fed in fast enough, we encounter "data-starvation", i.e. the actual training is idling, waiting for the data to be copied to the right internal places.

Tensorflow, since version 1.4, offers the `tf.data` module with its `Dataset` class. This module allows to perform all the preprocessing *within the computational graph* and conceptually uses three ingredients: tensors, transformations, and the iterator. We'll discuss each of them.

We discuss how to parse CSV files in the following in three steps:

1. Reading the files line by line
2. Restructuring the parsed data into arrays
3. Splitting the full dataset into batches for training
4. Iterating over the batches to feed each into the network

2.3.2.1 Parsing in the files

First of all, we need to tell tensorflow what those files actually are, i.e. we need to provide filenames.

```
filenames = ["test.csv"] # provide list of strings with (CSV) filenames
dataset = tf.data.Dataset.from_tensor_slices(filenames)
```

This is the initial information, the *tensor* ingredient, that the Dataset module needs. Here, some dataset in a file `test.csv`. However, we still need to parse the files. Tensorflow has the specialized classes `tf.data.TextLineDataset` and `tf.data.TFRecor` for this purpose. They parse text and tfrecored files. To allow using this in parallel tensorflow offers `tf.data.Dataset.interleav` our first *transformation*.

```
dataset = dataset.interleave(lambda filename: (
                    tf.data.TextLineDataset(filename)
                        .skip(1)
                        .filter(lambda line: tf.not_equal(tf.substr(line, 0, 1), '#'))),
                cycle_length=self.NUM_PARALLEL_CALLS)
```

You see a `lambda` expression in the interleave arguments. This unnamed function does thr following: First it instantiates one `TextLineDataset` class. Subsequently, we add transformations to the text line dataset. `skip()` tells it to remove the very first line (containing the header). Next, we `filter()` all lines that start with a comment sign (#). The function is called for each filename in `dataset`. The resulting file contents are weaved together allowing for parallel execution. `cycle_length` states how many threads are used. (From tf1.6 the API has changed to `parallel_interleave()`). The result is that we now have a dataset consisting of single lines read from the files.

Note
Do not be misled: We do not have a dataset in memory now! We have only added more nodes to the computational graph that *would* parse in the files and split them up into single lines, when triggered to do so in graph evaluation.

2.3.2.2 Restructuring the dataset

These parsed lines need to be split up (using the "," as separator) and sorted into features and labels. For this we need a small function that requires the number of feature and label columns (this can be provided from the header as well if it conforms to a certain column naming convention).

```
def decode_csv_line(line, defaults, input_dimension, output_dimension):
  items = tf.decode_csv(line, list(defaults.values()))

  # reshape into proper tensors
  features = tf.stack(items[0:input_dimension])
  label = tf.reshape(tf.convert_to_tensor(items[input_dimension: \
    input_dimension+output_dimension], dtype=tf.int32), [output_dimension])

  # return last element as label, rest as features
  return features, labels
```

Note that `tf.decode_csv` is a tensorflow-internal function for splitting up csv files. This function will fill missing values by using the `defaults` which is a dict with every column name as key and a respective default value.

This function is going to be used in the second "transformation", `map()`.

```
dataset = self.dataset.map(
                functools.partial(decode_csv_line, defaults=defaults,
                                  input_dimension=input_dimension,
                                  output_dimension=output_dimension),
                num_parallel_calls=self.NUM_PARALLEL_CALLS)
```

Here, we used `functools.partial` to partially fix some of the arguments of `decode_csv` such that it can be used as functor inside the `map()` call. Notice that we may again perform this transformation in parallel when setting `num_parallel_calls` appropriately.

2.3.2.3 Splitting into batches

Now, we are almost finished. The data is already available in the format recognized by tensorflow. However, we still need to do the typical tasks of shuffling, batching, and so on.

```
dataset = dataset.shuffle().batch(batch_size)
dataset = dataset.repeat(ceil(max_steps*batch_size/dimension))
```

Here, we need to two more pieces of informations from the user, `batch_size` and `max_steps` which give the size of the portions of the dataset used for feeding and the number of feeding steps in total.

Tip
`cache()` stores parsed in files and transformations and therefore greatly speeds up feeding after the first epoch (after the whole dataset has been parsed). This will give one or two orders of magnitude in performance. The best place is right after the last `map()`.

Tip
`prefetch()` will tell tensorflow to interleave operations of fetching and transforming data with subsequent matrix algebra tasks for a better parallel workload. This typically gives another factor of 2 in performance. The best place is at the very end of the `dataset` construction.

2.3.2.4 Iterating over the batches

Last but not least, we need an *iterator*. Iterators are a well-known concept in C++ and they are the same in tensorflow. Iterators point to specific batches of the dataset and deliver the batch on evaluation and advance, i.e. think of it as `i++` in C++ lingo.

Let us construct the iterator that will produce the batch of features (and labels) on evaluation which we need to feed into the network.

```
iterator = dataset.make_initializable_iterator()
batch_next = iterator.get_next()
```

We first construct an (initializable) iterator for our dataset and then we create the node that will produce the next batch of features and labels.

There are different flavours of iterators:

- The standard iterator may run through the dataset just once.
- The (re-)initializable iterator can go over the dataset multiple times if it is reset.
- Last, there are feedable iterators that can work on different datasets, which allow switching between train and test dataset.

Instantiating a session object, we could now take a look at the first batch as follows:

```
with tf.Session() as sess:
  sess.run(iterator.initializer)
  features, labels = sess.run(batch_next)
  print([features, labels])
```

Here, we first have to initialize the iterator (which also resets it to the beginning), next we evaluate and print the first batch.

Note
One could also have provided the filenames through placeholders. In this case one needs to supply additionally `feed_dict={filenames: ["..."]}` with the list of files.

2.3.3 Training

Now, we come to the last ingredient to neural network training, namely how to do the training itself.

Training is done by minimizing a function. The function in this context here is called the *loss*. It compares the predicted output of the network with the labels of the dataset. Moreover, we need *gradients*, i.e. derivatives of the loss function with respect to degrees of freedom of the network, weights and biases. Finally, we need an *update method* or optimizer that refines these degrees making use of gradients, e.g., Gradient Descent.

Let us look at the *optimizer* first and then fill in the other gaps. Furthermore, let us consider a concrete example, namely we want to find the approximate minimum of the convex function $2x^2 - 5x + 4$. Therefore, we know it has a single global extremum (a minimum).

```
import tensorflow as tf

# prepare the computational graph with x and f(x) ...
x = tf.Variable(10.0)
```

```
f_x = 2. * tf.pow(x,2.) - 5. * x + 4.

# ... and training
opt = tf.train.GradientDescentOptimizer(0.1).minimize(f_x)

# Then, performing training using the constructed graph
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(50):
        print(sess.run([opt, x, f_x]))
```

We first set up the variable `x` that is modified during the optimization. Next, we set up the function `f_x` to minimize. We instantiate the `tf.train.GradientDescentOptimizer` class, giving it a learning rate or step width of **0.1**, and we use `minimize()` to tell it which function we actually want to minimize.

Note

In place of the value **0.1** we could also give the class a placeholder as the learning rate. Then, we may flexibly provide the learning rate in each iteration step and could for example tune it adaptively.

Notice something? We did not give the variable `x` to the `minimize()` call. The reason is that tensorflow has an internal collection called "trainables", use `tf.trainable_variables()`. All variables are automatically added to this collection unless excluded (see end of Section 2.3.1). Gradients will always be calculated with respect to all variables in this collection.

Running the example for 50 steps, we obtain an approximate solution `x=1.2500001` with a function value of `f(x)=0.87500024`. In other words, the optimizer has been doing its job properly.

The gradients are determined completely internally. Each (activation) function has an encoded analytical derivative. Through chain rule, also known as back- propagation, the gradients are computed. There is *no* numerical derivation taking place.

Similar to the above example, we will now also train our neural network. The only thing we need to change is the loss function. Here, we will use the mean squared loss $||F_D(x) - y||^2$, where $F_D(x)$ is the black-box neural network function depending implicitly on the dataset. It returns a vector (or matrix) of all predictions. These are compared to $\mathbf{y}$, the labels of the dataset.

