climin Documentation

Release 0.1

climin developers

January 29, 2017

Contents

1 introduction **1**
- 1.1 Starting points . . . 1
- 1.2 Contact & Community . . . 2

2 Meta **3**
- 2.1 Manifest . . . 3
- 2.2 Installation . . . 3

3 Basics **5**
- 3.1 Tutorial . . . 5
- 3.2 Dealing with parameters . . . 9
- 3.3 Advanced data handling . . . 10
- 3.4 Interrupting and Resuming via Checkpoints . . . 11

4 Reference **13**
- 4.1 Optimizer overview . . . 13
- 4.2 Convenience, Utilities . . . 26

5 Indices and tables **33**

Bibliography **35**

Python Module Index **37**

CHAPTER 1

introduction

Optimization is a key ingredient in modern machine learning. While many models can be optimized in specific ways, several need very general gradient based techniques–e.g. neural networks. What's even worse is that you never know whether your model does not work or you just have not found the right optimizer.

This is where climin comes in. We offer you many different off-the-shelf optimizers, such as LBFGS, stochastic gradient descent with Nesterov momentum, nonlinear conjugate gradients, resilient propagation, rmsprop and more.

But what is best, is that we know that optimization is a process that needs to be analyzed. This is why climin does not offer you a black box function call that takes ages to run and might give you a good minimum of your training loss. Since this is not what you care about, climin takes you with you on its travel through the error landscape... in a classic for loop:

```
import climin

network = make_network()                  # your neural network
training_data = load_train_data()         # your training data
validation_data = load_validation_data()  # your validation data
test_data = load_test_data()              # your testing data

opt = climin.Lbfgs(network.parameters,
                   network.loss,
                   network.d_loss_d_parameters,
                   args=itertools.repeat((training_data, {})))

for info in opt:
    validation_loss = network.loss(network.parameters, validation_data)
    print info['loss'], validation_loss

print network.loss(test_data)
```

Climin works on the CPU (via numpy and scipy) and in parts on the GPU (via gnumpy).

1.1 Starting points

If you want to see how climin works and use climin asap, check out the Tutorial. Details on Installation are available. A list of the optimizers implemented can be found in the overview below. If you want to understand the design decisions of climin, read the Manifest.

1.2 Contact & Community

Climin was started by people from the Biomimetic robotics and machine learning group at the Technische Universitaet Muenchen. If you have any questions, there is a mailinglist at climin@librelist.com. Just send an email to subscribe or checkout the archive. The software is hosted over at our github repository, where you can also find our issue tracker.

CHAPTER 2

Meta

2.1 Manifest

Climin was started in the beginning with the observation that plain stochastic gradient descent was not able to find good solutions for sparse filtering *[sparsefiltering]*. The original article mentions the use of LBFGS as an optimization method, yet the implementation offered by scipy did not solve the problem for us immediately. Since matlab has a very powerful optimization library, we decided that it was time for Python to catch up in this respect.

We found several requirements for a good optimization library.

- Optimization in machine learning is mostly accompanied by online evaluation code: live plotting of error curves, parameters or sending you an email once your model has beaten the current state of the art. Also, you might have your own stopping criterions. We call this the "side logic" of an optimization. Every user has his own way of dealing with this side logic, and a lot of throw away code is being written for this. We wanted to make this part as easy as possible for the user.
- Do one thing, and do that right: climin is independent of your models and the way you work with optimization. We have a simple protocol: loss functions (and their derivatives) and a parameter array. Also, we do not force any framework on you on and come up with things that try to solve everything.
- Most of the optimizers, i.e. those that do not rely on too much linear algebra such as matrix inversions, should not only work on the CPU via numpy but also on the GPU via gnumpy.
- Optimizers should be easily switchable; if we have a model and a loss we wanted to be able to quickly experiment with different methods.
- Optimizers should be reasonably fast. Most of the computational work in machine learning is done within the models anyway. Yet, we want a clean python code base without C extensions. We also found that speeding up everything with Cython would be a good way to go where necessary. Since numba is around the corner, we wanted to decide this in a later version.
- Make development of new optimizers straight forward. The implementation of every optimizer has very little overhead, the most of it being assigning hyper parameters to class values.

The main idea of climin is to treat optimizers as iterators. This allows to have the logic surrounding it right in the same scope and written code block as the optimization. Also, callbacks are really ugly! Python has better tools for that.

2.2 Installation

Climin has been tested on Python 2.7 with numpy 1.8 and scipy 0.13. Currently, it is only available via git from github:

```
$ git clone https://github.com/BRML/climin.git
```

After that, pip can be used to install it on the Python path:

```
$ cd climin
$ pip install .
```

If you want to know whether everything works as we expect it, run the test suite:

```
$ nosetests tests/
```

for which nose is required.

CHAPTER 3

Basics

3.1 Tutorial

In the following we will explain the basic features of climin with a simple example. For that we will first use a multinomial logistic regression, which suffices to show much of climin's functionality.

Although we will use numpy in this example, we have spent quite some effort to make most optimizers work with gnumpy as well. This makes the use of GPUs possible. Check the reference documentation for specific optimizers whether the usage of GPU is supported.

3.1.1 Defining a Loss Function

At the heart of optimizitation lies the objective function we wish to optimize. In the case of climin, we will *always* be concernced with minimization. Even though algorithms are sometimes defined with respect to maximization (e.g.i Bayesian optimization or evolution strategies) in the literature. Thus, we will also be talking about loss functions.

A loss function in climin follows a simple protocol: a callable (e.g. a function) which takes an array with a single dimension as input and returns a scalar. An example would be a simple polynomial of degree 2:

```
def loss(x):
    return (x ** 2).sum()
```

In machine learning, this will mostly be the parameters of our model. Additionally, we will often have further arguments to the loss, the most important being the data that our model works on.

3.1.2 Logistic Regression

Optimizing a scalar quadratic with an iterative technique is all nice, but we want to do more complicated things. We will thus move on to use logistic regression.

In climin, the parameter vector will always be one dimensional. Your loss function will have to unpack that vector into the various parameters of different shapes. While this might seem tedious at first, it makes some calculations much easier and also more efficient.

Logistic regression has commonly two different parameter sets, the weight matrix (or coefficients) and the bias (or intercept). To unpack the parameters we define the following function:

```
import numpy as np

def unpack_parameters(pars):
    w = pars[:n_inpt * n_output].reshape((n_inpt, n_output))
```

```
    b = pars[n_inpt * n_output:].reshape((1, n_output))
    return w, b
```

We assume that inputs to our model will be an array of size `(n, d)`, where `n` refers to the number of samples and `d` to the dimensionality. Given some input we can then make predictions with the following function:

```
def predict(parameters, inpt):
    w, b = unpack_parameters(parameters)
    before_softmax = np.dot(inpt, w) + b
    softmaxed = np.exp(before_softmax - before_softmax.max(axis=1)[:, np.newaxis])
    return softmaxed / softmaxed.sum(axis=1)[:, np.newaxis]
```

For multiclass classification, we use the cross entropy loss:

```
def loss(parameters, inpt, targets):
    predictions = predict(parameters, inpt)
    loss = -np.log(predictions) * targets
    return loss.sum(axis=1).mean()
```

Gradient-based optimization requires not only the loss but also the first derivative with respect to the parameters. That gradient function has to return the gradients aligned with the parameters, which is why we concatenate them into a big array after we flattened out the weight matrix:

```
def d_loss_wrt_pars(parameters, inpt, targets):
    p = predict(parameters, inpt)
    g_w = np.dot(inpt.T, p - targets) / inpt.shape[0]
    g_b = (p - targets).mean(axis=0)
    return np.concatenate([g_w.flatten(), g_b])
```

Although this implementation can be optimized with no doubt, it suffices for this tutorial.

3.1.3 An Array for the Parameters

First we will need to allocate a region of memory where our parameters live. Climin tries to allocate as little additional memory as possible and will thus work inplace most of the time. After each optimization iteration, the current solution will always be in the array we created. This lets the user control as much as possible. We create an empty array for our solution:

```
import numpy as np
wrt = np.empty(7850)
```

where the `7850` refers to the dimensionality of our problem. We picked this number because we will be tackling the MNIST data set. It makes sense to initialize the parameters randomly (depending on the problem), even though the convexity of logistic regressions guarantees that we will always find the minimum. Climin offers convenience functions in its `initialize` module:

```
import climin.initialize
climin.initialize.randomize_normal(wrt, 0, 1)
```

This will populated the parameters with values drawn from a standard normal dostribution.

