Stochastic Gradient Ascent in maxLik

Ott Toomet

November 24, 2020

1 `maxLik` and Stochastic Gradient Ascent

`maxLik` is a package, primarily intended for Maximum Likelihood and related estimations. It includes several optimizers and associated tools for a typical Maximum Likelihood workflow.

However, as predictive modeling and complex (deep) models have gained popularity in the recend decade, `maxLik` also includes a few popular algorithms for stochastic gradient ascent, the mirror image for the more widely known stochastic gradient descent. This vignette gives a brief overview of these methods, and their usage in `maxLik`.

2 Stochastic Gradient Ascent

In machine learning literature, it is more common to describe the optimization problems as minimization and hence to talk about gradient descent. As `maxLik` is primarily focused on maximizing likelihood, it implements the maximization version of the method, stochastic gradient ascent (SGA).

The basic method is simple and intuitive, it is essentially just a careful climb in the gradient's direction. Given and objective function $f(\boldsymbol{\theta})$, and the initial parameter vector $\boldsymbol{\theta}_0$, the algorithm will compute the gradient $\boldsymbol{g}(\boldsymbol{\theta}_0) = \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})\big|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0}$, and update the parameter vector as $\boldsymbol{\theta}_1 = \boldsymbol{\theta}_0 + \rho \boldsymbol{g}(\boldsymbol{\theta}_0)$. Here ρ, the *learning rate*, is a small positive constant to ensure we do not overshoot the optimum. Depending on the task it is typically of order $0.1 \ldots 0.001$. In common tasks, the objective function $f(\boldsymbol{\theta})$ depends on data, "predictors" X and "outcome" $\boldsymbol{y}$ in an additive form $f(\boldsymbol{\theta}; \mathsf{X}, \boldsymbol{y}) = \sum_i f(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i)$ where i denotes "observations", typically arranged as the rows of the design matrix X. Observations are often considered to be independent of each other.

The overview above does not specify how to compute the gradient $\boldsymbol{g}(\boldsymbol{\theta}_0)$ in a sense of which observations i to include. A natural approach is to include the complete data and compute

$$\boldsymbol{g}_N(\boldsymbol{\theta}_0) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}; \boldsymbol{x}_i)\big|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0}. \tag{1}$$

In SGA context, this approach is called "full batch" and it has a number of advantages. In particular, it is deterministic (given data X and $\boldsymbol{y}$), and computing of the sum can be done in parallel. However, there are also a number of reasons why full-batch approach may not be desirable (see Bottou *et al.*, 2018):

- Data over different observations is often more or less redundant. If we use all the observations to compute the update then we spend a substantial effort on redundant calculations.
- Full-batch gradient is deterministic and hence there is no stochastic noise. While advantageous in the latter steps of optimization, the noise helps the optimizer to avoid local optima and overcome flat areas in the objective function early in the process.
- SGA achieves much more rapid initial convergence compared to the full batch method (although full-batch methods may achieve better final result).
- Cost of computing the full-batch gradient grows with the sample size but that of minibatch gradient does not grow.
- It is empirically known that large-batch optimization tend to find sharp optima (see Keskar *et al.*, 2016) that do not generalize well to validation data. Small batch approach leads to a better validation performance.

In contrast, SGA is an approach where the gradient is computed on just a single observation as

$$\boldsymbol{g}_1(\boldsymbol{\theta}_0) = \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i)\big|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0} \tag{2}$$

where i is chosen randomly. In applications, all the observations are usually walked through in a random order, to ensure that each observation is included once, and only once, in an *epoch*. Epoch is a full walk-through of the data, and in many ways similar to iteration in a full-batch approach.

As SGA only accesses a single observation at time, it suffers from other kind of performance issues. In particular, one cannot parallelize the gradient function (2), operating on individual data vectors may be inefficient compared to larger matrices, and while we gain in terms of gradient computation speed, we lose by running the optimizer for many more loops.

Minibatch approach offers a balance between the full-batch and SGA. In case of minibatch, we compute gradient not on individual observations but on *batches*

$$\boldsymbol{g}_m(\boldsymbol{\theta}_0) = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i)\big|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0} \tag{3}$$

where $\mathcal{B}$ is the batch, a set of observations that are included in the gradient computation. Normally the full data is partitioned into a series of minibatches and walked through sequentially in one epoch.

3 SGA in `maxLik` package

`maxLik` implements two different optimizers: `maxSGA` for simple SGA (including momentum), and `maxAdam` for the Adaptive Moments method (see Goodfellow *et al.*, 2016, p. 301). The usage of both methods mostly follows that of the package's main workhorse, `maxNR` (see Henningsen and Toomet, 2011), but their API has some important differences due to the different nature of SGA.

The basic usage of the maxSGA is as follows:

```
> maxSGA(fn, grad, start, nObs, control)
```

where `fn` is the objective function, `grad` is the gradient function, `nObs` is number of observations, and `control` is a list of control parameters. From the user's perspective, `grad` is typically the most important (and the most complex) argument.

