

Juliaと歩く 自動微分の世界

Julia Tokyo #11

@abap34

2023/02/03

自己紹介

@abap34

東工大 情報理工学院 情報工学系 B2
(もうすぐ消滅)

趣味

- 🤖 機械学習
- 🛡️ 個人開発
- ⚾ 野球をする/みる

GitHub: @abap34

Twitter: @abap34



自己紹介

✓ Juliaをこんなことに使
ってます！

1. データ分析

2. 機械学習や数学の勉強・調べ物

3. 競プロ

The screenshot shows a Julia development environment with three main panes:

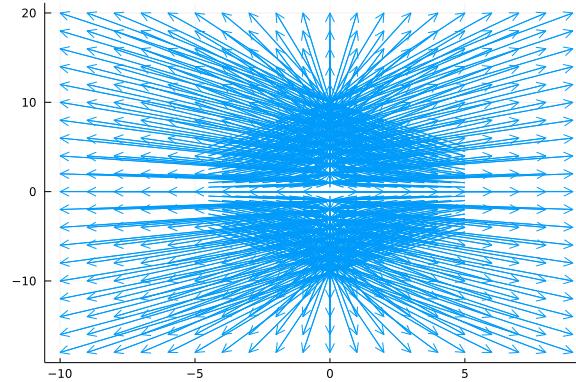
- gradient_descent.jl**: A code editor containing Julia code for gradient descent. It includes imports for `JITracer`, `ProgressBars`, and `Animation`. The code defines variables `lr`, `iters`, and `history`, and iterates over `iters` to update `x` and `y` using the Rosenbrock function. It also plots the history and saves an animation.
- Julia REPL (v1.8.4)**: A terminal window showing the execution of the code. It prints the value of `y` at each iteration from 1 to 100, and then generates an animation and a gif file named `6WMP.gif`.
- Julia Plots (180/180)**: A dependency graph visualization showing the relationships between various components. Nodes represent components like `Square`, `Sub`, `Mu`, and `Add`, and edges show their dependencies.

✓ Juliaのこんなところが気に入っています！

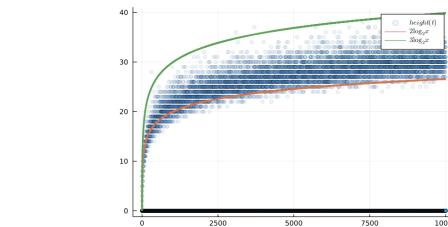
1. 綺麗な可視化・ベンチマークライブラリ
 - Plotまわり, @code_... マクロ, BenchmarkTools.jl たち
2. パッケージ管理ツール
 - 言語同梱で超便利（友人間で共有するのに一苦労の言語も）
3. すぐ書ける すぐ動く
 - Jupyter のサポート, 強力な REPL
4. 速い！！
 - 速度は正義
 - 裏が速いライブラリの「芸人」にならなくても、素直に書いてそのまま速い

Julia を使って解かれた・書かれたレポートたち

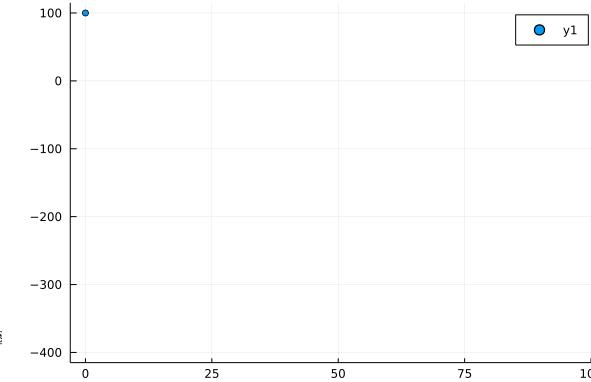
Introduction



② これでオセロ用に実装してみた。このように、一旦探索木をリモーブする機能が内蔵されている。



整列済みの長さ N の配列に対して生成される Treap の高さを調べたグラフである。この結果によると、Treap の高さは N の $\log_2 N$ に比例する。

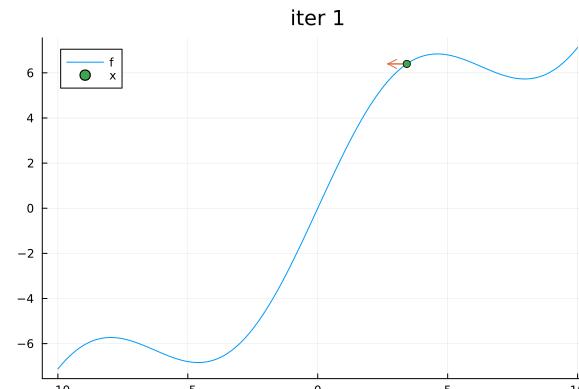
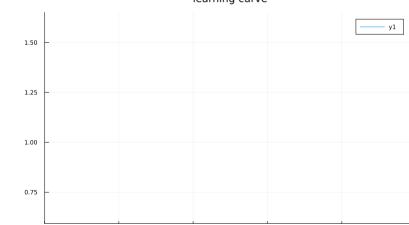
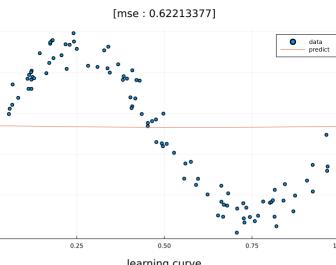


⇒ ✓ 16倍～高速化！！

実装 / オセロ自体の実装
吐かれるネイティブコードを見ると...
ピットボード 二次元配列を使った実装

ピットボードに対する操作は基本的なビット演算で済む
⇒ 少ない命令数で動く
例) 石の数を数える: popcnt命令のみでOK！

実装 / オセロ自体の実装
そのほかの操作もビット演算のみで行える
例1) 合法手の列挙: 6回右シフトを繰り返すことで行える



今日のお話

one of 興味があるもの

機械学習(特に深層学習)

の基盤

README MIT license

JITrench



Let's dive into the deep trenches of the loss function with JITrench.jl.

docs stable docs dev build passing

Install

```
jadd https://github.com/abap34/JITrench.jl
```

Automatic gradient calculation

```
julia> using JITrench
julia> f(x) = sin(x) + 1
f (generic function with 1 method)

julia> JITrench.@diff! f(x)
f' (generic function with 1 method)

julia> f'(π)
-1.0
```

:自作DLフレームワーク

one of 深層学習の基盤

自動微分

について話します。

こんなことがありますなんか？

1. 深層学習フレームワークを使っているけど、あまり中身がわかっていない。
2. 微分を求めたいことがあって既存のライブラリを使っているけど、どの場面でどれを使うのが適切かわからない
3. 自分の計算ルーチンに微分の計算を組み込みたいけどやり方がわからない
4. え、自動微分ってただ計算するだけじゃないの？何がおもしろいの？

[メインテーマ]

- 自動で微分を求めるもののモチベーション
- 自動で微分を求めるアルゴリズムたちの紹介と実装
- 一般的な自動微分の実装
- 自動微分の先進的な研究 (Julia まわりを中心に)
- 微分ライブラリの紹介

こんなワードが出てきます：

勾配降下法, 数値微分, 数式微分, 自動微分, 誤差評価, 深層学習フレームワーク, Define and/by Run
計算グラフ, Wengert List, Source Code Transformation(SCT), SSA形式

[1] 微分と連続最適化

- 1.1 微分のおさらい
- 1.2 勾配降下法
- 1.3 勾配降下法と機械学習

[2] 自動で微分

- 2.1 微分の近似—数値微分
- 2.2 誤差なしの微分—数式微分
- 2.3 式の微分からアルゴリズムの微分へ
- 2.4 自動微分とトレース
- 2.5 自動微分とソースコード変換

[3] Juliaに微分させる

- 3.1 FiniteDiff.jl/FiniteDifferences.jl
- 3.1 ForwardDiff.jl
- 3.2 Zygote.jl/Diffractor.jl
- 3.3 AbstractDifferentiation.jl

[4] Juliaの微分を拡張する

- 4.1 ChainRules

[5] まとめ

[6] 参考になる文献

[1] 微分と連続最適化

1.1 微分のおさらい

1.2 勾配降下法

1.3 勾配降下法と機械学習

微分の定義 from 高校数学

[定義1. 微分係数]

関数 f の x における微分係数は

$$\lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

- 偏微分の定義 from 大学数学

x_i について偏微分 $\leftrightarrow x_i$ 以外の変数を固定して微分

[定義2. 偏微分係数]

n 変数関数 f の (x_1, \dots, x_n) の x_i に関する偏微分係数

$$\frac{\partial f}{\partial x_i}(x_1, \dots, x_n) := \lim_{h \rightarrow 0} \frac{f(x_1, \dots, x_{i-1}, x_i + h, x_{i+1}, \dots, x_n) - f(x_1, \dots, x_n)}{h}$$