Let us setup the loss function and instantiate the optimizer.

```
dataset_labels = tf.Placeholder(tf.float32, [None, 10], name="labels")
mean_squared = tf.losses.mean_squared_error(labels=dataset_labels, predictions=output)
learning_rate = tf.Placeholder(tf.float32, name="learning_rate")
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(mean_squared)
```

First, we need a placeholder for the labels of the dataset in the same way we needed placeholders for the features of the dataset for the input layer. Next, we have created the loss function `mean_squared` that depends on these `dataset_labels` and the `output` layer of our network that was created previously. As a last step, we add the training method which receives the placeholder for the learning rate and tell it to minimize the loss function.

The `minimize()` call naturally does not perform any minimization itself but again just returns a node in the computational graph named `train_step`. This function call has added a lot of nodes to the graph that trigger calculating the gradients and modifying the weights and biases via the update rule. We will see how this works in detail in Section 3.1 covering how to extend the capabilities of tensorflow to allow for new sampling methods.

Evaluating `train_step` will trigger a single update step. Triggering it again will cause another iteration step. Of course, this calls for a loop.

```
with tf.Session() as sess:
  try:
    while(True):
      # parse next batch of dataset
      batch_features, batch_labels = session.run([features,labels])
      # perform single update step printing the current loss
      print(session.run([train_step,mean_squared],
        feed_dict={
```

```
          input: batch_features,
          dataset_labels: batch_labels,
          learning_rate: 0.1,
          }))
  except tf.errors.OutOfRangeError:
    print("End of dataset")
```

We instantiate a session object and perform basically an infinite loop. In the loop body we first evaluate the nodes `features` and `labels` which provide the current batch from the dataset in return. These go into the `feed_dict` for the second `session.run()` that performs a single update step.

As the iterator will raise a `tf.errors.OutOfRangeError` exception when it reaches the end of the dataset and we have told the dataset to repeat itself in such a way that we may obtain `max_steps` batches, it will perform exactly those number of steps and then exit the loop.

2.3.3.1 Chaining feeding and training

As a last note, we would like to show that it is possible to combine the two `run()` calls in the example training loop above into a single call.

As the `tf.data.Dataset` module simply adds nodes to the computational graph that perform the parsing of the data and therefore `features` and `labels` are nothing but nodes in this graph, we may connect them directly to the input layer.

In place of the input layer placeholders `x` we use `features` directly to construct our single-layer perceptron network.

```
# ...
pre_act = W * features +b
output = tf.nn.linear(pre_act)
```

Furthermore, when creating the loss function, instead of giving the placeholder `dataset_labels` for the labels we use `labels` directly.

```
# ...
mean_squared = tf.losses.mean_squared_error(labels=labels, predictions=output)
# ...
```

The loop looks then much simpler.

```
feed_dict = { learning_rate: 0.1}
with tf.Session() as sess:
  try:
    while(True):
      # get next batch and perform training step on it
      print(session.run([train_step,mean_squared],
        feed_dict=feed_dict))
  except tf.errors.OutOfRangeError:
    print("End of dataset")
```

Note that we still need a `feed_dict` for values such as `learning_rate`. However, as the dict does not change per iteration step we may construct it beforehand.

In principle, this second version should save an additional copy in memory when the provided datsaset batch is copied into the `batch_features` and `batch_labels` variables and then into the network's session instead of being directly referenced within the network's session. However, there is no change in performance. Both versions are equally fast.

Tip

Even though in many places in the tensorflow tutorials the use of `feed_dict` for feeding in the dataset is strongly discouraged and therefore one would expect this second approach to work much faster, we have not noticed any difference in performance between either version despite extensive tests and measurements.

We presume that `batch_features` and `batch_labels` are references to the internal arrays within the session object. This would explain why there is no difference in performance.

Caution
The second version has the drawback that the input pipeline has to be created first and the network crucially depends on it. It is not so easy anymore to exchange datasets as when using placeholders to feed the dataset. To some extent it is still possible using feedable iterators. Hence, the second approach sacrifices flexibility of the network.

Warning
The second version has another drawback: Whenever a node is evaluated in a separate `run()` call that depends on `features` or `labels` it will trigger the dataset's iterator to advance!

Chapter 3

Tensorflow extensions

In this section we will elaborate on different concepts in order to extend tensorflow to enable implementation of sampling methods. We will look ath these concepts on a concrete example: implementing a Geometric Langevin Algorithm sampler of 2nd order.

3.1 Sampling Methods

In the previous chapters we have learned what a computational graph is and how tensorflow, that is based on this concept, works in principle. Moreover, we have seen how to build a simple single-layer perceptron, how to parse and feed the dataset, and how to perform the actual training of its weights and biases.

Now, we want to take this one step further by explaining how to implement advanced sampling methods such as [SGLD], [GLA], or [HMC] using tensorflow.

We split this part of the guide into three sections.

1. First, we need to explain how an optimizer is implemented in tensorflow, namely we will be looking at the `tf.train.Gradient` implementation.
2. then, we will talk how to extend this implementation by deriving a new class from it which overrides certain functions.
3. finally, we elaborate on how to store information in local and global variables,
4. and how to do branching.

3.1.1 Optimizers in Tensorflow

A Gradient Descent optimizer is implemented in the class `tf.train.GradientDescentOptimizer`. We first look at the base class and then see how in the framework given by the base class a Gradient Descent update step is realized.

3.1.1.1 Base class Optimizer

The base class `Optimizer` contains several functions, see `tensorflow/python/training/optimizer.py` in your local tensorflow installation. We will describe some of those functions briefly. However, it is not necessary to know them by heart. We only go through this to elucidate the general setup as given by the framework.

In the following, the (list of) variables stands for the variables which are modified in order to minimize the loss, i.e. the list of trainables. The gradients are derivatives of said loss function with respect to these variables.

- `__init__()`

  Instantiates this class. This may be overridden to add more variables to the class that are needed for the sampling method, e.g., the inverse temperature might be supplied as a placeholder.

- `minimize()`

  Calls first `compute_gradients()` on the list of variables, afterwards these are fed into `apply_gradients()`.

- `compute_gradients()`

  Computes the gradients.

- `apply_gradients()`

  Performs the update step by calling `_apply_dense()` (and related functions) for each variable with its respective gradient.

- `_prepare()`

  This is called inside `_apply_gradients()` and used to convert any given python values in `__init__()` call to valid tensorflow tensors.

- `_apply_dense()`

  Adds nodes to the computational graph to perform the update of the a particular set of variables for a single step. There are different variants of this function such as `__apply_sparse()` or `_resource_apply_dense()` that distinguish between the character of the variable that is modified: Is it a dense or sparse tensor, is it a "special" resource variable and so on.

- `_create_slots()`

  Creates a new variable that is directly associated with a present tensor. These are called *slots*. Here, this tensor is typically one of the trainable variables.

- `_zeros_slot()`

  Creates a new variable as does `_create_slot()` but also sets its components to zero intially.

- `get_slot()` Returns a created slot by its (unique) name. The name is the identifier of each slot variable.

Note that there are three types of functions.

- initialization
- computing gradients and applying update
- creating extra variables ("slots")

Moreover, there is a certain hierarchy of functions for computing gradients and applyting them. `minimize()` is the function that we have used already. It simply combines the calls of two other functions. Then, we are leaving the official part of the Python interface and come to functions that are considered private.

Note
Python functions starting with an underscore are private by naming convention but they are not protected from any actual access.

One of the these two called functions is `_apply_gradients()` which uses `_prepare()` and `_apply_dense()` (we will ignore the other variants in this guide as the changes would be equivalent). The latter is the main work horse.

It is of utmost importance to fully realize the following bits:

- `_apply_dense()` is just called once per variable, e.g., for single-layer perceptron it will be called two times: once for the weights **W** and once for the biases **b**. That's it. In other words, you cannot perform any calculations in Python code which need to be done every step. You can, however, perform calculations that are required initially, i.e. before all steps.
- `_apply_dense()` is still called multiple times, once for each trainable variable. This means that you cannot distinguish in that function between computations you only want done for the biases and not for the weights. The same function will be called for both weights and biases.

