Gradient descent optimization using automatic differentiation

F. Cheysson, J. Chiquet, M. Mariandassou, T. Mary-Huard

Finist'R 2022

Context

Most inference problems in statistics/AI boil down to optimization.

Exemple Logistic regression
Assume $(X_i, Y_i) \in \mathbb{R}^d \times \{0, 1\}$, $i = 1, ..., n$, and

$$Y_i | x_i \sim B(p_{x_i}) \text{ ind, where } p_{x_i} = \frac{e^{x_i^T \theta}}{1 + e^{x_i^T \theta}}$$

Inference
Via Maximum Likelihood:

$$\begin{aligned}
\hat{\theta} &= \arg\min_{\theta} \left\{ -\sum_i \mathcal{L}(x_i, y_i; \theta) \right\} \\
&= \arg\min_{\theta} \left\{ -\sum_i y_i x_i^T \theta - \log\left(1 + e^{x_i^T \theta}\right) \right\} \\
&= \arg\min_{\theta} \; f(\theta)
\end{aligned}$$

Gradient descent

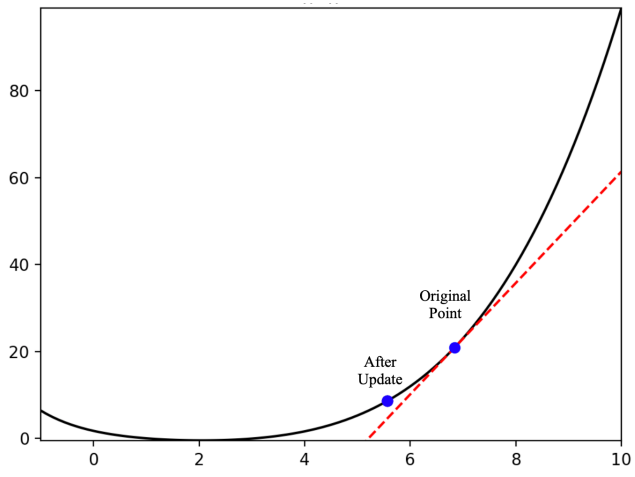
Let $f : \mathbb{R}^d \to \mathbb{R}$ be a convex and differentiable function.
Assume there exists

$$x^* = \arg\min_x \ f(x)$$

Gradient descent algorithm

Require: x_0, λ_t, η $Crit = 1$
  while $Crit > \eta$ **do**
    $x_{t+1} = x_t - \lambda_t \nabla f(x_t)$
    $Crit = ||x_{t+1} - x_t||$
  end while
  return x_{t+1}

x_0 initialization $\lambda_t > 0$ sequence of step sizes η precision



Gradient descent

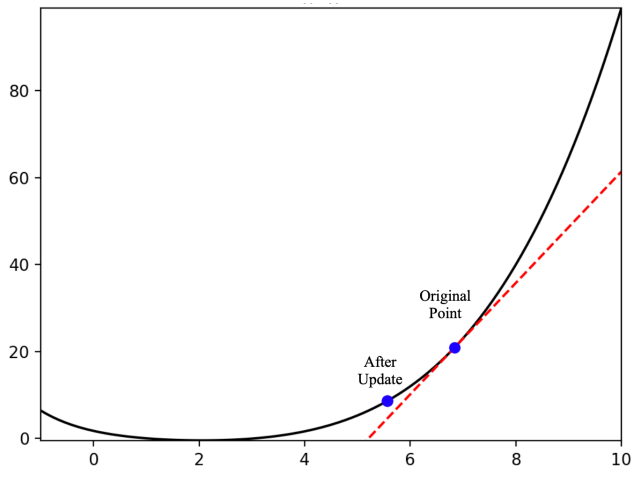
Let $f : \mathbb{R}^d \to \mathbb{R}$ be a convex and differentiable function.
Assume there exists

$$x^* = \arg\min_x \ f(x)$$

Gradient descent algorithm

Require: x_0, λ_t, η $Crit = 1$
  while $Crit > \eta$ **do**
    $x_{t+1} = x_t - \lambda_t \nabla f(x_t)$
    $Crit = ||x_{t+1} - x_t||$
  end while
  return x_{t+1}

x_0 initialization $\lambda_t > 0$ sequence of step sizes η precision



Why using gradient descent ?