Next, we describe the API and explain the differences between the `maxSGA` API and `maxNR` API, and thereafter give a few toy examples that demonstrate how to use `maxSGA` in practice.

3.1 The objective function

Unlike in `maxNR` and the related optimizers, SGA does not directly need the objective function `fn`. The function can still be provided (and perhaps will in most cases), but one can run the optimizer without it. If provided, the function can be used for printing the value at each epoch (by setting a suitable `printLevel` control option), and for stopping through *patience* stopping condition. If `fn` is not provided, do not forget to add the argument name for the gradient, `grad=`, as otherwise the gradient will be treated as the objective function with unexpected results!

If provided, the function should accept two (or more) arguments: the first must be the numeric parameter vector, and another one, named `index`, is the list of indices in the current minibatch.

As the function is not needed by the optimizer itself, it is up to the user to decide what it does. An obvious option is to compute the objective function value on the same minibatch as used for the gradient computation. But one can also opt for something else, for instance to compute the value on the validation data instead (and ignore the provided *index*). The latter may be a useful option if one wants to employ the patience-based stopping criteria.

3.2 Gradient function

Gradient is the work-horse of the SGA methods. Although `maxLik` can also compute numeric gradient using the finite difference method (this will be automatically done if the objective function is provided but the gradient isn't), this is not advisable, and may be very slow in high-dimensional problems. `maxLik` uses the numerator layout, i.e. the gradient should be a $1 \times K$ matrix where columns correspond to the components of the parameter vector $\boldsymbol{\theta}$. For compatibility with

other optimizers in `maxLik` it also accepts a observation-wise matrix where rows correspond to the individual observations and columns to the parameter vector components.

The requirements for the gradient function arguments are the same as for `fn`: the first formal argument must be the parameter vector, and it must also have an argument `index`, a numeric index for the observations to be included in the minibatch.

3.3 Stopping Conditions

`maxSGA` uses three stopping criteria:

- Number of epochs (control option `iterlim`): number of times all data is iterated through using the minibatches.
- Gradient norm. However, in case of stochastic approach one cannot expect the gradient at optimum to be close to zero, and hence the corresponding criterion (control option `gradtol`) is set to zero by default. If interested, one may make it positive.
- Patience. Normally, each new iteration has better (higher) value of the objective function. However, in certain situations this may not be the case. In such cases the algorithm does not stop immediately, but continues up to *patience* more epochs. It also returns the best parameters, not necessarily the last parameters.

  Patience can be controlled with the options `SG_patience` and `SG_patienceStep`. The former controls the patience itself–how many times the algorithm is allowed to produce an inferior result (default value `NULL` means patience criterion is not used). The latter controls how often the patience criterion is checked. If computing the objective function is costly, it may be useful to increase the patience step and decrease the patience.

3.4 Optimizers

`maxLik` currently implements two optimizers: *SGA*, the stock gradient ascent (including momentum), and *Adam*. Here we give some insight into the momentum, and into the Adam method, the basic gradient-only based optimization technique was explained in Section 2.

It is easy and intuitive to extend the SGA method with momentum. As implemented in `maxSGA`, the momentum μ ($0 < \mu < 1$) is incorporated into the the gradient update as

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{v}_t \quad \text{where} \quad \boldsymbol{v}_t = \mu \boldsymbol{v}_{t-1} + \rho \boldsymbol{g}(\boldsymbol{\theta}_t). \tag{4}$$

See Goodfellow *et al.* (2016, p. 288). The algorithm takes the initial "velocity" $\boldsymbol{v}_0 = \mathbf{0}$. It is easy to see that $\mu = 0$ is equivalent to no-momentum case, and if $\boldsymbol{g}(\boldsymbol{\theta})$ is constant, $\boldsymbol{v}_t \to \rho \boldsymbol{g}(\boldsymbol{\theta})/(1-\mu)$. So the movement speeds up in a region

with stable gradient. As a downside, it is also easier overshoot a maximum. But this behavior makes momentum-equipped SGA less prone of getting stuck in a local optimum. Momentum can be set by the control option `SG_momentum`, the default value is 0.

Adaptive Moments method, usually referred to as *Adam*, (Goodfellow *et al.*, 2016, p. 301) adapts the learning rate by variance of the gradient–if gradient components are unstable, it slows down, and if they are stable, it speeds up. The adaptation is proportional to the weighted average of the gradient divided by the square root of the weighted average of the gradient squared, all operations done component-wise. In this way a stable gradient component (where moving average is similar to the gradient value) will have higher speed than a fluctuating gradient (where the components frequently shift the sign and the average is much smaller). More specifically, the algorithm is as follows:

1. Initialize the first and second moment averages $\boldsymbol{s} = \boldsymbol{0}$ and $\boldsymbol{r} = \boldsymbol{0}$.
2. Compute the gradient $\boldsymbol{g}_t = \boldsymbol{g}(\boldsymbol{\theta}_t)$.
3. Update the average first moment: $\boldsymbol{s}_{t+1} = \mu_1 \boldsymbol{s}_t + (1-\mu_1)\boldsymbol{g}_t$. μ_1 is the decay parameter, the larger it is, the longer memory does the method have. It can be adjusted with the control parameter `Adam_momentum1`, the default value is 0.9.
4. Update the average second moment: $\boldsymbol{r}_{t+1} = \mu_2 \boldsymbol{r}_t + (1-\mu_2)\boldsymbol{g}_t \odot \boldsymbol{g}_t$ where $\odot$ denotes element-wise multiplication. The control parameter for the μ_2 is `Adam_momentum2`, the default value is 0.999.
5. As the algorithm starts with the averages $\boldsymbol{s}_0 = \boldsymbol{r}_0 = 0$, we also correct the resulting bias: $\hat{\boldsymbol{s}} = \boldsymbol{s}/(1-\mu_1^t)$ and $\hat{\boldsymbol{r}} = \boldsymbol{r}/(1-\mu_2^t)$.
6. Finally, update the estimate: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \rho\hat{\boldsymbol{s}}/(\delta + \sqrt{\hat{\boldsymbol{r}}})$ where division and square root are done element-wise and $\delta = 10^{-8}$ takes care of numerical stabilization.

Adam optimizer can be used with `maxAdam`.

3.5 Controlling Optimizers

Both `maxSGA` and `maxAdam` are designed to be similar to `maxNR`, and mostly expect similar arguments. In particular, both functions expect the objective function `fn`, gradient `grad` and Hessian function `hess`, and the initial parameter start values `start`. As these optimizers only need gradient, one can leave out both `fn` and `hess`. The Hessian is mainly included for compatibility reasons and only used to compute the final Hessian, if requested by the user. As SGA methods are typically used in contexts where Hessian is not needed, by default the algorithms do not return Hessian matrix and hence do not use the `hess` function even if provided. Check out the argument `finalHessian` if interested.

An important SGA-specific control options is `SG_batchSize`. This determines the batch size, or `NULL` for the full-batch approach.

Finally, unlike the traditional optimizers, stochastic optimizers need to know the size of data (argument `nObs`) in order to calculate the batches.

4 Example usage: Linear regression

4.1 Setting Up

We demonstrate the usage of `maxSGA` and `maxAdam` to solve a linear regression (OLS) problem. Although OLS is not a task where one commonly relies on stochastic optimization, it is a simple and easy-to understand model. We use the Boston housing data, a popular dataset where one traditionally attempts to predict the median house price across 500 neighborhoods using a number of neighborhood descriptors, such as mean house size, age, and proximity to Charles river. All variables in the dataset are numeric, and there are no missing values. The data is provided in *MASS* package.

First, we create the design matrix X and extract the house price y:

```
> i <- which(names(MASS::Boston) == "medv")
> X <- as.matrix(MASS::Boston[,-i])
> X <- cbind("const"=1, X)  # add constant
> y <- MASS::Boston[,i]
```

Although the model and data are simple, it is not an easy task for stock gradient ascent. The problem lies in different scaling of variables, the means are

```
> colMeans(X)

      const        crim          zn       indus
 1.00000000  3.61352356 11.36363636 11.13677866
       chas         nox          rm         age
 0.06916996  0.55469506  6.28463439 68.57490119
        dis         rad         tax     ptratio
 3.79504269  9.54940711 408.23715415 18.45553360
      black       lstat
356.67403162 12.65306324
```

One can see that *chas* has an average value 0.069 while that of *tax* is 408.237.

This leads to extremely elongated contours of the loss function: One can see that the ratio of the largest and the smallest eigenvalue is $\mathsf{X}^\mathsf{T}\mathsf{X} = 228400000$. Solely gradient-based methods, such as SGA, have trouble working in the resulting narrow valleys.

For reference, let's also compute the analytic solution to this linear regression model (reminder: $\hat{\boldsymbol{\beta}} = (\mathsf{X}^\mathsf{T}\mathsf{X})^{-1}\mathsf{X}^\mathsf{T}\boldsymbol{y}$):

```
> betaX <- solve(crossprod(X)) %*% crossprod(X, y)
> betaX <- drop(betaX)  # matrix to vector
> betaX
```

```
       const          crim            zn         indus
3.645949e+01 -1.080114e-01  4.642046e-02  2.055863e-02
        chas           nox            rm           age
2.686734e+00 -1.776661e+01  3.809865e+00  6.922246e-04
         dis           rad           tax       ptratio
-1.475567e+00  3.060495e-01 -1.233459e-02 -9.527472e-01
       black         lstat
9.311683e-03 -5.247584e-01
```

