Automatic Differentiation

On peut calculer des dérivées partielles de différentes manières :

1. De façon symbolique, en fixant une des variables et en dérivant les autres soit à la main, soit par ordinateur.
2. De façon numérique, avec la formule $f'(x) \approx (f(x+h) - f(x))/h$.
3. De façon algorithmique, soit forward, soit reverse, c'est ce que nous verons ici.

Pour illustrer, nous utiliserons l'exemple de classification de points de deux formes de lunes.

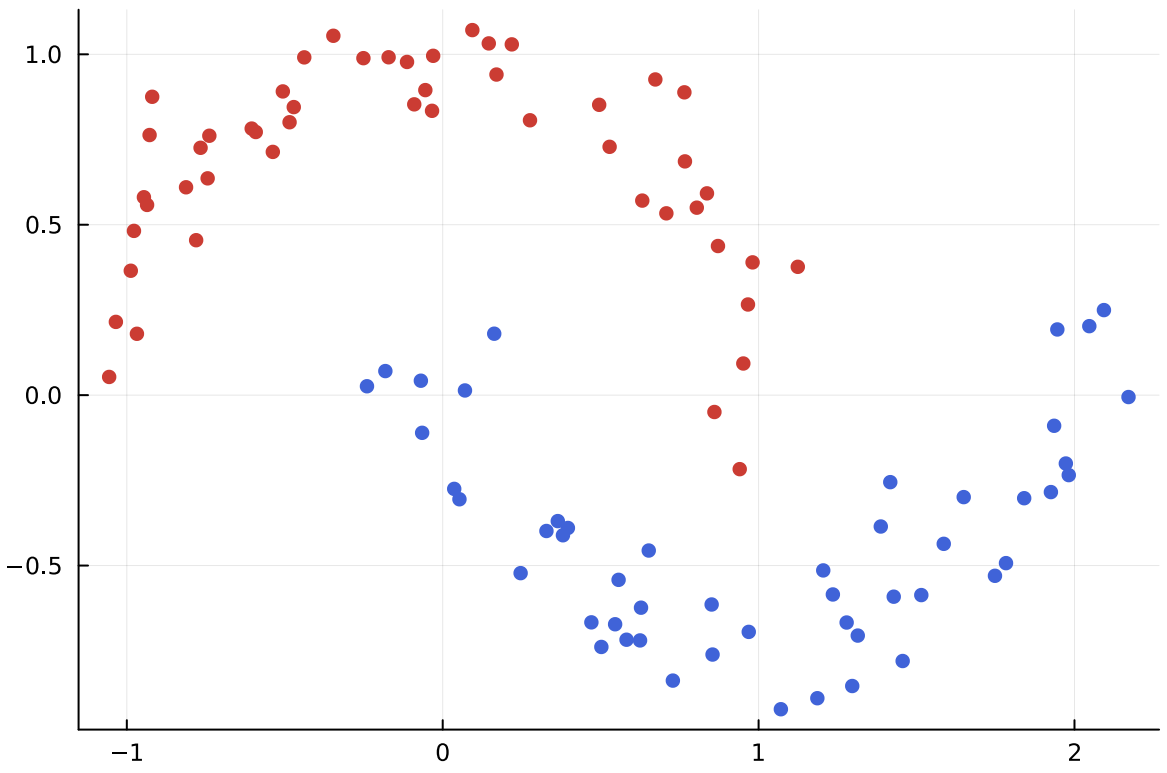
▸ (x1 , x2 , CategoricalArrays.CategoricalVector{Int64, UInt32, Int64, Cate

	x1	x2
1	-0.780545	0.454589
2	0.495337	0.851825
3	-0.0337664	0.834215
4	1.74827	-0.530045
5	0.0938209	1.07106
6	1.84076	-0.302403
7	0.980965	0.389551
8	-0.0650868	-0.110542
9	1.41684	-0.255285
10	1.51507	-0.586624

⋯ more

```
X_table, y_cat = MLJBase.make_moons(100, noise=0.1)
```

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




```
plot_w()
```

Nous travaillerons avec une matrice `X` contenant dans chaque ligne, les coordonnées d'un point.

```
X = 100×2 Matrix{Float64}:
    -0.780545    0.454589
     0.495337    0.851825
    -0.0337664   0.834215
     1.74827    -0.530045
     0.0938209   1.07106
     1.84076    -0.302403
     0.980965    0.389551
     ⋮
    -0.346186    1.0544
     0.708019    0.533406
     0.766845    0.685839
     1.58616    -0.436255
    -0.60468     0.782136
    -0.919653    0.875396
```

```
X = Tables.matrix(X_table)
```

Le vecteur `y` contiendra `1` pour les points de la lune bleue et `-1` pour les points de la lune rouge.

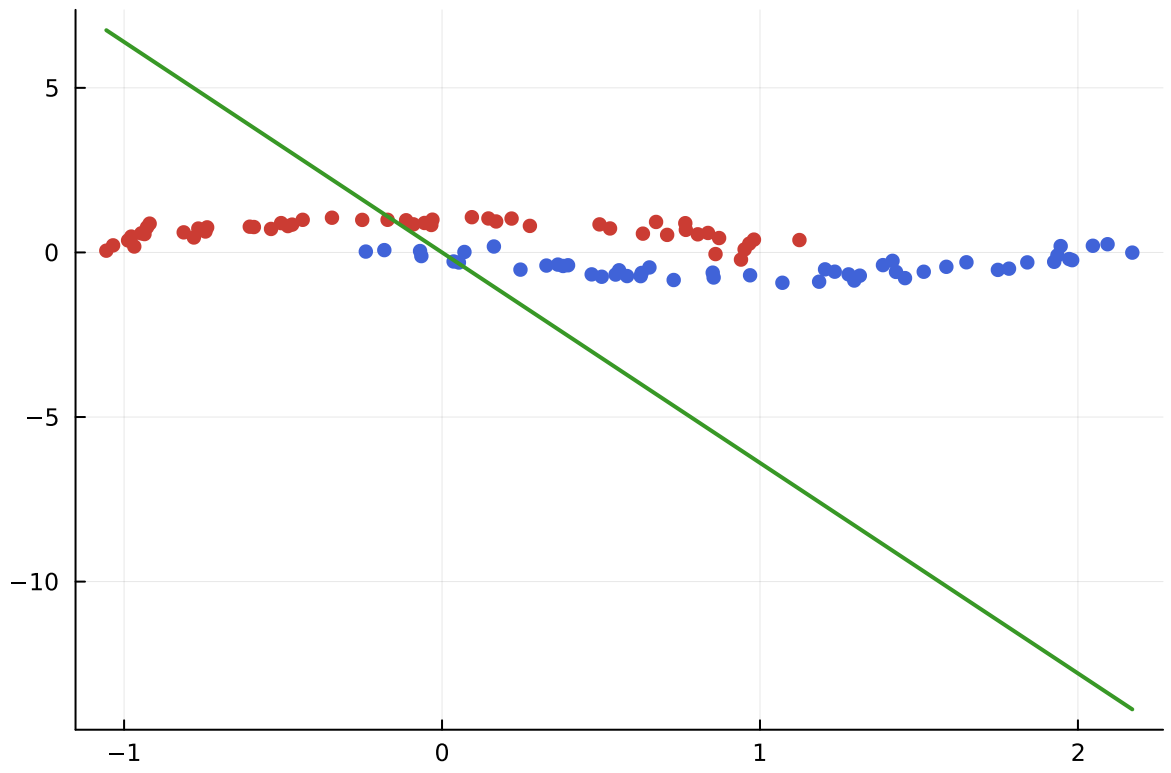
```
y =
▸ [-1.0, -1.0, -1.0, 1.0, -1.0, 1.0, -1.0, 1.0, 1.0, 1.0, -1.0, 1.0, 1.0, 1.0, 1.0, -1.0, 1.0,
```

```
y = 2(float.(y_cat.refs) .- 1.5)
```

Nous illustrons le calcul de dérivée automatique par l'entrainement du modèle linéaire $y \approx Xw$. Commençons avec des poids aléatoires. Le modèle n'est pour le moment pas très précis comme il est aléatoire.

```
w = [0.614529, 0.0961168]
```

```
w = rand(2)
```




```
plot_w(w)
```

En effet, les prédictions ne correspondent pas à `y`.

```
y_est =
[-0.435974, 0.386274, 0.0594316, 1.02342, 0.160603, 1.10214, 0.640274, -0.0506227, 0.8461
```

```
y_est = X * w
```

On peut regrouper les erreurs des estimations de tous les points en les comparant avec `y`.

```
errors =
[0.564026, 1.38627, 1.05943, 0.0234176, 1.1606, 0.102136, 1.64027, -1.05062, -0.15385, -0.
```

```
errors = y_est - y
```

Essayons de trouver des poids `w` qui minimisent la somme des carrés des erreurs (aka MSE) :

```
mean_squared_error = 0.7961996273322796
```

```
mean_squared_error = sum(errors.^2) / length(errors)
```

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

```
mse(w, X, y) = sum((X * w - y).^2 / length(y))
```

```
0.7961996273322797
```

```
mse(w, X, y)
```

Forward Differentiation

Commençons par définir la forward differentiation. Cette différentiation algorithmique se base sur l'observation que la chain rule permet de calculer la dérivée de n'importe quelle fonction dès lors qu'on connait son gradient et la dérivée de chacun de ses paramètres. En d'autres mots, supposons qu'on doive calculer

$$\frac{\partial}{\partial x} f(g(x), h(x))$$

Supposons que la fonction `f` soit une fonction `f(a, b)` simple (telle que `+`, `*`, `-`) dont on connait la formules des dérivée partielles $\partial f / \partial a$ en fonction de `a` et $\partial f / \partial b$ en fonction de `b`: La chain rule nous donne

$$\frac{\partial}{\partial x} f(g(x), h(x)) = \frac{\partial f}{\partial a}(g(x), h(x)) \frac{\partial g}{\partial x} + \frac{\partial f}{\partial b}(g(x), h(x)) \frac{\partial h}{\partial x}$$