各 x_i について偏微分係数を計算して並べたベクトルを **勾配ベクトル** と呼ぶ

$$\nabla f(x_1, \dots, x_n) := \left(\frac{\partial f}{\partial x_1}(x_1, \dots, x_n), \dots, \frac{\partial f}{\partial x_n}(x_1, \dots, x_n) \right)$$

例) $f(x, y) = x^2 + xy$ の $(1, 2)$ における勾配ベクトルは

$$\begin{cases} \frac{\partial f}{\partial x} = 2x + y \\ \frac{\partial f}{\partial y} = x \end{cases} \Rightarrow \underline{\nabla f(1, 2) = (4, 1)}$$

✓ 勾配ベクトルの重要なポイント

勾配ベクトルは関数の値が最も大きくなる方向を指示する

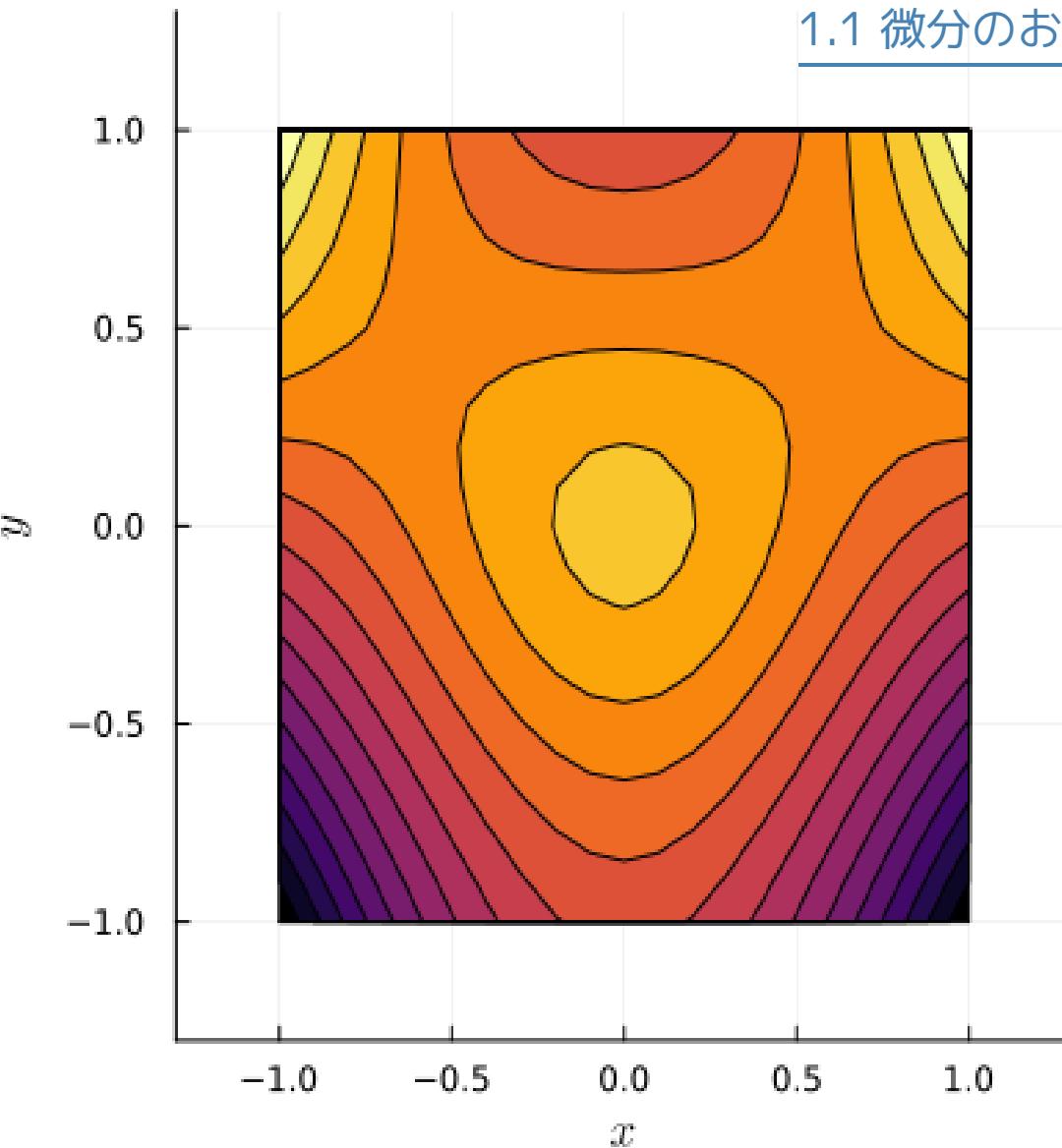
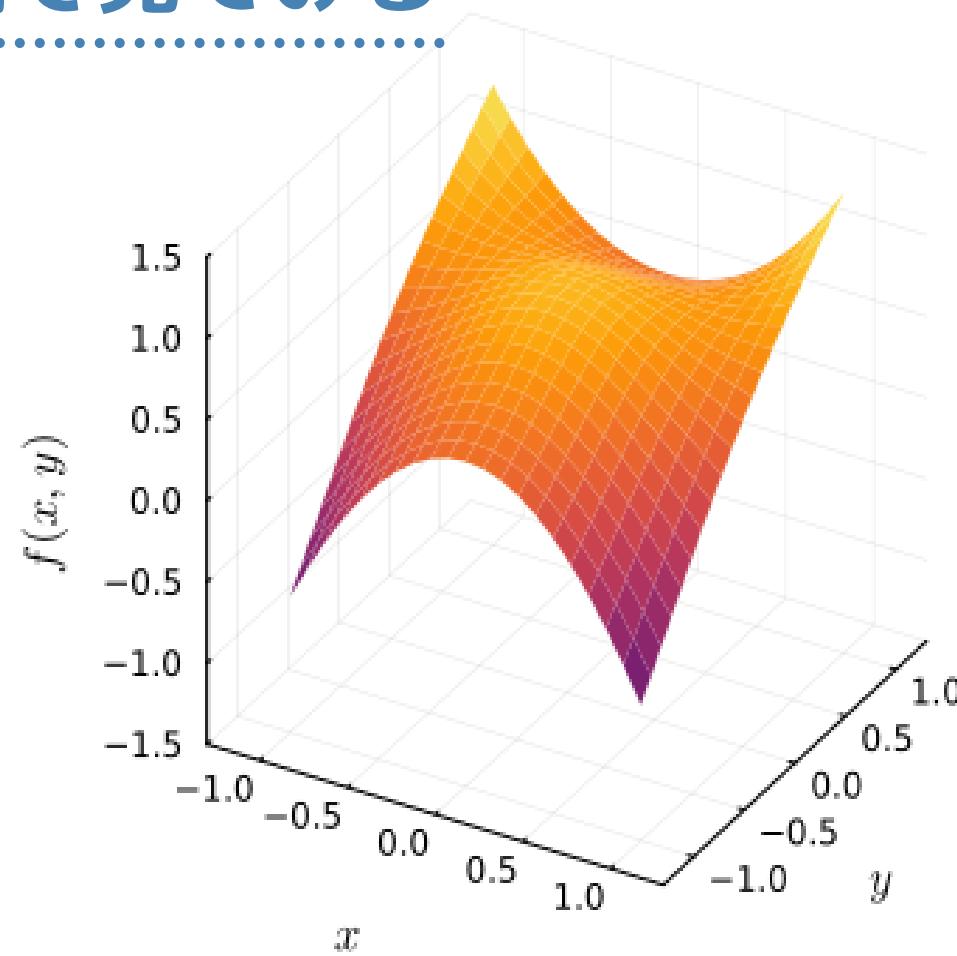
[定理1. 勾配ベクトルの性質]

$-\nabla f(\mathbf{x})$ は

$$g(\mathbf{v}) = \lim_{h \rightarrow 0} \frac{f(\mathbf{x}) - f(\mathbf{x} + h\mathbf{v})}{h}$$

の最大値を与える。

実例で見てみる



($f(x, y) = x^2y + \frac{1}{x^2+y^2+1}$ のプロット)

