Gradient Descent

ref. from book **"Data Science from Scratch"**, Chap 8

```
begin
  using Test
  using Plots
  using PlutoUI
  using Random
  using Distributions

  push!(LOAD_PATH, "./src")
  using YaLinearAlgebra
```

TOC

back to TOC

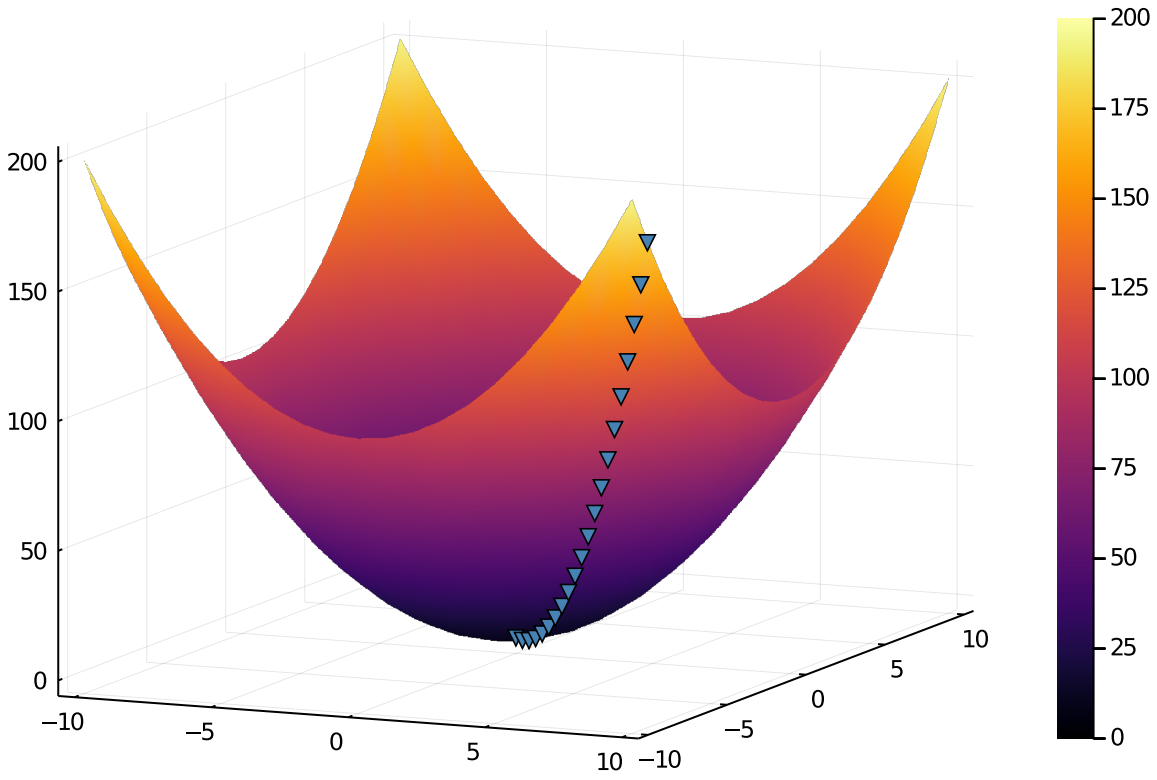
Context

The gradient (vector of partial derivatives) gives the direction in which the function most quickly increase.

In ML we are looking at a best model (relative to some criteria) to fit the data. In this context "best" is obtained by minimizing a quantity (ex. minimize MSE between prediction and ground truth in a supervised setting) or maximizing a quantity (ex. likelihood of the data). This means that the problem is cast as an optimization problem. One technique to solve those optimization problems, which scale rather weel is **Gradient Descent** (or **Gradient Ascent**).