Pour calculer cette expression, ils nous faut les valeurs de $g(x)$ et $h(x)$ ainsi que les dérivées $\partial g / \partial x$ et $\partial h / \partial x$.

```
Dual
```

```
begin
    struct Dual{T}
        value::T
        derivative::T
    end
    Dual(x, y) = Dual{typeof(x)}(x, convert(typeof(x), y))
end
```

L'implémentation générique du produit de matrice va appeler `zero`:

```
Base.zero(::Dual{T}) where {T} = Dual(zero(T), zero(T))
```

Par linéarité de la dérivée:

```
begin
    Base.:*(α::T, x::Dual{T}) where {T} = Dual(α * x.value, α * x.derivative)
    Base.:*(x::Dual{T}, α::T) where {T} = Dual(x.value * α, x.derivative * α)
    Base.:+(x::Dual{T}, y::Dual{T}) where {T} = Dual(x.value + y.value,
    x.derivative + y.derivative)
    Base.:-(x::Dual{T}, y::T) where {T} = Dual(x.value - y, x.derivative)
    Base.:/(x::Dual, α::Number) = Dual(x.value / α, x.derivative / α)
end
```

Par la product rule $(fg)' = f'g + fg'$:

```
Base.:*(x::Dual{T}, y::Dual{T}) where {T} = Dual(x.value * y.value, x.value *
y.derivative + x.derivative * y.value)
```

Pour l'exponentiation, on peut juste se rabattre sur le produit qu'on a déjà défini :

```
Base.:^(x::Dual, n::Integer) = Base.power_by_squaring(x, n)
```

```
▸ OneHotVector(::UInt32): [true, false]
```

```
onehot(1, 1:2)
```

```
▸ [1.0, 0.0]
```

```
float.(onehot(1, 1:2))
```

```
▸ [0.0, 1.0]
```

```
float.(onehot(2, 1:2))
```

```
▸ [Dual(0.614529, 1.0), Dual(0.0961168, 0.0)]
```

```
Dual.(w, onehot(1, 1:2))
```

```
▸ Dual(0.7961996273322797, 0.10378244461969546)
```

```
mse(Dual.(w, onehot(1, 1:2)), X, y)
```

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

```
function forward_diff(loss, w, X, y, i)
    loss(Dual.(w, onehot(i, eachindex(w))), X, y).derivative
end
```

```
0.10378244461969546
```

```
forward_diff(mse, w, X, y, 1)
```

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

```
function forward_diff(loss, w, X, y)
    [forward_diff(loss, w, X, y, i) for i in eachindex(w)]
end
```

Gradient descent

Cauchy-Schwarz inequality

$$\left(\sum_i x_i y_i\right)^2 = \left(\sum_i x_i^2\right)\left(\sum_i y_i^2\right)\cos(\theta)^2$$
$$\sum_i x_i y_i = \sqrt{\sum_i x_i^2}\sqrt{\sum_i y_i^2}\cos(\theta)$$
$$\langle x, y\rangle = \|x\|_2\|y\|_2\cos(\theta)$$
$$-\|x\|_2\|y\|_2 \leq \langle x, y\rangle \leq \|x\|_2\|y\|_2$$

Minimum atteint lorsque $x = -y$ et maximum atteint lorsque $x = y$. Dans les deux cas, x est **parallèle** à y, mais ce sont des **sens** différents.

Rappel dérivée directionnelle

Dérivée dans la direction d:

$$\langle d, \nabla f\rangle = d^\top \nabla f = d_1 \cdot \partial f/\partial x_1 + \cdots + d_n \cdot \partial f/\partial x_n$$

Etant donné un gradient ∇f, la direction d telle que $\|d\|_2 = 1$ qui a une dérivée minimale est $d = -\nabla f$.

Gradient:

```
∇ = [0.103782, 0.855627]
```

```
∇ = forward_diff(mse, w, X, y)
```

Gradient line search: pendant combien de temps doit-on suivre le gradient ?

```
step_sizes = -1.0:0.2222222222222222:1.0
```

```
step_sizes = range(-1, stop=1, length=num_n)
```

```
losses =
▸ [0.314122, 0.376175, 0.463986, 0.577554, 0.716878, 0.88196, 1.0728, 1.28939, 1.53175, 1.79
```

```
1 losses = [mse(w + η * ∇, X, y) for η in step_sizes]
```

```
best_idx = 1
```

```
1 best_idx = argmin(losses)
```

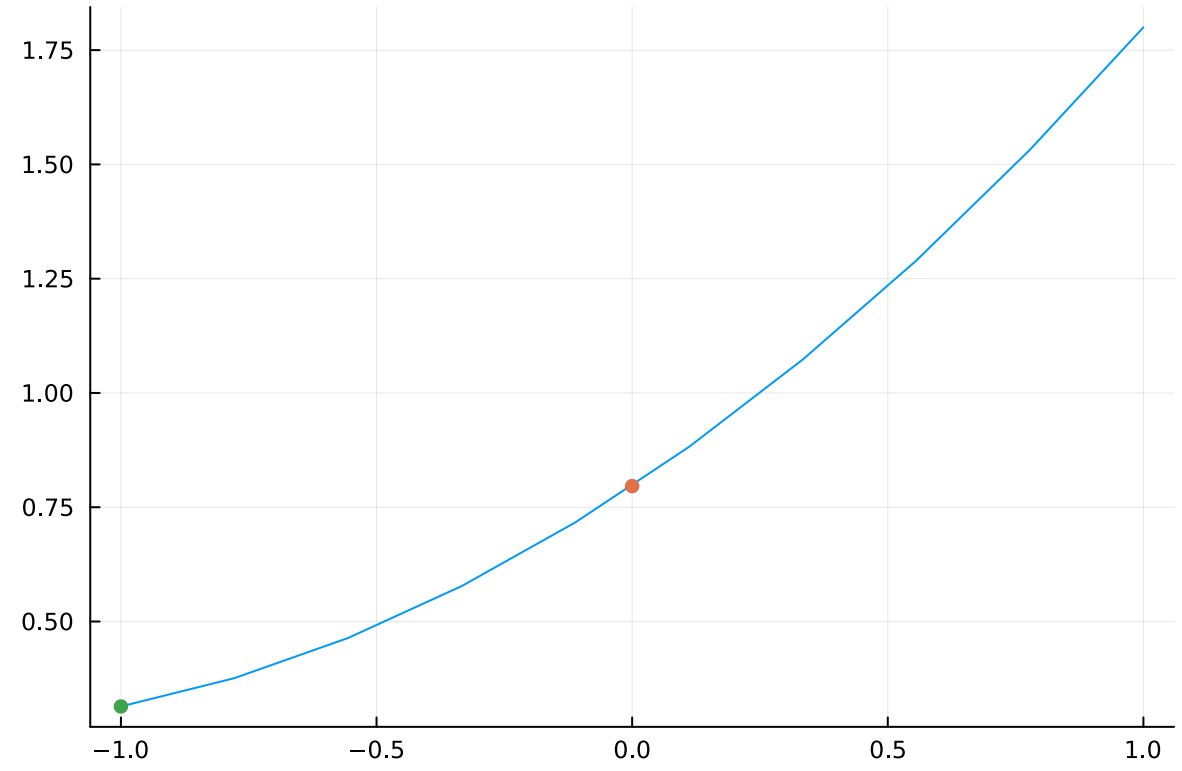
```
η_best = -1.0
```

```
1 η_best = step_sizes[best_idx]
```

```
best_loss = 0.31412162413575373
```

```
1 best_loss = losses[best_idx]
```

10




```
1 begin
2     plot(step_sizes, losses, label = "")
3     scatter!([0.0], [mse(w, X, y)], markerstrokewidth = 0, label = "")
4     scatter!([η_best], [best_loss], markerstrokewidth = 0, label = "")
5 end
```

```
w_improved = ▸ [0.510747, -0.75951]
```

```
1 w_improved = w + η_best * ∇
```