Next, we provide the gradient function. As a reminder, OLS gradient in numerator layout can be expressed as

$$\boldsymbol{g}_m(\boldsymbol{\theta}) = -\frac{2}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \left(y_i - \boldsymbol{x}_i^\top \cdot \boldsymbol{\theta} \right) \boldsymbol{x}_i^\top = -\frac{2}{|\mathcal{B}|} \left(y_{\mathcal{B}} - \mathsf{X}_{\mathcal{B}} \cdot \boldsymbol{\theta} \right)^\top \mathsf{X}_{\mathcal{B}} \tag{5}$$

where $y_{\mathcal{B}}$ and $\mathsf{X}_{\mathcal{B}}$ denote the elements of the outcome vector and the slice of the design matrix that correspond to the minibatch $\mathcal{B}$. We choose to divide the value by batch size $|\mathcal{B}|$ in order to have gradient values of roughly similar size, independent of the batch size. We implement it as:

```
> gradloss <- function(theta, index)  {
+    e <- y[index] - X[index,,drop=FALSE] %*% theta
+    g <- t(e) %*% X[index,,drop=FALSE]
+    2*g/length(index)
+ }
```

The `gradloss` function has two arguments: `theta` is the parameter vector, and `index` tells which observations belong to the current minibatch. The actual argument will be an integer vector, and hence we can use `length(index)` to find the size of the minibatch. Finally, we return the negative of (5) as `maxSGA` performs maximization, not minimization.

First, we demonstrate how the models works without the objective function. We have to supply the gradient function, initial parameter values (we use random normals below), and also `nObs`, number of observations to select the batches from. The latter is needed as the optimizer itself does not have access to data but still has to partition it into batches. Finally, we may also provide various control parameters, such as number of iterations, stopping conditions, and batch size. We start with only specifying the iteration limit, the only stopping condition we use here:

```
> library(maxLik)
> set.seed(3)
> start <- setNames(rnorm(ncol(X), sd=0.1), colnames(X))
>                             # add names for better reference
> res <- try(maxSGA(grad=gradloss,
+            start=start,
+            nObs=nrow(X),
```

```
+            control=list(iterlim=1000)
+            )
+     )

Iteration 63
Parameter:
[1] 3.47655620157556e+298, 1.55792823742485e+299, 3.46679084058245e+299, 4.20148533450887e+2
Gradient:
               [,1]           [,2]           [,3]
[1,] -2.176452e+304 -9.753202e+304 -2.170338e+305
               [,4]           [,5]           [,6]
[1,] -2.630284e+305 -1.489548e+303 -1.238498e+304
               [,7]
[1,] -1.359004e+305
 reached getOption("max.cols") -- omitted 7 columns
Error in maxSGACompute(fn = function (theta, fnOrig, gradOrig, hessOrig,  :
  NA/Inf in gradient
```

This run was a failure. We encountered a run-away growth of the gradient because the default learning rate $\rho = 0.1$ is too big for such strongly curved objective function. But before we repeat the exercise with a smaller learning rate, let's incorporate gradient clipping. Gradient clipping, performed with `SG_clip` control option, caps the L_2-norm of the gradient while keeping it's direction. We clip the squared norm at 10,000, i.e. the gradient norm cannot exceed 100:

```
> res <- maxSGA(grad=gradloss,
+               start=start,
+               nObs=nrow(X),
+               control=list(iterlim=1000,
+                            SG_clip=1e4)  # limit ||g|| <= 100
+               )
> summary(res)

--------------------------------------------
Stochastic Gradient Ascent
Number of iterations: 1000
Return code: 4
Iteration limit exceeded (iterlim)
Function value:
Estimates:
           estimate      gradient
const   -0.07999887 -1.749115e-05
crim     0.02785691 -7.755669e-05
zn       0.22208281 -1.754769e-04
indus    0.06456437 -2.106458e-04
chas     0.02077633 -1.198114e-06
```

```
nox      0.01196464 -9.941313e-06
rm       0.11108882 -1.092356e-04
age      1.20485974 -1.245784e-03
dis     -0.06026450 -6.282687e-05
rad      0.28567967 -1.921515e-04
tax      6.62820142 -7.689873e-03
ptratio  0.18507316 -3.261922e-04
black    5.81629057 -6.246515e-03
lstat    0.22176090 -2.326626e-04
-----------------------------------------
```