In a nutshell, Gradient Descent consists in picking a random starting point, computing the gradient (derivatives) of a given function w.r.t its parameters and take a small step in the opposite direction of the gradient (minimization setup, alternatively in the direction, for gradient ascent

and a maximization setup) and repeat the process with this updated point until some convergence criteria.




```
begin
  f(x, y) = x^2 + y^2
  x = -10:10
  y = x
  plot(x, y, f, linetype=:surface)

  xₛ = 0:0.5:10
  yₛ = 0:-.45:-9
  zₛ = f.(xₛ, yₛ)
  scatter!(xₛ, yₛ, zₛ, marker=:v, markersize=4, color=:steelblue, legend=false)
end
```

back to TOC

Estimating the gradient

If f is a function of one variable, its derivative at point x measures how f(x) changes as we make a very small change of x. The derivative is the limit of the difference of the quotients

```
Array{T,1} where T<:AbstractFloat
```

```
begin
  const AF =  AbstractFloat
  const VT = Vector{T} where T <: AbstractFloat
end
```

```
diff_quotient (generic function with 2 methods)
```

```
begin
  function diff_quotient(f::Function, x::AF, Δ::AF)::AF
    (f(x + Δ) - f(x)) / Δ
  end

  diff_quotient(f::Function, v::Vector{AF}, Δ::AF) = diff_quotient.(f, v, Δ)
```

The derivative is the slope of the tangent line at (x, f (x)) , while the difference quotient is the slope of the almost-tangent line that runs through (x + Δ, f (x + Δ)) . As Δ gets tends to o,, the almost-tangent line gets closer and closer to the tangent line.

For many function it is easy to compute the exact derivatives. For example for the square function we have:

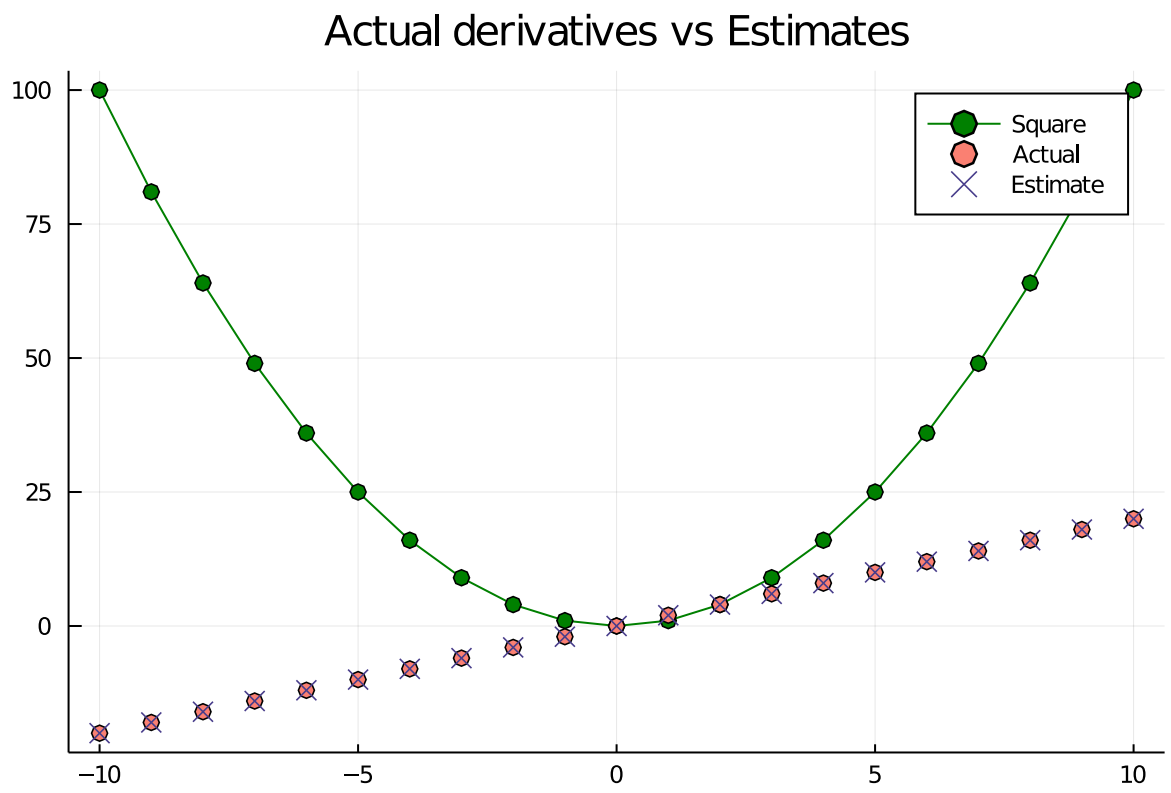
```
deriv_square (generic function with 2 methods)
```

```
begin
  square(x::AF) = x * x
  square(v::Vector{AF}) = square.(v)

  deriv_square(x::AF) = 2. * x
  deriv_square(v::VT) = deriv_square.(v)
```