実例で見てみる

$$f(x, y) = x^2y + \frac{1}{x^2+y^2+1}$$

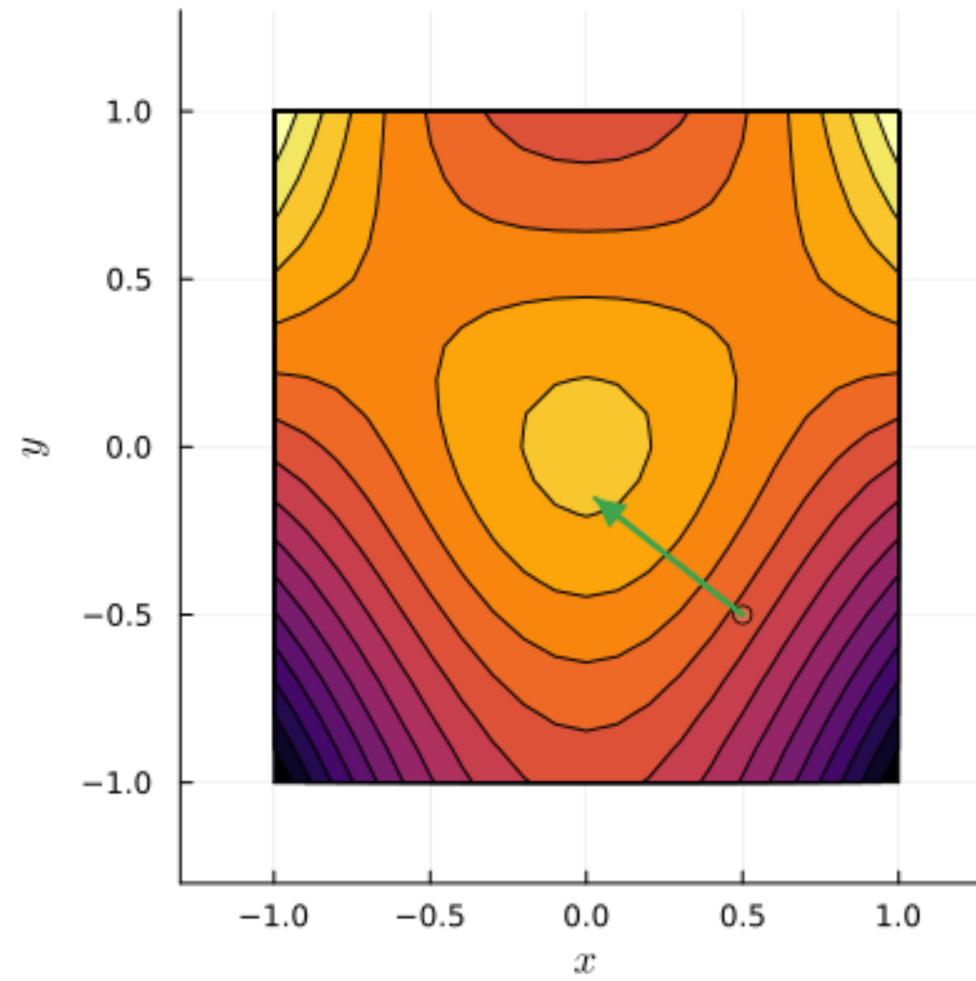
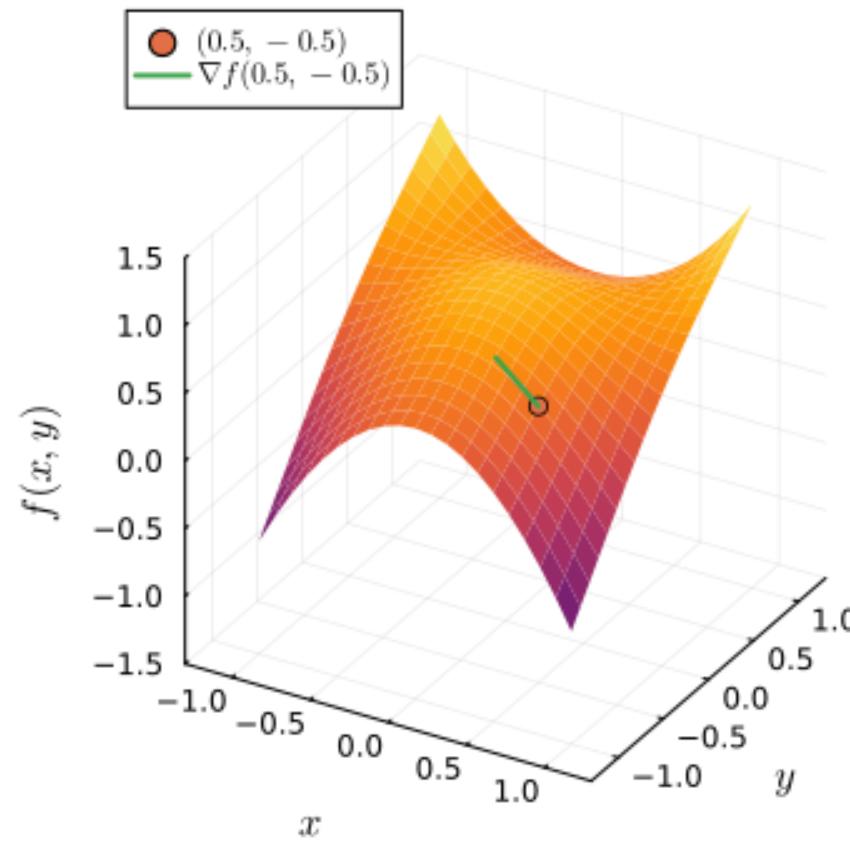
計算すると、

$$\begin{cases} \frac{\partial f}{\partial x} = 2xy - \frac{2x}{(x^2 + y^2 + 1)^2} \\ \frac{\partial f}{\partial y} = x^2 - \frac{2y}{(x^2 + y^2 + 1)^2} \end{cases}$$

なので

$$\nabla f(0.5, -0.5) = \left(\frac{-17}{18}, \frac{25}{36} \right) = (-0.94, 0.694)$$

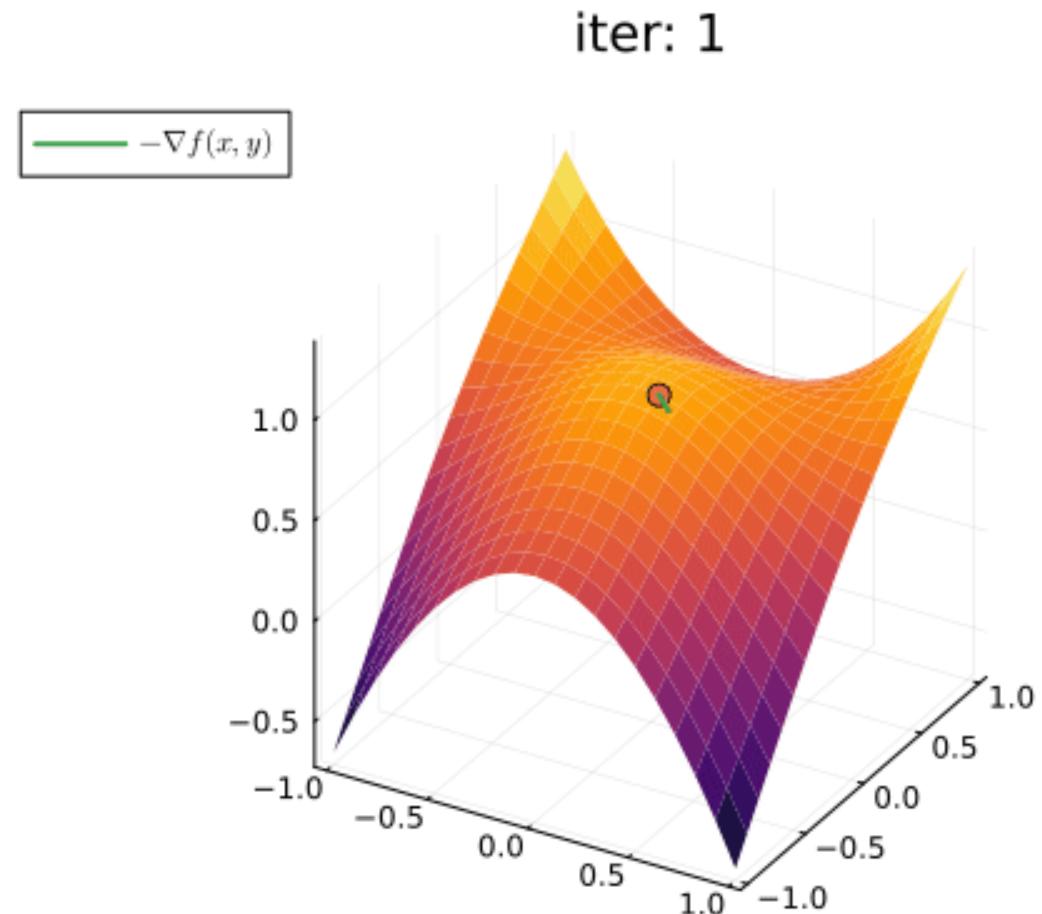
実例で見てみる



勾配降下法

✓ 勾配ベクトルは関数の値が大きくなる方向を指示する

⇒ $-\nabla f(x, y)$ の方向にちょっとづつ点を動かしていくけば関数のそこそこ小さい値を取る点を探しに行ける



[最急降下法]