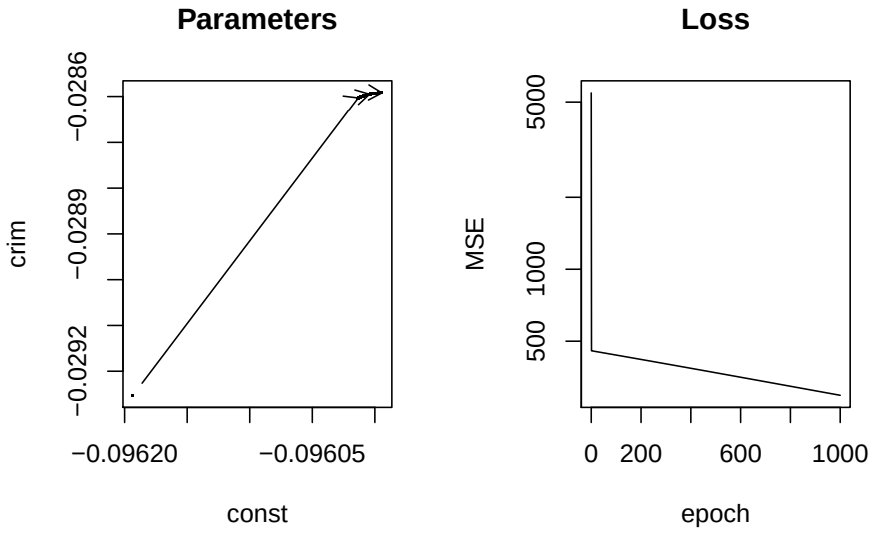
This time the gradient did not explode and we were able to get a result. But the estimates are rather far from the analytic solution shown above, e.g. the constant estimate -0.08 is very different from the corresponding analytic value 36.459. Let's analyze what is happening inside the optimizer. We can ask for both the parameter values and the objective function value to be stored for each epoch. But before we can store its value, in this case mean squared error (MSE), we have to supply an objective function to maxSGA. We compute MSE on the same minibatch as

```
> loss <- function(theta, index) {
+    e <- y[index] - X[index,] %*% theta
+    -crossprod(e)/length(index)
+ }
```

Now we can store the values with the control options `storeParameters` and `storeValues`. The corresponding numbers can be retrieved with `storedParameters` and `storedValues` methods. For `iterlim=R`, the former returns a $(R+1) \times K$ matrix, one row for each epoch and one column for each parameter component, and the latter returns a numeric vector of length $R + 1$ where R is the number of epochs. The first value in both cases is the initial value, so we have $R + 1$ values in total. Let's retrieve the values and plot both. We decrease the learning rate to 0.001 using the `SG_learningRate` control. Note that although we maximize negative loss, we plot positive loss.

```
> res <- maxSGA(loss, gradloss,
+               start=start,
+               nObs=nrow(X),
+               control=list(iterlim=1000,
+                            # will misbehave with larger numbers
+                            SG_clip=1e4,
+                            SG_learningRate=0.001,
+                            storeParameters=TRUE,
+                            storeValues=TRUE
+                            )
+               )
> par <- storedParameters(res)
```

```
> val <- storedValues(res)
> par(mfrow=c(1,2))
> plot(par[,1], par[,2], type="b", pch=".",
+      xlab=names(start)[1], ylab=names(start)[2], main="Parameters")
> ## add some arrows to see which way the parameters move
> iB <- c(40, nrow(par)/2, nrow(par))
> iA <- iB - 10
> arrows(par[iA,1], par[iA,2], par[iB,1], par[iB,2], length=0.1)
> ##
> plot(seq(length=length(val))-1, -val, type="l",
+      xlab="epoch", ylab="MSE", main="Loss",
+      log="y")
```



We can see how the parameters (the first and the second components, "const" and "crim" in this figure) evolve through the iterations while the loss is rapidly falling. One can see an initial jump where the loss is falling very fast, followed but subsequent slow movement. It is possible the initial jump be limited by gradient clipping.

4.2 Training and Validation Sets

However, as we did not specify the batch size, `maxSGA` will automatically pick the full batch (equivalent to control option `SG_batchSize = NULL`). So there was nothing stochastic in what we did above. Let us pick a small batch size–a single observation at time. However, as smaller batch sizes introduce more noise to the gradient, we also make the learning rate smaller and choose `SG_learningRate = 1e-5`.

But now the existing loss function, calculated just at the single observation, carries little meaning. Instead, we split the data into training and validation sets and feed batches of training data to gradient descent while calculating the loss on the complete validation set. This can be achieved with small modifications in the `gradloss` and `loss` function. But as the first step, we split the data:

```
> i <- sample(nrow(X), 0.8*nrow(X))  # training indices, 80% of data
> Xt <- X[i,]  # training data
> yt <- y[i]
> Xv <- X[-i,]  # validation data
> yv <- y[-i]
```

Thereafter we modify `gradloss` to only use the batches of training data while `loss` will use the complete validation data and just ignore `index`:

```
> gradloss <- function(theta, index)  {
+    e <- yt[index] - Xt[index,,drop=FALSE] %*% theta
+    g <- -2*t(e) %*% Xt[index,,drop=FALSE]
+    -g/length(index)
+ }
> loss <- function(theta, index) {
+    e <- yv - Xv %*% theta
+    -crossprod(e)/length(yv)
+ }
```