What if we cannot? Weel we can estimate the derivative by evaluating the difference quotient for very small Δ. For example:

```
Float64[-19.999, -17.999, -15.999, -13.999, -11.999, -9.999, -7.999, -5.999, -3
```

```
begin
  x_r = -10.:10.
  Δ = 1e-3

  y_r = square.(collect(x_r));
  dy_r = deriv_square.(x_r);
  dŷ_r = diff_quotient.(square, x_r, Δ);
```




```
begin
  plot(xᵣ, yᵣ, label="Square", marker=:o, color=:green)
  scatter!(xᵣ, dyᵣ, label="Actual", marker=:o, color=:salmon)
  scatter!(xᵣ, dŷᵣ, label="Estimate", marker=:x, color=:darkslateblue,
    markersize=5, title="Actual derivatives vs Estimates")
end
```

When a function has sevral variables, it has multiple *partila derivatives* each indicating how f change when a small change is made in one of the input variable (while keeping the rest constant).

```
∂_diff_quotient (generic function with 2 methods)
```

```
begin
  function ∂_diff_quotient(f::Function, v::VT, i::Integer, Δ::AF)::AF
    """
    Returns the i-th partial difference quotient of f at v
    """
    w = copy(v)
    w[i] += Δ
    (f.(w...) - f.(v...)) / Δ
  end

  ∂_diff_quotient(f::Function, v::Vector{AF}, Δ::AF)::AF =
    ∂_diff_quotient.(f, Tuple(v), 1:length(v), Δ)
end
```

With this definition we can then estimate the gradient.

```
estimate_∇ (generic function with 1 method)
```

```
function estimate_∇(f::Function, v::VT, Δ::AF)::VT
  [∂_diff_quotient(f, v, i, Δ) for i in 1:length(v)]
end
```

```
∂fnᵥ (generic function with 1 method)
```

```
begin
  fn(u::AF, v::AF) = u^2 + v

end
```

```
∂fn_u(u::AF, _v::AF) = 2. * u  # ∂fn w.r.t u
∂fn_v(_u::AF, v::AF) = 1       # ∂fn w.r.t v
```

```
begin
  Base.:≈(x::AF, y::AF; ϵ=1e-4) = abs(x - y) ≤ ϵ

  Base.:≈(u::Vector{T}, v::Vector{T}; ϵ=1e-4) where T <: AF = abs.(u - v) .≤ ϵ
end
```

```
Test Passed
```

```
begin
  Δ₁ = 1e-6
  vm = Float64[1., 2.]
  ix = 1
  @test ∂fn_u(vm...) ≈ ∂_diff_quotient(fn, vm, ix, Δ₁)
end
```

back to TOC

Using the Gradient

It is easy to see that the function `f` defines above is smallest when its input is a vector of zeros. But imagine let's say we do not know. In this case we can use the gradients to find the minimum (this function is bivariate convexe) among all three-dimensional vectors.

We will start by picking a random starting point and then take tiny steps in the opposite direction of the gradient until we reach a point where the gradient is very small (which is guarantee to happen)

```
∇_step (generic function with 1 method)
```

```
function ∇_step(v::VT, ∇::VT, η::AF)::VT
  @assert length(v) == length(∇)
  v + η * ∇
end
```

```
∇_sum_of_square (generic function with 1 method)
```

```
gradient_descent (generic function with 1 method)
```

```
function gradient_descent(fn;range=1000, rng=MersenneTwister(42), η=1e-2)
  v = rand(rng, 3)
  for epoch ∈ 1:range
    ∇ = fn(v)
    v = ∇_step(v, ∇, -η) ## take a step in negative direction of the gradient
    epoch % 100 == 0 && println("$(epoch) => v: $(v)")
  end
  v
end
```

```
100 => v: [0.07071049479836194, 0.060213142324845564, 0.00234561910412603]
200 => v: [0.009377594417257011, 0.00798544019414858, 0.0003110749638874428]
300 => v: [0.0012436524069777426, 0.0010590255321720962, 4.125462356072904e-5]
400 => v: [0.0001649326299008291, 0.00014044749575786822, 5.471169855229108e-6]
500 => v: [2.1873292130001645e-5, 1.862608451393872e-5, 7.255841164252459e-7]
600 => v: [2.90082628823702e-6, 2.470183056296691e-6, 9.622664328460298e-8]
700 => v: [3.847062940738188e-7, 3.2759457990481126e-7, 1.2761534697646886e-8]
800 => v: [5.1019577869986194e-8, 4.344544770050434e-8, 1.6924290641374127e-9]
900 => v: [6.766193759055349e-9, 5.761715979689615e-9, 2.244491908692764e-10]
1000 => v: [8.973296114237714e-10, 7.6411621441814e-10, 2.976635201402013 5e-11]
```

```
with_terminal() do
  gradient_descent(∇_sum_of_square)
```