3.1.4 Using data

Now that we have set up our model and loss and initialized the parameters, we need to manage the data.

In climin, we will always look at streams of data. Even if we do batch learning (as we do here), the recommended way of doing so is a repeating stream of the same data. How does that stream look? In Python, we have a convenient data structure which is the iterator. It can be thought of as a lazy list of infinite length.

The climin API expects that the loss function (and the gradient function) will accept the parameter array as the first argument. All further arguments can be as the user wants. When we initialize an optimizer, a keyword argument `args` can be specified. This is expected to be an iterator which yields pairs of `(a, kw)` which are then passed to the loss as `f(parameters, *a, **kw)` and `fprime(parameters, *a, *kw)` in case of the derivative.

We will be using the MNIST data set , which can be downloaded from here. We will first load it and convert the target variables to a one-of-k representation, which is what our loss functions expect:

```
# You can get this at http://www.iro.umontreal.ca/~lisa/deep/data/mnist/mnist.pkl.gz

datafile = 'mnist.pkl.gz'
# Load data.
with gzip.open(datafile,'rb') as f:
    train_set, val_set, test_set = cPickle.load(f)

X, Z = train_set
VX, VZ = val_set
TX, TZ = test_set

def one_hot(arr):
    result = np.zeros((arr.shape[0], 10))
    result[xrange(arr.shape[0]), arr] = 1.
    return result

Z = one_hot(Z)
VZ = one_hot(VZ)
TZ = one_hot(TZ)
```

To create our data stream, we will just repeat the training data `(X, Z)`:

```
import itertools
args = itertools.repeat(([X, Z], {}))
```

This certainly seems like overkill for logistic regression. Yet, even this simple model can often be sped up by estimating the gradients on "mini batches". Going even further, you might want to have a continuous stream that is read from the network, a data set that does not fit into RAM or which you want to transform on the fly. All these things can be elegantly implemented with iterators.

3.1.5 Creating an Optimizer

Now that we have set everything up, we are ready to create our first optimizer, a `GradientDescent` object:

```
import climin
opt = climin.GradientDescent(wrt, d_loss_wrt_pars, step_rate=0.1, momentum=.95, args=args)
```

We created a new object called `opt`. For initialization, we passed it several parameters:

- The parameters `wrt`. This will *always* be the first argument to any optimizer in climin.
- The derivative `d_loss_wrt_pars`; we do not need `loss` itself for gradient descent.
- A scalar to multiply the negative gradient with for the next search step, `step_rate`. This parameter is often referred to as learning rate in the literature.
- A momentum term `momentum` to speed up learning.

- Our data stream `args`.

The parameters `wrt` and `args` are consistent over optimizers. All others may vary wildly, according to what an optimizer expects.

3.1.6 Optimization as Iteration

Many optimization algorithms are iterative and so are all in climin. To transfer this metaphor into programming code, optimization with climin is as simple as iterating over our optimizer object:

```
for i in opt:   # Infinite loop!
    pass
```

This will result in an infinite loop. Climin does not handle stopping from within optimizer objects; instead, you will have to do it manually, since you know it much better. Let's iterate for a fixed number of iterations, say 100:

```
print loss(wrt, VX, VZ)   # prints something like 2.49771627484
for info in opt:
    if info['n_iter'] >= 100:
        break
print loss(wrt, VX, VZ)   # prints something like 0.324243334583
```

When we iteratore over the optimizer, we iterate over dictionaries. Each of these contains various information about the current state of the optimizer. The exact contents depend on the optimizer, but might contain the last step, gradient, etc. Here, we check the number of iterations that have already been performed.

3.1.7 Using different optimizers

The whole point of climin is to use different optimizers. How that goes, we will explain now. We have already seen Gradient Descent. Furthermore, there are Quasi-Newton (BFGS, L-BFGS), Conjugate Gradients, Resilient Propagation and rmsprop. Let's see how we can use each of them.

L-BFGS, RPROP and nonlinear conjugate Gradients all have the benefit that they work reasonably well without too much tuning of their hyper parameters. We can thus construct optimizers like this:

```
ncg = climin.NonlinearConjugateGradient(wrt, loss, d_loss_wrt_pars, args=args)
lbfgs = climin.Lbfgs(wrt, loss, d_loss_wrt_pars, args=args)
rprop = climin.Rprop(wrt, d_loss_wrt_pars, args=args)
rmsprop = climin.RmsProp(wrt, d_loss_wrt_pars, steprate=1e-4, decay=0.9, args=args)
```

As you can see, we now need to specify the loss function itself in case of `Lbfgs` and `NonlinearConjugateGradient`. That is because both utilize a line search after finding a search direction. `climin.RmsProp` has more hyper parameters and needs more fine grained tuning. Yet, `climin.GradientDescent` and `climin.RmsProp` work naturally with so a stochastic estimate of the objective and work very well with mini batches. For more details, see Advanced data handling.

3.1.8 Conclusion and Next Steps

This tutorial explained the basic functionality of climin. There is a lot more to explore to fully leverage the functionality of this library. Check the table of contents and the examples directory of your climin checkout.

3.2 Dealing with parameters

Parameters in climin are a long and one dimensional array. This might seem as a restriction at first, yet it makes things easier in other places. Consider a model involving complicated array dimensionalities; now consider how higher order derivatives of those might look like. Yes that's right, a pretty messy thing. Furthermore, letting paramters occupy consecutive regions of memory has further advantages from an implementation point of view. We can easier write it to disk, randomize its contents or similar things.

3.2.1 Creating parameter sets

Creating of parameter arrays need not be tedious, though. Climin comes with a nice convenience function, `climin.util.empty_with_views` which does most of the work. You just need to feed it the shapes of parameters you are interested in.

Let us use logistic regressiom from the Tutorial and see where it comes in handy. First, we will create a parameter array and the various views according to a template:

```
import numpy as np
import climin.util

tmpl = [(784, 10), 10]          # w is matrix and b a vector
flat, (w, b) = climin.util.empty_with_views(tmpl)
```

Now, `flat` is a one dimensional array. `w` and `b` are a two dimensional and a one dimensional array respectively. They share memory with `flat`, so any change we will do in `w` or `b` will be reflected in `flat` and vice versa. In order for a predict function to get the parameters out of the flat array, there is `climin.util.shaped_from_flat` which does the same job as `empty_with_views`, except that it receives `flat` and does not create it. In fact, the latter uses the former internally.

Let's adapt the `predict` function to use `w` and `b` instead:

```
def predict(parameters, inpt):
    w, b = climin.util.shaped_from_flat(parameters, tmpl)
    before_softmax = np.dot(inpt, w) + b
    softmaxed = np.exp(before_softmax - before_softmax.max(axis=1)[:, np.newaxis])
    return softmaxed / softmaxed.sum(axis=1)[:, np.newaxis]
```

This might seem like overkill for logistic regression, but becomes invaluable when complicated models with many different parameters are used.

3.2.2 Calculating derivatives in place

When calculating derivatives, you can make use of this as well–which is important because climin expects derivatives to be flat as well, nicely aligned with the parameter array:

```
def f_d_loss_wrt_pars(parameters, inpt, targets):
    p = predict(parameters, inpt)
    d_flat, d_w, d_b = climin.util.empty_with_views(tmpl)
    d_w[...] = np.dot(inpt.T, p - targets) / inpt.shape[0]
    d_b[...] = (p - targets).mean(axis=0)
    return d_flat
```

What are we doing here? First, we get ourselves a new array and preshaped views on it in the same way as the parameters. Then we overwrite the views in place with the derivatives and finally return the flat array as a result. The in place assignment is important. If we did it using `d_w = ...`, Python would just reassign the name and the changes would not turn up in `d_flat`.