- Theoretical guarantees (see Joon lecture),
- First order method $\Rightarrow$ scalable,
- Easy to implement,
- Many existing refinements (momentum, acceleration, stochastic gradient...).

Still...

Requires access to $\nabla f(x_t)$ at each step t.

- derive a close form expression for ∇f OR
- compute $\nabla f(x_t)$ numerically.

$\Rightarrow$ Automatic Differentiation!

Rmk: AD provides the exact value of the gradient at a given point (i.e. no numerical approximation).

Principle of Automatic differentiation

Composite function

Assume that we aim at finding

$$x^* = \arg\min_x \ f(x),$$

where f is a composite function, i.e.

$$f = f_K \circ f_{K-1} \circ ... \circ f_2 \circ f_1$$

with $f_k, \ k = 1, ..., K$ a "basic" function.

Chaining rule

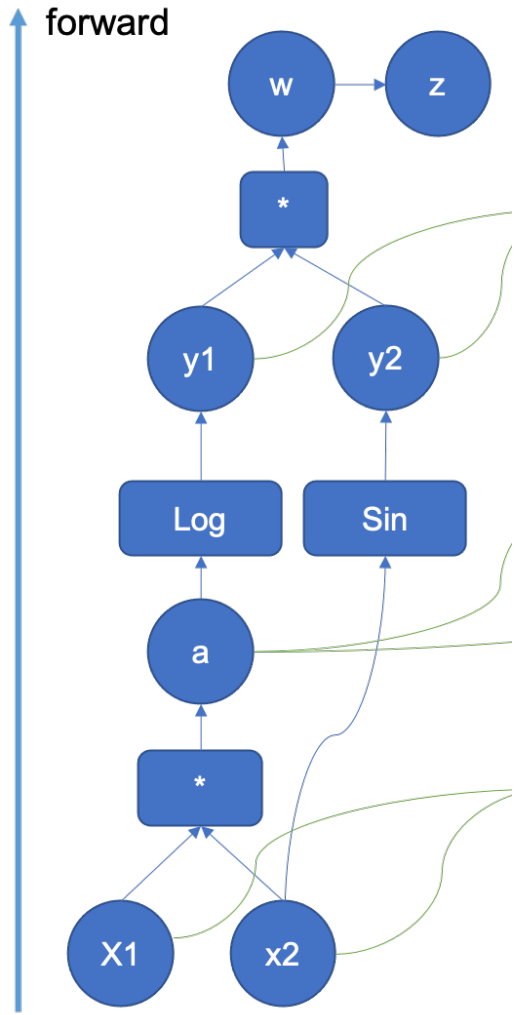
$$\begin{aligned} \text{Let } y_1 &= f_1(x) \\ y_2 &= f_2(y_1) \\ &... \\ y_K &= f_K(y_{K-1}) = f(x) \end{aligned}$$

Then

$$\frac{\partial f}{\partial x}(x) = \frac{\partial f_K}{\partial y_{K-1}}(y_{K-1}) \times \frac{\partial f_{K-1}}{\partial y_{K-2}}(y_{K-2}) \times ... \times \frac{\partial f_1}{\partial x}(x)$$

Computational graph

$$\begin{array}{rccl} \text{Consider } f: & \mathbb{R}^2 & \to & \mathbb{R} \\ & (x_1, x_2) & \mapsto & \log(x_1 x_2)\sin(x_2) = z \end{array}$$