syntax: extra token "0.93698" after end of expression

1. **top-level scope** @ *none:1*

```
begin
  @test all(Base.:≈(gradient_descent(∇_sum_of_square), zeros(Float64, 3); ϵ=1.e-7))
  @test all(gradient_descent(∇_sum_of_square) ≈ zeros(Float64, 3))
end9.93698
```

back to TOC

Choosing the right step

It is clear why we choose to move in the opposite direction of the gradient while minimizing an (objective) function. Determining the right step however is something like an art and its value depends on the context.

- Too large a step size and we are likely to overshoot and not converge to the optimum.
- Too small a step and it will take ages to converge...

Not easy. There are mutliple options like fixed step size, gradual shrinking of the step size over time and other fancier options... To select the right step (or rnage of steps) one can try different options on a validation set...

back to TOC

Using Gradient Descent (GD) to fit models

In this collection of notebooks, we will be using gradient descent to fit parameterized models to data.

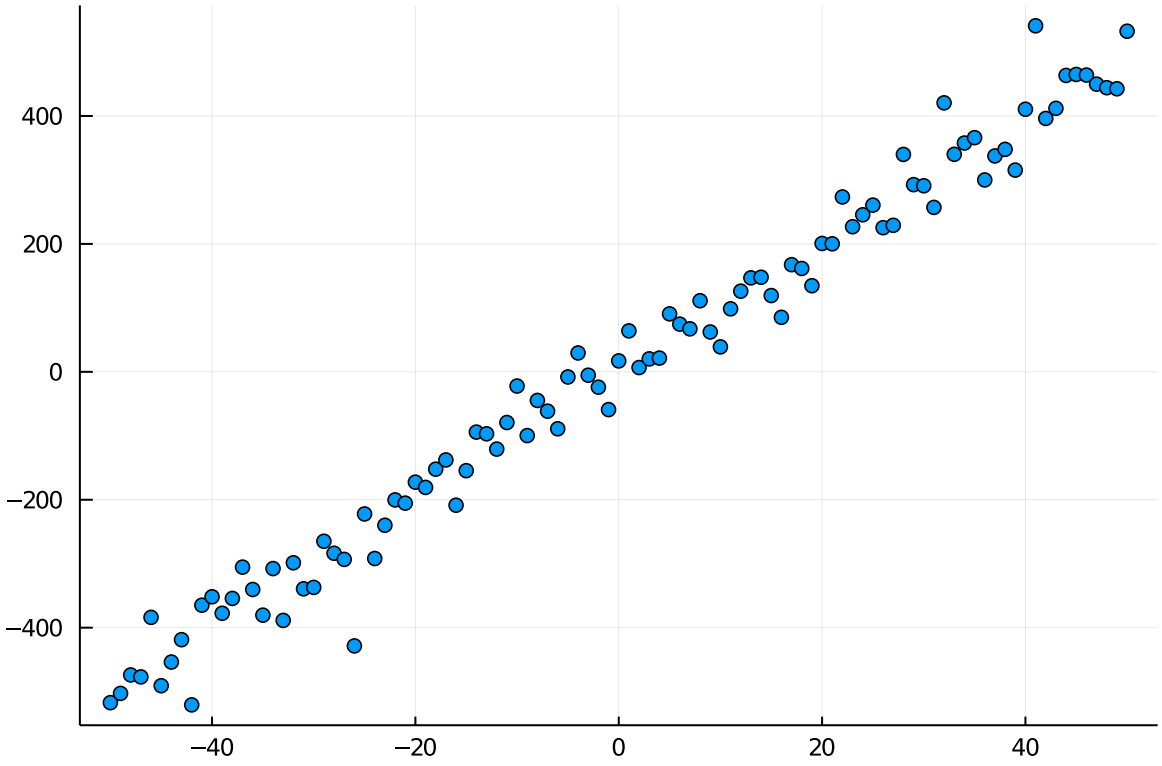
The usual case involves some dataset which we try to fit with a model (our hypothesis) that is characterized by some differentiable parameters. Also to measure the quality of our progress we will have a loss function.