3.1.1.2 GradientDescentOptimizer

Let us now turn to the implementation of the Gradient Descent optimizer in the tensorflow code base, see `tensorflow/python/tra`

- `__init__()`

  Overrides the base function to store the variable `learning_rate` received as parameter in a member variable.

- `_prepare()`

  Converts the received variable into a proper tensorflow tensor using `convert_to_tensor()`. This step is necessary as the `GradientDescentOptimizer` can also be a given a python `float` as the learning rate.

- `_apply_dense()`

  Implements the actual update step.

Note
We again ignore all the other implementations in the functions related to `_apply_dense()`

The actual code inside `_apply_dense()` is not very illustrative. There is a lot of abstraction code to switch between eager and static execution and so forth. Eventually and hidden deep inside the tensorflow code base, the code makes use of a very efficient implementation in C++ to perform the actual update step that performs well on both CPUs and GPUs.

Therefore, let's do the (python) implementation ourselves in the following example. Remember that the update step in Gradient Descent is $x_{n+1} = x_n - \lambda \nabla_x L(x_n)$, where we have called the current variable tensor $\mathbf{x}$, λ is the scalar learning rate, and $\nabla_x L(x_n)$ is the gradient of the loss function $\mathbf{L}$ with respect t the variable tensor.

Gradient Descent optimizer

```
from tensorflow.python.ops import control_flow_ops
from tensorflow.python.ops import state_ops
from tensorflow.python.training import optimizer

class GradientDescentOptimizer(optimizer):
...
  def _apply_dense(self, grad, var):
    scaled_gradient = grad * self._learning_rate_tensor # ❶
    var_update_t = state_ops.assign_sub(var, scaled_gradient) # ❷
    return control_flow_ops.group(*[var_update_t]) # ❸
...
```

❶ First, inside `_apply_dense()` we create a helper node that is referenced by `scaled_gradient` where we simply multiply the converted learning rate with the gradient tensor `grad`.

❷ Next, we perform the actual update of the variable tensor `var` using an assignment operation. Here, we use a special variant of it that subtracts the given value from the tensor.

❸ The last steps consists of returning a list of nodes that need to be executed to fully perform the update on this particular variable tensor. In our case it is just a single node, namely `var_update_t` that is referencing the update stop.

Note that `var_update_t` depends on `scaled_grad`. Hence, we do not have to give the latter in the returned list of nodes as it will be evaluated (and updated) automatically.

Again, this function is executed once and only once per variable adding all the nodes to the computational graph to perform a single training step.

3.1.2 Techniques for simple Samplers

We have seen how the [GD] optimizer was implemented in the last section. Next, we will implement our first sampler. In order to make this a little interesting, we look at [GLA]2, see [Trstanova, 2016, (1.59)].

Let us first look at the formulas that perform the descrete time integration.

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. $p_{n+1} = \alpha\widehat{p}_{n+1} + \sqrt{\frac{1-\alpha^2}{\beta}}\cdot\eta_n$

Here, η_n is the random noise at step **n**, p_n designates momentum, and moreover we have $\alpha = \exp(-\gamma\cdot\lambda)$ with the so-called friction constant γ.

3.1.2.1 Variables of the first kind: slots

First of all, we need more variables. x_n is given by `var` and $\nabla_x L(x_n)$ is given by `grad`. Both are simply parameters to the `_apply_dense()` function. However, we lack a variable to store the momenta p_n.

For this purpose, we can use *slots*. Slots are an official mechanism of tensorflow's `Optimizer` for this particular purpose as there are other, more advanced optimizers such as ADAM that actually use momenta as well. We will see how these are constructed in the new `_prepare()` function.

However before that, we still need more variables in the form of placeholders: inverse temperature β, friction constant γ and the learning rate λ which we will call *step width* to distinguish samplers from optimizers.

and there is one more: GLA2 contains a noise term η_n. To this end, we require a random number generator that produces random numbers of the same *shape* as `var` (or equivalently `grad`). These are created by `tf.random_uniform()` as you will see further below in `_prepare_dense()`. To reproducibly create the noise, this function takes a random number seed. This seed value needs to come from an extra placeholder.

Then, all of these are handed on to our new class `GLA2Sampler` in its constructor.

Constructor for class GLA2Sampler

```
def __init__(self, step_width, inverse_temperature, friction_constant,
    seed=None, use_locking=False, name="GLA2"):
    super(GLA2Sampler, self).__init__(use_locking, name)
    self._friction_constant = friction_constant
    self._step_width = step_width
    self._seed = seed
    self._inverse_temperature = inverse_temperature
```

You notice that we simply store all the obtained parameters using (private) member variables.

As these parameters could have been either python constants or tensorflow tensors, we convert each into a valid tensor in `_prepare()`.

Converting value to tensors

```
def _prepare():
  self._step_width_t = ops.convert_to_tensor(self._step_width,
      name="step_width")
  self._friction_constant = ops.convert_to_tensor(self._friction_constant,
      name="friction_constant")
  self._inverse_temperature_t = ops.convert_to_tensor(self._inverse_temperature,
      name="inverse_temperature")
```

Preparation is done in two steps: In the first step, see `_prepare()` above, we have converted any python value which we might have received in the `__init__()` call into full tensorflow tensors. In the second call, we cast tensors to the right type, e.g., `tf.float32` and produce the random number tensor which we need for the noise. Let us take a look.

Casting to the correct type and random number tensor

```
def _prepare_dense():
  step_width_t = math_ops.cast(self._step_width_t, var.dtype.base_dtype) # ❶
  friction_constant_t = math_ops.cast(self._friction_constant_t,
      var.dtype.base_dtype) # ❷
  inverse_temperature_t = math_ops.cast(self._inverse_temperature_t,
      var.dtype.base_dtype) # ❸
  if self._seed is None: # ❹
      random_noise_t = tf.random_normal(grad.get_shape(), mean=0.,stddev=1.,
          dtype=var.dtype.base_dtype) # ❺
  else:
      # increment such that we use different seed for each random tensor
      self._seed += 1 # ❻
      random_noise_t = tf.random_normal(grad.get_shape(), mean=0., stddev=1.,
          dtype=var.dtype.base_dtype, seed=self._seed) # ❼
  return step_width_t, inverse_temperature_t, friction_constant_t, random_noise_t
```

❶, ❷, ❸ In the first three lines we simply use `math_ops.cast()` to convert the given parameter tensor to the same type as `var`. This type is contained in `var.dtype.base_dtype`.

❹ we branch depending on whether the given `seed` value, stored in the member variable `_seed` is `None` or not.

❺ If the seed is `None`, i.e. not set, then we create the random noise tensor `random_noise_t` without specifying a seed. This will cause the random number generator to pick a different seed every time the program is executed.

❼ If a seed is given on the other hand, then it is used in the constructor call using `tf.random_normal(..)` which returns a normally distributed set of random variables of the same shape as `grad`. The shape is obtained through `grad.get_shape()`. Each variable has a `mean of *0.* and a standard deviation `stddev` of **1.**. Moreover, we use again the same type as that of `var`.

❻ We increment the seed as `_prepare_dense()` is called multiple times, ones per trainable variable just as `_apply_dense()` and we need to use different random numbers each time.

3.1.2.2 Rearranging integration steps

Next, having now all variables at hand, we may come to actual implementation of the time integration steps, see GLA2 for the formulas.

However, if we want to translate the set of equations directly into tensorflow instructions inside the body of `_apply_dense()` we face a restriction. Remember that `_apply_dense()` is called inside `_apply_gradients()` which is called after `compute_gradients()`. In other words, the gradients have been computed just before.

Hence, in the initial step we have an evaluation of term $\nabla_x L(x_n)$ and we cannot trigger a second evaluation inside `_apply_dense()` to compute $\nabla_x L(x_{n+1})$. If we wanted to do this, we would have to completely rewrite the base `Optimizer` class.

