Stochastic Gradient Ascent in maxLik

Ott Toomet

June 4, 2020

1 `maxLik` and Stochastic Gradient Ascent

`maxLik` is a package, primarily intended with Maximum Likelihood and related estimations. Besides of its name, it also includes a number of tools geared 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.

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 very easy, essentially just slow 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 typical tasks, the objective function $f(\boldsymbol{\theta})$ depends on data X in an additive form $f(\boldsymbol{\theta}; \mathsf{X}) = \sum_i f(\boldsymbol{\theta}; \boldsymbol{x}_i)$ where i denotes "observations", typically rows of the design matrix X that are independent of each other.

The above introduction did not specify how to compute the gradient $\boldsymbol{g}(\boldsymbol{\theta}_0)$ in terms of which data vectors $\boldsymbol{x}$ to include. A natural approach is to include the complete data and compute

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

This approach is called "full batch" and it has a number of advantages. In particular, it is deterministic (given data X), and the sum in (1) can be easily

parallelized in typical applications. However, there are also a number of reasons why full-batch approach may not be desirable (see Bottou *et al.*, 2018):

- In practical applications, there is often a lot of redundancy in different observations in data. When always using all the observations for the update means spending a substantial effort on redundant calculations.
- Full-batch gradient lacks the stochastic noise. While advantageous in the latter steps of optimization, the noise helps the optimizer to avoid local maxima 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).
- The advantage of SGA grows in sample size N. 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 (Keskar *et al.*, 2016) that do not generalize well to validation data. Small batch approach leads to better validation performance.

In what is usually referred to as "stochastic gradient ascent" in the literature, refers to the case where the gradient is computed on a single observation:

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

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*, a full walk-through of the data. In between the full-batch and stochastic gradient there is *minibatch* gradient

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

where $\mathcal{B}$ is the batch, a set of observations' indices, that are included in the gradient computation. In applications, one has to construct a series of minibatches that cover the complete data, and walk through those 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). Both methods mostly follow 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 and usage of SGA.

3.1 The objective function

Unlike in `maxNR` and related functions, SGA does not directly need the objective function values. The function can still be provided (and probably will in most cases), but one can run the optimizer with only gradient. If provided, the function can be used for printing the value at each epoch for following the process, and for stopping through *patience*: normally, the new iteration has better (higher) value of the objective function. However, in unfavorable situations it is often not the case. In such a case the algorithm continues not more than *patience* times before stopping (and returning the best parameters, not necessarily the last parameters).

If provided, the function should accept two (or more) arguments: the first is the numeric parameter vector, and the second, called `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 should do. 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 an 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 is not advisable, and may be very slow in high-dimensional problems. The gradient should be a $1 \times K$ matrix in numerator layout, i.e. each column corresponds to the corresponding component in the parameter vector $\boldsymbol{\theta}$.

4 Example usage cases

4.1 Linear regression

We demonstrate linear regression (OLS) as the first example, mostly for illustrative purposes, as OLS is an 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 variables, 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]
> eigenvals <- eigen(crossprod(X))$values
```

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 highly elliptical parabola of with the ratio of largest and smallest eigenvalues of $\mathsf{X}^\intercal\mathsf{X} = 228400000$. Solely gradient-based methods as SGA have trouble working in resulting narrow valleys.

For reference, let's also compute the analytic solution to this linear regression model (reminder: $\hat{\boldsymbol{\beta}} = (\mathsf{X}^\intercal\mathsf{X})^{-1}\mathsf{X}\boldsymbol{y}$):