Taking the data as being fixed, the loss function tells us how good (or bad) our model parameters are. We can use gradient descent to optimize this loss.

Let us try on a toy example.

```
Array{Float64,1}[Float64[-50.0, -517.241], Float64[-49.0, -502.775], Float64[-48.0
```

```
begin
  rng = MersenneTwister(42)
  data = [[x, 10x + 5. + 40. * randn(rng, 1)[1]] for x ∈ -50.:50.]
  # linear pattern with some (Normally distributed) noise
```




```
Array{Array{Float64,1},1}
```

```
linear_∇ (generic function with 1 method)
```

```
function linear_∇(x::AF, y::AF, θ::VT)::VT
  slope, intercept = θ
  ŷ = slope * x + intercept   # prediction model
  Δerr = (ŷ - y)
  # squared_Δerr = Δerr^2
  [2. * Δerr * x, 2. * Δerr]
```

We are going to:

1. Start with random initialisation of the parameters (θ)
2. Compute the mean of the gradient
3. Adjust θ
4. Repeat

```
∇_descent (generic function with 1 method)
```

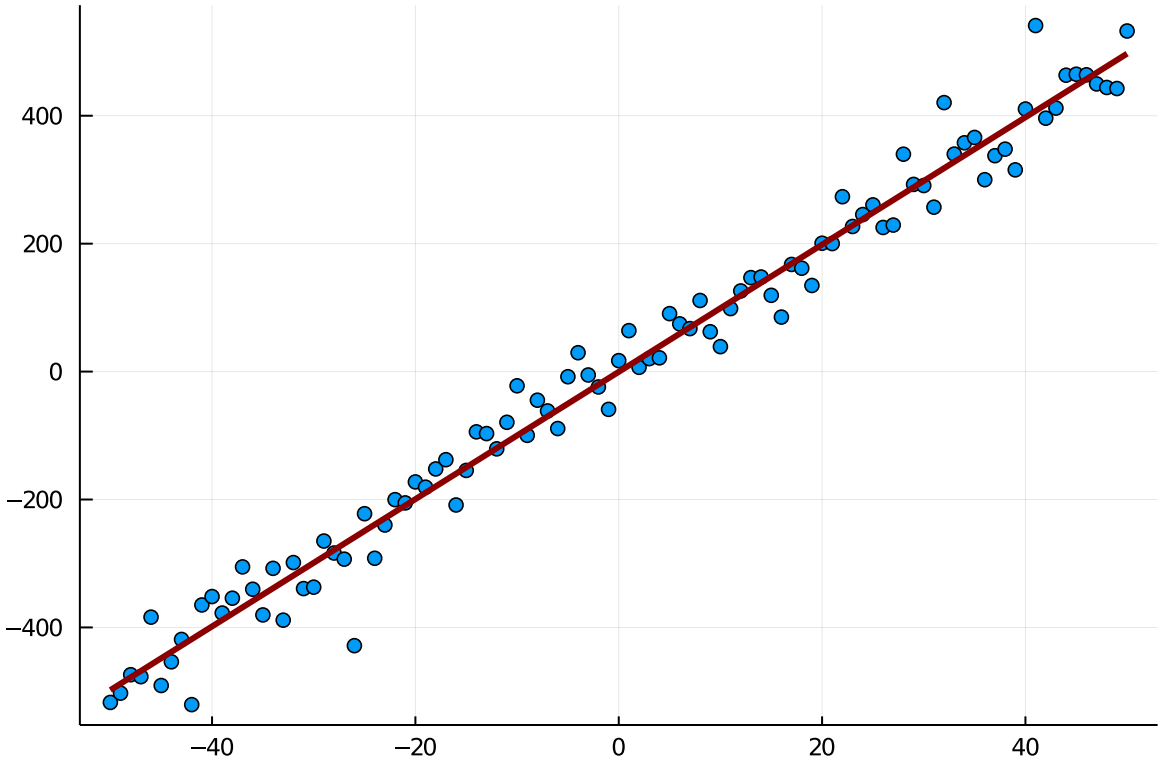
```
function ∇_descent(inputs::Vector{Vector{T}};
    η=1e-5, nepochs=5_000) where T <: AF
```

```
  u = Uniform(-1., 1.)
  θ = [rand(u) for _ ∈ 1:2]

  for epoch ∈ 1:nepochs
    ∇ = μ([linear_∇(x, y, θ) for (x, y) ∈ inputs])
    θ = ∇_step(θ, ∇, -η)
    epoch % 100 == 0 && println("$(epoch) => θ: $(θ)")
  end

  (slope, intercept) = θ
```

```
Test Passed
```

```
begin
  slope, intercept = ∇_descent(data; nepochs=1_000)
  @test 9.90 ≤ slope ≤ 10.1
  @test abs(intercept) ≤ 1.0
```