Warning
Rewriting tensorflow classes is perfectly possible. However, tensorflow only tries to keep the official public part of API unchanged. As most of the `Optimizer` class functions are private, their bodies and their signature may change with future versions of tensorflow and even without official notice. Tensorflow updates roughly every two months. In general, this would lock the applicability of such an implementation to a very specific tensorflow version.

However, there is another solution to this. Remember that the list of four steps above is continued for a certain number of steps, i.e. after step *4.* in GLA2 at iteration **n** follows step *1.* at **n+1**. In other words, we may cyclically rearrange the steps, ignoring that the first step is then no longer correct.

Caution
It is still important *where* to evaluate quantities such as kinetic energy. I.e. their evaluation must be cyclically rearranged in accordance.

We will implement the steps in the following order: 3., 4., evaluate kinetic energy and so on, then 1. and 2. In other words, instead of BABO we execute BOBA. See [Leimkuhler2012] for the nomenclature of the steps **A**, **B**, **O**.

Implementing GLA2 with tensorflow

```
def _apply_dense(self, grad, var):
  step_width, beta, gamma, random_noise_t = self._prepare_dense(grad, var) # ❶
  momentum = self.get_slot("momentum") # ❷
  scaled_gradient = grad * step_width_t # ❸

  # 3. \widehat{p}_{n+1} = p_{n+\tfrac 1 2} - \tfrac {\lambda}{2} \nabla_x L(x_{n+1})
  momentum_half_step = momentum - 0.5 * scaled_gradient # ❹

  # 4. p_{n+1} = \alpha \widehat{p}_{n+1} + \sqrt{\frac{1-\alpha^2}{\beta}} \cdot \eta_n
  alpha = tf.exp(-friction_constant * step_width) # ❺
  noise_scale = tf.sqrt((1.-tf.pow(alpha, 2))/inverse_temperature) # ❻
  scaled_noise = noise_scale * random_noise_t # ❼
  momentum_noise_step = alpha * momentum_half_step + scaled_noise # ❽

  # 1. p_{n+\tfrac 1 2} = p_n - \tfrac {\lambda}{2} \nabla_x L(x_n)
  momentum_t = momentum.assign(momentum_noise_step - 0.5 * scaled_gradient) # ❾

  # 2. x_{n+1} = x_n + \lambda p_{n+\tfrac 1 2}
  var_update_t = state_ops.assign_add(var, step_width * momentum_t) # ❿

  return control_flow_ops.group(*[var_update]) # ⓫
```

As this is quite a large example, let's go through the lines step by step.

❶ First, we use `_prepare_dense()` to obtain the three tensors containing our parameters `step_width` δt, `beta` β, and `gamma` γ. Moreover, we get the random number tensor that is our source of normally distributed noise η_n.

❷ Next, we get the slot variable `momentum`. We have obtained `grad` and `var` as parameters to `_apply_dense()`.

❸, ❹ Then comes the first momentum integration step, **B**. We create a helper node `scaled_gradient` that contains the gradient tensor scaled by the step width. Then, we assign the update to a temporary tensor named `momentum_half_step`.

❺, ❻, ❼, ❽ For the following **O** step, we need to compute α. To this end, we create several temporary nodes `alpha`, `noise_scale`, and finally `scaled_noise` to obtain the factor in the first in `alpha` and the second term in `scaled_noise` of step **4.** in GLA2.

❾ Now, we are actually at step **1.** of the split formulation for the Langevin dynamics time integration. We perform another **B** step. However, this time we assign the result back into the `momentum` slot. This action will be done whenever we evaluate `momentum_t`: it will *both* return the momentum after the second **B** step *and* assign its value to the slot `momentum`!

❿ Finally, we come to the **A** step where the variables in `var` are updated using the updated momentum in `momentum_t` just as in step **2.**.

⓫ The very last step consists of returning a list of nodes, here `[var_update_t]` to trigger the actual evaluation.

Let us elaborate a bit why in the last step we only need a single tensor in the list: `var_update_t` depends on `momentum_t`. Hence, it triggers the evaluation of that node. `momentum_t` depends on `momentum_noise_stept` and `scaled_gradient`. Hence, it will trigger those two nodes. `momentum_noise_step` in turn triggers evaluation of `alpha`, `momentum_half_step`, and `scaled_noise`. Finally, `scaled_noise` depends on `noise_scale` and `random_noise_t`.

So, all created nodes are actually evaluated when `var_update_t` 's evaluation is triggered.

And that's it. We are done with adding a GLA2 sampler.

Note
It is a good idea to adhere to a certain naming convention: Nodes that actually modify the state — by assigning a new value to a slot or by calling a random number generator which modifies its internal state, ... — are suffixed by `..._t` as a reminder that these are tensorflow **tensors**. All helper nodes in the above example do not have this suffix.

Caution
The above lines could principally be in any order as the order of execution comes purely from their dependence on one another in the computational graph. They are still in order because during construction we need the right python variables referencing tensorflow nodes at hand. We will come to this point in the next session.

3.1.3 Local and global variables

Having the sampling method implemented, we would like to test it. However, we can only access the loss at the moment. We do not know the kinetic energy or other quantities of interest because they are hidden away in some nodes in the computational graph.

Tensorflow maintains a dictionary of all nodes in its graph by which we could try to access the `momentum` slots. The key is always the `name` of the node. Therefore, we could access momenta by an additional `Session.run()` call after the sampling step. However, then we cannot evaluate the kinetic energy at the right point, namely just before step 5 and not after step 7 or before step 1.

In essence, we need *more variables* where we can store the contribution to the kinetic energy of the respective trainable variable and access it later on.

There will be four pieces to this puzzle which we discuss one by one.

- create the variable
- access and assign the variable
- evaluate the variable
- "reset" the variable (which is the same as assigning it)

3.1.3.1 Creating a resource variable

While the `momentum` slot is *local* to the specific `_apply_dense()` context, these variables will have a *global* character to them. Therefore, they are created outside of the `_apply_dense()` somewhere before we instantiate the sampler `GLA2Sampler`.

Resource variable

```
with tf.variable_scope("accumulate", reuse=False):
      tf.get_variable("kinetic_energy", shape=[], trainable=False,
                      initializer=tf.zeros_initializer,
                      use_resource=True, dtype=tf.float32)
```

Here, we first set a variable scope: all variables in that scope will have their names prefixed with "accumulate/", i.e. the name of the kinetic energy variable is "accumulate/kinetic_energy". We mark it as not trainable, after all it is designed as pure storage. Its shape defines it as a scalar quantity. Its `dtype` is simply `tf.float32`. Note this should match with the type of the trainable variables, see Output layer where we created them for the first time. Moreover, we initialize it to zero. See Variables on a reminder that variables need to be set to a specific value initially.

There is one element we have not encountered: `use_resource`. At the time of writing it is not clear to the author what this really does. Empirical evidence showed that it made using these variables as global variables more robust. Tensorflow's API referenced them for a long time as "experimental". Other variables are not created as *resource* variables by default.

3.1.3.2 Assigning to a resource variable

We can get hold of this variable in much the same way we got a slot before.

Getting a resource variable's reference

```
with tf.variable_scope("accumulate", reuse=True):
      kinetic_energy = tf.get_variable("kinetic_energy", dtype=tf.float32)
```

Before you strain your eyes too hard to spot the *important difference*, let us hightlight it: `reuse=True` means that if the variable is already present, tensorflow simply returns a reference. Moreover, this time we store the reference that `tf.get_variable()` returns in `kinetic_energy`.

Then, we may extend our present implementation of GLA2 in `_apply_dense()` in the following way to also accumulate the kinetic energy.