As a further note, `np.dot` supports an extra argument `out` which specifies where to write the result. To safe memory, we could perform the following instead:

```
np.dot(inpt.T, p - targets, out=d_w)
d_w  /= inpt.shape[0]
```

3.2.3 Initializing parameters

Initializing parameters with empty values is asking for trouble. You probably want to populate an array with random numbers or zeros. Of course it is easy to do so by hand:

```
flat[...] = np.random.normal(0, 0.1, flat.shape)
```

We found this quite tedious to write though; especially as soon as flat becomes the field of a nested object. Thus, we have a short hand in the initialize module which does exaclty that:

```
import climin.initialize
climin.initialize.randomize_normal(flat, 0, 0.1)
```

There are more functions to do similar things. Check out Initialization of Parameters.

3.3 Advanced data handling

We have already seen how to use batch learning in the Tutorial. The use of `itertools.repeat` to handle simple batch learning seemed rather over expressive; it makes more sense if we consider more advanced use cases.

3.3.1 Mini batches

Consider the case where we do not want to calculcate the full gradient in each iteration but estimate it given a mini batch. The contract of climin is that each item of the `args` iterator will be used in a single iteration and thus for one parameter update. The way to enable mini batch learning is thus to provide an `args` argument which is an infinite list of mini batches.

Let's revisit our example of logistic regression. Here, we created the args iterator using `itertools.repeat` on the same array again and again:

```
import itertools
args = itertools.repeat(([X, Z], {}))
```

What we want to do now is to have an infinite stream of slices of `X` and `Z`. How do we access the n'th batch of `X` and `Z`? We offer you a convenience function that will give you random (with or without replacement) slices from a container:

```
batch_size = 100
args = ((i, {}) for i in climin.util.iter_minibatches([train_set_x, train_set_y], batch_size, [0, 0]))
```

The last argument, `[0, 0]` gives the axes along which to slice `[X, Z]`. In some cases, samples might be aligned along axis `0` for the input, but along axis `1` in the target data.

3.3.2 External memory

What is nice about `climin.util.iter_minibatches` is that it needs only slices as a requirement for its arguments. We therefore only need to pass it a data structure which reads data from disk as soon as it is needed and disposes of it as soon as it is not any more.

HDF5 and its python package h5py are a perfect match for this. We have managed to use 6+ GB sized image data sets on GPUs with less than 2 GB of RAM with this simple recipe:

```
import climin.util
import gnumpy
import h5py

f = h5py.File('data.h5')
ds = f['inpts']
args = climin.util.iter_minibatches([ds], 100, [0])
args = (gnumpy.garray(i) for i in args)

# ...
```

This is in general not restricted by the size of the data set; it just show that going beyond the GPU RAM limit is achieved very naturally in climin.

3.3.3 Further usages

This architecture can be exploited in many different ways. E.g., a stream over a network can be directly used. A single pass over a file without keeping more than necessary is another option.

3.4 Interrupting and Resuming via Checkpoints

It is important to be able to interrupt optimization and continue right from where you left off. Reasons include scheduling on shared resources, branching the optimization with different settings or securing yourself against crashes in long-running processes.

Note: Currently, this is not supported by all optimizers. It is the case for gradient descent, rmsprop, adadelta and rprop.

Climin makes this in parts possible and leaves the responsibility to the user in other parts. More specifically, the user has to take over the serialization of the parameter vector (i.e. `wrt`), the objective function and its derivatives (e.g. `fprime`) and the data (i.e. `args`). The reason for this is that one cannot build a generic procedure for this. The data might be depending on an open file descriptor and only a subset of Python functions can be serialized, which is those that are defined at the top level.

3.4.1 Saving the state to disk (or somewhere else)

The idea is that the `info` dictionary which is the result of each optimization step carries all the information necesseray to resume. Thus a recipe to write your state to disk is as follows.

```
import numpy as np
import cPickle
from climin.gd import Gradient Descent

pars = make_pars()
fprime = make_fprime()
data = make_data()
opt = GradientDescent(pars, fprime, args=data)
for info in opt:
    with open('state.pkl', 'w') as fp:
```

```
        cPickle.dump(info, fp)
    np.savetxt('parameters.csv', pars)
```

This snippet first generates the necessery quantities from library functions which we assume given. We then create a `GradientDescent` object over which we iterate to optimize. In each iteration, we pickle the info dictionary to disk.

Note: Pickling an info dictionary directly to disk might be a bad idea in many cases. E.g. it will contain the current data element or a gnumpy array, which is not picklable. It is the users's responsibility to take care of that.

3.4.2 Loading the state from disk

We will now load the info dictionary from file, create an optimizer object an initialize it with values from the info dictionary.

```
import numpy as np
import cPickle
from climin.gd import Gradient Descent

pars = np.loadtxt('parameters.csv')
fprime = make_fprime()
data = make_data()
with open('state.pkl') as fp:
    info = cPickle.load(fp)

opt = GradientDescent(pars, fprime, args=data)
opt.set_from_info(info)
```

We can continue optimization right from where we left off.

CHAPTER 4

Reference

4.1 Optimizer overview

4.1.1 Gradient Descent

This module provides an implementation of gradient descent.

class `climin.gd.`**`GradientDescent`**(*wrt*, *fprime*, *step_rate=0.1*, *momentum=0.0*, *momentum_type='standard'*, *args=None*)

Classic gradient descent optimizer.

Gradient descent works by iteratively performing updates solely based on the first derivative of a problem. The gradient is calculated and multiplied with a scalar (or component wise with a vector) to do a step in the problem space. For speed ups, a technique called "momentum" is often used, which averages search steps over iterations.

Even though gradient descent is pretty simple it can be very effective if well tuned (in terms of its hyper parameters step rate and momentum). Sometimes the use of schedules for both parameters is necessary. See `climin.schedule` for basic schedules.

Gradient descent is also very robust to stochasticity in the objective function. This might result from noise injected into it (e.g. in the case of denoising auto encoders) or because it is based on data samples (e.g. in the case of stochastic mini batches.)

Given a step rate α and a function f' to evaluate the search direction the current paramters θ_t the following update is performed:

$$v_{t+1} = \alpha f'(\theta_t)$$
$$\theta_{t+1} = \theta_t - v_{t+1}.$$

If we also have a momentum β and are using standard momentum, we update the parameters according to:

$$v_{t+1} = \alpha f'(\theta_t) + \beta v_t$$
$$\theta_{t+1} = \theta_t - v_{t+1}$$

In some cases (e.g. learning the parameters of deep networks), using Nesterov momentum can be beneficial. In this case, we first make a momentum step and then evaluate the gradient at the location in between. Thus, there is an additional cost of an addition of the parameters.

$$\theta_{t+\frac{1}{2}} = \theta_t - \beta v_t$$
$$v_{t+1} = \alpha f'(\theta_{t+\frac{1}{2}})$$
$$\theta_{t+1} = \theta_t - v_{t+1}$$

which can be specified additionally by the initialization argument `momentum_type`.

Note: Works with gnumpy.

Attributes

wrt	(array_like) Current solution to the problem. Can be given as a first argument to `.fprime`.
fprime	(Callable) First derivative of the objective function. Returns an array of the same shape as `.wrt`.
step_rate	(float or array_like) Step rate to multiply the gradients with.
momentum	(float or array_like) Momentum to multiply previous steps with.
momentum_type	(string (either "standard" or "nesterov")) When to add the momentum term to the parameter vector; in the first case it will be done after the calculation of the gradient, in the latter before.

Methods

__init__ (*wrt*, *fprime*, *step_rate=0.1*, *momentum=0.0*, *momentum_type='standard'*, *args=None*)

Create a GradientDescent object.

Parameters **wrt** : array_like

Array that represents the solution. Will be operated upon in place. `fprime` should accept this array as a first argument.

fprime : callable

Callable that given a solution vector as first parameter and *args and **kwargs drawn from the iterations `args` returns a search direction, such as a gradient.

step_rate : float or array_like, or iterable of that

Step rate to use during optimization. Can be given as a single scalar value or as an array for a different step rate of each parameter of the problem.

Can also be given as an iterator; in that case, every iteration of the optimization takes a new element as a step rate from that iterator.

momentum : float or array_like, or iterable of that

Momentum to use during optimization. Can be specified analoguously (but independent of) step rate.

momentum_type : string (either "standard" or "nesterov")

When to add the momentum term to the paramter vector; in the first case it will be done after the calculation of the gradient, in the latter before.

args : iterable

Iterator of arguments which `fprime` will be called with.