1. $\mathbf{x}^{(0)}, \alpha$ を適当に決める
2. $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha \nabla f(\mathbf{x}^{(k)})$ として $\mathbf{x}^{(k+1)}$ を更新する
3. 収束したと思ったら $\mathbf{x}^{(k)}$ を出力して終了. そうでなければ 2. に戻る

✓ 一般の f について大域的な解を求められる保証はないが、
そこそこ小さい値を取る点を探しに行ける

勾配降下法を機械学習に応用する

機械学習の典型的な問題設定

学習データ $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ が与えられたとき、

損失関数

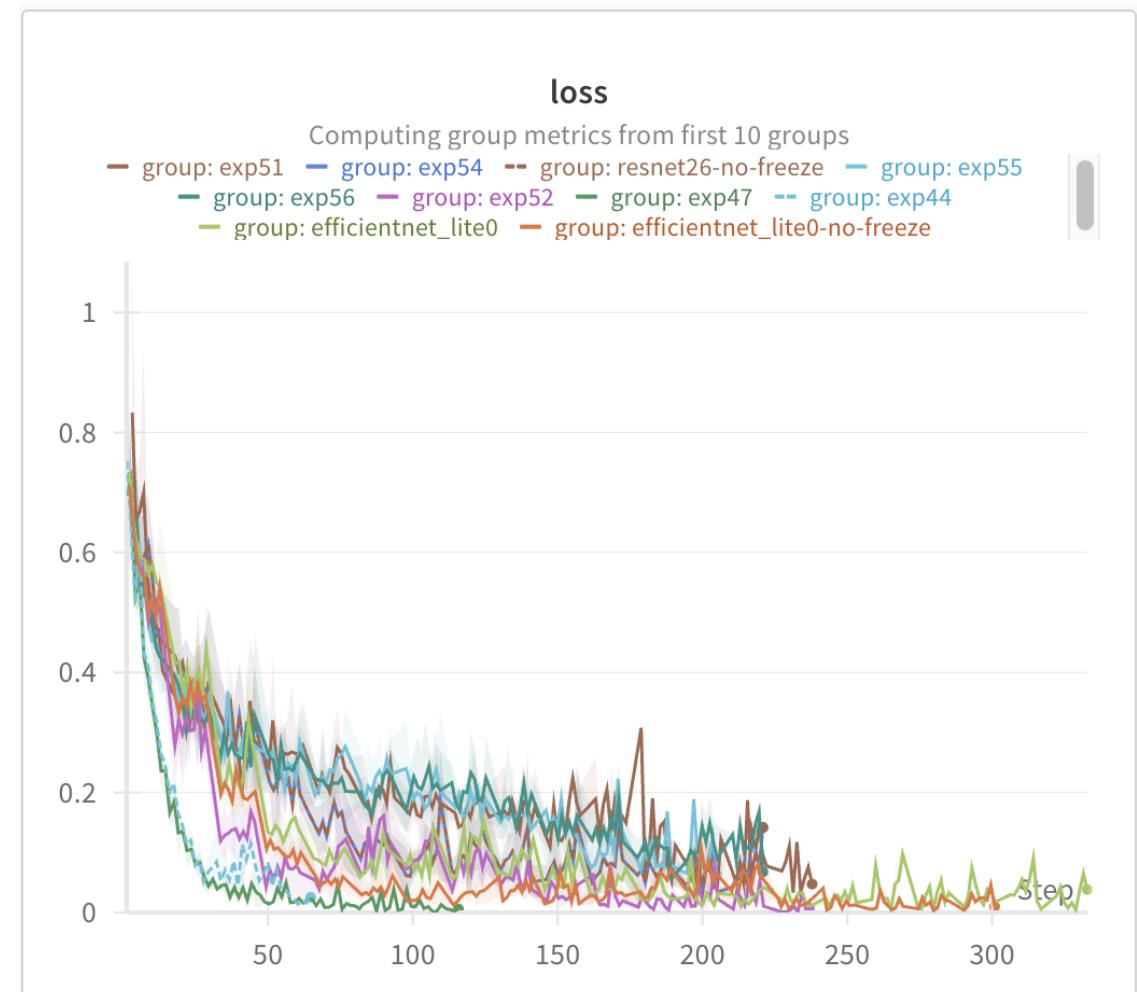
$$L(\mathbf{w}; \mathcal{D})$$

をなるべく小さくする \mathbf{w} を求めよ

勾配降下法と深層学習

勾配降下法を使った深層学習モデルのパラメータの最適化は、実際やってみると非常に上手くいく

⇒ 今この瞬間も世界中の計算機がせっせと勾配ベクトルを計算中



勾配降下法と深層学習（※ ギャグです）

1.3 勾配降下法と機械学習

- 2050年にはAI業務サーバの消費電力は 3000 TWh にのぼると予測されている [1]
- 日産リーフは 7.0 km/kWh で走るらしい

WOLFRAM言語とMATHEMATICAの開発元による

 WolframAlpha

3000 TWh \times 7.0 km/kWh

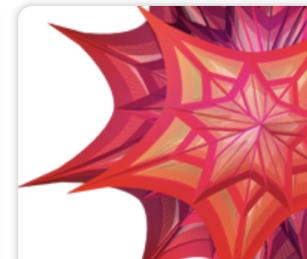
自然言語 数学入力 拡張キーボード 例を見る アップロード ランダムな例を使う

入力解釈
3000 TW h (テラワット時) \times 7. km/kWh (kilometers per kilowatt hours)

結果
21 兆 km (キロメートル)

ステップごとの解説

単位変換



[1] JST 低炭素社会戦略センター: 情報化社会の進展がエネルギー消費に与える影響 (Vol.2) –データセンター消費エネルギーの現状と将来予測および技術的課題
<https://www.jst.go.jp/lcs/pdf/fy2020-pp-03.pdf>

太陽～地球の距離は 1.5×10^8 km くらい。

$$(2.1 \times 10^{16}) / (1.5 \times 10^8) = 1.4 \times 10^5$$

⇒ 人類は、一年間で日産リーフを太陽に 140000 台送りこめる電力を勾配の計算に費やしている。

学習データ $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ が与えられたとき、

損失関数

$$L(\mathbf{w}; \mathcal{D})$$

をなるべく小さくする \mathbf{w} を求めよ

勾配降下法で解くには...

∇L を使って

\mathbf{w} を更新していくば良い

学習データ $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ が与えられたとき、

損失関数

$$L(\mathbf{w}; \mathcal{D})$$

をなるべく小さくする \mathbf{w} を求めよ

勾配降下法で解くには...