Let's plot our result:




```
begin
  x_ = [-50.0, 50.0]
  y_ = slope .* x_ .+ intercept
  #
  scatter(map(x -> x[1], data), map(y -> y[2], data), legend=false)
  plot!(x_, y_, lw=3, color=:darkred)
```

back to TOC

Mini-batch and Stochastic Gradient Descent

In the preceding approach we evaluated the gradient on the whole dataset. This was fine because our dataset was small. In practice datasets can be big/large and this technique becomes computationally expensive. This is why other techniques were devised, namely stochastic

gradient descent and mini-batch gradient descent (a good trade-off between full gradient descent and stochastic gradient descent).

In mini-batch gradient descent, we compute the gradient and take a gradient step based on a mini-batch sampled from the dataset:

```
begin
    """
    Generates batch-size sized min-batches from given dataset
    """

    struct MiniBatch{T <: Any}
        ds::Vector{Vector{T}}
        bsize::Integer  # batch-size

        function MiniBatch{T}(ds::Vector{Vector{T}}, bs::Integer) where T <: Any
            @assert 0 < bs ≤ length(ds)
            new(ds, bs)
        end
    end

    Base.collect(iter::MiniBatch) = iter.ds

    function Base.iterate(iter::MiniBatch,
            state=(view(iter.ds, 1:iter.bsize), iter.bsize + 1))
        batch, ix = state
        ##
        isnothing(batch) && return nothing
        ix ≥ length(iter.ds) && return (batch, (nothing, ix))
        ## otherwise
        jx = ix
        ix += iter.bsize
        ix = ix ≥ length(iter.ds) ? length(iter.ds) : ix
        (batch, (view(iter.ds, jx:ix - 1), ix))
    end

    Base.length(iter::MiniBatch) = length(iter.ds)
    Base.eltype(iter::MiniBatch) = eltype(iter.ds)
end
```

```
Test Passed
```

```
let _ix = 0, bsz = 15
    for batch ∈ MiniBatch{Float64}(data, bsz)
        @assert length(batch) == bsz || 0 ≤ length(batch) ≤ bsz
        _ix += 1
    end
    @test _ix == 7
end
```

```
Test Passed
```

```
let _ix = 0, bsz = 12
    for batch ∈ MiniBatch{Float64}(data, bsz)
        @assert length(batch) == bsz || 0 ≤ length(batch) ≤ bsz
        _ix += 1
    end
    @test _ix == 9
end
```

let's rewrite our gradient(∇) descent with min-batch:

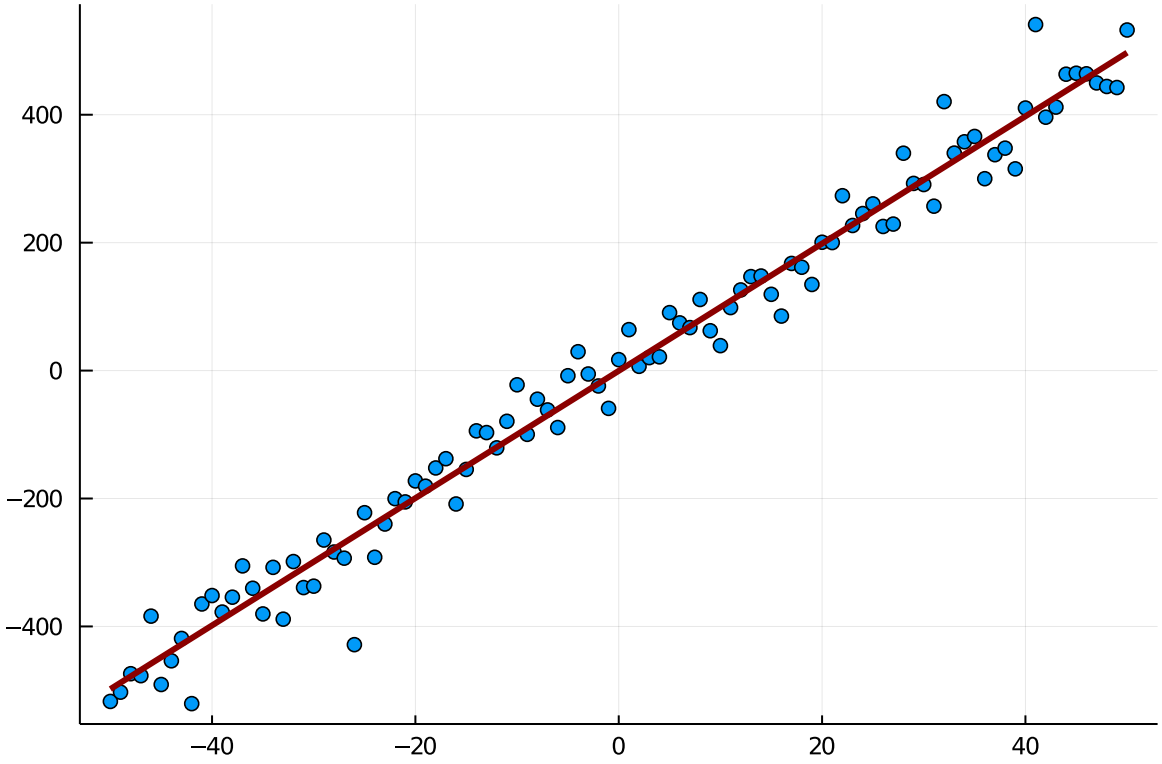
```
minibatch_∇_descent (generic function with 1 method)
```

```
function minibatch_∇_descent(inputs::Vector{Vector{T}};
```

```
    η=1e-5, nepochs=5_000, bsize=10) where T <: AF
  u = Uniform(-1., 1.)
  θ = [rand(u) for _ ∈ 1:2]

  for epoch ∈ 1:nepochs
        for batch ∈ MiniBatch{eltype(inputs[1][1])}(inputs, bsize)
            ∇ = μ([linear_∇(x, y, θ) for (x, y) ∈ inputs])
            θ = ∇_step(θ, ∇, -η)
        end
    epoch % 100 == 0 && println("$(epoch) => θ: $(θ)")
  end

  (slope, intercept) = θ
```

```
Test Passed
```

```
begin
  slope₂, intercept₂ = minibatch_∇_descent(data; nepochs=100) ## only 100 epochs
  @test 9.90 ≤ slope₂ ≤ 10.1
  @test abs(intercept₂) ≤ 1.0
```




```
begin
  # x_₂ = [-50.0, 50.0]
  y_₂ = slope .* x_ .+ intercept
  #
  scatter(map(x -> x[1], data), map(y -> y[2], data), legend=false)
  plot!(x_, y_₂, lw=3, color=:darkred)
```

In this instance our mini-batch gradient descent worked pretty well, finding the optimal parameters in just 100 iteration (and actually less)...

Basing gradient step on a small min-batch or just on one sampel (in the case of stochastic gradient descent) allows to take more steps, but the gradient can fluctuate a lot...