4.1.2 rmsprop

This module provides an implementation of rmsprop.

class climin.rmsprop.**RmsProp**(*wrt*, *fprime*, *step_rate*, *decay=0.9*, *momentum=0*, *step_adapt=False*, *step_rate_min=0*, *step_rate_max=inf*, *args=None*)

RmsProp optimizer.

RmsProp *[tieleman2012rmsprop]* is an optimizer that utilizes the magnitude of recent gradients to normalize the gradients. We always keep a moving average over the root mean squared (hence Rms) gradients, by which we divide the current gradient. Let $f'(\theta_t)$ be the derivative of the loss with respect to the parameters at time step t. In its basic form, given a step rate α and a decay term γ we perform the following updates:

$$\begin{aligned} r_t &= (1-\gamma)\ f'(\theta_t)^2 + \gamma r_{t-1}, \\ v_{t+1} &= \frac{\alpha}{\sqrt{r_t}} f'(\theta_t), \\ \theta_{t+1} &= \theta_t - v_{t+1}. \end{aligned}$$

In some cases, adding a momentum term β is beneficial. Here, Nesterov momentum is used:

$$\begin{aligned} \theta_{t+\frac{1}{2}} &= \theta_t - \beta v_t, \\ r_t &= (1-\gamma)\ f'(\theta_{t+\frac{1}{2}})^2 + \gamma r_{t-1}, \\ v_{t+1} &= \beta v_t + \frac{\alpha}{\sqrt{r_t}} f'(\theta_{t+\frac{1}{2}}), \\ \theta_{t+1} &= \theta_t - v_{t+1} \end{aligned}$$

Additionally, this implementation has adaptable step rates. As soon as the components of the step and the momentum point into the same direction (thus have the same sign) the step rate for that parameter is multiplied with `1 + step_adapt`. Otherwise, it is multiplied with `1 - step_adapt`. In any way, the minimum and maximum step rates `step_rate_min` and `step_rate_max` are respected and exceeding values truncated to it.

RmsProp has several advantages; for one, it is a very robust optimizer which has pseudo curvature information. Additionally, it can deal with stochastic objectives very nicely, making it applicable to mini batch learning.

Note: Works with gnumpy.

Attributes

wrt	(array_like) Current solution to the problem. Can be given as a first argument to `.fprime`.
fprime	(Callable) First derivative of the objective function. Returns an array of the same shape as `.wrt`.
step_rate	(float or array_like) Step rate of the optimizer. If an array, means that per parameter step rates are used.
momentum	(float or array_like) Momentum of the optimizer. If an array, means that per parameter momentums are used.
step_adapt	(float or bool) Constant to adapt step rates. If False, step rate adaption is not done.
step_rate_min	(float, optional, default 0) When adapting step rates, do not move below this value.
step_rate_max	(float, optional, default inf) When adapting step rates, do not move above this value.

Methods

__init__(*wrt*, *fprime*, *step_rate*, *decay=0.9*, *momentum=0*, *step_adapt=False*, *step_rate_min=0*, *step_rate_max=inf*, *args=None*)

Create an RmsProp object.

Parameters **wrt** : array_like

Array that represents the solution. Will be operated upon in place. `fprime` should accept this array as a first argument.

fprime : callable

Callable that given a solution vector as first parameter and *args and **kwargs drawn from the iterations `args` returns a search direction, such as a gradient.

step_rate : float or array_like

Step rate to use during optimization. Can be given as a single scalar value or as an array for a different step rate of each parameter of the problem.

decay : float

Decay parameter for the moving average. Must lie in [0, 1) where lower numbers means a shorter "memory".

momentum : float or array_like

Momentum to use during optimization. Can be specified analoguously (but independent of) step rate.

step_adapt : float or bool

Constant to adapt step rates. If False, step rate adaption is not done.

step_rate_min : float, optional, default 0

When adapting step rates, do not move below this value.

step_rate_max : float, optional, default inf

When adapting step rates, do not move above this value.

args : iterable

Iterator over arguments which `fprime` will be called with.

4.1.3 Adadelta

This module provides an implementation of adadelta.

class `climin.adadelta.`**Adadelta**(*wrt*, *fprime*, *step_rate=1*, *decay=0.9*, *momentum=0*, *offset=0.0001*, *args=None*)

Adadelta optimizer.

Adadelta *[zeiler2013adadelta]* is a method that uses the magnitude of recent gradients and steps to obtain an adaptive step rate. An exponential moving average over the gradients and steps is kept; a scale of the learning rate is then obtained by their ration.

Let $f'(\theta_t)$ be the derivative of the loss with respect to the parameters at time step t. In its basic form, given a step rate α, a decay term γ and an offset ϵ we perform the following updates:

$$g_t = (1 - \gamma)\ f'(\theta_t)^2 + \gamma g_{t-1}$$

where $g_0 = 0$. Let $s_0 = 0$ for updating the parameters:

$$\Delta\theta_t = \alpha \frac{\sqrt{s_{t-1} + \epsilon}}{\sqrt{g_t + \epsilon}} f'(\theta_t),$$
$$\theta_{t+1} = \theta_t - \Delta\theta_t.$$

Subsequently we adapt the moving average of the steps:

$$s_t = (1 - \gamma) \Delta\theta_t^2 + \gamma s_{t-1}.$$

To extend this with Nesterov's accelerated gradient, we need a momentum coefficient β and incorporate it by using slightly different formulas:

$$\theta_{t+\frac{1}{2}} = \theta_t - \beta\Delta\theta_{t-1},$$
$$g_t = (1 - \gamma) f'(\theta_{t+\frac{1}{2}})^2 + \gamma g_{t-1},$$
$$\Delta\theta_t = \alpha \frac{\sqrt{s_{t-1} + \epsilon}}{\sqrt{g_t + \epsilon}} f'(\theta_{t+\frac{1}{2}}).$$

In its original formulation, the case $\alpha = 1, \beta = 0$ was considered only.

Methods

__init__ (*wrt*, *fprime*, *step_rate=1*, *decay=0.9*, *momentum=0*, *offset=0.0001*, *args=None*)
Create an Adadelta object.

Parameters **wrt** : array_like

Array that represents the solution. Will be operated upon in place. `fprime` should accept this array as a first argument.

fprime : callable

Callable that given a solution vector as first parameter and *args and **kwargs drawn from the iterations `args` returns a search direction, such as a gradient.

step_rate : scalar or array_like, optional [default: 1]

Value to multiply steps with before they are applied to the parameter vector.

decay : float, optional [default: 0.9]

Decay parameter for the moving average. Must lie in [0, 1) where lower numbers means a shorter "memory".

momentum : float or array_like, optional [default: 0]

Momentum to use during optimization. Can be specified analoguously (but independent of) step rate.

offset : float, optional, [default: 1e-4]

Before taking the square root of the running averages, this offset is added.

args : iterable

Iterator over arguments which `fprime` will be called with.

4.1.4 Adam

This module provides an implementation of Adam.

class `climin.adam.`**Adam**(*wrt*, *fprime*, *step_rate=0.0002*, *decay=None*, *decay_mom1=0.1*, *decay_mom2=0.001*, *momentum=0*, *offset=1e-08*, *args=None*)

Adaptive moment estimation optimizer. (Adam).

Adam is a method for the optimization of stochastic objective functions.

The idea is to estimate the first two moments with exponentially decaying running averages. Additionally, these estimates are bias corrected which improves over the initial learning steps since both estimates are initialized with zeros.

The rest of the documentation follows the original paper *[adam2014]* and is only meant as a quick primer. We refer to the original source for more details, such as results on convergence and discussion of the various hyper parameters.