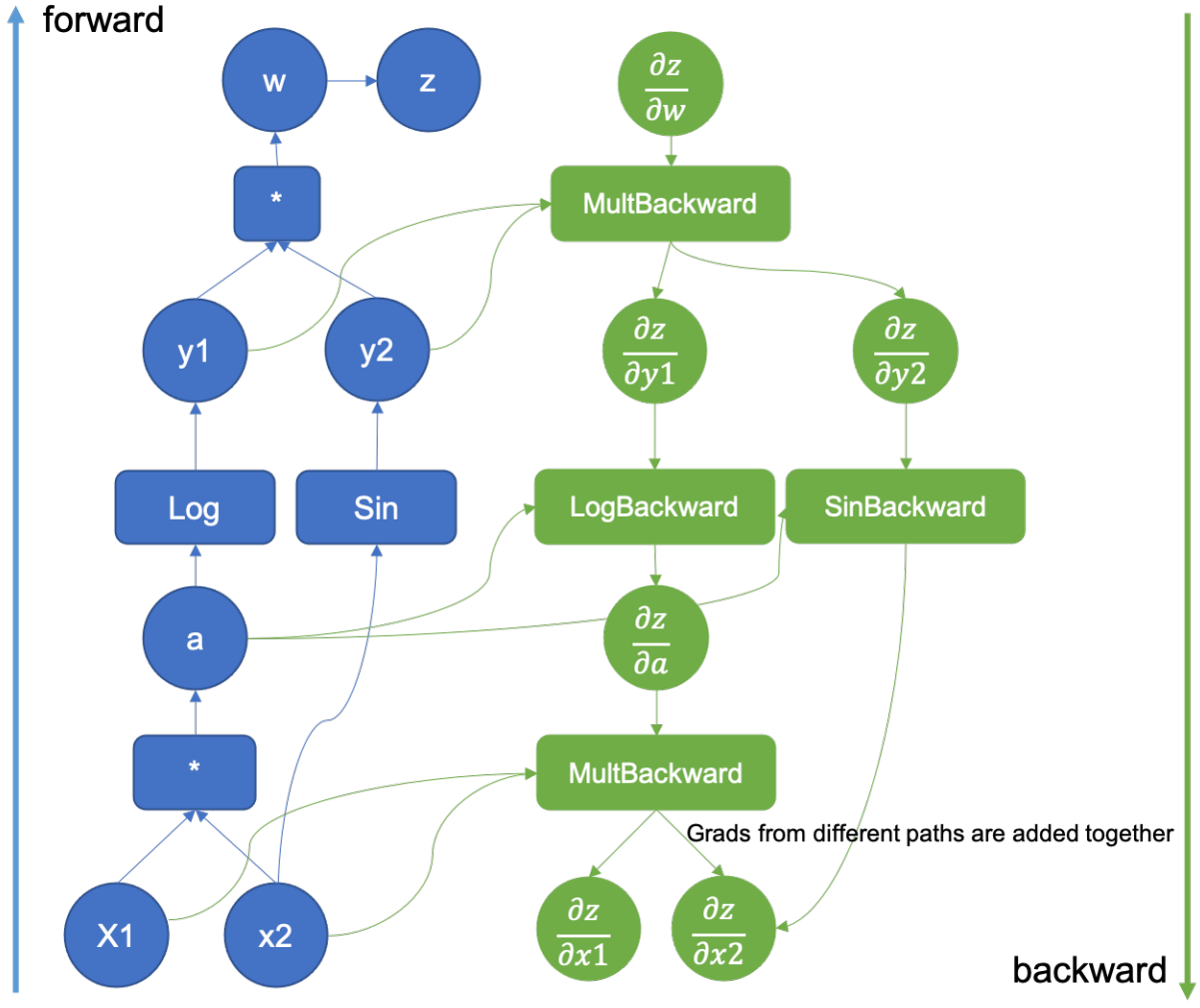
Behind the forward path: Autograd

- Every time the engine executes an operation in the graph, the derivative of that operation is added to the graph.
- These derivatives will be executed later in the backward path.
- This assumes that the engine knows the derivatives of the basic functions.

$\Rightarrow$ At the end of a forward pass, the **computational graph** the **results** (i.e. the computed outputs) and the **backward path** are ready to be used.

Computational Graph extended with the backward path

Consider $f(x_1, x_2) = \sin(x_2)\log(x_1 x_2) = z$



Backward operators of basic functions...

... are not regular derivatives !

Why ?

To flow through the backward path, at operator node f_k one needs to output

$$\frac{\partial f_K}{\partial y_k}(y_{K-1}) = \frac{\partial f_K}{\partial y_{K-1}}(y_{K-1}) \times \frac{\partial f_{K-1}}{\partial y_{K-2}}(y_{K-2}) \times ... \times \frac{\partial f_k}{\partial y_{k-1}}(y_{k-1})$$

rather than $\frac{\partial f_k}{\partial y_{k-1}}(y_{k-1})$ only.

