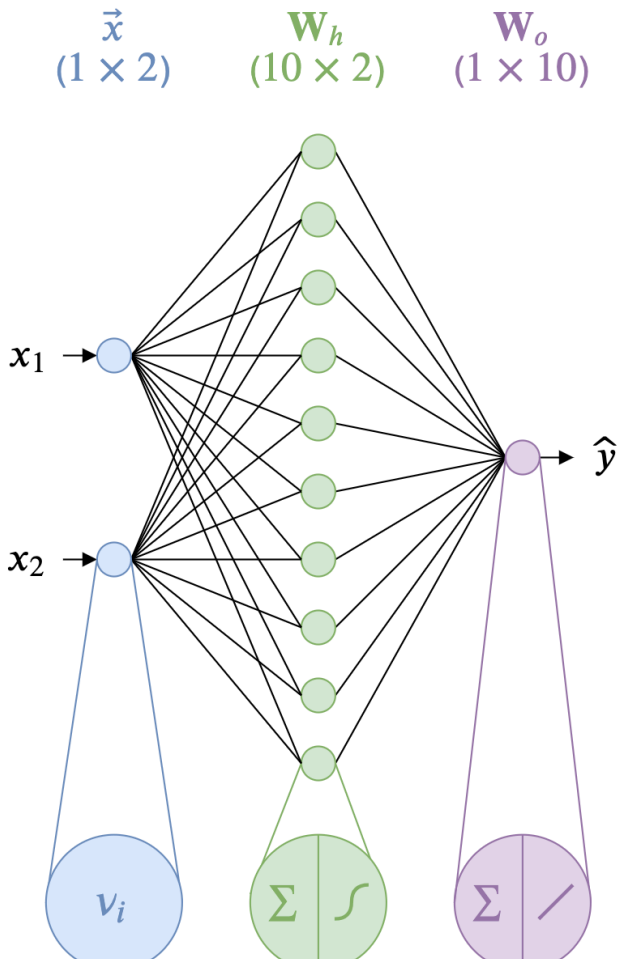
Algorytmy w Inżynierii Danych (2024)

Automatyczne różniczkowanie

Bartosz Chaber

Algorytm propagacji wstecznej błędu

Jak nauczyć prostą sieć neuronową z jedną warstwą ukrytą? Sieć ma za zadanie znaleźć aproksymację funkcji $y = f(\boldsymbol{x})$.



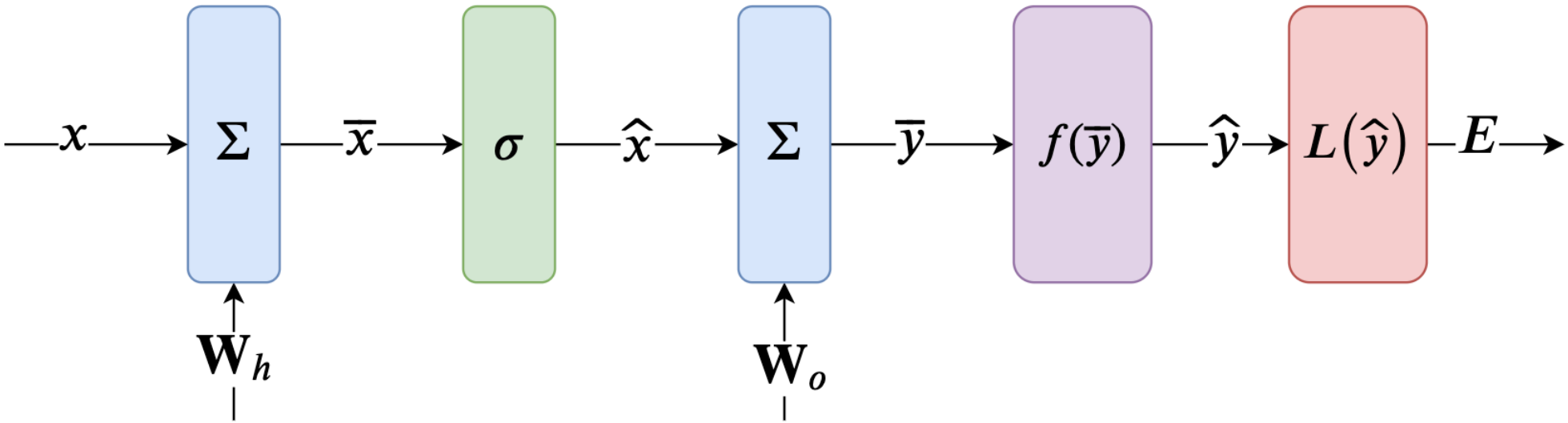
```
dense(w, n, m, v, f)     = f.(reshape(w, n, m) * v)
mean_squared_loss(y, ŷ) = sum(0.5(y - ŷ).^2)
sigmoid(x) = one(x) / (one(x) + exp(-x))
linear(x) = x
Wh = randn(10,2)
Wo = randn(1,10)
x, y = [1.98; 4.434], [0.064]
function net(x, wh, wo, y)
    x̂ = dense(wh, 10, 2, x, sigmoid)
    ŷ = dense(wo, 1, 10, x̂, linear)
    E = mean_squared_loss(y, ŷ)
end # Grant Sanderson, "Ale czym są sieci neuronowe?"
# https://www.youtube.com/watch?v=aircAruvnKk
```