Let $f'_t(\theta_t)$ be the derivative of the loss with respect to the parameters at time step t. In its basic form, given a step rate α, decay terms β_1 and β_2 for the first and second moment estimates respectively and an offset ϵ we initialise the following quantities

$$\begin{aligned} m_0 &\leftarrow 0 \\ v_0 &\leftarrow 0 \\ t &\leftarrow 0 \end{aligned}$$

and perform the following updates:

$$\begin{aligned} t &\leftarrow t + 1 \\ g_t &\leftarrow f'_t(\theta_{t-1}) \\ m_t &\leftarrow \beta_1 \cdot g_t + (1 - \beta_1) \cdot m_{t-1} \\ v_t &\leftarrow \beta_2 \cdot g_t^2 + (1 - \beta_2) \cdot v_{t-1} \\ \hat{m}_t &\leftarrow \frac{m_t}{(1 - (1 - \beta_1)^t)} \\ \hat{v}_t &\leftarrow \frac{v_t}{(1 - (1 - \beta_2)^t)} \\ \theta_t &\leftarrow \theta_{t-1} - \alpha \frac{\hat{m}_t}{(\sqrt{\hat{v}_t} + \epsilon)} \end{aligned}$$

As suggested in the original paper, the last three steps are optimized for efficieny by using:

$$\begin{aligned} \alpha_t &\leftarrow \alpha \frac{\sqrt{(1 - (1 - \beta_2)^t)}}{(1 - (1 - \beta_1)^t)} \\ \theta_t &\leftarrow \theta_{t-1} - \alpha_t \frac{m_t}{(\sqrt{v_t} + \epsilon)} \end{aligned}$$

The quantities in the algorithm and their corresponding attributes in the optimizer object are as follows.

Symbol	Attribute	Meaning
t	`n_iter`	Number of iterations, starting at 0.
m_t	`est_mom_1_b`	Biased estimate of first moment.
v_t	`est_mom_2_b`	Biased estimate of second moment.
$\hat{m}_t$	`est_mom_1`	Unbiased estimate of first moment.
$\hat{v}_t$	`est_mom_2`	Unbiased estimate of second moment.
α	`step_rate`	Step rate parameter.
β_1	`decay_mom1`	Exponential decay parameter for first moment estimate.
β_2	`decay_mom2`	Exponential decay parameter for second moment estimate.
ϵ	`offset`	Safety offset for division by estimate of second moment.

Additionally, using Nesterov momentum is possible by setting the momentum attribute of the optimizer to a value other than 0. We apply the momentum step before computing the gradient, resulting in a similar incorporation of Nesterov momentum in Adam as presented in *[nadam2015]*.

Note: The use of decay parameters β_1 and β_2 differs from the definition in the original paper *[adam2014]*: With β_i^* referring to the parameters as defined in the paper, we use β_i with $\beta_i = 1 - \beta_i^*$

Methods

__init__(*wrt*, *fprime*, *step_rate=0.0002*, *decay=None*, *decay_mom1=0.1*, *decay_mom2=0.001*, *momentum=0*, *offset=1e-08*, *args=None*)

Create an Adam object.

Parameters **wrt** : array_like

Array that represents the solution. Will be operated upon in place. `fprime` should accept this array as a first argument.

fprime : callable

Callable that given a solution vector as first parameter and *args and **kwargs drawn from the iterations `args` returns a search direction, such as a gradient.

step_rate : scalar or array_like, optional [default: 1]

Value to multiply steps with before they are applied to the parameter vector.

decay_mom1 : float, optional, [default: 0.1]

Decay parameter for the exponential moving average estimate of the first moment.

decay_mom2 : float, optional, [default: 0.001]

Decay parameter for the exponential moving average estimate of the second moment.

momentum : float or array_like, optional [default: 0]

Momentum to use during optimization. Can be specified analogously (but independent of) step rate.

offset : float, optional, [default: 1e-8]

Before taking the square root of the running averages, this offset is added.

args : iterable

Iterator over arguments which `fprime` will be called with.

4.1.5 Resilient Propagation

This module contains the Resilient propagation optimizer.

class climin.rprop.**Rprop**(*wrt*, *fprime*, *step_shrink=0.5*, *step_grow=1.2*, *min_step=1e-06*, *max_step=1*, *changes_max=0.1*, *args=None*)

Rprop optimizer.

Resilient propagation is an optimizer that was originally tailored towards neural networks. It can however be savely applied to all kinds of optimization problems. The idea is to have a parameter specific step rate which is determined by sign changes of the derivative of the objective function.

To be more precise, given the derivative of the loss given the parameters $f'(\theta_t)$ at time step t, the i th component of the vector of steprates α is determined as

$$\alpha_i \leftarrow \begin{cases} \alpha_i \cdot \eta_{\text{grow}} \text{ if } f'(\theta_t)_i \cdot f'(\theta_{t-1})_i > 0 \\ \alpha_i \cdot \eta_{\text{shrink}} \text{ if } f'(\theta_t)_i \cdot f'(\theta_{t-1})_i < 0 \\ \alpha_i \end{cases}$$

where $0 < \eta_{\text{shrink}} < 1 < \eta_{\text{grow}}$ specifies the shrink and growth rates of the step rates. Typically, we will threshold the step rates at minimum and maximum values.

The parameters are then adapted according to the sign of the error gradient:

$$\theta_{t+1} = -\alpha \operatorname{sgn}(f'(\theta_t)).$$

This results in a method which is quite robust. On the other hand, it is more sensitive towards stochastic objectives, since that stochasticity might lead to bad estimates of the sign of the gradient.

Note: Works with gnumpy.

Attributes

wrt	(array_like) Current solution to the problem. Can be given as a first argument to `.fprime`.
fprime	(Callable) First derivative of the objective function. Returns an array of the same shape as `.wrt`.
step_shrink	(float) Constant to shrink step rates by if the gradients of the error do not agree over time.
step_grow	(float) Constant to grow step rates by if the gradients of the error do agree over time.
min_step	(float) Minimum step rate.
max_step	(float) Maximum step rate.

Methods

__init__(*wrt*, *fprime*, *step_shrink=0.5*, *step_grow=1.2*, *min_step=1e-06*, *max_step=1*, *changes_max=0.1*, *args=None*)

Create an Rprop object.

Parameters **wrt** : array_like

Current solution to the problem. Can be given as a first argument to `.fprime`.

fprime : Callable

First derivative of the objective function. Returns an array of the same shape as `.wrt`.

step_shrink : float

Constant to shrink step rates by if the gradients of the error do not agree over time.

step_grow : float

Constant to grow step rates by if the gradients of the error do agree over time.

min_step : float

Minimum step rate.

max_step : float

Maximum step rate.

args : iterable

Iterator over arguments which `fprime` will be called with.

4.1.6 Conjugate Gradients

Module containing functionality for conjugate gradients.

Conjugate gradients is motivated from a first order Taylor expansion of the objective:

$$f(\theta_t + \alpha_t d_t) \approx f(\theta_t) + \alpha_t d_t^T f'(\theta_t).$$

To locally decrease the objective, it is optimal to set $d_t \propto -f'(\theta_t)$ and find α_t with a line search algorithm, which is known as steepest descent. Yet, a well known disadvantage of this approach is that directions found at t will often interfere with directions found for $t' < t$.

The solution to this problem is to chose d_t in a way that it does not interfere with previous updates. If the dimensions of our problem were independent, we could just move along these dimensions. If they were independent up to rotation, we would have to chose directions which are orthogonal to each other. This is exactly the case when the Hessian of the problem, A is diagonal. If it is not diagonal, we have to move along directions which are called *conjugate* to each other with respect to the matrix A.

The conjugate gradients algorithms provide methods to do so efficiently. The linear conjugate gradients algorithm assumes that the objective is a quadratic and can thus determine α exactly. Nonlinear conjugate gradients works on arbitrary functions (yet, the Taylor expansion assumption above has to be reasonable). Since the Hessian A is not constant in this case, the previous directions (to which a new direction has to be conjugate) have to be reset from time to time. Additionally, we need to perform a line search to solve for α_t.

class `climin.cg.`**`ConjugateGradient`**(*wrt*, *H=None*, *b=None*, *f_Hp=None*, *min_grad=1e-14*, *precond=None*)

ConjugateGradient class.

Minimize a quadratic objective of the form

$$f(\theta) = \frac{1}{2}\theta^T A\theta + \theta^T b + c.$$

The minimization will take place by moving along conjugate directions of steepest decrease in the error. This will take at most as many steps as the dimensionality of the problem.

Note: In most cases it is better to use `scipy.optimize.solve`. Only use this function if you want to monitor intermediate quantities and are not entirely interested in optimization of a quadratic objective, but in a different error measure. E.g. as in Hessian free optimization.