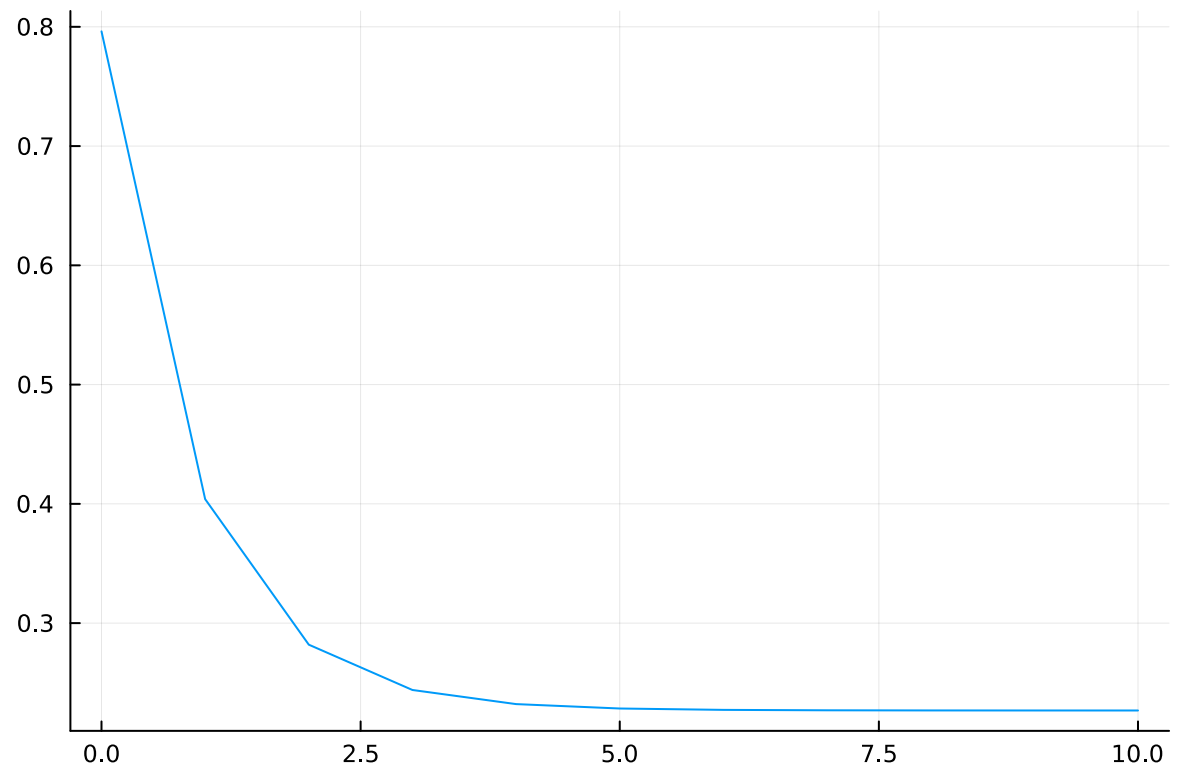
```
train! (generic function with 1 method)
```

```
function train!(diff, loss, w0, X, y, η, num_iters)
    w = copy(w0)
    training_losses = [loss(w, X, y)]
    for _ in 1:num_iters
        ∇ = diff(loss, w, X, y)
        w .= w .- η .* ∇
        push!(training_losses, loss(w, X, y))
    end
    return w, training_losses
end
```

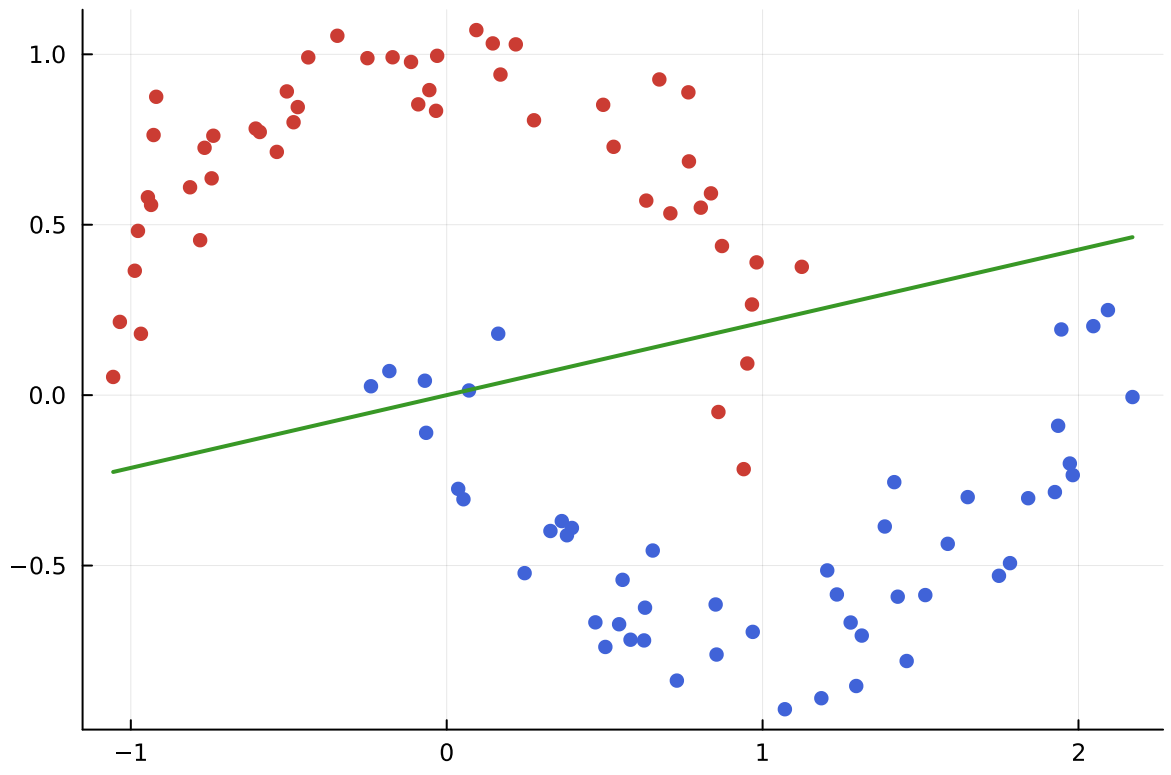
```
([0.253483, -1.1873], [0.7962, 0.403979, 0.281925, 0.243925, 0.232089, 0.228401, 0.227252,
```

```
w_trained, training_losses = train!(forward_diff, mse, w, X, y, 0.7, num_iters)
```

num_iters = 10




```
plot(0:num_iters, training_losses, label = "")
```




```
plot_w(w_trained)
```

Kernel trick

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

```
lift(x) = [1.0, x[1], x[2], x[1]^2, x[1] * x[2], x[2]^2, x[1]^3, x[1]^2 * x[2],
x[1] * x[2]^2, x[2]^3]
```

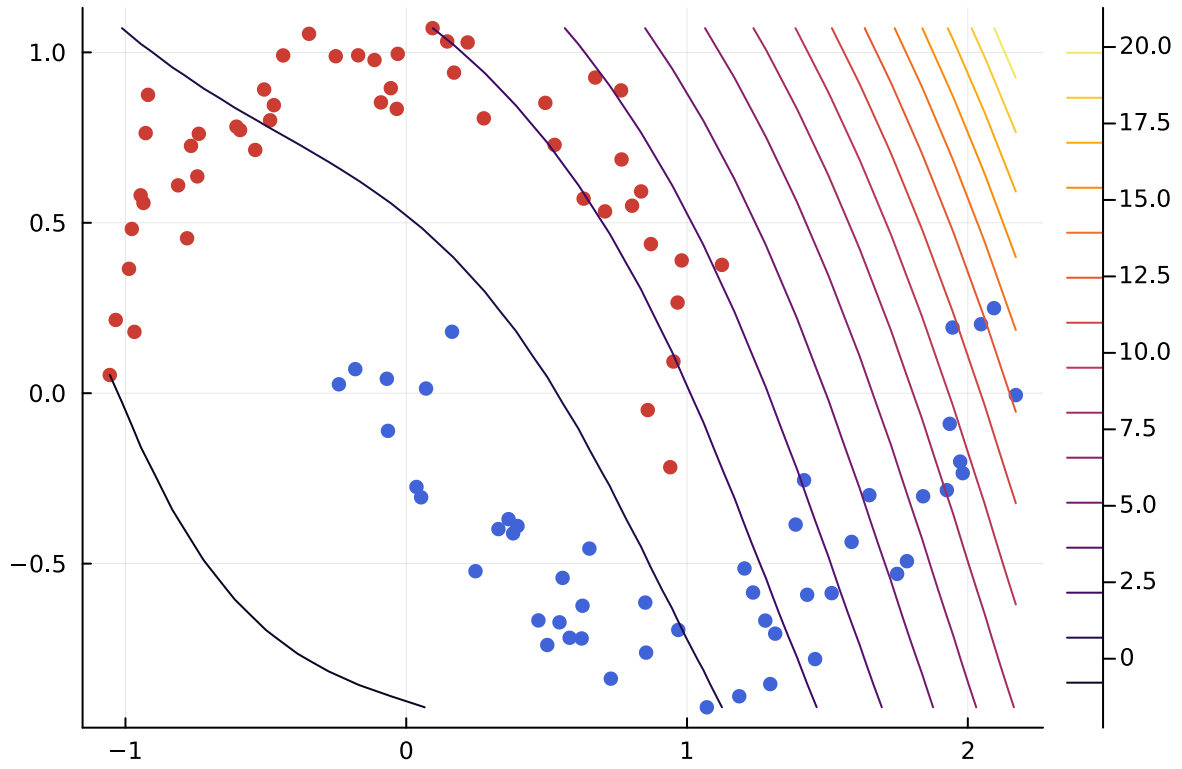
```
X_lift =
100×10 Matrix{Float64}:
 1.0  -0.780545    0.454589  0.60925      …   0.276958     -0.1613      0.0939412
 1.0   0.495337    0.851825  0.245359         0.209003      0.35942     0.61809
 1.0  -0.0337664   0.834215  0.00114017       0.000951145  -0.0234985   0.580542
 1.0   1.74827    -0.530045  3.05645         -1.62006       0.491173   -0.148915
 1.0   0.0938209   1.07106   0.00880237       0.00942789    0.107629    1.2287
 1.0   1.84076    -0.302403  3.3884       …  -1.02466       0.168333   -0.0276539
 1.0   0.980965    0.389551  0.962292         0.374862      0.148861    0.0591143
 ⋮                                      ⋱
 1.0  -0.346186    1.0544    0.119845         0.126365     -0.384878    1.17225
 1.0   0.708019    0.533406  0.501291    …    0.267392      0.201447    0.151766
 1.0   0.766845    0.685839  0.588051         0.403308      0.360705    0.322601
 1.0   1.58616    -0.436255  2.51591         -1.09758       0.301876   -0.0830274
 1.0  -0.60468     0.782136  0.365638         0.285979     -0.369905    0.478461
 1.0  -0.919653    0.875396  0.845762         0.740377     -0.704747    0.670832
```

```
X_lift = reduce(vcat, transpose.(lift.(eachrow(X))))
```

w_lift =

▸ [0.0647981, 0.580938, 0.885052, 0.620712, 0.681065, 0.34864, 0.832972, 0.759907, 0.476721

```
w_lift = rand(size(X_lift, 2))
```