Rmk A basic function is an autograd function with both a forward **and** a backward methods available.

Summary

To perform automatical differentiation, one needs,

- Well defined basic functions with forward and backward methods
- Automatic construction of the computational graph and backward path
- Efficient storage of function outputs and by-products

$\Rightarrow$ R package `torch`!

Easy to install, no dependence to python or reticulate.

torch objects

Compared with basic R, torch deals with tensors:

```
library(torch)
x.r <- rnorm(3)
y.r <- as.matrix(rnorm(3))
x.torch <- torch_tensor(x.r)
y.torch <- torch_tensor(y.r)
```

Very similar to R, but pay attention to details...

```
x.r + y.r
```

```
          [,1]
[1,] 0.2489226
[2,] 0.7104369
[3,] 1.9373319
```

```
x.torch + y.torch
```

```
torch_tensor
 0.2489  2.1009  0.3943
-1.1415  0.7104 -0.9961
 1.7919  3.6439  1.9373
[ CPUFloatType{3,3} ]
```

Trainable `torch` objects

$$\text{Recall} \quad \hat{\theta} = \arg\min_{\theta} \left\{ -\sum_{i} \mathcal{L}(x_i, y_i; \theta) \right\}.$$

- (x_i, y_i) are static values (i.e. data) that do not require gradient,
- θ is a trainable value (i.e. learnable parameter) that requires gradient.

Example

```
x = torch_tensor(3,requires_grad = TRUE)
y = x %>% torch_log %>% torch_square ## y=f(x)=log(x)^2
x
```

```
torch_tensor
 3
[ CPUFloatType{1} ][ requires_grad = TRUE ]
```

```
y
```

```
torch_tensor
 1.2069
[ CPUFloatType{1} ][ grad_fn = <PowBackward0> ]
```

Gradient evaluation

- The gradient computation requires the computational graph, stored in the function output:

```
y$grad_fn
```

```
PowBackward0
```

```
y$grad_fn$next_functions
```

```
[[1]]
LogBackward0
```

- Gradient is computed when the backward pass is applied, and stored in the input variable:

```
y$backward()
x$grad
```

```
torch_tensor
 0.7324
[ CPUFloatType{1} ]
```

Application: logistic regression

$$\text{Recall} \quad \hat{\theta} = \arg\min_{\theta} \left\{ -\sum_i y_i x_i^T \theta - \log\left(1 + e^{x_i^T \theta}\right) \right\}$$

```
logistic_loss <- function(theta, x, y) {
  ## Compute x_i*\theta
  odds <- torch_matmul(x, theta)
  log_lik <- torch_dot(y, odds) - torch_sum(torch_log(1 + torch_exp(odds)))
  return(-log_lik)
}
```

Application: logistic regression

Create the different `torch` objects

```
## Build data tensors from existing R objects X and Y
x <- torch_tensor(X)
y <- torch_tensor(Y)
## Build the trainable tensor theta
theta_current <- torch_tensor(rep(0, ncol(X)), requires_grad = TRUE)
```

and choose your favorite gradient descent method

```
## Choose among
theta_optimizer <- optim_adam(theta_current)
theta_optimizer <- optim_sgd(theta_current,lr=0.01)
theta_optimizer <- optim_rprop(theta_current)
### ...
```