Attributes

wrt	(array_like) Parameters of the problem.
H	(array_like, 2 dimensional, square) Curvature term of the quadratic, the Hessian.
b	(array_like) Linear term of the quadratic.
f_Hp	(callable) Function to calculcate the dot product of a Hessian with an arbitrary vector.
min_grad	(float, optional, default: 1e-14) If all components of the gradient fall below this threshold, stop optimization.
precond	(array_like) Matrix to precondition the problem. If a vector, is taken to represent a diagonal matrix.

Methods

__init__(*wrt*, *H=None*, *b=None*, *f_Hp=None*, *min_grad=1e-14*, *precond=None*)
Create a ConjugateGradient object.

Parameters **wrt** : array_like

Parameters of the problem.

H : array_like, 2 dimensional, square

Curvature term of the quadratic, the Hessian.

b : array_like

Linear term of the quadratic.

f_Hp : callable

Function to calculcate the dot product of a Hessian with an arbitrary vector.

min_grad : float, optional, default: 1e-14

If all components of the gradient fall below this threshold, stop optimization.

precond : array_like

Matrix to precondition the problem. If a vector, is taken to represent a diagonal matrix.

class climin.cg.**NonlinearConjugateGradient**(*wrt*, *f*, *fprime*, *min_grad=1e-06*, *args=None*)
NonlinearConjugateGradient optimizer.

NCG minimizes functions by following directions which are conjugate to each other with respect to the Hessian. Since the curvature changes if the objective is nonquadratic, the Hessian will not be accurate and thus the conjugacy of successive search directions as well. Furthermore, the optimal step length cannot be found in closed form and has to be obtained with a line search.

During optimization, we sometimes perform a restart. That means we give up on trying to find conjugate directions and use the gradient as a new search direction. This is done whenever two successive directions are far from orthogonal, an indicator that the quadratic assumption is either inaccurate or the Hessian has changed too much lately.

Attributes

wrt	(array_like) Array of parameters of the problem.
f	(callable) Objective function.
fprime	(callable) First derivative of the objective function.
min_grad	(float) If all components of the gradient fall below this value, stop minimization.
line_search	(LineSearch object.) Line search object to perform line searches with.
args	(iterable) Iterable of arguments passed on to the objective function and its derivatives.

Methods

__init__(*wrt*, *f*, *fprime*, *min_grad=1e-06*, *args=None*)

Create a NonlinearConjugateGradient object.

Parameters **wrt** : array_like

Array of parameters of the problem.

f : callable

Objective function.

fprime : callable

First derivative of the objective function.

min_grad : float

If all components of the gradient fall below this value, stop minimization.

args : iterable, optional

Iterable of arguments passed on to the objective function and its derivatives.

4.1.7 Quasi-Newton (BFGS, L-BFGS)

This module provides an implementation of Quasi-Newton methods (BFGS, sBFGS and l-BFGS).

The Taylor expansion up to second order of a function $f(\theta_t)$ allows a local quadratic approximiation of $f(\theta_t + d_t)$:

$$f(\theta_t + d_t) \approx f(\theta_t) + d_t^T f'(\theta_t) + \frac{1}{2} d_t^T H_t d_t$$

where the symmetric positive definite matrix H_t is the Hessian at θ_t. The minimizer d_t of this convex quadratic model is:

$$d_t = -H^{-1} f'(\theta_t).$$

For large scale problems both computing/storing the Hessian and solving the above linear system is computationally demanding. Instead of recomputing the Hessian from scratch at every iteration, quasi-Newton methods utilize successive measurements of the gradient to build a sufficiently good quadratic model of the objective function. The above formula is then applied to yield a direction d_t. The update done is then of the form

$$\theta_{t+1} = \alpha_t d_t + \theta_t$$

where α_t is obtained with a line search.

Note: The classes presented here are not working with gnumpy.

class `climin.bfgs.Bfgs`(*wrt*, *f*, *fprime*, *initial_inv_hessian=None*, *line_search=None*, *args=None*)

BFGS (Broyden-Fletcher-Goldfarb-Shanno) is one of the most well-knwon quasi-Newton methods. The main idea is to iteratively construct an approximate inverse Hessian B_t^{-1} by a rank-2 update:

$$B_{t+1}^{-1} = B_t^{-1} + (1 + \frac{y_t^T B_t^{-1} y_t}{y_t^T s_t}) \frac{s_t s_t^T}{s_t^T y_t} - \frac{s_t y_t^T B_t^{-1} + B_t^{-1} y_t s_t^T}{s_t^T y_t},$$

where $y_t = f(\theta_{t+1}) - f(\theta_t)$ and $s_t = \theta_{t+1} - \theta_t$.

The storage requirements for BFGS scale quadratically with the number of variables. For detailed derivations, see *[nocedal2006a]*, chapter 6.

Attributes

wrt	(array_like) Current solution to the problem. Can be given as a first argument to `.f` and `.fprime`.
f	(Callable) The object function.
fprime	(Callable) First derivative of the objective function. Returns an array of the same shape as `.wrt`.
initial_inv_hessian	(array_like) The initial estimate of the approximiate Hessian.
line_search	(LineSearch object.) Line search object to perform line searches with.
args	(iterable) Iterator over arguments which `fprime` will be called with.

Methods

__init__(*wrt*, *f*, *fprime*, *initial_inv_hessian=None*, *line_search=None*, *args=None*)

Create a BFGS object.

Parameters **wrt** : array_like

Array that represents the solution. Will be operated upon in place. `f` and `fprime` should accept this array as a first argument.

f : callable

The objective function.

fprime : callable

Callable that given a solution vector as first parameter and *args and **kwargs drawn from the iterations `args` returns a search direction, such as a gradient.

initial_inv_hessian : array_like

The initial estimate of the approximiate Hessian.

line_search : LineSearch object.

Line search object to perform line searches with.

args : iterable

Iterator over arguments which `fprime` will be called with.

class `climin.bfgs.`**`Lbfgs`**(*wrt*, *f*, *fprime*, *initial_hessian_diag=1*, *n_factors=10*, *line_search=None*, *args=None*)

l-BFGS (limited-memory BFGS) is a limited memory variation of the well-known BFGS algorithm. The storage requirement for BFGS scale quadratically with the number of variables, and thus it tends to be used only for smaller problems. Limited-memory BFGS reduces the storage by only using the l latest updates (factors) in computing the approximate Hessian inverse and representing this approximation only implicitly. More specifically, it stores the last l BFGS update vectors y_t and s_t and uses these to implicitly perform the matrix operations of BFGS (see *[nocedal2006a]*).

Note: In order to handle simple box constraints, consider `scipy.optimize.fmin_l_bfgs_b`.

Attributes

wrt	(array_like) Current solution to the problem. Can be given as a first argument to `.f` and `.fprime`.
f	(Callable) The object function.
fprime	(Callable) First derivative of the objective function. Returns an array of the same shape as `.wrt`.
initial_hessian_diag	(array_like) The initial estimate of the diagonal of the Hessian.
n_factors	(int) The number of factors that should be used to implicitly represent the inverse Hessian.
line_search	(LineSearch object.) Line search object to perform line searches with.
args	(iterable) Iterator over arguments which `fprime` will be called with.

Methods

`__init__`(*wrt*, *f*, *fprime*, *initial_hessian_diag=1*, *n_factors=10*, *line_search=None*, *args=None*)

Create an Lbfgs object.

Attributes

wrt	(array_like) Current solution to the problem. Can be given as a first argument to `.f` and `.fprime`.
f	(Callable) The object function.
fprime	(Callable) First derivative of the objective function. Returns an array of the same shape as `.wrt`.
initial_hessian_diag	(array_like) The initial estimate of the diagonal of the Hessian.
n_factors	(int) The number of factors that should be used to implicitly represent the inverse Hessian.
line_search	(LineSearch object.) Line search object to perform line searches with.
args	(iterable) Iterator over arguments which `fprime` will be called with.

4.2 Convenience, Utilities

4.2.1 Schedules

This module holds various schedules for parameters such as the step rate or momentum for gradient descent.

A schedule is implemented as an iterator. This allows it to have iterators of infinite length. It also makes it possible to manipulate scheduls with the `itertools` python module, e.g. for chaining iterators.