Algorytm propagacji wstecznej błędu

Wejściem sieci są argumenty funkcji aproksymowanej, natomiast wyjściem jest wynik aproksymacji $\hat{y} = \hat{f}(\boldsymbol{x})$.

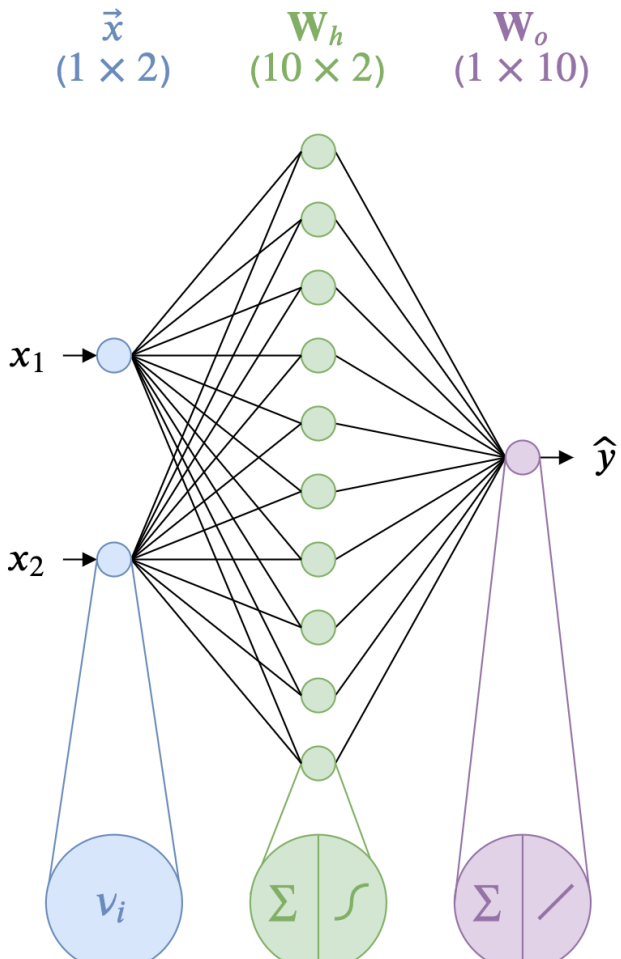
Podczas uczenia z nauczycielem mamy dostęp jeszcze do referencyjnej wartości y, dlatego możemy policzyć funkcję straty $E = \sum_i \left(y_i - \hat{y}_i\right)^2$. Wartość funkcji straty E zależy pośrednio od wag sieci (bo na ich podstawie wyliczane jest $\hat{y}$).

Jeżeli policzymy macierz Jacobiego funkcji $E(\boldsymbol{W}_h, \boldsymbol{W}_o)$ znajdziemy pochodne względem $\boldsymbol{W}_h$ oraz $\boldsymbol{W}_o$. Pochodne te oznaczają *kierunek* zmian wartości, które **zwiększają** błąd.


x
Σ
$\overline{x}$
σ
$\hat{x}$
Σ
$\overline{y}$
$f(\overline{y})$
$\hat{y}$
$L(\hat{y})$
E
$\mathbf{W}_h$
$\mathbf{W}_o$