```
> betaX <- solve(crossprod(X)) %*% crossprod(X, y)
> betaX <- drop(betaX)
> 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

$$\boldsymbol{g}_m(\boldsymbol{\theta}) = -\frac{2}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}\left(y_i - \boldsymbol{x}_i^\intercal\cdot\boldsymbol{\theta}\right)\boldsymbol{x}_i^\intercal = -\frac{2}{|\mathcal{B}|}\left(y_\mathcal{B} - \mathsf{X}_\mathcal{B}\cdot\boldsymbol{\theta}\right)^\intercal\mathsf{X}_\mathcal{B} \tag{4}$$

where $y_\mathcal{B}$ and $\mathsf{X}_\mathcal{B}$ denote the outcomes and rows of the design matrix that corresponds to the minibatch $\mathcal{B}$. We implement it as:

```
> gradloss <- function(theta, index)  {
+    e <- y[index] - X[index,,drop=FALSE] %*% theta
+    g <- -2*t(e) %*% X[index,,drop=FALSE]
+    -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 (4) 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, 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, 1.55792823742484e+299, 3.46679084058244e+299, 4.20148533450886e+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 and parameter values. This is 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 try gradient clipping. The clipping, performed with `SG_clip` control option, limits the squared L_2-norm of the gradient with the given value while keeping it's direction:

```
> res <- maxSGA(grad=gradloss,
+               start=start,
```

```
+               nObs=nrow(X),
+               control=list(iterlim=1000,
+                            SG_clip=1e4)  # allow ||g|| <= 100
+               )
> summary(res)

------------------------------------------
Stochastic Gradient Ascent
Number of iterations: 1000
Return code: 4
Iteration limit exceeded.
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
------------------------------------------
```

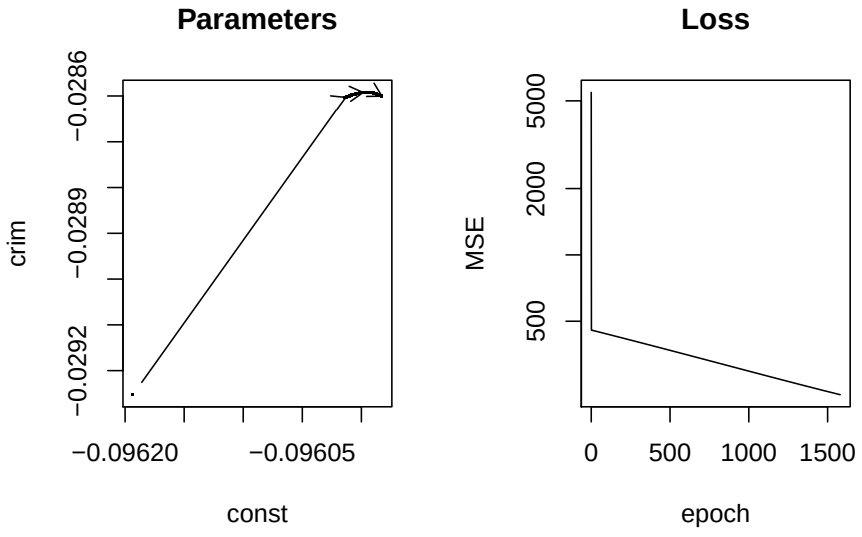
This time the gradient did not explode and we were able to get a result. But the estimates are rather far from the analytic solution shown above, in particular the constant estimate -0.08 is very different from the corresponding analytic value 36.459. Let's analyze what is happening inside the optimizer. We can ask for both the parameter values and the objective function value to be stored for each epoch. But before we can store the objective function (MSE) value, we have to supply it. We compute MSE on the same dataset 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. The former returns a $K \times R+1$ 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, that's why we have $R+1$ values in total. Let's retrieve the values and plot both. We also decrease the learning rate to 0.001 using the `SG_learningRate` control:

```
> res <- maxSGA(loss, gradloss,
+               start=start,
+               nObs=nrow(X),
+               control=list(iterlim=1580,
+                            # will misbehave with larger numbers
+                            SG_clip=1e4,  # allow ||g|| <= 100
+                            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")
> 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 (first and second component here) 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 is 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, for instance 1, a single observation at time. However, as smaller batch sizes introduce more noise to the gradient, we also make the learning rate accordingly smaller and choose `SG_learningRate = 2e-4`.