`climin.schedule.`**`decaying`**(*start*, *decay*)
Return an iterator of exponentially decaying values.

The first value is `start`. Every further value is obtained by multiplying the last one by a factor of `decay`.

Examples

```
>>> from climin.schedule import decaying
>>> s = decaying(10, .9)
>>> [next(s) for i in range(5)]
[10.0, 9.0, 8.100000000000001, 7.290000000000001, 6.561]
```

`climin.schedule.`**`linear_annealing`**(*start*, *stop*, *n_steps*)
Return an iterator that anneals linearly to a point linearly.

The first value is `start`, the last value is `stop`. The annealing will be linear over `n_steps` iterations. After that, `stop` is yielded.

Examples

```
>>> from climin.schedule import linear_annealing
>>> s = linear_annealing(1, 0, 4)
>>> [next(s) for i in range(10)]
[1.0, 0.75, 0.5, 0.25, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

`climin.schedule.`**`repeater`**(*iter*, *n*)
Return an iterator that repeats each element of *iter* exactly *n* times before moving on to the next element.

Examples

```
>>> from climin.schedule import repeater
>>> s = repeater([1, 2, 3], 2)
>>> [next(s) for i in range(6)]
[1, 1, 2, 2, 3, 3]
```

class `climin.schedule.`**`SutskeverBlend`**(*max_momentum*, *stretch=250*)
Class representing a schedule that step-wise increases from zero to a maximum value, as described in *[sutskever2013importance]*.

Examples

```
>>> from climin.schedule import SutskeverBlend
>>> s = iter(SutskeverBlend(0.9, 2))
>>> [next(s) for i in range(10)]
[0.5, 0.75, 0.75, 0.8333333333333333, 0.8333333333333333, 0.875, 0.875, 0.9, 0.9, 0.9]
```

4.2.2 Initialization of Parameters

Module that contains functionality to initialize parameters to starting values.

climin.initialize.**sparsify_columns**(*arr*, *n_non_zero*, *keep_diagonal=False*, *random_state=None*)

Set all but `n_non_zero` entries to zero for each column of `arr`.

This is a common technique to find better starting points for learning deep and/or recurrent networks.

Parameters **arr** : array_like, two dimensional

Array to work upon in place.

n_non_zero : integer

Amount of non zero entries to keep.

keep_diagonal : boolean, optional [default: False]

If set to True and `arr` is square, do keep the diagonal.

random_state : numpy.random.RandomState object, optional [default

If set, random number generator that will generate the indices corresponding to the zero-valued columns.

Examples

```
>>> import numpy as np
>>> from climin.initialize import sparsify_columns
>>> arr = np.arange(9).reshape((3, 3))
>>> sparsify_columns(arr, 1)
>>> arr
array([[0, 0, 0],
       [0, 4, 5],
       [6, 0, 0]])
```

climin.initialize.**bound_spectral_radius**(*arr*, *bound=1.2*)

Set the spectral radius of the square matrix `arr` to `bound`.

This is performed by scaling eigenvalues of `arr`.

Parameters **arr** : array_like, two dimensional

Array to work upon in place.

bound : float, optional, default: 1.2

Examples

```
>>> import numpy as np
>>> from climin.initialize import bound_spectral_radius
>>> arr = np.arange(9).reshape((3, 3)).astype('float64')
>>> bound_spectral_radius(arr, 1.1)
>>> arr
array([[ -7.86816957e-17,   8.98979486e-02,   1.79795897e-01],
       [  2.69693846e-01,   3.59591794e-01,   4.49489743e-01],
       [  5.39387691e-01,   6.29285640e-01,   7.19183588e-01]])
```

climin.initialize.**randomize_normal**(*arr*, *loc=0*, *scale=1*, *random_state=None*)

Populate an array with random numbers from a normal distribution with mean *loc* and standard deviation *scale*.

Parameters **arr** : array_like

Array to work upon in place.

loc : float

Mean of the random numbers.

scale : float

Standard deviation of the random numbers.

random_state : np.random.RandomState object, optional [default

Random number generator that shall generate the random numbers.

Examples

```
>>> import numpy as np
>>> from climin.initialize import randomize_normal
>>> arr = np.empty((3, 3))
>>> randomize_normal(arr)
>>> arr
array([[ 0.18076413,  0.60880657,  1.20855691],
       [ 1.7799948 , -0.82565481,  0.53875307],
       [-0.67056028, -1.46257419,  1.17033425]])
>>> randomize_normal(arr, 10, 0.1)
>>> arr
array([[ 10.02221481,  10.0982449 ,  10.02495358],
       [  9.99867829,   9.99410111,   9.8242318 ],
       [  9.9383779 ,   9.94880091,  10.03179085]])
```

4.2.3 Line searches

Module containing various line searches.

Line searches are at the heart of many optimizers. After finding a suitable search direction (e.g. the steepest descent direction) we are left with a one-dimensional optimization problem, which can then be solved by a line search.

class climin.linesearch.**BackTrack**(*wrt*, *f*, *decay=0.9*, *max_iter=inf*, *tolerance=1e-20*)

Class implementing a back tracking line search.

The idea is to jump to a starting step length t and then shrink that step length by multiplying it with γ until we improve upon the loss.

At most `max_iter` attempts will be done. If the largest absolut value of a component of the step falls below `tolerance`, we stop as well. In both cases, a step length of 0 is returned.

To not possibly iterate forever, the field *tolerance* holds a small value (1E-20 per default). As soon as the absolute value of every component of the step (direction multiplied with the scalar from *schedule*) is less than *tolerance*, we stop.

Attributes

wrt	(array_like) Parameters over which the optimization is done.
f	(Callable) Objective function.
decay	(float) Factor to multiply trials for the step length with.
tolerance	(float) Minimum absolute value of a component of the step without stopping the line search.

Methods

__init__ (*wrt*, *f*, *decay=0.9*, *max_iter=inf*, *tolerance=1e-20*)
Create BackTrack object.

Parameters **wrt** : array_like

Parameters over which the optimization is done.

f : Callable

Objective function.

decay : float

Factor to multiply trials for the step length with.

max_iter : int, optional, default infinity

Number of step lengths to try.

tolerance : float

Minimum absolute value of a component of the step without stopping the line search.

search (*direction*, *initialization=1*, *args=None*, *kwargs=None*, *loss0=None*)
Return a step length `t` given a search direction.

Perform the line search along a direction. Search will start at `initialization` and assume that the loss is `loss0` at `t == 0`.

Parameters **direction** : array_like

Has to be of the same size as .wrt. Points along that direction will tried out to reduce the loss.

initialization : float

First attempt for a step size. Will be reduced by a factor of `.decay` afterwards.

args : list, optional, default: None

list of optional arguments for `.f`.

kwargs : dictionary, optional, default: None

list of optional keyword arguments for `.f`.

loss0 : float, optional

Loss at the current parameters. Will be calculated of not given.

class climin.linesearch.**StrongWolfeBackTrack**(*wrt*, *f*, *fprime*, *decay=None*, *c1=0.0001*, *c2=0.9*, *tolerance=1e-20*)

Class implementing a back tracking line search that finds points satisfying the Strong Wolfe conditions.

The idea is to jump to a starting step length t and then shrink that step length by multiplying it with γ until the strong Wolfe conditions are satisfied. That is the Armijo rule

$$f(\theta_t + \alpha_t d_t) \leq f(\theta) + c_1 \alpha_t d_t^T f'(\theta),$$

and the curvature condition

$$\left| d_k^T T f('\theta_t + \alpha_t d_t) \right| \leq c_2 \left| d_t^T f'(\theta_t) \right|.$$

At most `max_iter` attempts will be done. If the largest absolut value of a component of the step falls below `tolerance`, we stop as well. In both cases, a step length of 0 is returned.

To not possibly iterate forever, the field *tolerance* holds a small value (1E-20 per default). As soon as the absolute value of every component of the step (direction multiplied with the scalar from *schedule*) is less than *tolerance*, we stop.

Attributes

wrt	(array_like) Parameters over which the optimization is done.
f	(Callable) Objective function.
decay	(float) Factor to multiply trials for the step length with.
tolerance	(float) Minimum absolute value of a component of the step without stopping the line search.
c1	(float) Constant in the strong Wolfe conditions.
c2	(float) Constant in the strong Wolfe conditions.

Methods

__init__(*wrt*, *f*, *fprime*, *decay=None*, *c1=0.0001*, *c2=0.9*, *tolerance=1e-20*)

Create StrongWolfeBackTrack object.

Parameters **wrt** : array_like

Parameters over which the optimization is done.

f : Callable

Objective function.

decay : float

Factor to multiply trials for the step length with.

tolerance : float

Minimum absolute value of a component of the step without stopping the line search.

class climin.linesearch.**ScipyLineSearch**(*wrt*, *f*, *fprime*)

Wrapper around the scipy line search.

Methods

class climin.linesearch.**WolfeLineSearch**(*wrt*, *f*, *fprime*, *c1=0.0001*, *c2=0.9*, *maxiter=25*, *min_step_length=1e-09*, *typ=4*)

Port of Mark Schmidt's line search.

Methods

4.2.4 Utility functions

climin.util.**empty_with_views**(*shapes*, *empty_func=<built-in function empty>*)

Create an array and views shaped according to `shapes`.

The `shapes` parameter is a list of tuples of ints. Each tuple represents a desired shape for an array which will be allocated in a bigger memory region. This memory region will be represented by an array as well.

For example, the shape speciciation `[2, (3, 2)]` will create an array `flat` of size 8. The first view will have a size of `(2,)` and point to the first two entries, i.e. `flat`[:2]`, while the second array will have a shape of ``(3, 2)` and point to the elements `flat[2:8]`.

Parameters **spec** : list of tuples of ints

Specification of the desired shapes.

empty_func : callable

function that returns a memory region given an integer of the desired size. (Examples include `numpy.empty`, which is the default, `gnumpy.empty` and `theano.tensor.empty`.

Returns **flat** : array_like (depending on `empty_func`)

Memory region containing all the views.

views : list of array_like

Variable number of results. Each contains a view into the array `flat`.

Examples