Algorytm propagacji wstecznej błędu

Przykładowo, załóżmy, że mamy **magiczną** funkcję `dnet`, która zwraca pochodne funkcji straty względem wag sieci:



```
julia> typeof(Wh), typeof(Wh[:])
(Array{Float64,2}, Array{Float64,1})
julia> x = [1.98;4.434]
julia> y = [0.064]
julia> E = net(x, Wh[:], Wo[:], y) # => 5.680084
julia> dWh, dWo = dnet(x, Wh[:], Wo[:], y)
julia> dWh
10×2 Array{Float64,2}:
  2.51645    5.63532
  0.00114    0.00257
  ⋮
 -0.14419   -0.32290
  0.04442    0.09949
```

Aby zmniejszyć błąd pójdziemy w kierunku **przeciwnym** do gradientu:

```
julia> Wh -= 0.1dWh
julia> Wo -= 0.1dWo
julia> E = net(x, Wh[:], Wo[:], y) # => 0.915599
```

Ręczne różniczkowanie

Znając strukturę sieci i mając wystarczająco dużo czasu możemy ręcznie napisać funkcję zwracającą gradienty funkcji straty względem wag:

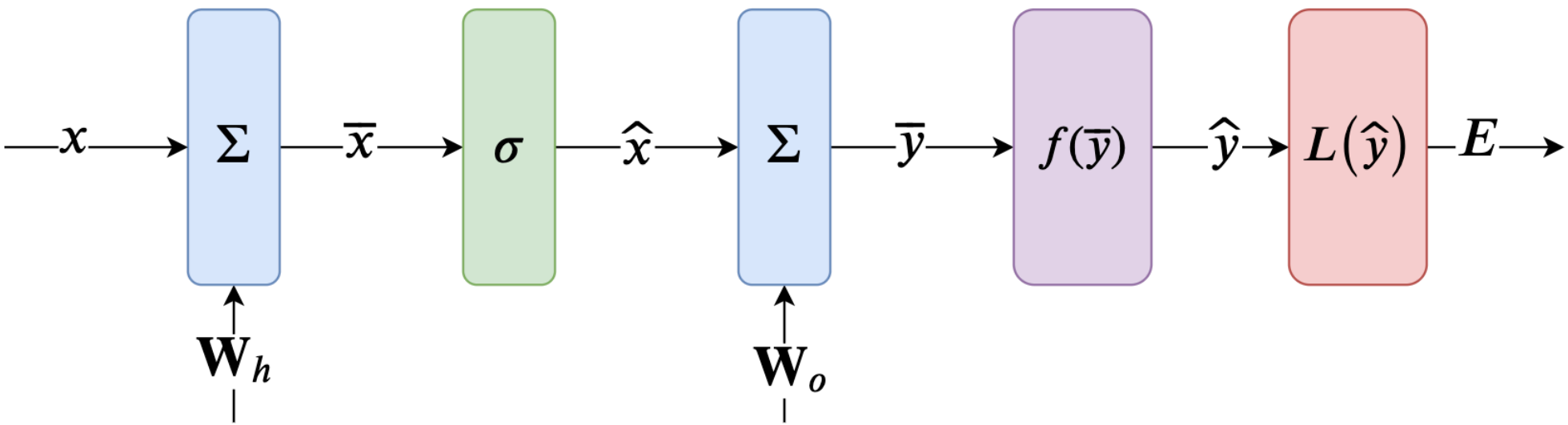
```
import LinearAlgebra: diagm
eye(n) = diagm(ones(n))
diagonal(v) = diagm(v)

julia> eye(3)
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

```
julia> diagonal([1., 2., 3.,
4.])
4×4 Array{Float64,2}:
 1.0  0.0  0.0  0.0
 0.0  2.0  0.0  0.0
 0.0  0.0  3.0  0.0
 0.0  0.0  0.0  4.0
```

```
function ∇W(x, x̂, ŷ, y, Wo)
  # mean_squared_loss
  Eŷ  = ŷ - y
  # liniowa funkcja aktywacji
  ŷȳ  = ȳ  |> length |> eye
  # sumowanie (W*x)
  ȳWo = x̂  |> transpose
  ȳx̂  = Wo |> transpose
  # sigmoidalna f. aktywacji
  x̂x̄  = x̂  .* (1.0 .- x̂) |> diagonal
  # sumowanie (W*x) wzg. wag
  x̄Wh = x  |> transpose
  # reguła łańcuchowa
  Eȳ  = ŷȳ * Eŷ
  Ex̂  = ȳx̂ * Eȳ
  Ex̄  = x̂x̄ * Ex̂
  EWo = Eȳ * ȳWo
  EWh = Ex̄ * x̄Wh
  return EWo, EWh
end
```

Pochodne na proste funkcje aktywacji są dobrze opisane:

arunmallya.github.io/writeups/nn/backprop.html



Dzięki temu możemy podczas ewaluacji sieci obliczać też pochodne względem wag.