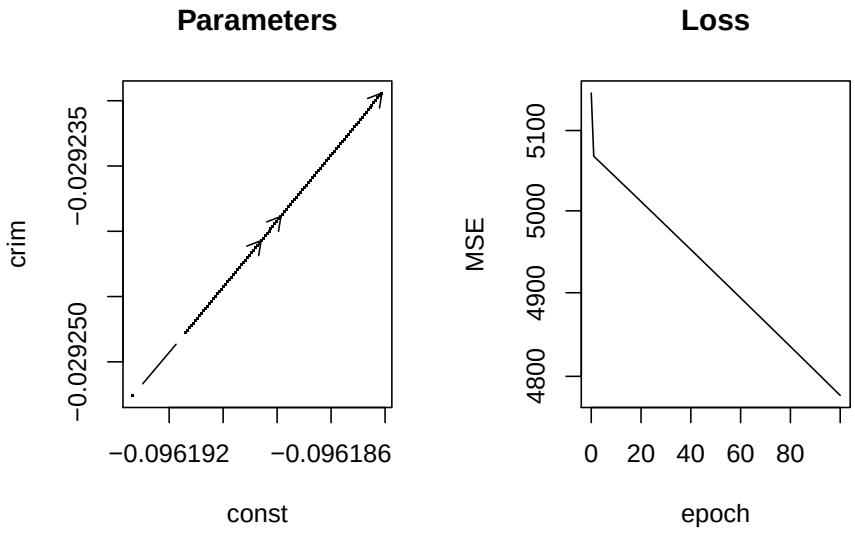
Note that because the optimizer only uses training data, the `nObs` argument now must equal to the size of training data in this case.

Another thing to discuss is the computation speed. `maxLik` implements SGA in a fairly complex loop that does printing, storing, and complex function calls, computes stopping conditions and does many other checks. Hence a smaller batch size leads to many more such auxiliary computations per epoch and the algorithm gets considerably slower. This is less of a problem for complex objective functions or larger batch sizes, but for linear regression, the slow-down is very large. For demonstration purposes we lower the number of epochs from 1000 to 100.

How do the convergence properties look now with the updated approach?

```
> res <- maxSGA(loss, gradloss,
+               start=start,
+               nObs=nrow(Xt),  # note: only training data now
+               control=list(iterlim=100,
+                            SG_batchSize=1,
+                            SG_learningRate=1e-5,
+                            SG_clip=1e4,
+                            storeParameters=TRUE,
+                            storeValues=TRUE
+                            )
+               )
```

```
> par <- storedParameters(res)
> val <- storedValues(res)
> par(mfrow=c(1,2))
> plot(par[,1], par[,2], type="b", pch=".",
+      xlab=names(start)[1], ylab=names(start)[2], main="Parameters")
> iB <- c(40, nrow(par)/2, nrow(par))
> iA <- iB - 1
> arrows(par[iA,1], par[iA,2], par[iB,1], par[iB,2], length=0.1)
> plot(seq(length=length(val))-1, -val, type="l",
+      xlab="epoch", ylab="MSE", main="Loss",
+      log="y")
```

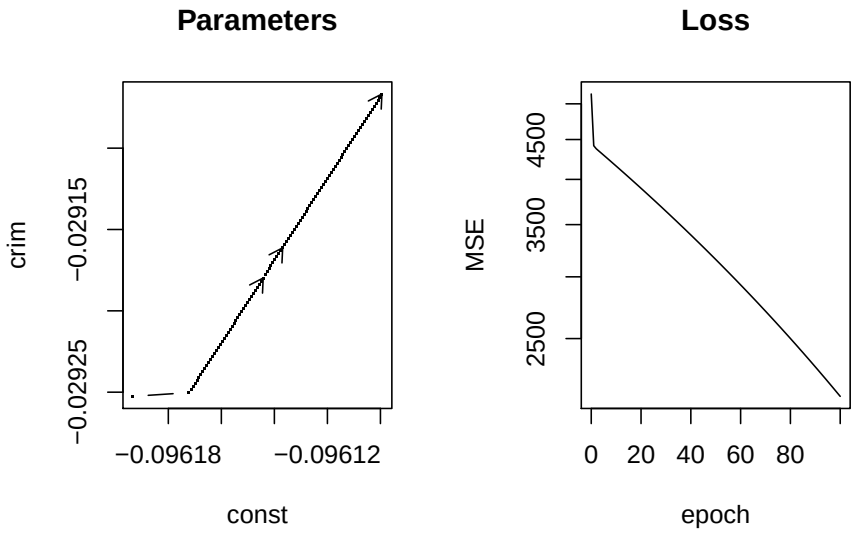


We can see the parameters evolving and loss decreasing over epochs. The convergence seems to be smooth and not ruptured by gradient clipping.