Adding kinetic energy accumulation to GLA2.

```
def _apply_dense(self, grad, var):
  ...
  momentum_noise_step = alpha_t * momentum_half_step_t + scaled_noise

  with tf.variable_scope("accumulate", reuse=True): # ❶
    kinetic_energy = tf.get_variable("kinetic_energy", dtype=dds_basetype) # ❷
    # 1/2 p_{n}^t p_{n}
    momentum_sq = 0.5 * tf.reduce_sum(tf.multiply(momentum_noise_step, # ❸
      momentum_noise_step))
    kinetic_energy_t = tf.assign_add(kinetic_energy, momentum_sq) # ❹

  # 1. p_{n+\tfrac 1 2} = p_n - \tfrac {\lambda}{2} \nabla_x L(x_n)
  with tf.control_dependencies([kinetic_energy_t]): # ❺
      momentum_t = momentum.assign(momentum_noise_step - 0.5 * scaled_gradient)
  ....
  # 2. x_{n+1} = x_n + \lambda p_{n+\tfrac 1 2}
  var_update_t = state_ops.assign_add(var, step_width_t * momentum_t)

  return control_flow_ops.group(*[var_update, kinetic_energy_t]) # ❻
```

❶, ❷ We obtain a reference to the already created resource variable.

❸, ❹ Next, we add the contribution from $\frac{1}{2}p_n^t p_n$ to the resource variable.

❺ Adding a `tf.control_dependencies()` in a `with` context instructs tensorflow that all nodes in the list have to be evaluated before any of the statements inside the context are evaluated. This ensures that we add the kinetic energy contribution before continuing with step **1.** in GLA2. Remember that evaluation is determined by dependence and not by order of appearance in the python code.

❻ We need to make sure that the assignment actually takes place by adding it to the list of returned variables, as `var_update_t` does not depend on it.

3.1.3.3 Evaluating a resource variable

Evaluating the variable is simply the same as any other node in tensorflow, using the `run()` statement of a session object. Assume we have obtained the reference to the kinetic energy as `kinetic_energy`, see Assigning a resource.

Evaluating the kinetic energy

```
with tf.Session() as sess:
  print(sess.run(kinetic_energy))
```

3.1.3.4 Zeroing to a resource variable

There is one last step: If we want to have the current kinetic energy per step and not some average, then we need to zero the resource variable before continuing the next sampling step.

To this end, we need to create an assignment node *before* we instantiate the `Session`. The assignment will place the constants scalar of **0.** in the resource variable whenever it is evaluated. Again, we assume having a reference to the variable in `kinetic_energy` already present.

Evaluating the kinetic energy

```
zero_kinetic_energy_t = kinetic_energy.assign(0.)

with tf.Session() as sess:
  sess.run(zero_kinetic_energy_t)
```

Now, that you can access the kinetic energy, do try to access other quantities. Running averages are simply obtained by not setting to zero and dividing by the number of steps on output. Virials can be computed by looking at the scalar product of gradients and variables, `tf.reduce_sum(tf.multiply(grad,var))`. Norms of gradients, noise, momenta, and so on can be obtained in the same way. For each a diferent resource fvariable is required.

3.1.4 Branching

As the last topic in this guide we come to an issue that has some peculiarities about it. We would like to add nodes to the graph such that either one node or another is evaluated depending on a given condition.

Let us look at a simple example in familiar python code.

Branching in python

```
import numpy as np
np.random.seed(426)

steps = 100
threshold = 0.5
lower_sum = 0
higher_sum = 0

for i in range(0, 100):
  value = np.random.random() # ❶
  if value < threshold:
    lower_sum += value # ❷
  else:
    higher_sum += value # ❸

print([lower_sum/steps, higher_sum/steps])
```

This is really a contrived example:

❶ We throw a random die to uniformly produce a number in [0,1].

❷ When the value is less than **0.5**, we add it to `lower_sum`.

❸ If it is equal or larger, then we add it to `higher_sum`.

Let us look at how to implement this in tensorflow. Naturally, we need to convert all expressions to tensorflow statements. The `if` condition is replaced by `tf.cond()` which takes three arguments: the condition statement, a function for the true case, a function for the false case. Let us say that again: You need to give function handlers as second and third argument, *not* tensorflow nodes! However, you can wrap your node in a dummy function that returns it. This is the first peculiarity but there is another. The first is not too bad because tensorflow will throw an error informing you that it needs a function in place of a node. The second one is more subtle to debug.

Let us first take a look at **how not to do it**, i.e. let us deliberately stumble over the second peculiarity.

Naive branching implementation

```
steps=100
threshold = tf.constant(0.5, dtype=tf.float32) # ❶
lower_sum = tf.Variable(0., trainable=False) # ❷
higher_sum = tf.Variable(0., trainable=False) # ❸
random_t = tf.random_uniform(min=0., max=1., dtype=tf.float32) # ❹

def accept_low(): # ❺
  return lower_sum.assign(lower_sum + random_t)

def accept_high(): # ❻
  return higher_sum.assign(higher_sum + random_t)

branching_t = tf.cond(tf.less(random_t, threshold),
  accept_low, accept_high)  # ❼

with tf.Session() as sess:
  for i in range(0,steps):
    sess.run(branching_t) # ❽
  print(sess.run([lower_sum/steps, higher_sum/steps])) # ❾
```

❶, ❷, ❸ First, we create several variables in the same way as we instantiated python variables before.

❹ The first difference is that we have to create a source of random numbers. However, this should be quite familiar as we had to do the same thing for the sampler implementation above.

❺, ❻ Next, circumventing pecularity number one we define two functions that sum onto the `lower_sum` and `higher_sum` variables through assignment.

❼ Then we come to the branching statement. Using `tf.less()` we make a comparison node between the current random number in `random_t` and the constant `threshold`. `tf.cond()` then should execute either `accept_low` or `accept_high`.

❽ We evaluate the branching node for 100 steps.

❾ And finally we print the two resulting averages for either sum.

And the result is: something close to **0.5, 0.5**?! We had expected this to be **0.25, 0.75**.

What has happened? When tensorflow evaluates the `tf.cond()` statement it needs to look at *both* the possible true and false branches in deciding which nodes they depend on. Somehow this also seems to make it necessary to evaluate them. In other words, in the naive way above both branches are executed and we obtain two equivalent sums.

The workaround is to hide away all *side effects*, i.e. changing the state, of your branches inside a `tf.control_dependencies()` statement and only return a dummy node, e.g., using `tf.identity()`.

Changes for a working branching implementation

```
...
def accept_low():
  with tf.control_dependencies([lower_sum.assign(lower_sum + random_t)]):
  return tf.identity(random_t)

def accept_high():
  with tf.control_dependencies([higher_sum.assign(higher_sum + random_t)]):
  return tf.identity(random_t)
...
```

This now returns the expected output.

In our corrections we have circumvented another peculiarity: Inside the control dependency there *must not appear nodes*. Tensorflow will not throw an error but these will not be triggered. Depenedencies must be always given as statements. After all, this is why this workaround is working at all.

To make this concrete, *the following does nothing*.

Node should not be used in control dependencies when branching

```
...
accept_low_t = lower_sum.assign(lower_sum + random_t)
def accept_low():
  with tf.control_dependencies([accept_low_t]):
  return tf.identity(random_t)
...
```

3.1.5 More flow control

Note that tensorflow basically has support for any kind of flow control. Using the branching statement one could easily implement a loop. There is also a `tf.while_loop` statement right away. See Control Flow Operations for a complete list of all control flow operatipons that tensorflow supports.

Note
Not every one of them is useful. In our case of the sampler it is perfectly acceptable to have a hybrid python/tensorflow implementation as each train step depends on the former and therefore no parallelization is possible there.

Chapter 4

Debugging tensorflow code

Here, we show tips and tricks on how to debug tensorflow code.

Tensorflow code relies on the concept of a computational graph which is often harder to grasp intuitively than the typical programming paradigm such as found when writing python programs.

Typical debugging makes use of print statements that show the control flow and internal states of the program on the console for the user to follow. Tensorflow has similar statements but they work in the particular manner of the computational graph.

4.1 Tips

The tips and guidelines revolve around the following concepts:

- print statements,
- debugger,
- toy models and minimum working examples.

4.1.1 Print statement

Our first advice is to make use of `tf.Print()` when something does not look right. It serves a similar purpose as would a `print()` statement do in normal python code: We want to inspect the value of a certain variable or we want to see when the program reaches a certain point in its control flow.

Because of the nature of the computational graph, we cannot simply add a node using `tf.Print()` and be done. We have to make sure that the node is actually executed in the evaluation of the graph. In other words, it needs to be inserted in the proper place.