```
>>> from climin.util import empty_with_views
>>> flat, (w, b) = empty_with_views([(3, 2), 2])
>>> w[...] = 1
>>> b[...] = 2
>>> flat
array([ 1.,  1.,  1.,  1.,  1.,  1.,  2.,  2.])
>>> flat[0] = 3
>>> w
array([[ 3.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
```

climin.util.**shaped_from_flat**(*flat*, *shapes*)

Given a one dimensional array `flat`, return a list of views of shapes `shapes` on that array.

Each view will point to a distinct memory region, consecutively allocated in flat.

Parameters **flat** : array_like

Array of one dimension.

shapes : list of tuples of ints

Each entry of this list specifies the shape of the corresponding view into `flat`.

Returns **views** : list of arrays

Each entry has the shape given in `shapes` and points as a view into `flat`.

climin.util.**minibatches**(*arr*, *batch_size*, *d=0*)

Return a list of views of the given arr.

Each view represents a mini bach of the data.

Parameters **arr** : array_like

Array to obtain batches from. Needs to be slicable. If `d > 0`, needs to have a `.shape` attribute from which the number of samples can be obtained.

batch_size : int

Size of a batch. Last batch might be smaller if `batch_size` is not a divisor of `arr`.

d : int, optional, default: 0

Dimension along which the data samples are separated and thus slicing should be done.

Returns **mini_batches** : list

Each item of the list is a view of `arr`. Views are ordered.

climin.util.**iter_minibatches**(*lst*, *batch_size*, *dims*, *n_cycles=None*, *random_state=None*, *discard_illsized_batch=False*)

Return an iterator that successively yields tuples containing aligned minibatches of size *batch_size* from slicable objects given in *lst*, in random order without replacement. Because different containers might require slicing over different dimensions, the dimension of each container has to be givens as a list *dims*.

Parameters **lst** : list of array_like

Each item of the list will be sliced into mini batches in alignment with the others.

batch_size : int

Size of each batch. Last batch might be smaller.

dims : list

Aligned with `lst`, gives the dimension along which the data samples are separated.

n_cycles : int, optional [default: None]

Number of cycles after which to stop the iterator. If `None`, will yield forever.

random_state : a numpy.random.RandomState object, optional [default

Random number generator that will act as a seed for the minibatch order.

discard_illsized_batch : bool, optional [default

If `True` and the length of the sliced dimension is not divisible by `batch_size`, the leftover samples are discarded.

Returns **batches** : iterator

Infinite iterator of mini batches in random order (without replacement).

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Bibliography

[sparsefiltering] Ngiam, Jiquan, et al. “Sparse filtering.” Advances in Neural Information Processing Systems. 2011.

[tieleman2012rmsprop] Tieleman, T. and Hinton, G. (2012), Lecture 6.5 - rmsprop, COURSERA: Neural Networks for Machine Learning

[zeiler2013adadelta] Zeiler, Matthew D. “ADADELTA: An adaptive learning rate method.” arXiv preprint arXiv:1212.5701 (2012).

[adam2014] Kingma, Diederik, and Jimmy Ba. “Adam: A Method for Stochastic Optimization.” arXiv preprint arXiv:1412.6980 (2014).

[nadam2015] Dozat, Timothy “Incorporating Nesterov Momentum into Adam.” Stanford University, Tech. Rep (2015).

[riedmiller1992rprop] M. Riedmiller und Heinrich Braun: Rprop - A Fast Adaptive Learning Algorithm. Proceedings of the International Symposium on Computer and Information Science VII, 1992

[nocedal2006a] Nocedal, J. and Wright, S. (2006), Numerical Optimization, 2nd edition, Springer.

[sutskever2013importance] On the importance of initialization and momentum in deep learning, Sutskever et al (ICML 2013)

C

climin.adadelta, 16
climin.adam, 18
climin.bfgs, 23
climin.cg, 21
climin.gd, 13
climin.initialize, 27
climin.linesearch, 28
climin.rmsprop, 14
climin.rprop, 20
climin.schedule, 26

Symbols

__init__() (climin.adadelta.Adadelta method), 17
__init__() (climin.adam.Adam method), 19
__init__() (climin.bfgs.Bfgs method), 24
__init__() (climin.bfgs.Lbfgs method), 25
__init__() (climin.cg.ConjugateGradient method), 22
__init__() (climin.cg.NonlinearConjugateGradient method), 23
__init__() (climin.gd.GradientDescent method), 14
__init__() (climin.linesearch.BackTrack method), 29
__init__() (climin.linesearch.StrongWolfeBackTrack method), 30
__init__() (climin.rmsprop.RmsProp method), 16
__init__() (climin.rprop.Rprop method), 20

A

Adadelta (class in climin.adadelta), 16
Adam (class in climin.adam), 18

B

BackTrack (class in climin.linesearch), 28
Bfgs (class in climin.bfgs), 24
bound_spectral_radius() (in module climin.initialize), 27

C

climin.adadelta (module), 16
climin.adam (module), 18
climin.bfgs (module), 23
climin.cg (module), 21
climin.gd (module), 13
climin.initialize (module), 27
climin.linesearch (module), 28
climin.rmsprop (module), 14
climin.rprop (module), 20
climin.schedule (module), 26
ConjugateGradient (class in climin.cg), 21

D

decaying() (in module climin.schedule), 26

E

empty_with_views() (in module climin.util), 31

G

GradientDescent (class in climin.gd), 13

I

iter_minibatches() (in module climin.util), 32

L

Lbfgs (class in climin.bfgs), 25
linear_annealing() (in module climin.schedule), 26

M

minibatches() (in module climin.util), 31

N

NonlinearConjugateGradient (class in climin.cg), 22

R

randomize_normal() (in module climin.initialize), 28
repeater() (in module climin.schedule), 26
RmsProp (class in climin.rmsprop), 14
Rprop (class in climin.rprop), 20

S

ScipyLineSearch (class in climin.linesearch), 30
search() (climin.linesearch.BackTrack method), 29
shaped_from_flat() (in module climin.util), 31
sparsify_columns() (in module climin.initialize), 27
StrongWolfeBackTrack (class in climin.linesearch), 29
SutskeverBlend (class in climin.schedule), 26

W

WolfeLineSearch (class in climin.linesearch), 30