```
dWo = similar(Wo)
dWh = similar(Wh)
function net(x, wh, wo, y)
    x̂ = dense(wh, 10, 2, x, sigmoid)
    ŷ = dense(wo, 1, 10, x̂, linear)
    EWo, EWh = ∇W(x, x̂, ŷ, y, Wo)
    dWo .= EWo
    dWh .= EWh
    E = mean_squared_loss(y, ŷ)
end
```

Teraz wywołując wielokrotnie w pętli ewaluację sieci oraz uaktualnienie wag nasza sieć powinna się uczyć (dla jednego przypadku `x` i `y` jest to trywialne, więc przypadki uczące trzeba losować z pewnego zbioru):

```
epochs = 10
for i=1:epochs
  E   = net(x, Wh[:], Wo[:], y)
  Wh -= 0.1dWh
  Wo -= 0.1dWo
end
```

Wykorzystując informację o pochodnych funkcji straty względem wag poszczególnych połączeń między neuronami możemy zmodifykować te wagi tak, żeby **minimalizować** tę stratę

Automatyczne różniczkowanie

Przy dużej liczbie warstw ręczne wyprowadzanie zależności na różne funkcje aktywacji jest trudne do zrobienia. Potrzebny jest algorytm do automatycznego różniczkowania.

```
struct Dual <:Number
     v::Number
    dv::Number
end
```

```
struct Dual{T <:Number} <:Number
     v::T
    dv::T
end
```

Lepiej jest mieć typ parametryzowany.

Dodamy do tego przeciążone podstawowe operacje na liczbach dualnych:

```
import Base: +, -, *, /
 -(x::Dual)           = Dual(-x.v,       -x.dv)
 +(x::Dual, y::Dual) = Dual( x.v + y.v, x.dv + y.dv)
 -(x::Dual, y::Dual) = Dual( x.v - y.v, x.dv - y.dv)
 *(x::Dual, y::Dual) = Dual( x.v * y.v, x.dv*y.v + x.v*y.dv)
 /(x::Dual, y::Dual) = Dual( x.v / y.v,(x.dv*y.v - x.v*y.dv)/
y.v^2)
```

```
import Base: abs, sin, cos, tan, exp, sqrt, isless
abs(x::Dual)  = Dual(abs(x.v),sign(x.v)*x.dv)
sin(x::Dual)  = Dual(sin(x.v), cos(x.v)*x.dv)
cos(x::Dual)  = Dual(cos(x.v),-sin(x.v)*x.dv)
tan(x::Dual)  = Dual(tan(x.v), one(x.v)*x.dv +
tan(x.v)^2*x.dv)
exp(x::Dual)  = Dual(exp(x.v), exp(x.v)*x.dv)
sqrt(x::Dual) = Dual(sqrt(x.v),.5/sqrt(x.v) * x.dv)
isless(x::Dual, y::Dual) = x.v < y.v;
```

Automatyczne różniczkowanie

Dodamy do tego ładne wypisywania naszego typu:

```
import Base: show
show(io::IO, x::Dual) =
    print(io, "(", x.v, ") + [", x.dv, "ϵ]");
value(x::Dual)    = x.v;
partials(x::Dual) = x.dv;
```

I zdefiniujemy reguły konwersji i promocji typów:

```
import Base: convert, promote_rule
convert(::Type{Dual{T}}, x::Dual) where T =
  Dual(convert(T, x.v), convert(T, x.dv))

convert(::Type{Dual{T}}, x::Number) where T =
  Dual(convert(T, x), zero(T))

promote_rule(::Type{Dual{T}}, ::Type{R}) where {T,R} =
  Dual{promote_type(T,R)}
```

Obliczanie pochodnej funkcji jednej zmiennej

Jak policzyć pochodną funkcji `f(x) = sin(x*x)`?