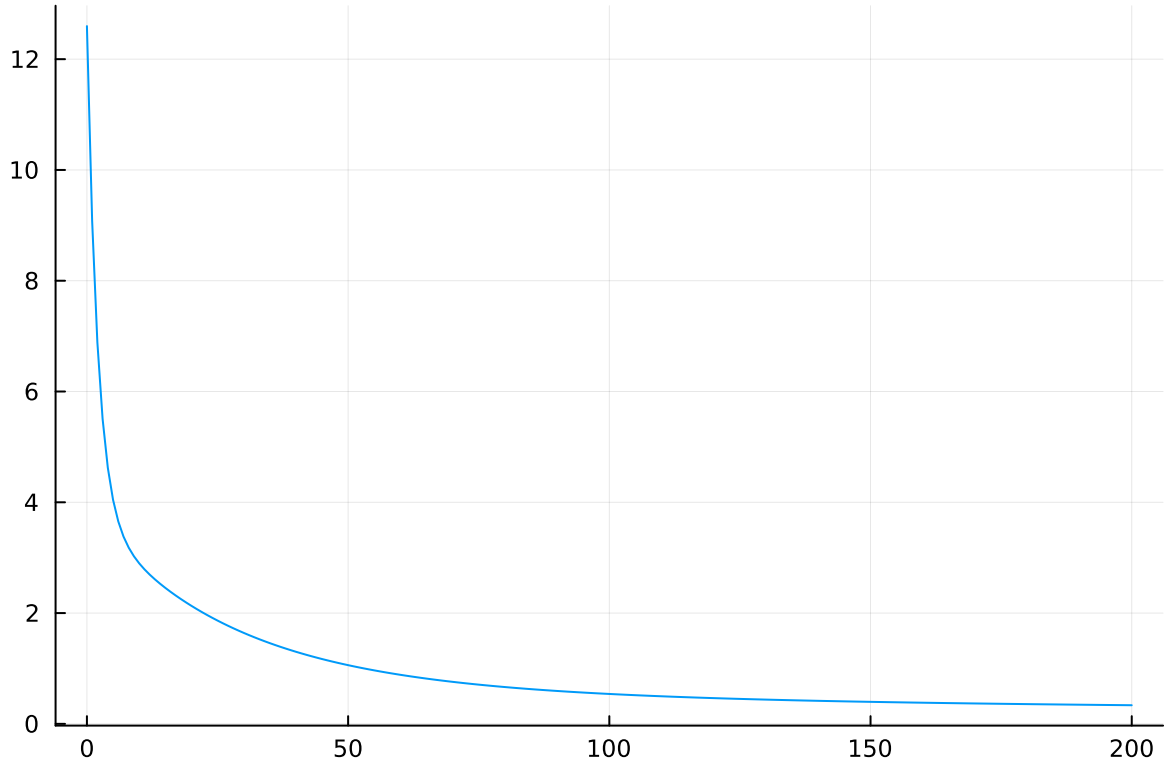

```
plot_w(w_lift)
```

η_lift = 0.01

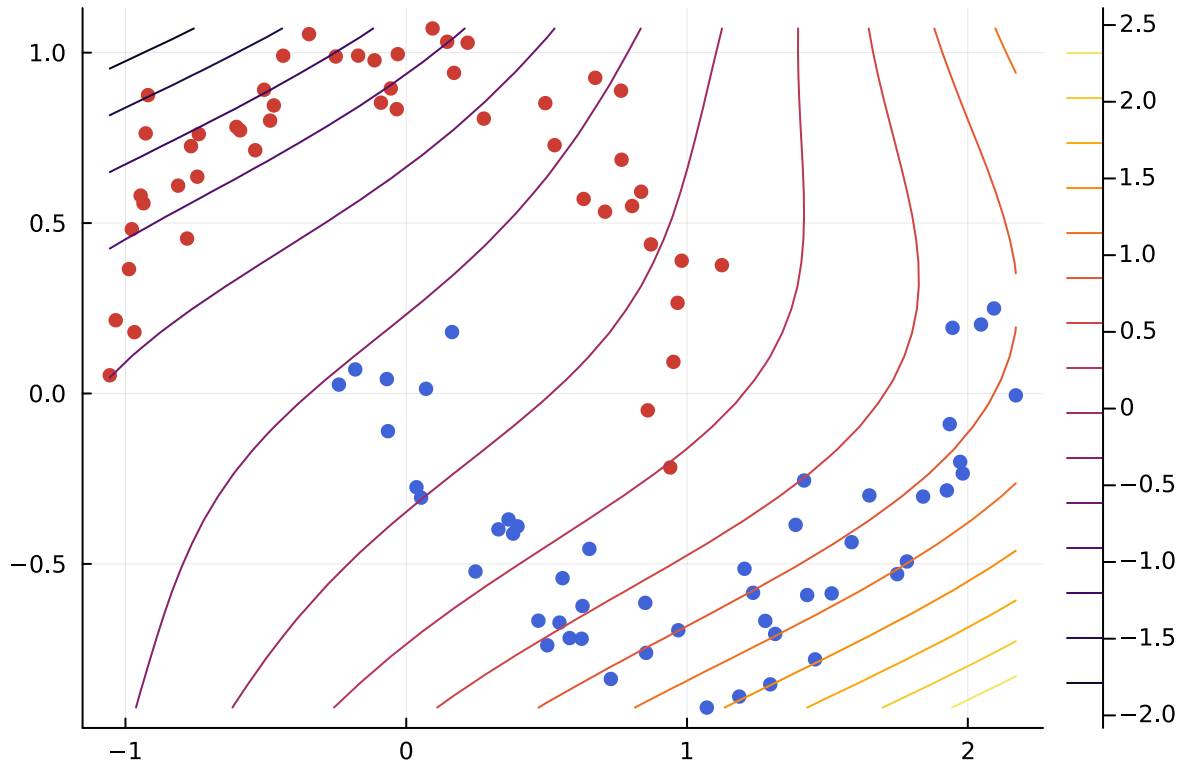
num_iters_lift = 200

▸ ([-0.207272, 0.331503, -0.478055, 0.00657985, -0.0270123, -0.00234553, 0.0363923, -0.0017

```
w_lift_trained, training_losses_lift = train!(forward_diff, mse, w_lift, X_lift, y, η_lift, num_iters_lift)
```



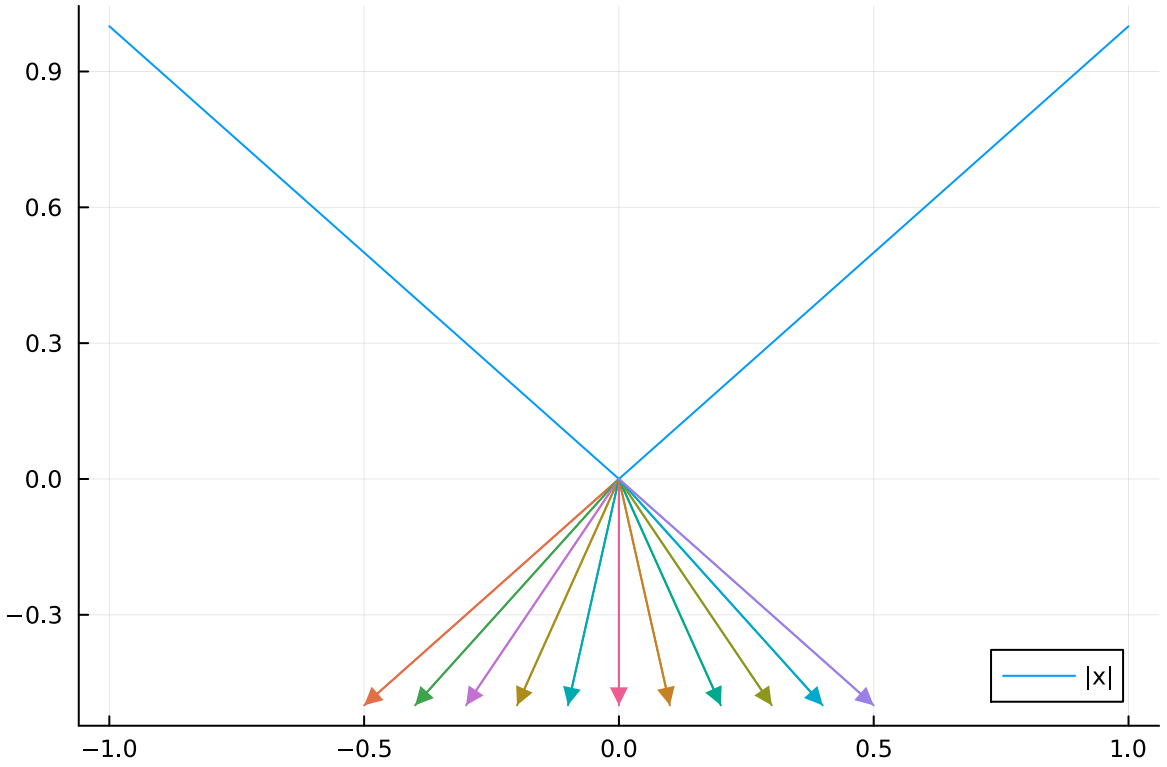

```
plot(0:num_iters_lift, training_losses_lift, label = "")
```




```
plot_w(w_lift_trained)
```