∇L を使って

\mathbf{w} を更新していくば良い

勾配の計算法を考える

$$\text{さっきは } f(x, y) = x^2y + \frac{1}{x^2+y^2+1}$$

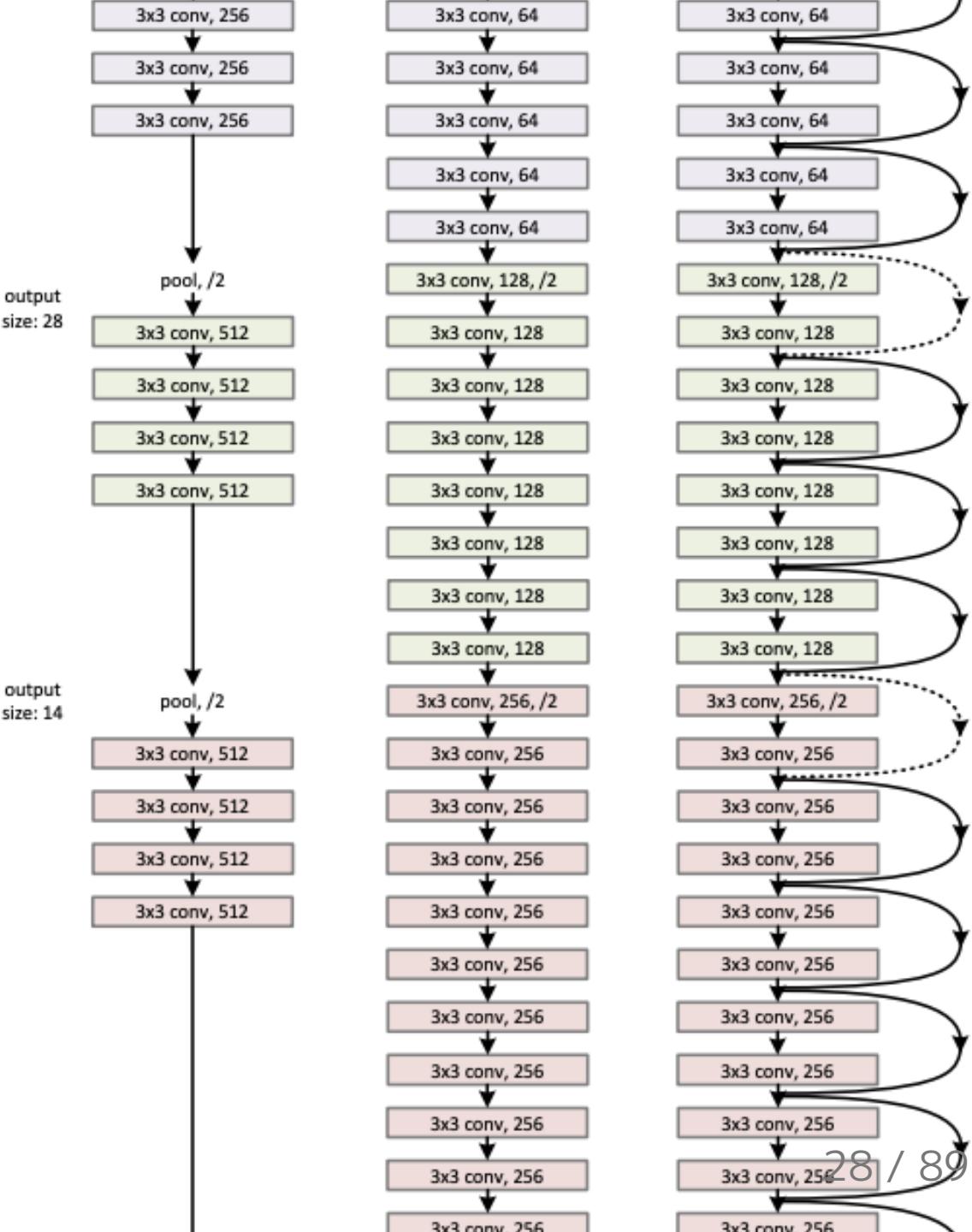
⇒ 頑張って手で ∇f を求められた

深層学習の複雑なモデル...

$$L(\mathbf{w}; \mathbf{x}, y) = \text{VeryComplicatedf}(\mathbf{w}; \mathbf{x}, y)$$



画像: He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. ArXiv. /abs/1512.03385



 < やりますよ



計算機に自動で微分させよう！

[2] 自動微分

2.1 微分の近似—数値微分

2.2 誤差なしの微分—数式微分

2.3 式の微分からアルゴリズムの微分へ

2.4 自動微分とトレース

2.5 自動微分とソースコード変換

2.1 微分の近似—数値微分

```
function numerical_derivative(f::Function, x::Number)::Number
    g = numerical_operation(f, x)
    return g
end
```

数値微分のアイデア

微分の定義

$$\lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

をそのまま近似する

数値微分の実装

```
function numerical_derivative(f, x; h=1e-8)
    g = (f(x+h) - f(x)) / h
    return g
end

f(x) = sin(x)
f'(x) = cos(x)
x = π / 3

numerical_derivative(f, x) # 0.4999999969612645
f'(x)                      # 0.5000000000000001
```

+ 数値微分のメリット

- ✓ 実装が極めて容易
- ✓ f がなんでも 計算可能.

数値微分のメリット ~ 実装が容易

これだけで完了

```
function numerical_derivative(f, x; h=1e-8)
    g = (f(x+h) - f(x)) / h
    return g
end

f(x) = sin(x)
f'(x) = cos(x)
x = π / 3

numerical_derivative(f, x) # 0.499999969612645
f'(x)                      # 0.5000000000000001
```

数値微分のメリット ~ 実装が容易

多変数関数への拡張 … i 番目を固定して繰り返し計算

```
function numerical_gradient(f, x::Vector; h=1e-8)
    n = length(x)
    g = zeros(n)
    y = f(x...)
    for i in 1:n
        x[i] += h
        g[i] = (f(x...) - y) / h
        x[i] -= h
    end
    return g
end
```



数値微分のメリット ~ なんでも計算可能

```
function numerical_derivative(f, x; h=1e-8)
    g = (f(x+h) - f(x)) / h
    return g
end
```

f がどんなに訳のわからない演算でも
 $f(x + h)$ さえ $f(x)$ が計算できればOK

数値微分のデメリット

1. 打ち切り誤差が生じる
2. 衍落ちも起こる
3. 計算コストが高い

数値微分の誤差 ~ 打ち切り誤差

本来は極限を取るのに小さい値で
誤魔化すので誤差が発生



実際どれくらいの誤差が発生する？

$$\lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

数値微分の誤差

[定理2. 数値微分の誤差]

$$f'(x) - \frac{f(x+h) - f(x)}{h} = O(h)$$

[証明]

テイラー展開すると、

$$\begin{aligned} f'(x) - \frac{1}{h}(f(x+h) - f(x)) &= f'(x) - \frac{1}{h}(f(x) + f'(x)h + O(h^2) - f(x)) \\ &= O(h) \end{aligned}$$

誤差の最小化

実験:

$O(h)$ なら、 h をどんどん小さくすればいくらでも精度が良くなるはず？

```
H = [0.1^i for i in 4:0.5:10]
E = similar(H)

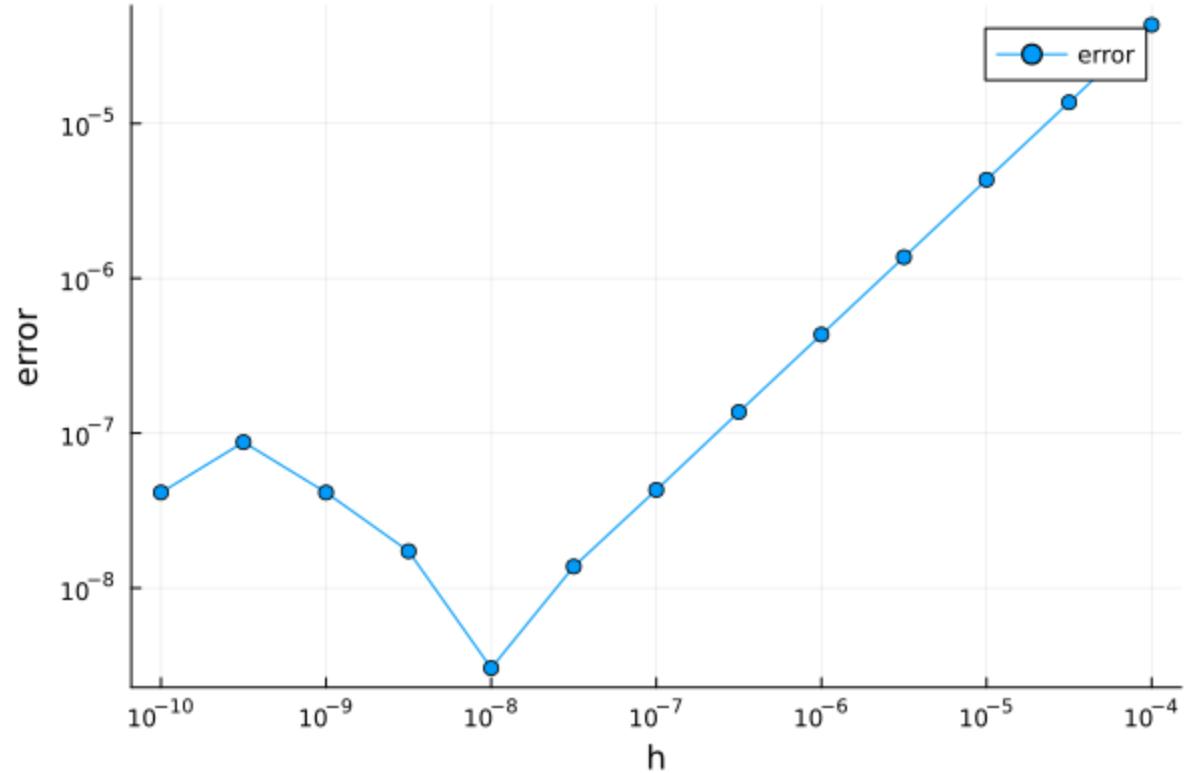
for i in eachindex(H)
    d = numerical_derivative(f, x, h=H[i])
    E[i] = abs(d - f'(x))
end

plot(H, E)
```