Next, we try to improve the convergence by introducing momentum. We add momentum $\mu = 0.95$ to the gradient and decrease the learning rate down to $1 \cdot 10^{-6}$:

```
> res <- maxSGA(loss, gradloss,
+               start=start,
+               nObs=nrow(Xt),
+               control=list(iterlim=100,
+                            SG_batchSize=1,
+                            SG_learningRate=1e-6,
+                            SG_clip=1e4,
+                            SGA_momentum = 0.99,
+                            storeParameters=TRUE,
```

```
+                             storeValues=TRUE
+                             )
+               )
> par <- storedParameters(res)
> val <- storedValues(res)
> par(mfrow=c(1,2))
> plot(par[,1], par[,2], type="b", pch=".",
+      xlab=names(start)[1], ylab=names(start)[2], main="Parameters")
> iB <- c(40, nrow(par)/2, nrow(par))
> iA <- iB - 1
> arrows(par[iA,1], par[iA,2], par[iB,1], par[iB,2], length=0.1)
> plot(seq(length=length(val))-1, -val, type="l",
+      xlab="epoch", ylab="MSE", main="Loss",
+      log="y")
```

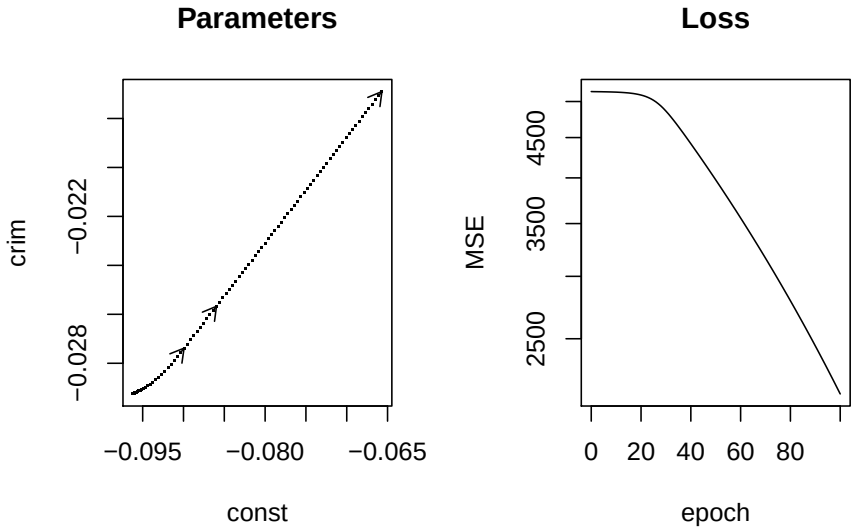


We achieved a lower loss but we are still far from the correct solution.

As the next step, we use Adam optimizer. Adam has two momentum parameters but we leave those untouched at the initial values. `SGA_momentum` is not used, so we remove that argument.

```
> res <- maxAdam(loss, gradloss,
+                start=start,
+                nObs=nrow(Xt),
+                control=list(iterlim=100,
+                             SG_batchSize=1,
+                             SG_learningRate=1e-6,
+                             SG_clip=1e4,
```

```
+                             storeParameters=TRUE,
+                             storeValues=TRUE
+                             )
+               )
> par <- storedParameters(res)
> val <- storedValues(res)
> par(mfrow=c(1,2))
> plot(par[,1], par[,2], type="b", pch=".",
+      xlab=names(start)[1], ylab=names(start)[2], main="Parameters")
> iB <- c(40, nrow(par)/2, nrow(par))
> iA <- iB - 1
> arrows(par[iA,1], par[iA,2], par[iB,1], par[iB,2], length=0.1)
> plot(seq(length=length(val))-1, -val, type="l",
+      xlab="epoch", ylab="MSE", main="Loss",
+      log="y")
```



As visible from the figure, Adam was marching toward the solution without any stability issues.

4.3 Sequence of Batch Sizes

The OLS' loss function is globally convex and hence there is no danger to get stuck in a local maximum. However, when the objective function is more complex, the noise that is generated by the stochastic sampling helps the algorithm to leave local maxima. A suggested strategy is to increase the batch size over time to achieve good exploratory properties early in the process and stable convergence later (see Smith *et al.*, 2018, for more information). This

approach is in some ways similar to Simulated Annealing.

Here we introduce such an approach by using batch sizes $B = 1$, $B = 10$ and $B = 100$ in succession. We also introduce patience stopping condition. If the objective function value is worse than the best value so far for more than *patience* times then the algorithm stops. Here we use patience value 5. We also store the loss values from all the batch sizes into a single vector `val`. If the algorithm stops early, some of the stored values are left uninitialized (`NA`-s), hence we use `na.omit` to include only the actual values in the final `val`-vector. We allow the algorithm to run for 200 epochs, but as we now have introduced early stopping through patience, the actual number of epochs may be less than that.