L1 norm

La fonction $|x|$ n'est pas différentiable lorsque $x = 0$. Si on s'approche par la gauche (c'est à dire $x < 0$, la fonction est $-x$) donc la dérivée vaut -1. Si on s'approche par la droite (c'est à dire $x > 0$, la fonction est x) donc la dérivée vaut 1. Il n'y a pas de gradient valide ! Par contre, n'importe quel nombre entre -1 et 1 est un **subgradient** valide ! Alors que le gradient est la normale à la tangente **unique**, le subgradient est un élément du **cone tangent**.




```
let
    x = range(-1, stop = 1, length = 11)
    p = plot(x, abs, axis = :equal, label = "|x|")
    for λ in range(0, stop = 1, length = 11)
        plot!([0, λ/2 - (1 - λ)/2], [0, -1/2], arrow = Plots.arrow(:closed), label
        = "")
    end
    p
end
```

⚠ Skipped xaxis arg equal

⚠ Skipped yaxis arg equal

⚠ Skipped zaxis arg equal

```
L1_loss (generic function with 1 method)
```

```
L1_loss(w, X, y) = sum(abs.(X * w - y)) / length(y)
```

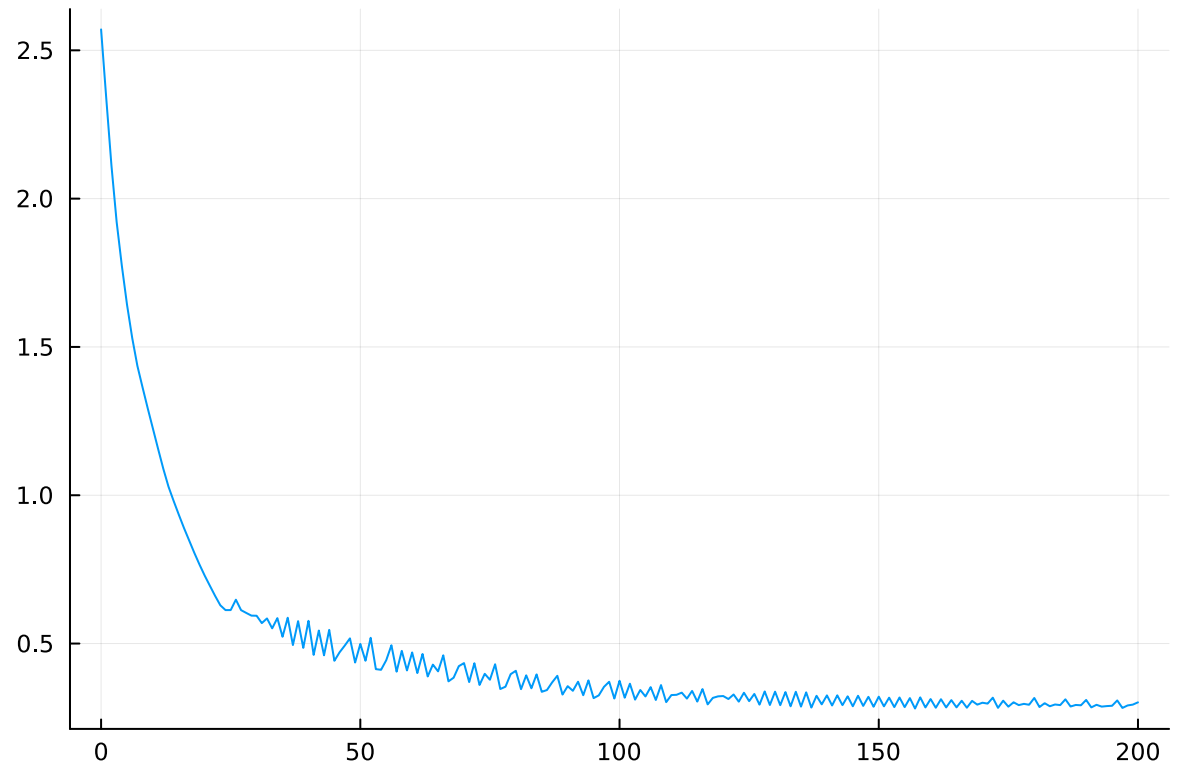
```
function Base.abs(d::Dual)
    if d.value < 0
        return -1.0 * d
    else
        return d
    end
end
```

η_L1 = 0.1

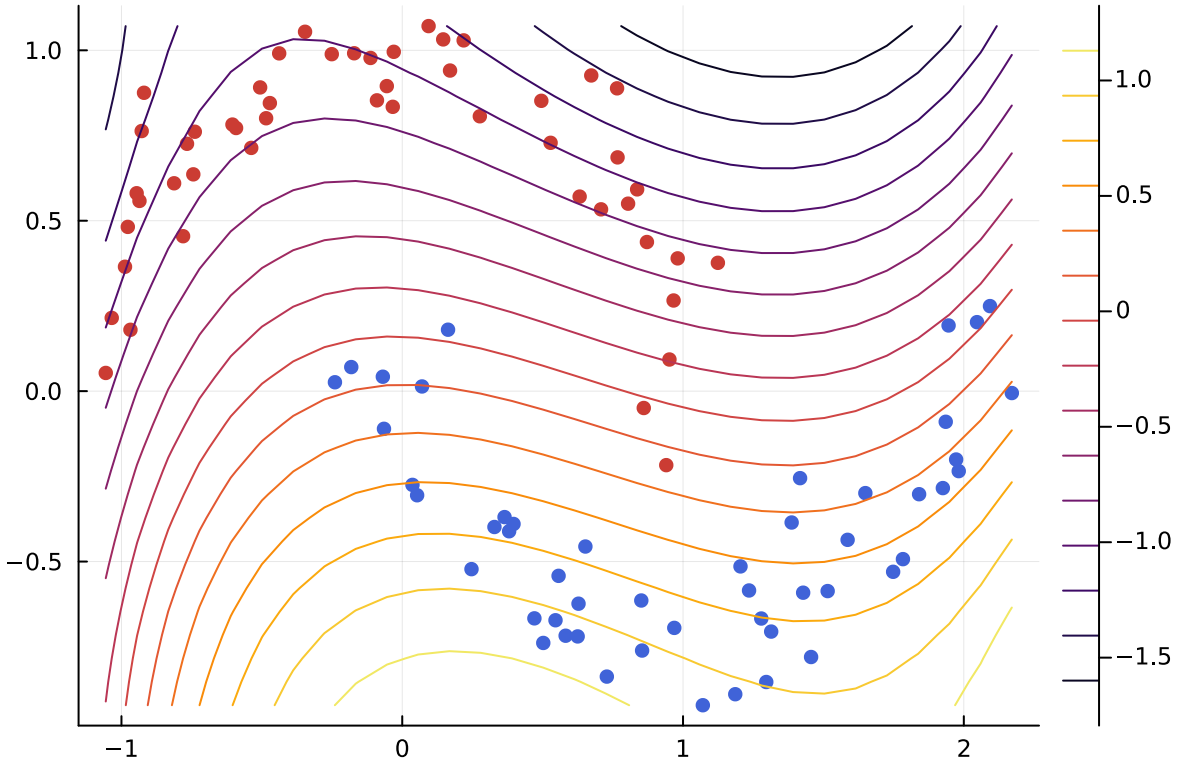
num_iters_L1 = 200

```
([0.180798, 0.0147275, -1.37894, -0.606945, -0.347741, -0.0638433, 0.296703, 0.157887, -0
```

```
w_trained_L1, training_losses_L1 = train!(forward_diff, L1_loss, w_lift, X_lift, y, η_L1, num_iters_L1)
```




```
plot(0:num_iters_L1, training_losses_L1, label = "")
```




```
1 plot_w(w_trained_L1)
```

Reverse diff

Le désavantage de la forward differentiation, c'est qu'il faut recommencer tout le calcul pour calculer la dérivée par rapport à chaque variable. La *reverse differentiation*, aussi appelée *backpropagation*, résoud ce problème en calculant la dérivée par rapport à toutes les variables en une fois !

Chain rule

Exemple univarié

Commençons par un exemple univarié pour introduire le fait qu'il existe un choix dans l'ordre de la multiplication des dérivées. La liberté introduite par ce choix donne lieu à la différence entre la différentiation *forward* et *reverse*.

Supposons qu'on veuille dériver la fonction $\tan(\cos(\sin(x)))$ pour $x = \pi/3$. La Chain Rule nous donne :

$$\begin{aligned}(\tan(\cos(\sin(x))))' &= (\tan(x))'|_{x=\cos(\sin(x))}(\cos(\sin(x)))' \\ &= (\tan(x))'|_{x=\cos(\sin(x))}(\cos(x))'|_{x=\sin(x)}(\sin(x)))' \\ &= \frac{1}{\cos^2(\cos(\sin(x)))}(-\sin(\sin(x)))\cos(x)\end{aligned}$$

La dérivée pour $x = \pi/3$ est donc :

$$\begin{aligned}(\tan(\cos(\sin(x))))'|_{x=\pi/3} &= (\tan(x))'|_{x=\cos(\sin(\pi/3)))}(\cos(x))'|_{x=\sin(\pi/3))}(\sin(x)))'|_{x=\pi/3} \\ &= \frac{1}{\cos^2(\cos(\sin(\pi/3)))}(-\sin(\sin(\pi/3)))\cos(\pi/3)\end{aligned}$$

Pour calculer ce produit de 3 nombres, on a 2 choix.

La première possibilité (qui correspond à forward diff) est de commencer par calculer le produit

$$\begin{aligned}(\cos(\sin(x)))'|_{x=\pi/3} &= (\cos(x))'|_{x=\sin(\pi/3))}(\sin(x)))'|_{x=\pi/3} \\ &= (-\sin(\sin(\pi/3)))\cos(\pi/3)\end{aligned}$$

puis de le multiplier avec $(\tan(x))'|_{x=\cos(\sin(\pi/3)))} = \frac{1}{\cos^2(\cos(\sin(\pi/3)))}$.