To enable placement in arbitrary positions, `tf.Print()` works the same way as `tf.identity()` does, i.e. it is a pass-thru node that simply passes the value of its argument on, see Figure Pass-thru node. However, it has the side-effect of printing a `message` together with `data`.

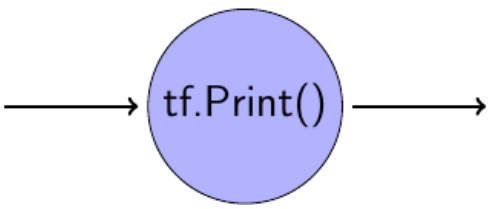


Figure 4.1: Pass-thur node: `tf.Print()` simply passes thru input to output

Imagine we wanted to inspect whether the random number tensor in our Branching example before would not actually be evaluated two times. Or even three times in the naive example: Once for evaluating `branching_t` and two more times when adding onto either sum.

Does this value coincide each time? Or has a new random number been triggered? We can deduce the behavior from the result **0.25, 0.75**. Nonetheless, let us do this properly.

Therefore, let us modify the `branching_t` statement and the function definitions as follows:

Print debugging in the tensorflow lingo

```
def accept_low():
  with tf.control_dependencies([lower_sum.assign(lower_sum + random_t)]):
  return tf.Print(random_t, [random_t], message="random number low") # ❶

def accept_high():
  with tf.control_dependencies([higher_sum.assign(higher_sum + random_t)]):
  return tf.Print(random_t, [random_t], message="random number high") # ❷

branching_t = tf.cond(tf.less(
    tf.Print(random_t, [random_t], message="random number"), threshold),
  accept_low, accept_high) # ❸
```

❶, ❷ Here, we have simply replaced the identity by the print statement.

❸ In the branching statement we do not use `random_t` directly but let it pass thru the print statement first.

This way we obtain two messages per loop iteration and we can easily compare the values of the message with "random number" to the one from "random number low" or "random number high".

4.1.2 Using the debugger

`tfdbg` is a specialized debugger for inspecting tensorflow's computational graphs.

It is added when the `Session` object is modified in the following way:

```
sess = tf_debug.LocalCLIDebugWrapperSession(sess, ui_type=FLAGS.ui_type)
```

The debugger is more powerful when inspecting multiple values. Execute the program through `run`. This will stop once per encountered `session.run()` statement. `lt` lists all tensors. `pt` prints the value of a particular tensor. We refer to above linked tutorial on how to use the debugger.

Debugging is also possible through TensorBoard which is a webserver specialized on displaying inputs, outputs and states of the tensorflow graph.

It is added when the `Session` is modified like this.

```
sess = tf_debug.TensorBoardDebugWrapperSession( sess,
    FLAGS.tensorboard_debug_address)
```

This debugger is essentially a graphical interface with many of the features of the command-line (CLI) debugger. It requires a running tensorboard session. Again, we refer the reader to the tutorial.

4.1.3 Understanding through toy models

One last way of debugging a program is in making a *minimum working example*.

This could for example be a toy model such as simple harmonic oscillator where analytical properties such as the mean loss are known that we may compute and compare against.

In general, when something is not working right, then try to make the code that is not working as small as possible. Throw away any other code that has nothing to do with the problem.

As a start, you can use the code examples in this guide to build a light-weight GLA2 sampler on a single-layer perceptron. By feeding a dataset consisting of a single **0.** as feature and a single **0.** as its label and using the `mean_squared` loss, you obtain a harmonic oscillator in the single bias variable.

But a toy model may also be much simpler, consisting only of two variables and a statement combining the two. It may help understanding how a tensorflow statement works, see Constructing a graph.

Chapter 5

Setup of TATi

In this chapter we want to explain the program structure of TATi, namely its

- directory and python module structure
- build system
- version control system (git)

5.1 Structure

Understanding the structure of TATi's directories and python modules/files is essential when contributing.

5.1.1 Directories

Let us explain each folder in turn:

- doc
  Contains documentation such as the userguide and also configuration files for doxygen to produce the reference documentation.
- examples
  This folder is for IPython/Jupyter notebooks on tutorials or short examples on using TATi.
- src
  Contains the source tree of TATi, i.e. all python modules that are installed are located in this folder.
- tests
  Contains all tests, i.e. everything needed to execute `make check`.
- util
  This folder contains helper python scripts that are not general enough to be placed into the `src` folder but still serve a purpose close to TATi's nature.

There are probably several other folders: `autom4te.cache`, `build-aux`, and `m4`. These belong to the autoconf build system are explained in Section 5.2.

In the following we look more closely at each folder. Each of them is connected to a different toolchain as well.

5.1.2 Documentation

TATi's *userguide and programmer's guide* in `doc/userguide` consist of simple text files that use a special "markdown" notation in the asciidoc format. See the userguide there for a full reference and plenty of examples.

All *example scripts* are full python scripts that are tested in the folder `tests/userguide`. They reside in subfolders `cmdline`, `python`, and `simulation` for the command-line interface, for the general python interface and for the `simulation` interface. This is to make sure that all examples are always up-to-date and working.

All *images* in the guides reside in the subfolder `pictures` and typically consist of **png** files. If they have been created using tikz or pgfplots, then the source `.tex` files have been placed in there as well.

The program packages `asciidoc` and `dblatex` are used to convert the text files first fto html and xml and the latter is finally converted one more time to a pdf file. Instructions are found `doc/userguide/Makefile.am`.

Moreover, TATi's *reference documentation* is automatically created from the source files located in folder `src` using doxygen. It pulls in additional files that explain the source code in general from `src/documentation`. The configuration files for doxygen are located in `doc/Doxyfile`.

5.1.3 Examples

Examples as Jupyter notebooks reside in the `examples` folder. They are currently still untested, i.e. there are not checked to still comply with the current API of TATi.

In principle it is possible to extract the pure python code from notebooks, see this stackoverflow question. However, a few issues are difficult to overcome:

- comments in cells cannot be extracted without jupyter.
- jupyter notebooks lack proper syspath in general.
- plot commands (`plt.show()`, `ax.scatter`) are difficult to prevent in this non-interactive environment.

To this end, they remain still untested until these can be resolved.

5.1.4 Python modules

The folder `src` contains two folders: `TATi` and `documentation`.

The latter simply contains additional doxygen files to explain general parts and concepts of the source code.

The folder `TATi` consists of all python modules for the `TATi` module. The structure in the source tree is exactly the same as it is installed, i.e. a module `TATi.models.model` can be found in the folder `TATi/models/model.py`.

The folder itself contains several subfolders which we briefly enumerate and explain.

- datasets

  This contains generators for toy example datasets.

- diffusion_maps

  Here, we have code for performing diffusion map analysis. Similar functionality also exists in the package `pydiffmap`, see Analysis section in the quickstart part of the userguide.

- exploration

  In exploration we code for obtained several sampling and training trajectories and analysing them. To this end, there is a "queue" implementation there and each part (e.g., sampling trajectory) is encoded as a "job". To allow for parallel runs, there is also an extension where each job is run as an independent "process" if it relies in tensorflow.

- models

  Models contains the general python interface class `model.py` and helper classes to construct the actual "model", i.e. the input pipeline in subfolder `input` and the neural network.

- runtime

  This folder contains a helper class to measure the runtime of events and is used in the command-line tools.

- samplers

  Here, all sampler implementations are gathered. They all derive from `walkerensembleoptimizer.py` which contains functionality to have multiple copies of the neural network, one for each walker.

- tools

  The tools folder contains the python script that form the toolkit part of TATi. They are small command-line programs that accept arguments using the `argparse` package and each perform a very specific task, e.g., executing obtaining a single sampling trajectory.

Moreover, the folder `TATi` itself contains a few special files. There is `common.py` which is a general module where small helper functions are placed that have not found a place anywhere else, i.e. in any proper class.

There is `TrajectoryAnalyser.py` which contains functionality for analysing trajectories.

Last but not least, there is `version.py` which contains helper functions to give the name version and build hash of TATi. Similarly, `TATi.__version__` is contained in *`__init__`*`.py` which is produced from requesting the current version from the git repository using `git describe`.