```
julia> ϵ = Dual(0., 1.)
(0.0) + [1.0ϵ]
julia> x = 5.0 + ϵ
(5.0) + [1.0ϵ]
julia> y = f(x)
(-0.13235175009777303) + [9.912028118634735ϵ]
julia> sin(25.0), 10cos(25.0)
(-0.13235175009777303,    9.912028118634735)
```

Z poprzedniego wykładu pamiętamy, że:

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\mathrm{d}f}{\mathrm{d}g} \cdot \frac{\mathrm{d}g}{\mathrm{d}x} \cdot \left(\frac{\mathrm{d}x}{\mathrm{d}x} \right)$$

Prześledźmy jak propaguje się pochodna po x ($\frac{\mathrm{d}x}{\mathrm{d}x} = 1\epsilon$):

```
# *(x::Dual, y::Dual) =
Dual(x.v * y.v,
     x.dv * y.v + x.v * y.dv)
# sin(x::Dual)  =
 Dual(sin(x.v), cos(x.v)*x.dv)
#
```

```
#(5.0)+[1.0ϵ] * (5.0)+[1.0ϵ]
Dual(5.0 * 5.0,
     1.0 * 5.0 + 5.0 * 1.0)
# => (25.0) + [10.0ϵ]
Dual(sin(25), cos(25) * 10)
# => (-0.132...) + [9.912...ϵ]
```

Obliczanie pochodnych funkcji kilku zmiennych

Spróbujmy to samo wykonać dla funkcji dwóch zmiennych. Poniżej definicja funkcji oraz jej pochodnych po obydwu zmiennych:

```
f(x,y)      = x*x*y
dfdx(x, y) = 2x*y
dfdy(x, y) = x*x
```

Dodamy zarodek do zmiennej x ($\frac{\mathrm{d}x}{\mathrm{d}x} = 1\epsilon$):

```
ϵ = Dual(0., 1.);
x = 1.0 + ϵ
y = 3.0
@show f(x,y), dfdx(1.0, 3.0);
# => (3.0) + [6.0ϵ], 6.0
```

Pochodną po y liczymy dodając zarodek do y ($\frac{\mathrm{d}y}{\mathrm{d}y} = 1\epsilon$):

```
x = 1.0
y = 3.0 + ϵ
@show f(x,y), dfdy(1.0,3.0))
# => (3.0) + [1.0ϵ], 1.0
```

Macierz Jacobiego

Żeby z kolei policzyć macierz Jacobiego funkcji możemy napisać następującą funkcję: (na następnym slajdzie)

```
J = function jacobian(f, args::Vector{T}) where {T <:Number}
    jacobian_columns = Matrix{T}[]
    for i=1:length(args)
        x = Dual{T}[]
        for j=1:length(args)
            push!(x, (i == j) ? Dual(args[j], one(args[j])) :
                                Dual(args[j],zero(args[j])) )
        end
        column = partials.([f(x)...])
        push!(jacobian_columns, column[:,:])
    end
    hcat(jacobian_columns...)
end
```

Widać, że `column = partials.([f(x)...])` wywołuje się tyle razy, ile jest elementów w wektorze x.

Poniżej, dwa triki (na zamianę liczby/wektora w wektor i zamianę wektora w macierz):

```
n = 5.2                    #
v = [3.1,3.3]              #
@show [n...];              [n...] = [5.2]
@show [v...];              [v...] = [3.1, 3.3]
@show typeof(v);           typeof(v) = Array{Float64,1}
@show typeof(v[:,:]);      typeof(v[:, :]) = Array{Float64,2}
```

Uczenie sieci neuronowej...

...z automatycznym różniczkowaniem

```
dense(w, n, m, v, f)     = f.(reshape(w, n, m) * v)
mean_squared_loss(y, ŷ) = sum(0.5(y - ŷ).^2)
linear(x) = x
sigmoid(x) = one(x) / (one(x) + exp(-x))

Wh  = randn(10,2)
Wo  = randn(1,10)
dWh = similar(Wh)
dWo = similar(Wo)
x, y = [1.98;4.434], [0.064]
```