La deuxième possibilité (qui correspond à reverse diff) est de commencer par calculer le produit

$$\begin{aligned}(\tan(\cos(x)))'|_{x=\sin(\pi/3)} &= (\tan(x))'|_{x=\cos(\sin(\pi/3)))}(\cos(x))'|_{x=\sin(\pi/3))} \\ &= \frac{1}{\cos^2(\cos(\sin(\pi/3)))}(-\sin(\sin(\pi/3)))\end{aligned}$$

puis de le multiplier avec $\cos(\pi/3)$.

Vous remarquerez que dans l'équation ci-dessus, comme mis en évidence en rouge, les valeurs auxquelles les dérivées doivent être évaluées dépendent de $\sin(\pi/3)$. L'approche utilisée par reverse diff de multiplier de gauche à droite ne peut donc pas être effectuée sans prendre en compte la valeur qui doit être évaluée de droite à gauche.

Pour appliquer reverse diff, il faut donc commencer par une *forward pass* de droite à gauche qui calcule $\sin(\pi/3)$ puis $\cos(\sin(\pi/3))$ puis $\tan(\cos(\sin(\pi/3)))$. On peut ensuite faire la *backward pass* qui multiplie les dérivées de gauche à droite. Afin d'être disponibles pour la backward pass, les valeurs calculées lors de la forward pass doivent être **stockées** ce qui implique un **coût mémoire**. En revanche, comme forward diff calcule la dérivée dans le même sens que l'évaluation, les dérivées et évaluations peuvent être calculées en même temps afin de ne pas avoir besoin de stocker les évaluations. C'est effectivement ce qu'on a implémenté avec `Dual` précédemment.

Au vu de ce coût mémoire supplémentaire de reverse diff par rapport à forward diff, ce dernier paraît préférable en pratique. On va voir maintenant que dans le cas multivarié, dans certains cas, ce désavantage est contrebalancé par une meilleure complexité temporelle qui rend reverse diff indispensable !

Exemple multivarié

Prenons maintenant un exemple multivarié, supposons qu'on veuille calculer le gradient de la fonction $f(g(h(x_1, x_2)))$ qui compose 3 fonctions f, g et h. Le gradient est obtenu via la chain rule comme suit :

$$\begin{aligned}
\frac{\partial}{\partial x_1} f(g(h(x_1, x_2))) &= \frac{\partial f}{\partial g} \frac{\partial g}{\partial h} \frac{\partial h}{\partial x_1} \\
\frac{\partial}{\partial x_2} f(g(h(x_1, x_2))) &= \frac{\partial f}{\partial g} \frac{\partial g}{\partial h} \frac{\partial h}{\partial x_2} \\
\nabla_{x_1, x_2} f(g(h(x_1, x_2))) &= \begin{bmatrix} \frac{\partial f}{\partial g} \frac{\partial g}{\partial h} \frac{\partial h}{\partial x_1} & \frac{\partial f}{\partial g} \frac{\partial g}{\partial h} \frac{\partial h}{\partial x_2} \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial f}{\partial g} \end{bmatrix} \begin{bmatrix} \frac{\partial g}{\partial h} \end{bmatrix} \begin{bmatrix} \frac{\partial h}{\partial x_1} & \frac{\partial h}{\partial x_2} \end{bmatrix}
\end{aligned}$$

On voit que c'est le produit de 3 matrices. Forward diff va exécuter ce produit de droite à gauche :

$$\begin{aligned}
\nabla_{x_1, x_2} f(g(h(x_1, x_2))) &= \begin{bmatrix} \frac{\partial f}{\partial g} \end{bmatrix} \begin{bmatrix} \frac{\partial g}{\partial h} \frac{\partial h}{\partial x_1} & \frac{\partial g}{\partial h} \frac{\partial h}{\partial x_2} \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial f}{\partial g} \end{bmatrix} \begin{bmatrix} \frac{\partial g}{\partial x_1} & \frac{\partial g}{\partial x_2} \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial f}{\partial g} \frac{\partial g}{\partial x_1} & \frac{\partial f}{\partial g} \frac{\partial g}{\partial x_2} \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} \end{bmatrix}
\end{aligned}$$

L'idée de reverse diff c'est d'effectuer le produit de gauche à droite:

$$\begin{aligned}
\nabla_{x_1, x_2} f(g(h(x_1, x_2))) &= \begin{bmatrix} \frac{\partial f}{\partial g} \frac{\partial g}{\partial h} \end{bmatrix} \begin{bmatrix} \frac{\partial h}{\partial x_1} & \frac{\partial h}{\partial x_2} \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial f}{\partial h} \end{bmatrix} \begin{bmatrix} \frac{\partial h}{\partial x_1} & \frac{\partial h}{\partial x_2} \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} \end{bmatrix}
\end{aligned}$$

Pour calculer $\partial f / \partial x_1$ via forward diff, on part donc de $\partial x_1 / \partial x_1 = 1$ et $\partial x_2 / \partial x_1 = 0$ et on calcule ensuite $\partial h / \partial x_1$, $\partial g / \partial x_1$ puis $\partial f / \partial x_1$. Effectuer la reverse diff est un peu moins intuitif. L'idée est de partir de la dérivée du résultat par rapport à lui-même $\partial f / \partial f = 1$ et de calculer $\partial f / \partial g$ puis $\partial f / \partial h$ et ensuite $\partial f / \partial x_1$. L'avantage de reverse diff c'est qu'il n'y a que la dernière

étape qui est spécifique à x_1. Tout jusqu'au calcul de $\partial f/\partial h$ peut être réutilisé pour calculer $\partial f/\partial x_2$, il n'y a plus qu'à multiplier ! Reverse diff est donc plus efficace pour calculer le gradient d'une fonction qui a une seule output par rapport à beaucoup de paramètres, comme détaillé dans la discussion à la fin de ce notebook.

Forward pass : Construction de l'expression graph

Pour implémenter reverse diff, il faut construire l'expression graph pour garder en mémoire les valeurs des différentes expressions intermédiaires afin de pouvoir calculer les dérivées locales $\partial f/\partial g$ et $\partial g/\partial h$. Le code suivant définit un noeud de l'expression graph. Le field `derivative` correspond à la valeur de $\partial f_{\text{final}}/\partial f_{\text{node}}$ où f_{final} est la dernière fonction de la composition et f_{node} est la fonction correspondant au node.

Node

```
begin
    mutable struct Node
        op::Union{Nothing,Symbol}
        args::Vector{Node}
        value::Float64
        derivative::Float64
    end
    Node(op, args, value) = Node(op, args, value, NaN)
    Node(value) = Node(nothing, Node[], value)
end
```

L'opérateur overloading suivant sera sufficant pour construire l'expression graph dans le cadre de ce notebook, vous l'étendrez pendant la séance d'exercice.

```
begin
    Base.zero(x::Node) = Node(0.0)
    Base.:*(x::Node, y::Node) = Node(:*, [x, y], x.value * y.value)
    Base.:+(x::Node, y::Node) = Node(:+, [x, y], x.value + y.value)
    Base.:-(x::Node, y::Node) = Node(:-, [x, y], x.value - y.value)
    Base.:/(x::Node, y::Number) = x * Node(inv(y))
    Base.:^(x::Node, n::Integer) = Base.power_by_squaring(x, n)
    Base.sin(x::Node) = Node(:sin, [x], sin(x.value))
    Base.cos(x::Node) = Node(:cos, [x], cos(x.value))
end
```

On crée les leafs correspondant aux variables x_1 et x_2 de valeurs 1 et 2 respectivement. Les valeurs correspondent aux valeurs de x_1 et x_2 auxquelles on veut dériver la fonction. On a choisi 1 et 2 pour pouvoir les reconnaitre facilement dans le graphe.

```
x_nodes = ▸[Node(nothing, [], 1.0, NaN), Node(nothing, [], 2.0, NaN)]
```

```
1 x_nodes = Node.([1, 2])
```

```
expr = ▸Node(:cos, [Node(:sin, [⋯ more], 0.909297, NaN)], 0.6143002821164822, NaN)
```

```
1 expr = cos(sin(prod(x_nodes)))
```

```
_nodes! (generic function with 1 method)
```

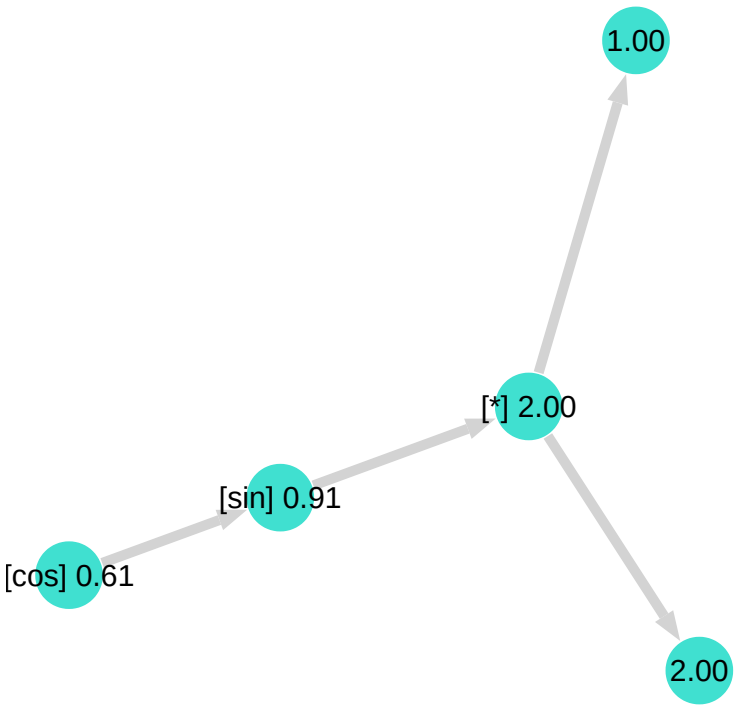
```
_edges (generic function with 1 method)
```