5.2 Build system

The build systems used the **autootols** suite for dependency generation, testing and packaging. It is the standard build system for all GNU software and has a very long history.

It based on the scripting language *m4* that is a bit outdated from a modern viewpoint.

Note that there are other build systems such as cmake. However, the principal author of TATi at its creation felt that autotools has the superior test system and TATi would be solidly based in the UNIX world and less in Windows environments that are scarcely found in scientific working environments. Cmake needs to make some compromises to fully support the Windows platform, while autotools basically admits the full set of GNU utilities which makes testing especially powerful.

5.2.1 General concept

The build system has the following responsibilities:

- installation into a target folder (`make install`)
- packaging of releases as so-called tarballs (`make dist`)
- testing of the software package (`make check`)
- creation of guides and API documentation (`make doc`)

As you notice, the central tool is (GNU) `make`. The central concept of make are rules. Each rule revolves around a *target*, *dependencies* for the target and a prescription how to use the dependencies to produce the target. All in all this creates a dependency graph through which make automatically recreates targets even when "distant" dependencies have changed.

The autotools suite is then mainly responsible for creating a set of +Makefile+s containing all necessary targets that tell make how to build/test/deploy the package.

Note

Automake focuses on creating packages from C/C++ code. It can be easily extended to be useful for other languages, too. This has been done for Python. However, C needs to be compiled which Python as an interpreted language does not. Therefore, there are some steps that seem superfluous for Python.

5.2.1.1 Automake

Writing `Makefile` s typically can be quite cryptic. It becomes much simpler by using `automake`. In automake one defines sets of specific variables in a file *Makefile.am* from which it then creates a fully fletched `Makefile`.

Let us alook at an example, namely `src/TATi/runtime/Makefile.am`.

```
TATI_runtime_PYTHON = \
        runtime.py \
        __init__.py

TATI_runtimedir = $(pythondir)/TATi/runtime
```

You notice that we simply define two variables, **TATI_runtime_PYTHON** and **TATI_runtimedir**. The first tells automake that it deals with files written in the python language as the variables are suffixed with **PYTHON**. The naming **TATI_runtime** is arbitrary but we want to reflect the module structure as this module would be imported by `import TATI.runtime`. The variable ending in **..dir** tells automake the installation folder.

Automake automatically notes that the files listed in the first variabe also need to go into the tarball when doing a release.

All +Makefile.am+s are written by simply defining specific variables. Most of the problems revolve around finding the correct name.

5.2.1.2 Autoconf

Not all Unix systems are alike. And even like systems such as Linux-based variants tend to differ among each other, e.g., in the way the handle packages.

To this end, autoconf is a tool for setting up variables such as **$(PYTHON)** to contain the path to the python interpreter. Autoconf takes files with suffix `.in` and replaces these variables by their contents.

Let us take a look at `src/TATi/tools/Makefile.am`.

```
do_substitution = sed -e 's,[@]pythondir[@],$(pythondir),g' \
        -e 's,[@]PACKAGE[@],$(PACKAGE),g' \
        -e 's,[@]PYTHON[@],$(PYTHON),g' \
        -e 's,[@]VERSION[@],$(VERSION),g'
```

There you see that we define a *macro* (i.e. a variable being a a function) that will use the command-line program `sed` to replace the placeholder **@PYTHON@** by **$(PYTHON)** which in turn is replaced by the python interpreter that autoconf found.

Autoconf parses the file `configure.ac` in the package's root folder and creates a script called `configure`.

This script contains a bunch of tests that define all these variables. Their contents can be inspected in `build64/config.log` if `build64` is the folder for the out-of-source build, see Installation procedure.

These tests are instantiated by small functions written in the **m4** language that can typically be found in a folder installed alongside with autoconf or on the web.

Let us take a look at `configure.ac` and the part where we check for the python interpreter.

```
AM_PATH_PYTHON([2.5])
```

This calls the macro **AM_PATH_PYTHON** to look for a python interpreter in various folders encoded in the macro that is at least version **2.5**. This macro is contained in the autoconf installation.

The macro can be overridden by the environment variable **PYTHON** which is why

```
configure -C PYTHON=/usr/bin/python3
```

works. Note that these variables should go *after* configure as they are cached (**-C**). Caching speeds up configure quite a lot.

One macro is not contained, the one looking for **doxygen**, see `m4/ac_doxygen.m4`. These need to be placed in the **m4** folder.

5.2.1.3 Autotest

When executing `make check`, then the autotest part of the autotools suite is used. It is not a tool by itself but again a set of macros written in m4 which implement the test driver.

We just would like to gather a few tricks here:

- use `make -j4 check` to execute tests in parallel
- you can execute tests individually: Look for the line above the tests header. This is line is pre- and postfixed by lines looking like this

```
## ------------------------------------------------------------ ##
```

  The line contains a call to `testsuite` along with **$nrjobs**, followed by defining the variable **AUTOTEST_PATH**. Copy that line and replace **$nrjobs** by the following arguments: `-d -k sampler -j4`. This will execute all tests matching the keyword **sampler** (see **AT_KEYWORD** in the test case files) on four processes and leave all output (**-d**).

You can inspect the output of all failed tests by looking at the file `testsuite.dir/01/testsuite.log` if **01** is the number of your test as printed by the `make check` or `testsuite` run. In that folder also all temporary files can be found.

5.2.1.4 Other tools

There are more tools like `aclocal` that need not be of interest here.

5.2.2 Adding new files

Clashes with the build system will occur when trying to add a new file.

In this section we go through the few standard cases one by one.

5.2.2.1 Source file

You should know about the Directory structure by now. Therefore you know where to put your file in the source tree.

However, you also have to tell automake where it belongs.

Look at the `Makefile.am` that should already by present in the folder. IN there, there is a variable ending in **.._PYTHON**. Add it to the list and it will get installed.

If you have added a new folder with a file in it, then a few things need to be done:

- create the init file in the folder: `touch __init__.py`
- create the `Makefile.am`, see Section 5.2.1.1 for an example
- add the folder to the **SUBDIRS** of the `Makefile.am` in the folder below. Otherwise, make will not enter this directory.

5.2.2.2 Tool

Tools residing in the folder **tools** have the special ending **.in** because they still need some variables replaced. Typically, it is **@PYTHON@** and **@pythondir@** required when `tati` is installed at a non-default location.

These files need to be added in `configure.ac`. Look close to the bottom for statements **AC_CONFIG_FILES** that list the tools (without the *.in suffix). The statement causes them to be marked as executable. Furthermore, add the target statement in the `Makefile.am` in `src/TATi/tools` in the same way as for the tools already present.

In case you need to test the new tool, then you have to define a wrapper script. They are located in `tests` and have the same name as the tool itself. Simply look at one of those for an example and adapt the names.

5.2.2.3 Test file

In `tests` there is the following distinction between the various tests:

- integration

  These tests combine several tools or parts of TATi. For example they might sample a trajectory and feed it to the `LossFunctionSamp`
  This ensures that the tools understand the file formats among one another.

- regression

  The regression tests either check for encountered bugs and make sure that these do not occur again. Or they check specific capabilities of TATI, e.g., they make sure that every sampler behaves the same by checking its output against a stored one. This is to ensure that changes in the code do not change the behavior of the package.

- simulation

  Here, we test specific parts of the `simulation` module with examples that will not go in the userguide.

- userguide

  All examples given in the userguide are tested in here one by one.

- tensorflow

  Here, we check for certain tensorflow functionality or rather dysfunctiontality, meaning that certain behavior of tensorflow is non-intuitive. These tests whether the behavior changes with future tensorflow versions which would allow to remove certain boilerplate code in overcoming the unexpected.

Each test cases resides in a unique file `testsuite-....at* where the directory structure also reflects in the name, e.g., +testsuite-userguide-simulation-complex.at`,

These test cases are included in a tree-like structure up till the topmost `testsuite.at` through `m4_include()` statements, see for example `tests/userguide/testsuite.at`.

A general introduction to this *autotest* part of autotools can be found here. Therefore, we do not want to cover its details here.