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, feed batches of training data to gradient but calculate the loss on the complete validation set. This can be achieved with small modifications in the `gradloss` and `loss` function. But first, let's split data:

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

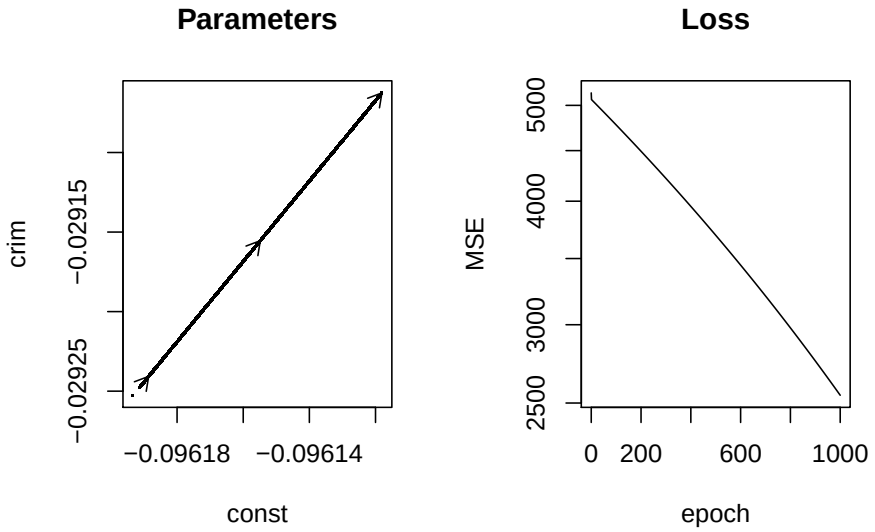
Thereafter we modify `gradloss` to only use the batches of training data while `loss` will use the validation data and just ignore `index`:

```
> 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)
+ }
```

Another thing to note is the speed. `maxLik` implements SGA in a fairly complex loop that does printing, storing, stopping conditions, and complex function calls. Hence smaller batch size means much more such auxiliary computations per epoch and the algorithm slows considerably down. 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=1000,
```

```
+                             SG_batchSize=1,
+                             SG_learningRate=1e-5,
+                             # will misbehave with larger numbers
+                             SG_clip=1e4,  # allow ||g|| <= 100
+                             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. With momentum μ, the gradient update is

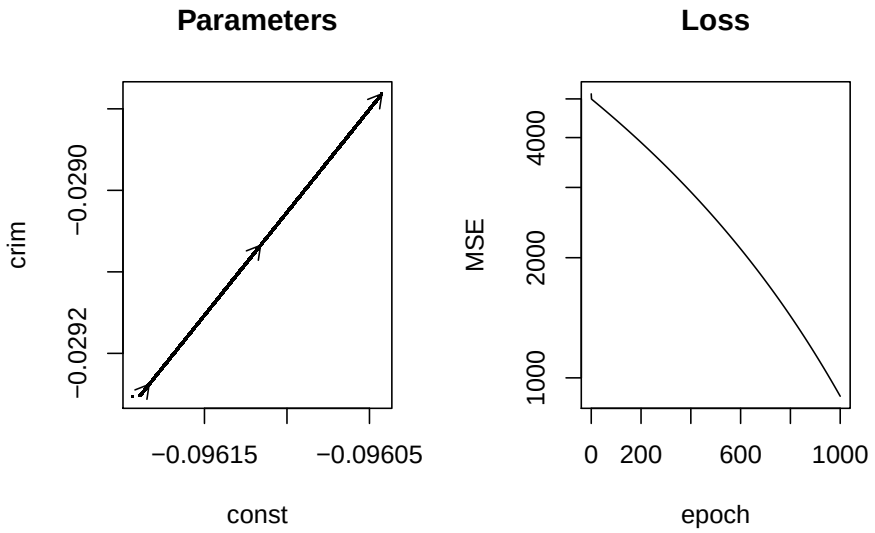
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{v}_t \quad \text{where} \quad \boldsymbol{v}_t = \mu \boldsymbol{v}_{t-1} + \rho \boldsymbol{g}(\boldsymbol{\theta}). \tag{5}$$

See Goodfellow *et al.* (2016, p. 288). The algorithm takes the initial "velocity"

$\boldsymbol{v}_0 = \mathbf{0}$.