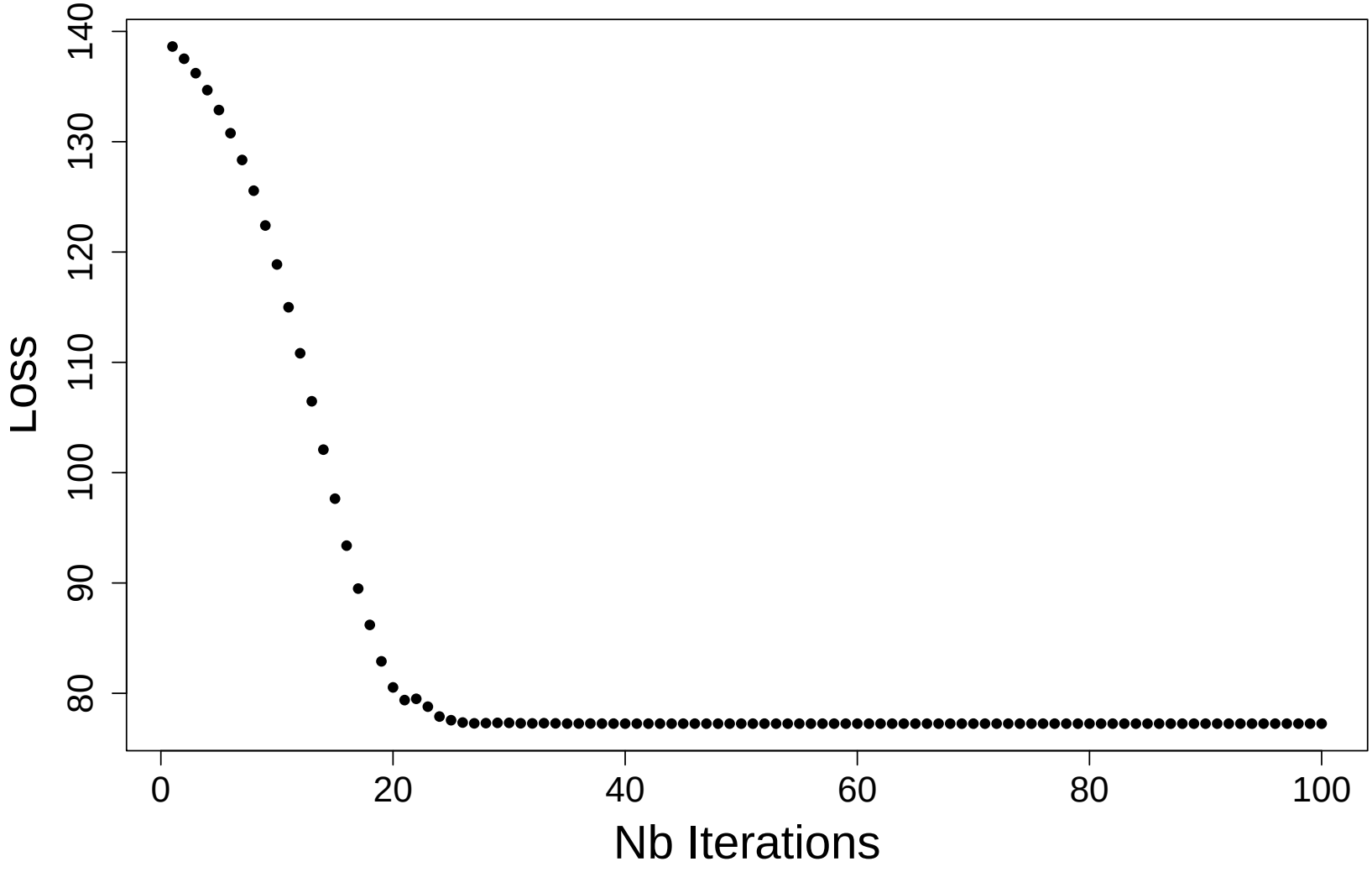
Application: logistic regression

Now loop !

```
## Parameters
num_iterations <- 100
loss_vector <- vector("numeric", length = num_iterations)

for (i in 1:num_iterations) {

  ## Set the derivatives at 0
  theta_optimizer$zero_grad()

  ## Forward
  loss <- logistic_loss(theta_current, x, y)

  ## Backward
  loss$backward()

  ## Update parameter
  theta_optimizer$step()

  ## Store the current loss for graphical display
  loss_vector[i] <- loss %>% as.numeric()
}
```

Application: logistic regression

Convergence ?



Automatic Differentiation for Neural Networks ?

Stay tuned on State Of The R!

Upcoming session on **Variational Auto Encoders** including

- Personalized datasets,
- Mini Batches,
- Convolutional NN,
- Dropout.

Some references

https://stateofther.github.io/finistR2022/autodiff.html

https://pytorch.org/blog/overview-of-pytorch-autograd-engine/

https://pytorch.org/blog/computational-graphs-constructed-in-pytorch/

https://pytorch.org/blog/how-computational-graphs-are-executed-in-pytorch/