Basically, the test cases consist of shell commands that wrapped into `AT_CHECK()` statements that take an expected return code (0 - success, else - failure) and whether to capture stdout and stderr (e.g., **[stdout]**) or to ignre them (**[ignore]**).

Typically, we use `grep` to make sure certain output was printed. We use `diff` to compare files and the tool `NumericalDiff` for comparing files allowing for specific numerical variations. Moreover, `sed` can be used to reformat output and `awk` to extract certain elements based on delimiters (e.g., **-F","** for comma-based spitting).

When adding a new test case, make sure to do the following:

- write a new test case file
- properly include the new **testsuite-...at** file in the upper level testsuite file
- add the file to the `Makefile.am`, look for **TESTSCRIPTS**.
- check that your tests is executed, see Section 5.2.1.3

5.2.2.4 Documentation

The documentation is contained `doc/userguide`. See all files ending **.txt** in there.

When adding new files make sure they are properly included in other asciidoc files. If it is a new root file, add it to the **all:** target in +Makefile.am*.

In any case, add it to the variable **DEPENDENCIES** in +Makefile.am*.

If your new piece of documentation also has an example code piece, then put the code into a distinct file and use the `include:::[]` statement of asciidoc. Add the filename to the variable to the variable **PYTHON_EXAMPLES** in +Makefile.am*.

Then, you need to create a test case. Look at one of the already present test cases in `tests/userguide`. These make use of a `run` script that invokes python with the correct **PYTHONPATH** such that the module `tati` can be found even when installed on non-default locations.

5.2.2.5 Other files

If the added file is none of the above but it should still go into the release tarball, then add it to the **EXTRA_DIST** variable or add a statement such as `EXTRA_DIST = foo.bar` to the folder's `Makefile.am` if your file is called **foo.bar**.

5.3 Version control

TATi's source code is maintained with git as its version control system. The "central" repository is located at GitHub.

Git records changes to the source code in *commits*. Each commit forms a node in a graph. A commit is uniquely identified by its *hash* (a 7-digit hex code).

Development should usually commence in a distinct *branch* that can be easily created by `git checkout -b DevBranch` with any name in place for **DevBranch**. The top of a branch is referred to as head and is the most current commit to this branch.

Important
The branch `master` should only be modified by the package maintainer and is basically only advanced onto the next release branch, see below.

5.3.1 Release policy

Releases should be frequent and not too large.

To this end, a branch `Candidate_vVERSION` is checked out from the current `master` branch where **VERSION** is the upcoming version number, e.g., 0.9. All development branches that are fit for release are merged using `git merge --no-ff DevBranch` (if **DevBranch** is the branch's name) such that the original branch name is encoded in the merge commit message.

Conflicts have to be resolved in the merge. Note that the test policy also and especially applies to this candidate branch.

At the very end, we need to test whether `make dist` produces a usable tarball, i.e. whether unpacking, `configure`, `make && make install && make check` runs flawlessly.

Finally, the version number **VERSION** is placed into the **AC_INIT** statement in `configure.ac` and this should always be done in a single commit not containing any other changes. This last commit is tagged using `git tag -a vVERSION` and the tag message should follow the example below.

```
Version VERSION contains the following branches:
HASH  Candidate_vVERSION
HASH1 BranchName1
HASH2 BranchName2
```

We put a list of all merged branches including the hash value of their respective head. This way even when branches are deleted but have been merged into **master**, then they can still be tracked.

Subsequently, all development branches are deleted from the central repository using `git branch -D DevBranch` (again assuming the branch's name to be **DevBranch**).

5.3.2 Test policy

Using the version control system we retain a strict test policy in the following way: **For every commit `make && make install && make check` needs to run flawlessly.**

This ensures that the userguide compiles and TATi's code complies with the result expected in every test.

If this should not be possible **within development branches**, then test cases can be marked as *expected to fail (XFAIL)* **temporarily** by adding the statement

```
AT_XFAIL_IF($BIN_TRUE)
```

just after the **AT_KEYWORDS** statement. By the very latest these issues must be resolved till the next release.

It is generally advised to modify commits such that adhere to the above test policy. The commit history can be partially rewritten using `git rebase`.

As an extreme measure, i.e. when too many commits would be affected, the tests can be deactivated **temporarily** by removing the respective test folder from the **SUBDIRS** statements in `tests/Makefile.am`.

Chapter 6

Tensorflow Flaws

It may be thought a little too much to dedicate a whole chapter to the potential flaws in a framework that forms the basis of this software. However, this particular framework has given me so much hardship and failed in such unexpected ways that I have to make a list.

Note that this list is up-to-date with respect to the tests employed in the code and is currently focused at tensorflow version 1.4

Most of issues simply made it hard to have reproducible runs which made it diffult to maintain my testsuite. Moreover, most of them are made on purpose for the sake of speed over deterministic behavior.

- Non-deterministic `reduce_sum`

  See the issue at Github. Non-deterministic is obviously faster than deterministic, so that's what they are going for. Sadly, no determinstic alternative for calculating norms of 1-dimensional tensors or scalar products is offered. This is very hurtful for reproducibility. The current workaround is to set `inter_op_thread` to one, eliminating any use of multiple cores.

- `tf.set_global_seed` not useful

  This is working as intended: it sets the global seed in such a way that all operations requiring randomness derive their seed in a deterministic fashion from it. And this is valid as long as it is *exactly* the same graph. If just a single node is added that does not even need to be relevant for the operation, all seeds will change because the derivation of seeds probably depends on some random order of nodes and not on the name of the node or any other unique property.

- `tf.float64` is flawed

  I encountered issues with precision when ascertaining theoretical properties of the samplers. One remedy I though might solve the issue was to switch from tf.float32 to tf.float64, i.e. from 1e-8 to 1e-18 internal floating point precision.

  What I found was that suddenly I could not recover the theoretical properties any more. Even simple sampling (i.e. central limit theorem and expected convergence rates of $\frac{1}{sqrt(n)}$ would not bring up slopes of -0.5 as expected in log-log plots but also -0.4.

  I went to great length to check that all values are tf.float64. If I had forgotten one, either the internal type checker would admonish it, or the precision should just be the one I had with tf.float32. However, the quality of the values had changed. My only guess is that there must be some weird bug hidden deep in the C parts of the tensorflow code.

- Parsing from CSV file despite caching tenfold slower

  With tensorflow 1.4 the Dataset module arrived (no longer being "contrib") and I happily switched to this as means of constructing my input pipeline. So far, I had just been looking at small test datasets which fit in memory without issues. As the datasets were so small, I did not expect any much slowing down of my code switching to parsing CSV files and feeding them.

  However, both the old "queues" input pipeline and the new "Dataset" pipeline (the latter even with caches) experienced a tenfold decrease of runtime with respect to in/memory.

  I must admit though that the Dataset module at least made it possible to let the user decide between in-memory storing and file parsing.

- `tf.if` conditional working in funny way

  For the Hamiltonian Monte Carlo sampler an "if" block is required inside the gradient evaluation that decides on whether the current short trajectory run using Hamiltonian Dynamics is accepted or rejected. When I tried make this work, I failed utterly, until I hit this answer on stackoverflow. In a comment even one from the tensorflow team admits that he finds this behavior confusing.

- Shuffling (in queues) shuffles over all repeated datasets causing duplicate items.

  I gues this is for speed reasons as well but it is really a pain in the arse. I guess the reason is a reshuffled dataset in every epoch, hence the reshuffle over all repeated sets instead of reshuffling at the start of the epoch. Probably it is simpler to implement with really large datasets in multiple files.

  However, for small datasets suddenly your gradients change (not using mini-batches) because one element is missing as another is in the set twice. Again, bad for reproducibility.

- `tf.concat` dropping variable character

  This is more of a nuisance but a painful one that has quite strong effects on the efficiency of the **simulations** interface part: You cannot simply concatenate four variable tensors and then set them all using a single placeholder of the right dimension as the "variable" character is lost in the concatenation. It can only be read, see stackoverflow. See also this related issue.

Chapter 7

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].

Chapter 8

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