```
function net(x, wh, wo, y)
    x̂ = dense(wh, 10, 2, x, sigmoid)
    ŷ = dense(wo, 1, 10, x̂, linear)
    E = mean_squared_loss(y, ŷ)
end
dnet_Wh(x, wh, wo, y) = J(w -> net(x, w, wo, y), wh);
dnet_Wo(x, wh, wo, y) = J(w -> net(x, wh, w, y), wo);
```

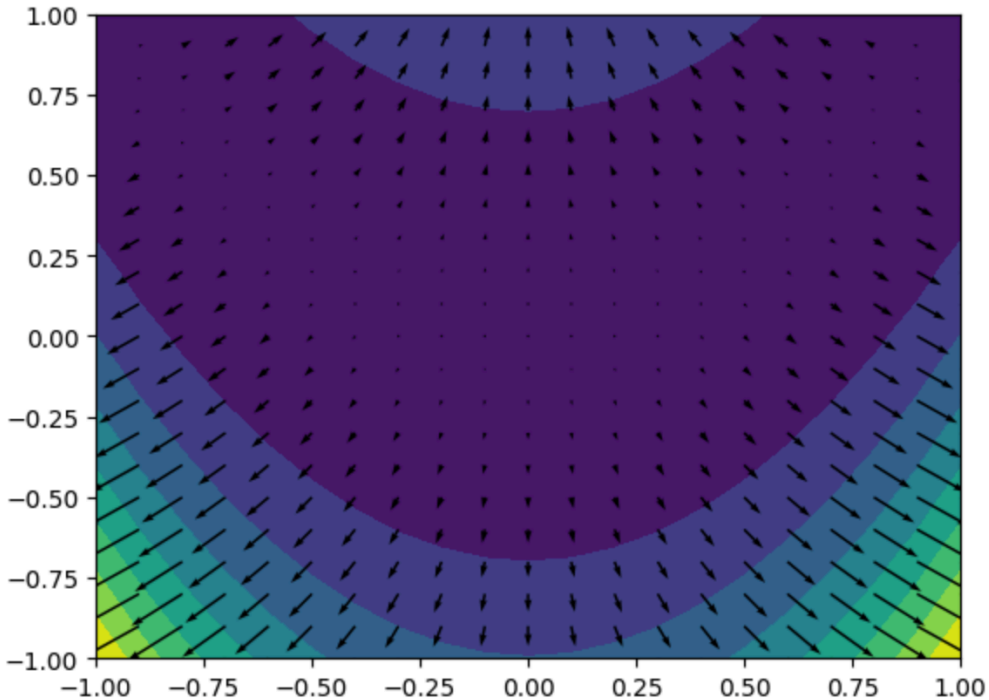
```
epochs=10
for i=1:epochs
  E       =  net(x, Wh[:], Wo[:], y)
  dWh[:] = dnet_Wh(x, Wh[:], Wo[:], y)
  dWo[:] = dnet_Wo(x, Wh[:], Wo[:], y)
  Wh     -= 0.1dWh
  Wo     -= 0.1dWo
end
```

Policzymy gradient funkcji Rosenbrocka (klasyczna funkcja do testowania algorytmów optymalizacyjnych), $f : \mathbb{R}^2 \to \mathbb{R}$

```
rosenbrock(x, y) = (1.0 - x*x) + 100.0*(y - x*x)*(y - x*x)
v  = -1:.1:+1
n  = length(v)
xv = repeat(v, 1, n)
yv = repeat(v',n, 1)
dz = rosenbrock.(xv .+ ϵ, yv)
dzdx = partials.(dz)
dz = rosenbrock.(xv, yv .+ ϵ)
dzdy = partials.(dz)
z = value.(dz)
```

```
using PyPlot
contourf(xv, yv, z)
quiver(xv, yv, dzdx, dzdy)
```

Jak efektywnie policzyć macierz Jacobiego dla wielu zmiennych?



Zarodek musi być dodany do kolejnych argumentów różniczkowanej funkcji.

Wymaga to **tylu przejść, ile argumentów** przyjmuje funkcja.

Jest to odpowiednie dla funkcji
$f : \mathbb{R}^n \to \mathbb{R}^m, n < m$

Można to ominąć stosując wielowymiarowe liczby dualne: $\epsilon \rightarrow \epsilon_1 + \epsilon_2 + \epsilon_3 ...$

Literatura

- Mike Innes, 2019, “Differentiation for Hackers: Implementing Forward Mode”, url: https://github.com/MikeInnes/diff-zoo/blob/notebooks/forward.ipynb, dostęp: 25.03.2020
- Jarrett Revels, Miles Lubin, Theodore Papamarkou, 2016, “Forward-Mode Automatic Differentiation in Julia”, url: https://arxiv.org/abs/1607.07892, dostęp: 23.03.2020
- Jarrett Revels, 2017, “How ForwardDiff Works”, url: https://github.com/JuliaDiff/ForwardDiff.jl/blob/master/docs/src/dev/how_it_works.md, dostęp: 24.03.2020

- Jeffrey Mark Siskind, Barak A. Pearlmutter, 2005, “Perturbation Confusion and Referential Transparency: Correct Functional Implementation of Forward-Mode AD”, url: http://www.bcl.hamilton.ie/~barak/papers/ifl2005.pdf, dostęp: 25.03.2020

Dziękuję za uwagę