Optimization algorithms

The following algorithms are provided:

- *Stochastic Gradient Descent*
- *Averaged Stochastic Gradient Descent*
- *L-BFGS*
- *Congugate Gradients*
- *AdaDelta*
- *AdaGrad*
- *Adam*
- *AdaMax*
- *FISTA with backtracking line search*
- *Nesterov's Accelerated Gradient method*
- *RMSprop*
- *Rprop*
- *CMAES*

All these algorithms are designed to support batch optimization as well as stochastic optimization.
It's up to the user to construct an objective function that represents the batch, mini-batch, or single sample on which to evaluate the objective.

Some of these algorithms support a line search, which can be passed as a function (*L-BFGS*), whereas others only support a learning rate (*SGD*).

General interface:

```
x*, {f}, ... = optim.method(opfunc, x[, config][, state])
```

sgd(opfunc, x[, config][, state])

An implementation of *Stochastic Gradient Descent* (*SGD*).

Arguments:

- `opfunc` : a function that takes a single input `X` , the point of a evaluation, and returns

`f(X)` and `df/dX`
- `x` : the initial point
- `config` : a table with configuration parameters for the optimizer
- `config.learningRate` : learning rate
- `config.learningRateDecay` : learning rate decay
- `config.weightDecay` : weight decay
- `config.weightDecays` : vector of individual weight decays
- `config.momentum` : momentum
- `config.dampening` : dampening for momentum
- `config.nesterov` : enables Nesterov momentum
- `state` : a table describing the state of the optimizer; after each call the state is modified
- `state.learningRates` : vector of individual learning rates

Returns:

- `x*` : the new x vector
- `f(x)` : the function, evaluated before the update

asgd(opfunc, x[, config][, state])

An implementation of *Averaged Stochastic Gradient Descent* (*ASGD*):

```
x = (1 - lambda eta_t) x - eta_t df / dx(z, x)
a = a + mu_t [ x - a ]

eta_t = eta0 / (1 + lambda eta0 t) ^ 0.75
mu_t = 1 / max(1, t - t0)
```

Arguments:

- `opfunc` : a function that takes a single input `X` , the point of evaluation, and returns `f(X)` and `df/dX`
- `x` : the initial point
- `config` : a table with configuration parameters for the optimizer
- `config.eta0` : learning rate
- `config.lambda` : decay term
- `config.alpha` : power for eta update
- `config.t0` : point at which to start averaging

Returns:

- `x*` : the new x vector
- `f(x)` : the function, evaluated before the update
- `ax` : the averaged x vector

lbfgs(opfunc, x[, config][, state])

An implementation of *L-BFGS* that relies on a user-provided line search function (`state.lineSearch`).
If this function is not provided, then a simple learning rate is used to produce fixed size steps. Fixed size steps are much less costly than line searches, and can be useful for stochastic problems.

The learning rate is used even when a line search is provided.
This is also useful for large-scale stochastic problems, where opfunc is a noisy approximation of `f(x)`.
In that case, the learning rate allows a reduction of confidence in the step size.

Arguments:

- `opfunc` : a function that takes a single input `X`, the point of evaluation, and returns `f(X)` and `df/dX`
- `x` : the initial point
- `config` : a table with configuration parameters for the optimizer
- `config.maxIter` : Maximum number of iterations allowed
- `config.maxEval` : Maximum number of function evaluations
- `config.tolFun` : Termination tolerance on the first-order optimality
- `config.tolX` : Termination tol on progress in terms of func/param changes
- `config.lineSearch` : A line search function
- `config.learningRate` : If no line search provided, then a fixed step size is used

Returns:
* `x*` : the new `x` vector, at the optimal point
* `f` : a table of all function values:
* `f[1]` is the value of the function before any optimization and
* `f[#f]` is the final fully optimized value, at `x*`

cg(opfunc, x[, config][, state])

An implementation of the *Conjugate Gradient* method which is a rewrite of `minimize.m` written by Carl E. Rasmussen.
It is supposed to produce exactly same results (give or take numerical accuracy due to some changed order of operations).
You can compare the result on rosenbrock with `minimize.m`.

```
x, fx, c = minimize([0, 0]', 'rosenbrock', -25)
```

Note that we limit the number of function evaluations only, it seems much more important in practical use.