Let's add momentum $\mu = 0.95$ to the gradient (and also decrease the learning rate):

```
> res <- maxSGA(loss, gradloss,
+               start=start,
+               nObs=nrow(Xt),
+               control=list(iterlim=1000,
+                            SG_batchSize=1,
+                            SG_learningRate=2e-7,
+                            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")
```

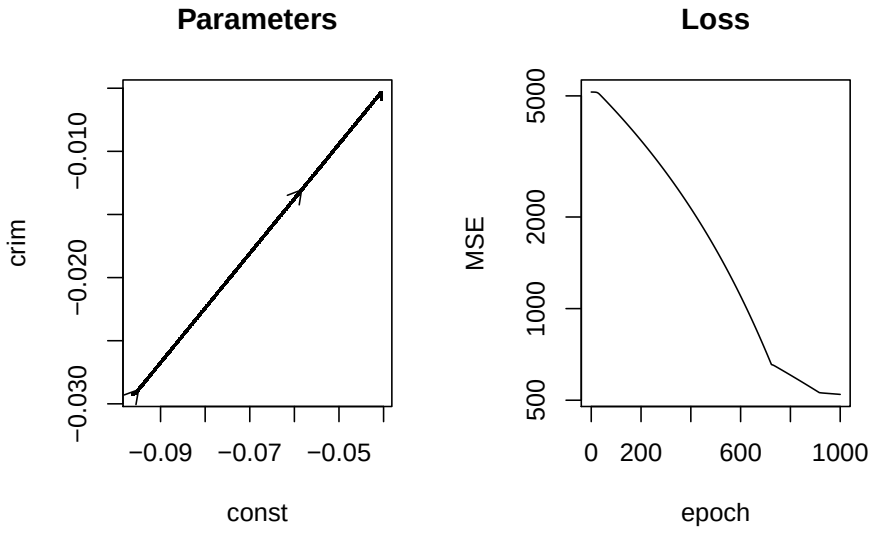


We achieved a lower loss but we are still far from the correct solution.