誤差の最小化

実際

$h < 10^{-8}$ くらいになるとむしろ精度が悪化する



丸め誤差と打ち切り誤差のトレードオフ

h が小さくなると、分子の引き算が非常に近い値の引き算になる

⇒ 衍落ちが発生し全体として悪化

$$\frac{f(x + h) - f(x)}{h}$$

数値微分の改良

誤差への対応

1. 打ち切り誤差 \Rightarrow 計算式の変更
2. 衍落ち $\Rightarrow h$ の調整？

数値微分の改良 ~ 打ち切り誤差の改善

1. 打ち切り誤差への対応

微分の(一般的な)定義をそのまま計算する方法:

$$f'(x) \approx \frac{f(x + h) - f(x)}{h}$$

は **前進差分** と呼ばれる

数値微分の改良 ~ 打ち切り誤差の改善

ところで、

$$\lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x - h)}{2h}$$



これを近似してみても良さそう？

中心差分による2次精度の数値微分

実はこれの方が精度がよい！

[定理3. 中心差分の誤差]

$$f'(x) - \frac{f(x+h) - f(x-h)}{2h} = O(h^2)$$

同じようにテイラー展開をするとわかる

また、簡単な計算で一般の n について誤差 $O(h^n)$ の近似式を得られる（下参照）

中心差分と同様に x から左右に $\frac{h}{2}$ ずつとってこの評価の重みつき和を考えてみます。

すると、テイラー展開の各項を足し合わせて $f'(x)$ 以外の係数を 0 にすることで公比が各列 $-\frac{n}{2}, -\frac{n-1}{2}, \dots, \frac{n}{2}$ で初項 1 のヴァンデルモンド行列を A として $Ax = e_2$ を満たす x を h で割ったのが求めたい重みとわかります。あとはこれの重み付き和をとればいいです。同様に k 階微分の近似式も得られます。

数値微分の改良 ~ 衍落ちへの対応

2. 衍落ちへの対応

Q. 打ち切り誤差と丸め誤差のトレードオフで h を小さくすればいいというものじゃないことはわかった。じゃあ、最適な h は見積もれる？

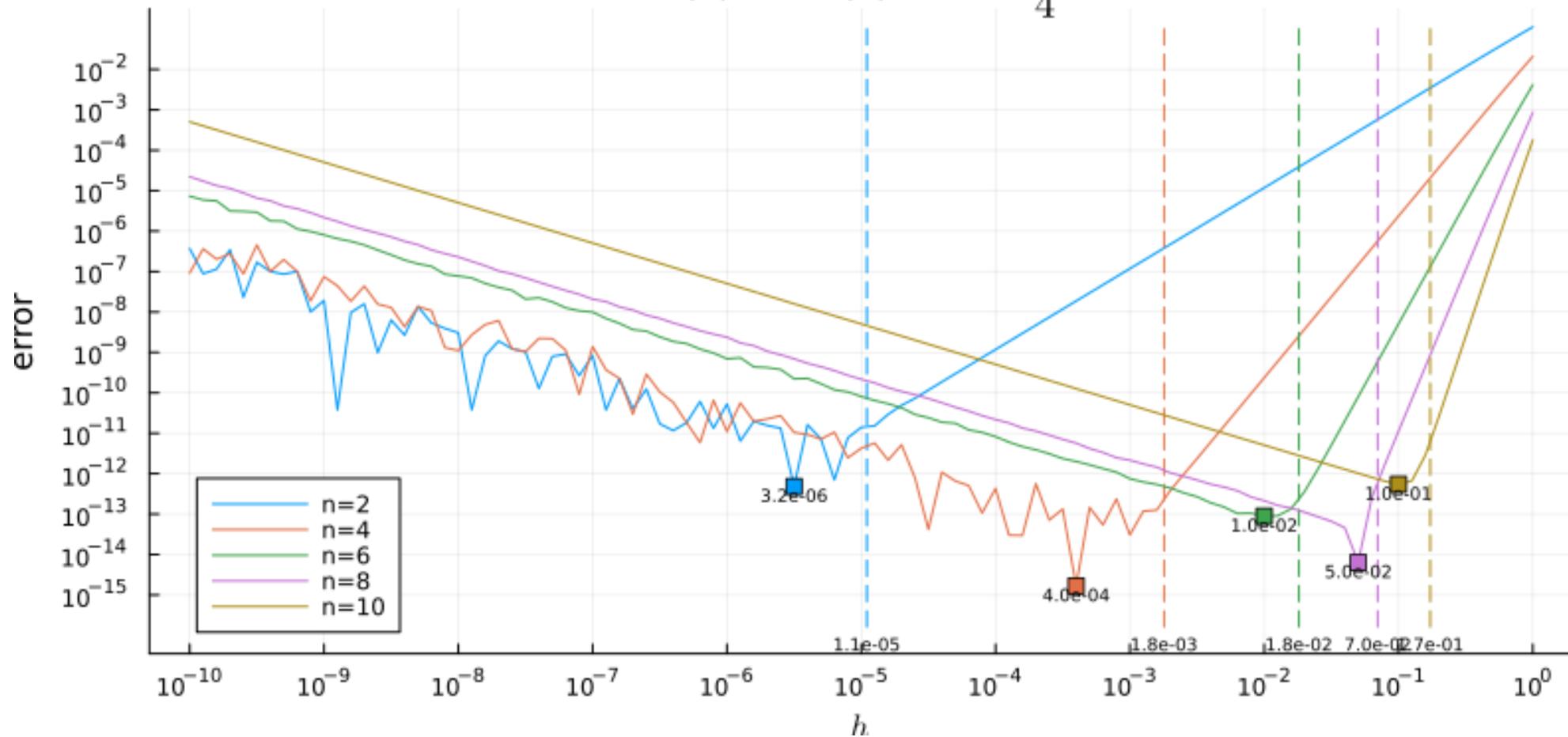
A. 最適な h は f の n 階微分の大きさに依存するから簡単ではない。

例) 中心差分 $\frac{f(x+h) - f(x-h)}{h}$ は $h_{best} \approx \sqrt[3]{\frac{3\sqrt{2}\varepsilon}{|f'''(x)|}}$ くらい？

⇒ $f'(x)$ がわからないのに $f'''(x)$ を使った式を使うのは現実的でない。

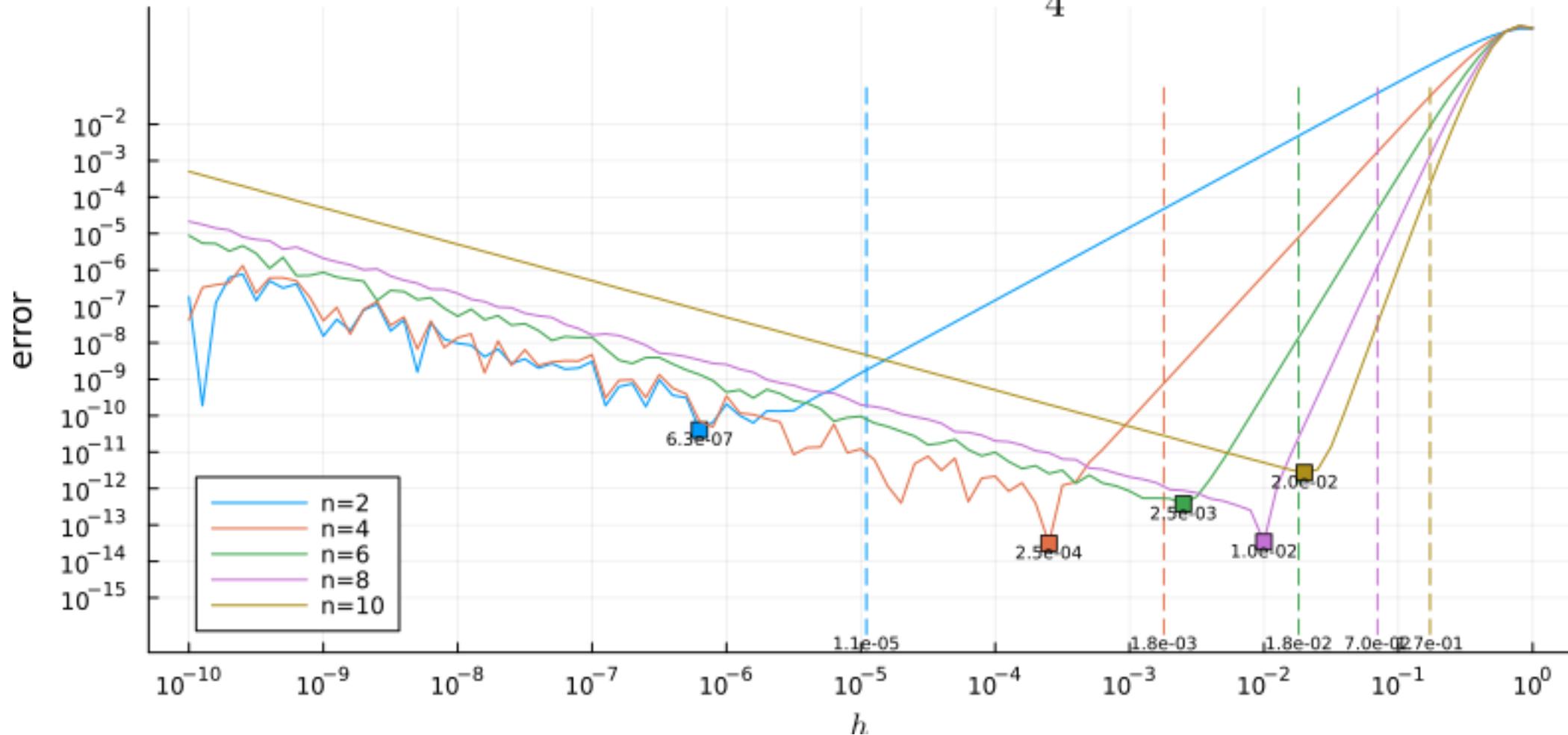
しううがないので $h = \left(\frac{(n+1)!}{\sqrt{n}f(x)}\varepsilon\right)^{\frac{1}{n+1}}$ に線を引いてみると…