Arguments:

- `opfunc` : a function that takes a single input, the point of evaluation.
- `x` : the initial point
- `config` : a table with configuration parameters for the optimizer
- `config.maxEval` : max number of function evaluations
- `config.maxIter` : max number of iterations
- `state` : a table of parameters and temporary allocations.
- `state.df[0, 1, 2, 3]` : if you pass `Tensor` they will be used for temp storage
- `state.[s, x0]` : if you pass `Tensor` they will be used for temp storage

Returns:

- `x*` : the new `x` vector, at the optimal point
- `f` : a table of all function values where
  - `f[1]` is the value of the function before any optimization and
  - `f[#f]` is the final fully optimized value, at `x*`

adadelta(opfunc, x[, config][, state])

AdaDelta implementation for *SGD* http://arxiv.org/abs/1212.5701.

Arguments:

- `opfunc` : a function that takes a single input `X`, the point of evaluation, and returns `f(X)` and `df/dX`
- `x` : the initial point
- `config` : a table of hyper-parameters
- `config.rho` : interpolation parameter

- `config.eps` : for numerical stability
- `state` : a table describing the state of the optimizer; after each call the state is modified
- `state.paramVariance` : vector of temporal variances of parameters
- `state.accDelta` : vector of accummulated delta of gradients

Returns:

- `x*` : the new x vector
- `f(x)` : the function, evaluated before the update

adagrad(opfunc, x[, config][, state])

AdaGrad implementation for *SGD*.

Arguments:

- `opfunc` : a function that takes a single input `X` , the point of evaluation, and returns `f(X)` and `df/dX`
- `x` : the initial point
- `config` : a table with configuration parameters for the optimizer
- `config.learningRate` : learning rate
- `state` : a table describing the state of the optimizer; after each call the state is modified
- `state.paramVariance` : vector of temporal variances of parameters

Returns:

- `x*` : the new `x` vector
- `f(x)` : the function, evaluated before the update

adam(opfunc, x[, config][, state])

An implementation of *Adam* from http://arxiv.org/pdf/1412.6980.pdf.

Arguments:

- `opfunc` : a function that takes a single input `X` , the point of a evaluation, and returns `f(X)` and `df/dX`
- `x` : the initial point

- `config` : a table with configuration parameters for the optimizer
- `config.learningRate` : learning rate
- `config.learningRateDecay` : learning rate decay
- `config.beta1` : first moment coefficient
- `config.beta2` : second moment coefficient
- `config.epsilon` : for numerical stability
- `state` : a table describing the state of the optimizer; after each call the state is modified

Returns:

- `x*` : the new x vector
- `f(x)` : the function, evaluated before the update

adamax(opfunc, x[, config][, state])

An implementation of *AdaMax* http://arxiv.org/pdf/1412.6980.pdf.

Arguments:

- `opfunc` : a function that takes a single input `X` , the point of a evaluation, and returns `f(X)` and `df/dX`
- `x` : the initial point
- `config` : a table with configuration parameters for the optimizer
- `config.learningRate` : learning rate
- `config.beta1` : first moment coefficient
- `config.beta2` : second moment coefficient
- `config.epsilon` : for numerical stability
- `state` : a table describing the state of the optimizer; after each call the state is modified

Returns:

- `x*` : the new `x` vector
- `f(x)` : the function, evaluated before the update

FistaLS(f, g, pl, xinit[, params])

Fista with backtracking *Line Search*:

- `f` : smooth function
- `g` : non-smooth function
- `pl` : minimizer of intermediate problem Q(x, y)
- `xinit` : initial point
- `params` : table of parameters (**optional**)
- `params.L` : 1/(step size) for ISTA/FISTA iteration (0.1)
- `params.Lstep` : step size multiplier at each iteration (1.5)
- `params.maxiter` : max number of iterations (50)
- `params.maxline` : max number of line search iterations per iteration (20)
- `params.errthres` : Error thershold for convergence check (1e-4)
- `params.doFistaUpdate` : true : use FISTA, false: use ISTA (true)
- `params.verbose` : store each iteration solution and print detailed info (false)

On output, `params` will contain these additional fields that can be reused.
* `params.L` : last used L value will be written.