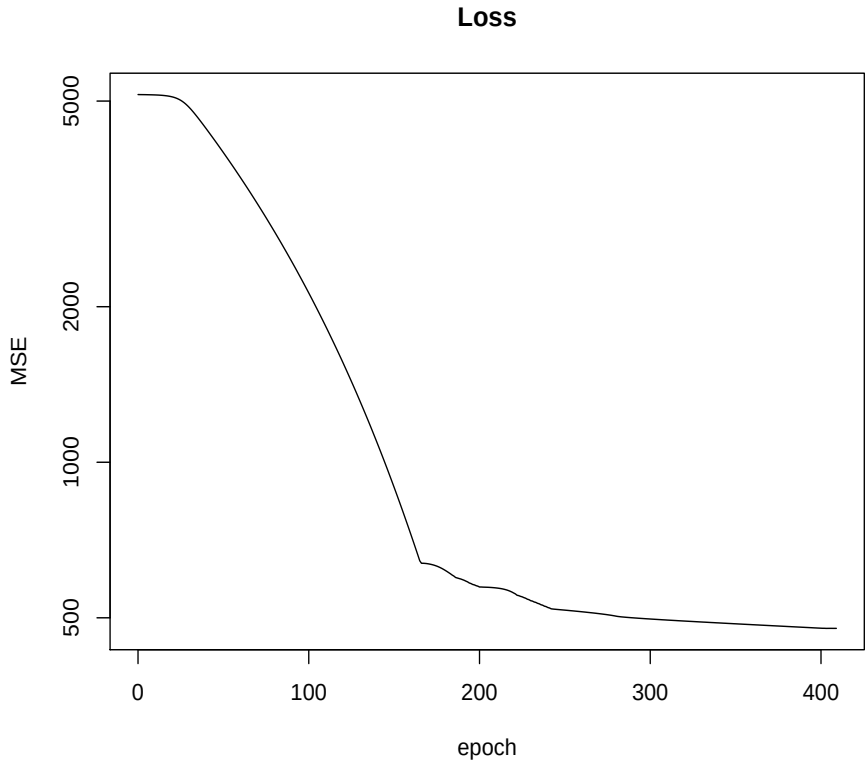
```
> val <- NULL
> # loop over batch sizes
> for(B in c(1,10,100)) {
+    res <- maxAdam(loss, gradloss,
+                   start=start,
+                   nObs=nrow(Xt),
+                   control=list(iterlim=200,
+                                SG_batchSize=1,
+                                SG_learningRate=1e-6,
+                                SG_clip=1e4,
+                                SG_patience=5,
+                              # worse value allowed only 5 times
+                                storeValues=TRUE
+                                )
+                   )
+    cat("Batch size", B, ",", nIter(res),
+        "epochs, function value", maxValue(res), "\n")
+    val <- c(val, na.omit(storedValues(res)))
+    start <- coef(res)
+ }

Batch size 1 , 200 epochs, function value -573.5616
Batch size 10 , 200 epochs, function value -477.1564
Batch size 100 , 7 epochs, function value -476.9687

> plot(seq(length=length(val))-1, -val, type="l",
+      xlab="epoch", ylab="MSE", main="Loss",
+      log="y")
> summary(res)

------------------------------------------
Stochastic Gradient Ascent/Adam
Number of iterations: 7
Return code: 10
Lost patience (SG_patience)
```

```
Function value: -476.9687
Estimates:
            estimate     gradient
const   -0.039415021 1.315825e-05
crim    -0.006873007 4.571544e-05
zn       0.097802487 0.000000e+00
indus   -0.066295960 2.381643e-04
chas     0.057364690 1.315825e-05
nox      0.058696481 9.447622e-06
rm       0.065297664 1.155294e-04
age      0.164299062 1.090819e-03
dis     -0.070026405 2.506251e-05
rad      0.171508473 3.157979e-04
tax     -0.019731897 8.763393e-03
ptratio -0.056611015 2.657966e-04
black   -0.007726219 4.665257e-03
lstat    0.074810043 6.960713e-05
--------------------------------------------
```



Two first batch sizes run through all 200 epochs, but the last run stopped

early after 7 epochs only. The figure shows that Adam works well for approximately 170 epochs, thereafter the steady pace becomes uneven. It may be advantageous to slow down the movement further.

As explained above, this dataset is not an easy task for methods that are solely gradient-based, and so we did not achieve a result that is close to the analytic solution. But our task here is to demonstrate the usage of the package, not to solve a linear regression exercise. We believe every *R*-savy user can adapt the method to their needs.

References

Bottou, L., Curtis, F. and Nocedal, J. (2018) Optimization methods for large-scale machine learning, *SIAM Review*, **60**, 223–311.

Goodfellow, I. J., Bengio, Y. and Courville, A. (2016) *Deep Learning*, MIT Press.

Henningsen, A. and Toomet, O. (2011) maxlik: A package for maximum likelihood estimation in r, *Computational Statistics*, **26**, 443–458, 10.1007/s00180-010-0217-1.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M. and Tang, P. T. P. (2016) On large-batch training for deep learning: Generalization gap and sharp minima, *ArXiv*, **abs/1609.04836**.

Smith, S. L., Kindermans, P.-J. and Le, Q. V. (2018) Don't decay the learning rate, increase the batch size, *ArXiv*, **abs/1711.00489**.