$$f(x) = \sin(x), \quad x = \frac{\pi}{4}$$



結構いい感じ？

$$f(x) = \sin(5x), \quad x = \frac{\pi}{4}$$



デメリット3. 計算コストが高い

多変数関数への拡張

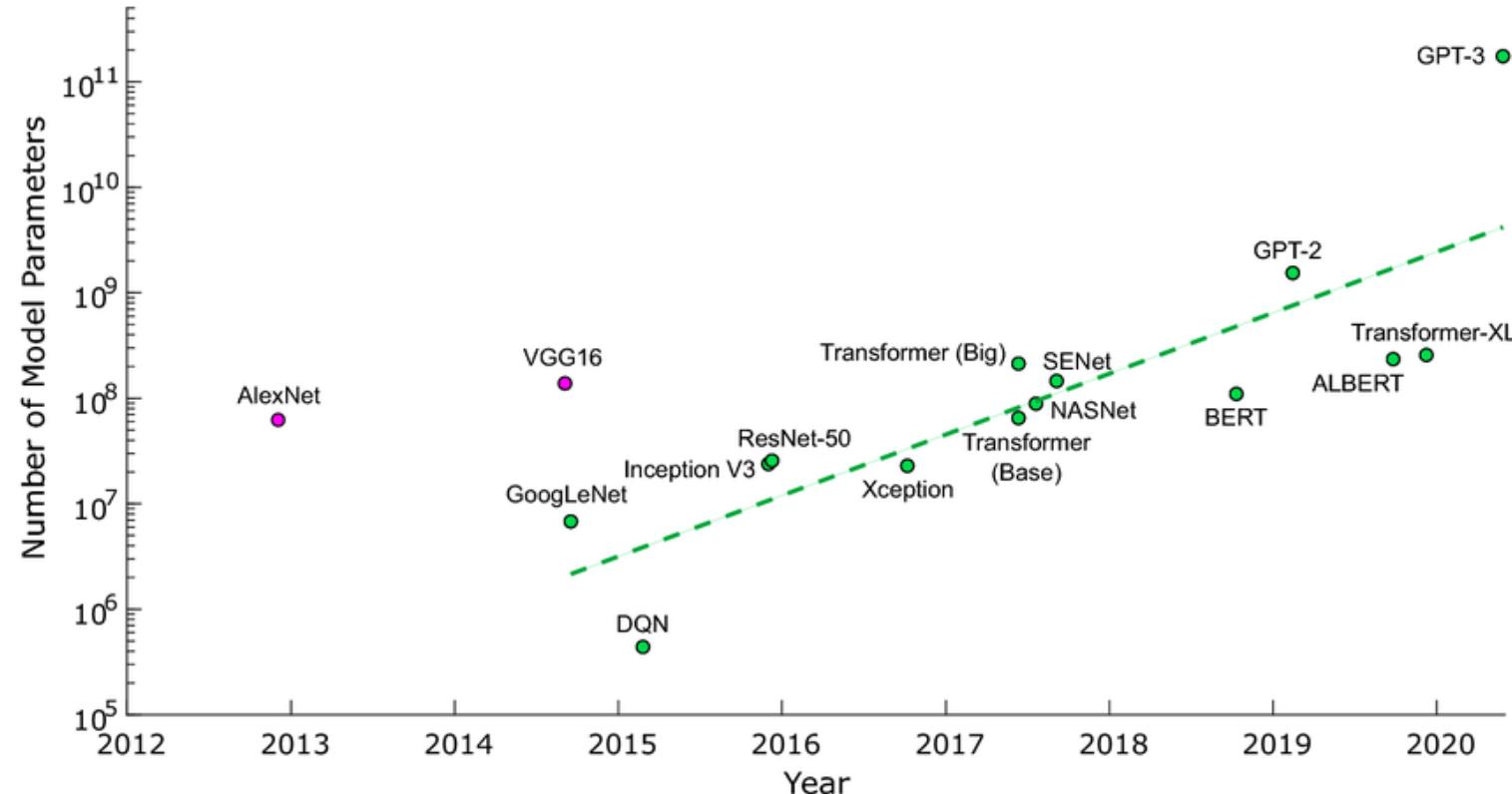
$f : \mathbb{R}^n \rightarrow \mathbb{R}$ の x における勾配ベクトル $\nabla f(x)$ を求める

```
function numerical_gradient(f, x::Vector; h=1e-8)
    n = length(x)
    g = zeros(n)
    y = f(x...)
    for i in 1:n
        x[i] += h
        g[i] = (f(x...) - y) / h
        x[i] -= h
    end
    return g
end
```