These are temporary storages needed by the algo and if the same params object is passed a second time, these same storages will be used without new allocation.
* `params.xkm` : previous iterarion point
* `params.y` : fista iteration
* `params.ply` : `ply = pl(y * 1/L grad(f))`

Returns the solution `x` and history of `{function evals, number of line search , ...}`.

Algorithm is published in http://epubs.siam.org/doi/abs/10.1137/080716542

nag(opfunc, x[, config][, state])

An implementation of *SGD* adapted with features of *Nesterov's Accelerated Gradient method*, based on the paper "On the Importance of Initialization and Momentum in Deep Learning" (Sutskever et. al., ICML 2013) http://www.cs.toronto.edu/~fritz/absps/momentum.pdf.

Arguments:

- `opfunc` : a function that takes a single input `X` , the point of evaluation, and returns `f(X)` and `df/dX`
- `x` : the initial point
- `config` : a table with configuration parameters for the optimizer
- `config.learningRate` : learning rate
- `config.learningRateDecay` : learning rate decay

- `config.weightDecay` : weight decay
- `config.momentum` : momentum
- `config.learningRates` : vector of individual learning rates

Returns:

- `x*` : the new `x` vector
- `f(x)` : the function, evaluated before the update

rmsprop(opfunc, x[, config][, state])

An implementation of *RMSprop*.

Arguments:

- `opfunc` : a function that takes a single input `X` , the point of a evaluation, and returns `f(X)` and `df/dX`
- `x` : the initial point
- `config` : a table with configuration parameters for the optimizer
- `config.learningRate` : learning rate
- `config.alpha` : smoothing constant
- `config.epsilon` : value with which to initialise m
- `state` : a table describing the state of the optimizer; after each call the state is modified
- `state.m` : leaky sum of squares of parameter gradients,
- `state.tmp` : and the square root (with epsilon smoothing)

Returns:

- `x*` : the new x vector
- `f(x)` : the function, evaluated before the update

rprop(opfunc, x[, config][, state])

A plain implementation of *Rprop* (Martin Riedmiller, Koray Kavukcuoglu 2013).

Arguments:

- `opfunc` : a function that takes a single input `X` , the point of evaluation, and returns

`f(X)` and `df/dX`
- `x` : the initial point
- `config` : a table with configuration parameters for the optimizer
- `config.stepsize` : initial step size, common to all components
- `config.etaplus` : multiplicative increase factor, > 1 (default 1.2)
- `config.etaminus` : multiplicative decrease factor, < 1 (default 0.5)
- `config.stepsizemax` : maximum stepsize allowed (default 50)
- `config.stepsizemin` : minimum stepsize allowed (default 1e-6)
- `config.niter` : number of iterations (default 1)

Returns:

- `x*` : the new x vector
- `f(x)` : the function, evaluated before the update

cmaes(opfunc, x[, config][, state])

An implementation of *CMAES* (*Covariance Matrix Adaptation Evolution Strategy*), ported from https://www.lri.fr/~hansen/barecmaes2.html.

CMAES is a stochastic, derivative-free method for heuristic global optimization of non-linear or non-convex continuous optimization problems.
Note that this method will on average take much more function evaluations to converge then a gradient based method.

Arguments:

If `state` is specified, then `config` is not used at all.
Otherwise `state` is `config`.

- `opfunc` : a function that takes a single input `X`, the point of evaluation, and returns `f(X)` and `df/dX`. Note that `df/dX` is not used and can be left 0
- `x` : the initial point
- `state` : a table describing the state of the optimizer; after each call the state is modified
- `state.sigma` : float, initial step-size (standard deviation in each coordinate)
- `state.maxEval` : int, maximal number of function evaluations
- `state.ftarget` : float, target function value
- `state.popsize` : population size. If this is left empty, `4 + int(3 * log(|x|))` will be used
- `state.ftarget` : stop if `fitness < ftarget`
- `state.verb_disp` : display info on console every verb_disp iteration, 0 for never

Returns:
* `x*` : the new `x` vector, at the optimal point
* `f` : a table of all function values:
* `f[1]` is the value of the function before any optimization and
* `f[#f]` is the final fully optimized value, at `x*`