As the next step, we use Adaptive Moment (Adam) optimizer (Goodfellow *et al.*, 2016, p. 301). Adam 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 moving average of the gradient divided by the square root of the moving 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 first and second moment averages $\boldsymbol{s} = \boldsymbol{0}$ and $\boldsymbol{r} = \boldsymbol{0}$.
2. Compute the gradient $\boldsymbol{g} = \boldsymbol{g}(\boldsymbol{\theta}_t)$.
3. Update the moving average for the first moment: $\boldsymbol{s}_{t+1} = \mu_1 \boldsymbol{s}_t + (1 - \mu_1)\boldsymbol{g}$. 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` with the default value 0.9.
4. ... and the second moment: $\boldsymbol{r}_{t+1} = \mu_1 \boldsymbol{r}_t + (1 - \mu_1)\boldsymbol{g} \odot \boldsymbol{g}$ where $\odot$ denotes element-wise multiplication. The control parameter for the μ_2 is `Adam_momentum2` with the default value 0.999.
5. As the algorithm starts with the moving averages equal to zero, we also correct the resulting bias: $\hat{\boldsymbol{s}} = \boldsymbol{s}/(1 - \mu_1^t)$ and $\hat{r} = \boldsymbol{s}/(1 - \mu_1^t)$.
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.

```
> res <- maxAdam(loss, gradloss,
+                start=start,
+                nObs=nrow(Xt),
+                control=list(iterlim=1000,
+                             SG_batchSize=1,
+                             SG_learningRate=2e-7,
+                             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, we achieved a lower loss (approximately 500) but run into stability problems at the end.

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 optimum. However, when the objective function is more complex, the noise, 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 many 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 more than *patience* times, 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`.

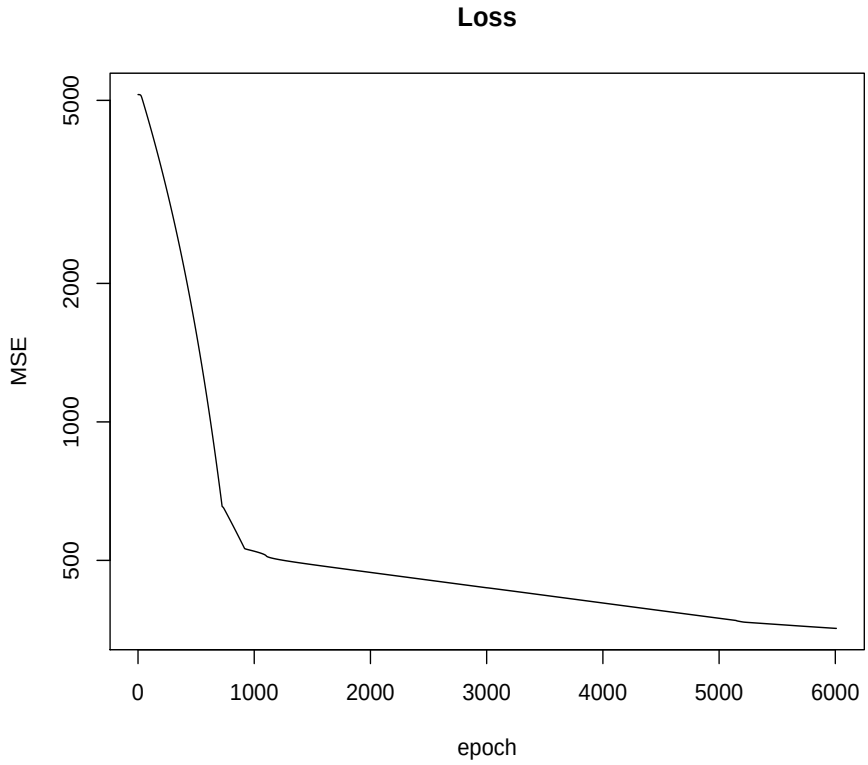
```
> val <- NULL
> for(B in c(1,10,100)) {
+    res <- maxAdam(loss, gradloss,
+                   start=start,
+                   nObs=nrow(Xt),
+                   control=list(iterlim=3000,
+                                SG_batchSize=1,
+                                SG_learningRate=2e-7,
+                                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 , 3000 epochs, function value -435.957
Batch size 10 , 3000 epochs, function value -355.8042
Batch size 100 , 7 epochs, function value -355.774

> 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
```

```
Function value: -355.774
Estimates:
            estimate     gradient
const   -0.038400125 1.315825e-05
crim    -0.015655654 4.571544e-05
zn       0.241572420 0.000000e+00
indus   -0.066340367 2.381643e-04
chas     0.141215599 1.315825e-05
nox      0.059187895 9.447622e-06
rm       0.070048257 1.155294e-04
age      0.164455271 1.090819e-03
dis     -0.063369610 2.506251e-05
rad      0.171111901 3.157979e-04
tax     -0.019615094 8.763393e-03
ptratio -0.055830146 2.657966e-04
black   -0.003546472 4.665257e-03
lstat    0.071757242 6.960713e-05
-------------------------------------------
```



One can see that both first loops ($B = 1$ and $B = 10$) run over all 3000

epochs, but the last one only iterated 7 times–apparently it almost immediately overshot the maximum. A solution might be to decrease the learning rate. But we stop here as we believe every *R*-savy user can adapt the method to their need.

As explained above, this dataset is not an easy task for methods that are solely gradient-based. However, our task here is to demonstrate the usage of the package, not to solve a linear regression exercise.

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**.