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

Pour le visualiser, on le convertit en graphe en utilisant la structure de données de Graphs.jl pour pouvoir utiliser `gplot`.

```
▸({5, 4} directed simple Int64 graph, ["[cos] 0.61", "[sin] 0.91", "[*] 2.00", "1.00", "2.
```

```
1 expr_graph, labels = graph(expr)
```




```
1 gplot(expr_graph, nodelabel = labels)
```

Combinaison des dérivées

Que faire si plusieurs expressions dépendent d'une même variable ? Considérons l'exemple $f(x) = \sin(x)\cos(x)$ qui correspond à $f(g,h) = gh$, $g(x) = \sin(x)$ et $h(x) = \cos(x)$. La chain rule donne

$$f'(x) = \frac{\partial f}{\partial g} g'(x) + \frac{\partial f}{\partial h} h'(x)$$

Une fois la valeur $\partial f / \partial g$ calculée, on peut la multiplier par $g'(x)$ pour avoir la première partie partie de $f'(x)$. Idem pour h. Ces deux contributions seront calculée séparément lors de la backward pass sur le noeud g et h. On voit par la formule de la chain rule que ces deux contributions doivent être sommées. Lors de la backward pass, on initialise donc toutes les dérivées à 0. Pour chaque contribution, on ajoute la dérivée avec `+=`. On s'assure ensuite qu'on ne procède pas à la backward pass sur un noeud avant qu'il ait fini d'accumuler les contributions de toutes les expressions qui en dépendent via un *tri topologique*.

```
x_node = ▸ Node(nothing, [], 1.0, NaN)
```

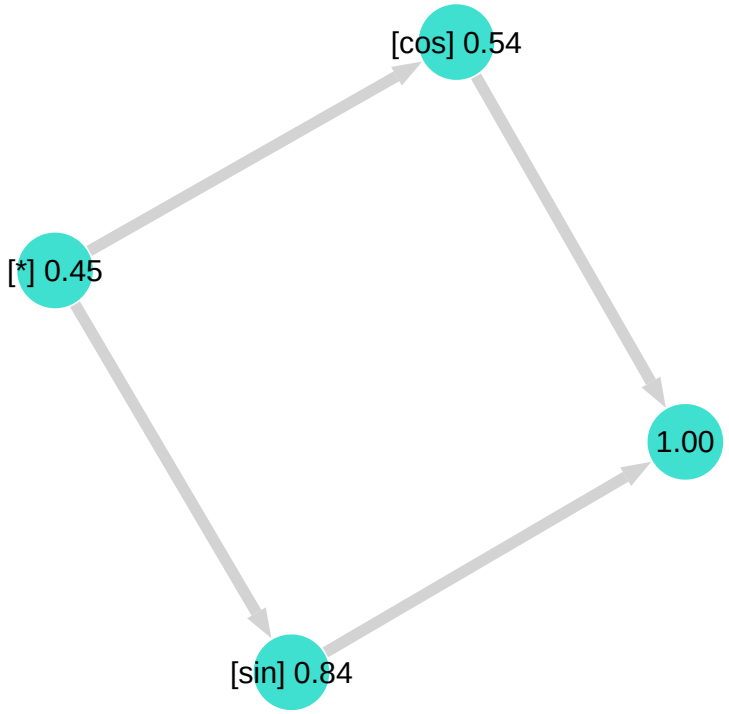
```
1 x_node = Node(1)
```

```
sin_cos =
▸ Node(:*, [Node(:sin, [⋯ more], 0.841471, NaN), Node(:cos, [⋯ more], 0.540302, NaN)], 0.454
```

```
1 sin_cos = sin(x_node) * cos(x_node)
```

```
▸ ({4, 4} directed simple Int64 graph, ["[*] 0.45", "[sin] 0.84", "1.00", "[cos] 0.54"])
```

```
1 sin_cos_graph, sin_cos_labels = graph(sin_cos)
```




```
1 gplot(sin_cos_graph, nodelabel = sin_cos_labels)
```

Backward pass : Calcul des dérivées

La fonction suivante propage la dérivée $\partial f_{\text{final}}/\partial f_{\text{node}}$ à la dérivée des arguments de la fonction f_{node}. Comme les arguments peuvent être utilisés par d'autres fonction, on somme la dérivée avec `+=`.

```
_backward! (generic function with 1 method)
```

```
function _backward!(f::Node)
    if isnothing(f.op)
        return
    elseif f.op == :+
        for arg in f.args
            arg.derivative += f.derivative
        end
    elseif f.op == :- && length(f.args) == 2
        f.args[1].derivative += f.derivative
        f.args[2].derivative -= f.derivative
    elseif f.op == :* && length(f.args) == 2
        f.args[1].derivative += f.derivative * f.args[2].value
        f.args[2].derivative += f.derivative * f.args[1].value
    elseif f.op == :sin
        f.args[].derivative += f.derivative * cos(f.args[].value)
    elseif f.op == :cos
        f.args[].derivative -= f.derivative * sin(f.args[].value)
    else
        error("Operator `$(f.op)` not supported yet")
    end
end
```

La fonction `_backward!` ne doit être appelée que sur un noeud pour lequel `f.derivative` a déjà été calculé. Pour cela, `_backward!` doit avoir été appelé sur tous les noeuds qui représentent des fonctions qui dépendent directement ou indirectement du résultat du noeud. Pour trouver l'ordre dans lequel appeler `_backward!`, on utilise donc un tri topologique (nous reviendrons sur les tris topologiques dans la partie graphe).

```
backward! (generic function with 1 method)
```

```
function backward!(f::Node)
    topo = typeof(f)[]
    topo_sort!(Set{typeof(f)}(), topo, f)
    reverse!(topo)
    for node in topo
        node.derivative = 0
    end
    f.derivative = 1
    for node in topo
        _backward!(node)
    end
    return f
end
```

```
topo_sort! (generic function with 1 method)
```

```
function topo_sort!(visited, topo, f::Node)
    if !(f in visited)
        push!(visited, f)
        for arg in f.args
            topo_sort!(visited, topo, arg)
        end
        push!(topo, f)
    end
end
```

```
▸ Node(:cos, [Node(:sin, [⋯ more], 0.909297, -0.789072)], 0.6143002821164822, 1.0)
```

```
backward!(expr)
```

On a maintenant l'information sur les dérivées de `x_nodes` :

```
▸ [Node(nothing, [], 1.0, 0.65674), Node(nothing, [], 2.0, 0.32837)]
```

```
x_nodes
```

Comparaison avec Forward Diff dans l'exemple moon

Revenons sur l'exemple utilisé pour illustrer la forward diff et essayons de calculer la même dérivée mais à présent en utilisant reverse diff.

```
w_nodes = ▸ [Node(nothing, [], 0.614529, NaN), Node(nothing, [], 0.0961168, NaN)]
```

```
w_nodes = Node.(w)
```

```
mse_expr = ▸ Node(:*, [Node(:*, [⋯ more], 1.0, 1.0), Node(nothing, [], 1.0, 1.0)], 1.0, 1.0)
```

```
mse_expr = backward!(mse(Node.(ones(1)), Node.(2ones(1, 1)), Node.(3ones(1))))
```

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

```
function reverse_diff(loss, w, X, y)
    w_nodes = Node.(w)
    expr = loss(w_nodes, Node.(X), Node.(y))
    backward!(expr)
    return [w.derivative for w in w_nodes]
end
```

```
▸ [0.103782, 0.855627]
```

```
@time forward_diff(mse, w, X, y)
```

```
  0.000016 seconds (25 allocations: 13.359 KiB)
```

```
▸ [0.103782, 0.855627]
```

```
1 @time reverse_diff(mse, w, X, y)
```

```
  0.087064 seconds (226.25 k allocations: 10.933 MiB, 99.80% compilation time)
```

On l'exécute une seconde fois, pour éliminer le temps de compilation :

```
▸ [0.103782, 0.855627]
```

```
1 @time reverse_diff(mse, w, X, y)
```

```
  0.000142 seconds (4.15 k allocations: 240.234 KiB)
```

On remarque que reverse diff est plus lent ! Il y a un certain coût mémoire lorsqu'on construit l'expression graph. Pour cette raison, si on veut calculer plusieurs dérivées consécutives pour différentes valeurs de x_1 et x_2, on a intérêt à garder le graphe et à uniquement changer la valeur des variables plutôt qu'à reconstruire le graphe à chaque fois qu'on change les valeurs. Alternativement, on peut essayer de condenser le graphe en exprimant les opérations sur des larges matrices ou même tenseurs, c'est l'approche utilisée par pytorch ou tensorflow.

num_data = 32

num_features = 32

num_hidden = 32

```
mse2 (generic function with 1 method)
```

```
1 mse2(W1, W2, X, y) = sum((X * W1 * W2 - y).^2 / length(y))
```

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

```
function bench(num_data, num_features, num_hidden)
    X = rand(num_data, num_features)
    W1 = rand(num_features, num_hidden)
    W2 = rand(num_hidden)
    y = rand(num_data)

    @time for i in axes(W1, 1)
        for j in axes(W1, 2)
            mse2(
                Dual.(W1, onehot(i, axes(W1, 1)) * onehot(j, axes(W1, 2))'),
                W2,
                X,
                y,
            )
        end
    end
    expr = @time mse2(Node.(W1), Node.(W2), Node.(X), Node.(y))
    @time backward!(expr)
    return
end
```

```
bench(num_data, num_features, num_hidden)
```

```
  0.007931 seconds (16.38 k allocations: 19.133 MiB)
  0.000120 seconds (17.55 k allocations: 737.734 KiB)
  0.000658 seconds (38 allocations: 284.438 KiB)
```

Comment choisir entre forward et reverse diff ?

Suppose that we need to differentiate a composition of functions: $(f_n \circ f_{n-1} \circ \cdots \circ f_2 \circ f_1)(w)$. For each function, we can compute a jacobian given the value of its input. So, during a forward pass, we can compute all jacobians. We now just need to take the product of these jacobians:

$$J_n J_{n-1} \cdots J_2 J_1$$

While the product of matrices is associative, its computational complexity depends on the order of the multiplications! Let $d_i \times d_{i-1}$ be the dimension of J_i.

Forward diff: from right to left

If the product is computed from right to left:

$$
\begin{aligned}
J_{1,2} &= J_2 J_1 && \Omega(d_2 d_1 d_0) \\
J_{1,3} &= J_3 J_{1,2} && \Omega(d_3 d_2 d_0) \\
J_{1,4} &= J_4 J_{1,3} && \Omega(d_4 d_3 d_0) \\
&\vdots \quad \vdots \\
J_{1,n} &= J_n J_{1,(n-1)} && \Omega(d_n d_{n-1} d_0)
\end{aligned}
$$

we have a complexity of

$$\Omega(\sum_{i=2}^{n} d_i d_{i-1} d_0).$$

Reverse diff: from left to right

Reverse differentation corresponds to multiplying the adjoint from right to left or equivalently the original matrices from left to right. This means computing the product in the following order:

$$
\begin{aligned}
J_{(n-1),n} &= J_n J_{n-1} && \Omega(d_n d_{n-1} d_{n-2}) \\
J_{(n-2),n} &= J_{(n-1),n} J_{n-2} && \Omega(d_n d_{n-2} d_{n-3}) \\
J_{(n-3),n} &= J_{(n-2),n} J_{n-3} && \Omega(d_n d_{n-3} d_{n-4}) \\
&\vdots \quad \vdots \\
J_{1,n} &= J_{2,n} J_1 && \Omega(d_n d_1 d_0)
\end{aligned}
$$

We have a complexity of

$$\Omega(\sum_{i=1}^{n-1} d_n d_i d_{i-1}).$$

Mixed : from inward to outward

Suppose we multiply starting from some d_k where $1 < k < n$. We would then first compute the left side:

$$
\begin{aligned}
J_{k+1,k+2} &= J_{k+2} J_{k+1} && \Omega(d_{k+2} d_{k+1} d_k) \\
J_{k+1,k+3} &= J_{k+3} J_{k+1,k+2} && \Omega(d_{k+3} d_{k+2} d_k) \\
&\vdots \quad \vdots \\
J_{k+1,n} &= J_n J_{k+1,n-1} && \Omega(d_n d_{n-1} d_k)
\end{aligned}
$$

then the right side:

$$
\begin{aligned}
J_{k-1,k} &= J_k J_{k-1} && \Omega(d_k d_{k-1} d_{k-2}) \\
J_{k-2,k} &= J_{k-1,k} J_{k-2} && \Omega(d_k d_{k-2} d_{k-3}) \\
&\vdots \quad \vdots \\
J_{1,k} &= J_{2,k} J_1 && \Omega(d_k d_1 d_0)
\end{aligned}
$$

and then combine both sides:

$$
J_{1,n} = J_{k+1,n} J_{1,k} \qquad \Omega(d_n d_k d_0)
$$

we have a complexity of

$$
\Omega(d_n d_k d_0 + \sum_{i=1}^{k-1} d_k d_i d_{i-1} + \sum_{i=k+2}^{n} d_i d_{i-1} d_k).
$$

Comparison

We see that we should find the minimum d_k and start from there. If the minimum is attained at $k = n$, this corresponds mutliplying from left to right, this is reverse differentiation. If the minimum is attained at $k = 0$, we should multiply from right to left, this is forward differentiation. Otherwise, we should start from the middle, this would mean mixing both forward and reverse diff.

What about neural networks ? In that case, d_0 is equal to the number of entries in W_1 added with the number of entries in W_2 while d_n is 1 since the loss is scalar. We should therefore clearly multiply from left to right hence do reverse diff.

Pour la séance d'exercices:

tanh

Passer un réseau de neurone avec fonction d'activation `tanh`

```
W1 =
2×32 Matrix{Float64}:
 0.928212  0.272494   0.302023  0.681915  …  0.525217  0.628835  0.929544   0.742009
 0.443788  0.0114539  0.288565  0.167577     0.713208  0.104145  0.0493637  0.093836
```

```
W1 = rand(size(X, 2), num_hidden)
```

```
W2 =
▸ [0.438865, 0.615116, 0.380671, 0.786789, 0.136726, 0.983433, 0.498938, 0.831164, 0.318933
```

```
W2 = rand(num_hidden)
```

```
y_est_tanh =
▸ [-3.2372, 7.78992, 4.37956, 8.18643, 6.23771, 9.45206, 8.56814, -1.18715, 7.95551, 6.88116
```

```
y_est_tanh = tanh.(X * W1) * W2
```

ReLU

Passer un réseau de neurone avec fonction d'activation ReLU

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

```
function relu(x)
    if x < 0
        return 0
    else
        return x
    end
end
```

```
y_est_relu =
▸ [0.342842, 9.12834, 4.79874, 10.8767, 7.25594, 12.8968, 10.1955, 0.0, 9.79686, 8.73857, 0.
```

```
y_est_relu = relu.(X * W1) * W2
```

One-Hot encoding et cross-entropy

```
Y = 100×2 BitMatrix:
 1  0
 1  0
 1  0
 0  1
 1  0
 0  1
 1  0
 ⋮
 1  0
 1  0
 1  0
 0  1
 1  0
 1  0
```

```
Y = unique!(sort(y_cat.refs))' .== y_cat.refs
```

```
W = 2×2 Matrix{Float64}:
 0.600246  0.678143
 0.930934  0.583002
```

```
W = rand(size(X, 2), size(Y, 2))
```

```
Y_1 = 100×2 Matrix{Float64}:
      -0.0453267  -0.264295
       1.09032     0.832525
       0.756331    0.463451
       0.555955    0.876559
       1.05341     0.688057
       0.823393    1.072
       0.951467    0.892343
       ⋮
       0.773783    0.379956
       0.921552    0.791115
       1.09877     0.919876
       0.545963    0.821307
       0.36516     0.0459275
       0.262918   -0.113298
```

```
1 Y_1 = X * W
```

```
Y_2 = 100×2 Matrix{Float64}:
       0.955685  0.767747
       2.97522   2.29912
       2.13045   1.58955
       1.74361   2.40262
       2.8674    1.98984
       2.27822   2.92121
       2.58951   2.44084
       ⋮
       2.16795   1.46222
       2.51319   2.20586
       3.00046   2.50898
       1.72627   2.27347
       1.44074   1.047
       1.30072   0.892884
```

```
1 Y_2 = exp.(X * W)
```

```
sums = 100×1 Matrix{Float64}:
        1.7234325859364996
        5.2743360170192
        3.7199947618237728
        4.146224754898377
        4.8572445100968205
        5.199427659716082
        5.03034819249333
        ⋮
        3.6301716094908416
        4.719043342014801
        5.50944220811363
        3.9997405189755266
        2.4877433117458767
        2.19360440553851
```

```
1 sums = sum(Y_2, dims=2)
```

```
Y_est = 100×2 Matrix{Float64}:
        0.554524  0.445476
        0.564093  0.435907
        0.572701  0.427299
        0.420529  0.579471
        0.590335  0.409665
        0.438167  0.561833
        0.514777  0.485223
        ⋮
        0.597204  0.402796
        0.532563  0.467437
        0.544604  0.455396
        0.431596  0.568404
        0.579137  0.420863
        0.59296   0.40704
```

```
Y_est = Y_2 ./ sums
```

```
cross = 100×2 Matrix{Float64}:
        0.554524  0.0
        0.564093  0.0
        0.572701  0.0
        0.0       0.579471
        0.590335  0.0
        0.0       0.561833
        0.514777  0.0
        ⋮
        0.597204  0.0
        0.532563  0.0
        0.544604  0.0
        0.0       0.568404
        0.579137  0.0
        0.59296   0.0
```

```
cross = Y_est .* Y
```

```
cross_entropies = 100×1 Matrix{Float64}:
                  0.5896447058458247
                  0.572535303354674
                  0.5573913182091242
                  0.5456388366035338
                  0.5270657177227734
                  0.5765505684782167
                  0.6640223977625751
                  ⋮
                  0.5154970745641536
                  0.6300539882003595
                  0.6076968062871445
                  0.5649222536673819
                  0.5462158211425789
                  0.5226280331227549
```

```
cross_entropies = -log.(sum(Y_est .* Y, dims=2))
```

```
▸ [0.589645, 0.572535, 0.557391, 0.545639, 0.527066, 0.576551, 0.664022, 0.676591, 0.598504
```

```
-log.(getindex.(Ref(Y_est), axes(Y_est, 1), y_cat.refs))
```

Acknowledgements and further readings

- `Dual` est inspiré de ForwardDiff
- `Node` est inspiré de micrograd
- Une bonne intro à l'automatic differentiation est disponible ici

Utils

```
1 import MLJBase, Colors, Tables
```

```
1 using Graphs, GraphPlot, Printf
```

```
1 using Plots, OneHotArrays, PlutoUI
```