⇒ 関数を n 回評価する必要がある。

n 回評価は致命的

✓ 應用では f が重く, n が大きくなりがち $\Rightarrow n$ 回評価は高コスト

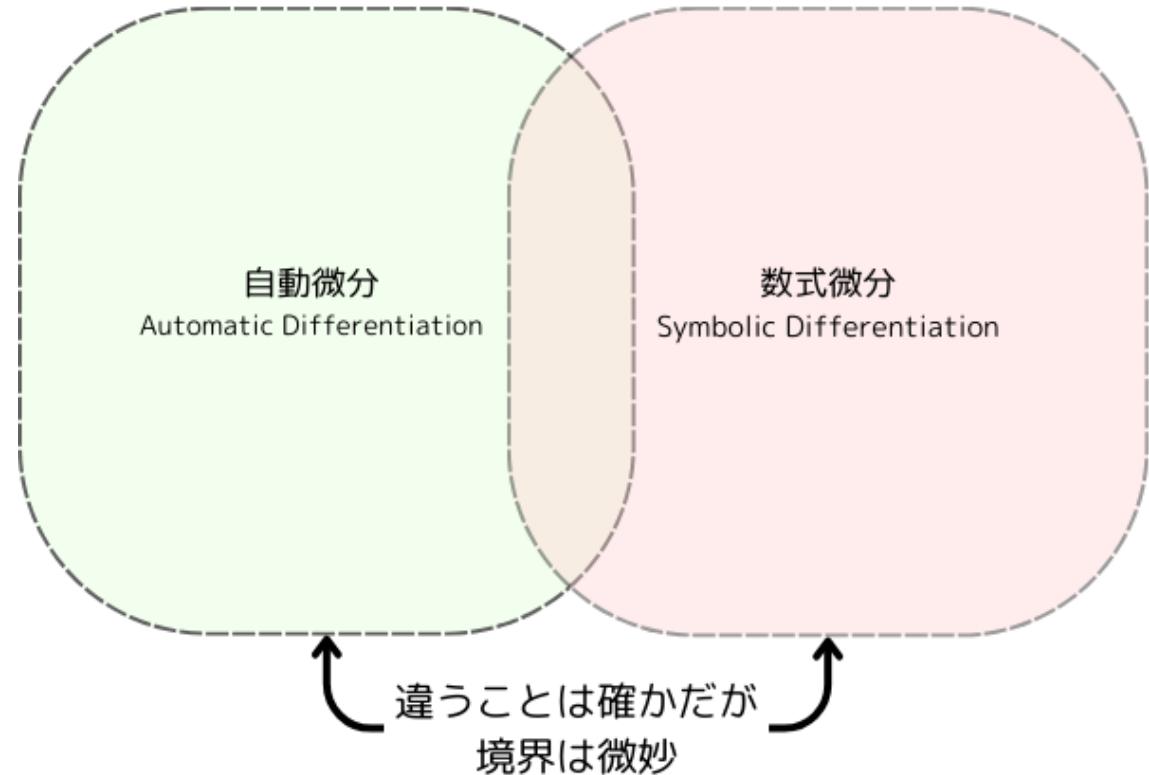


結論：実装の手軽さや f がなんでもいけるのは非常に有用！

一方、機械学習などで応用するのはきびしそう

数式微分と自動微分は違う？

各ツールに対して
数式微分 / 自動微分のどちらかを
きっちり分類するのは不毛ぎみ？



3 つの主要な手法

1. 数値微分

2. 数式微分

3. 自動微分

- 数式微分

```
function symbolic_derivative(f::Expr)::Expr
    df = symbolic_operation(f)
    return df
end
```

数式微分の方針

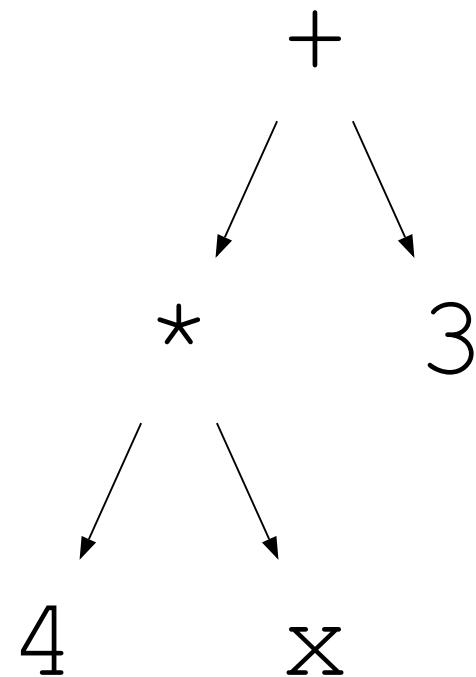
式 (Expr) を入力として、式 (Expr) を出力する

式・プログラムの表現

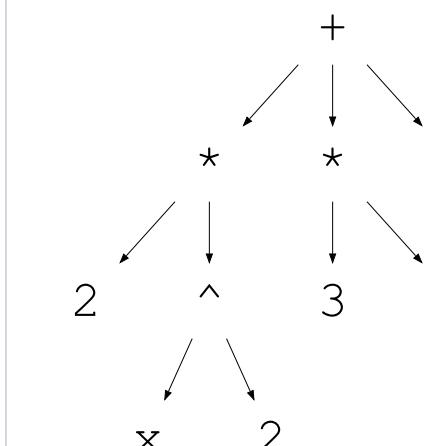
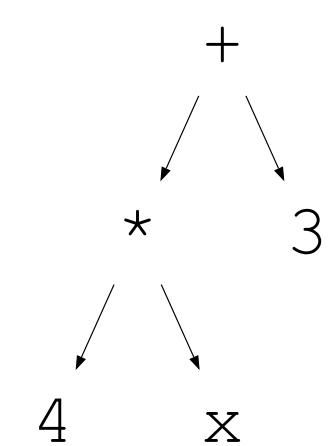
式, プログラムは木構造で表現できる

$f(x) = 4x + 3$ は ...

```
@to_graphviz 4x+3
```



✓ 木に操作を頑張って行うことでの
導関数を表す式を構築する

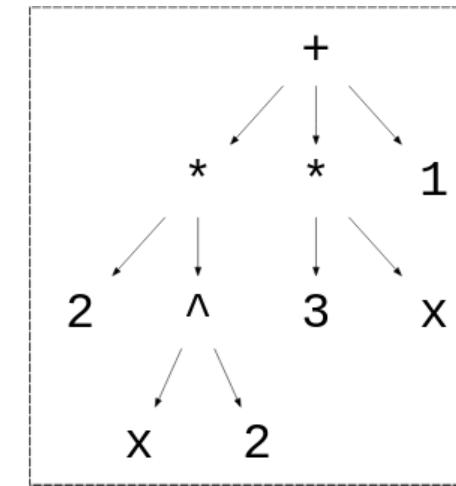
$f(x) = 2x^2 + 3x + 1$	$f'(x) = 2x + 3$
	

数式微分の目標

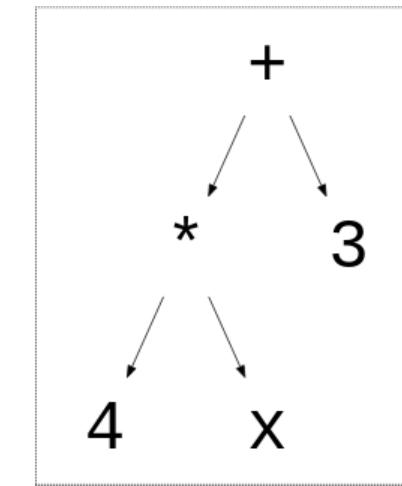
2.4 数式微分

$$f(x) = 2x^2 + 3x + 1$$

変換



操作



✓ Julia ならプログラムそのものをデータとして扱う機構が整っている！

- Expr 型 ... Julia のプログラムを表現する型

```
julia> ex = Meta.parse("3 + 4")
:(3 + 4)
```

```
julia> dump(ex)
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 3
    3: Int64 4
```

```
julia> eval(ex)
```

✓ Juliaなら、特別な準備なく（数学の意味での）式をそのまま扱って微分できる

数式微分の簡単な実装

足し算と掛け算に関するルールを実装してみる

$$1. \frac{d}{dx}(f(x) + g(x)) = f'(x) + g'(x)$$

$$2. \frac{d}{dx}(f(x) \cdot g(x)) = f'(x) \cdot g(x) + f(x) \cdot g'(x)$$

数式微分の簡単な実装

```

derivative(ex::Symbol) = 1      # dx/dx = 1
derivative(ex::Int64) = 0       # 定数の微分は 0

function derivative(ex::Expr)::Expr
    op = ex.args[1]
    if op == :+
        return Expr(:call, :+, derivative(ex.args[2]), derivative(ex.args[3]))
    elseif op == :*
        return Expr(
            :call,
            :+,
            Expr(:call, :*, ex.args[2], derivative(ex.args[3])),
            Expr(:call, :*, derivative(ex.args[2]), ex.args[3])
        )
    end
end

```

※ Juliaは `2 * x * x` のような式を、`(2 * x) * x` でなく `*(2, x, x)` として表現するのでこのような式については上は正しい結果を返しません。(スペース不足)
このあたりもちゃんとやる場合をいちおう掲載しておきます: [gist](#)

数式微分の簡単な実装

```
f = :(x * x + 3)
df = derivative(f) # :((x * 1 + 1x) + 0)
```

```
x = 2
eval(df) # 4
```

```
x = 10
eval(df) # 20
```



数式微分の注意点？



不用意な実装 だと、導関数の式が爆発してまう [1]

$$\frac{d}{dx}(f(x) \cdot g(x)) = f'(x) \cdot g(x) + f(x) \cdot g'(x)$$

... 項が二つに「分裂」するので、再帰的に微分していくと項が爆発的に増える

[1] よく数式微分の固有・不可避っぽい問題だ、みたいな文脈で語られるのですが、数値微分自体の問題ではないという指摘もあります。僕もそう思います。
参考: Laue, S. (2019). On the Equivalence of Automatic and Symbolic Differentiation. ArXiv. /abs/1904.02990

- ✓ Juliaなら、特別な準備なく（数学の意味での）式をそのまま扱って微分できる

- ✓ Juliaなら、特別な準備なく（数学の意味での）式をそのまま扱って微分できる

式の微分からアルゴリズムの微分へ

需要: 制御構文とともに許して柔軟な記述をしたい。

```
x = [1, 2, 3]
y = [2, 4, 6]

function linear_regression_error(coef)
    pred = x * coef
    error = 0
    for i in eachindex(y)
        error += (y[i] - pred[i])^2
    end
    return error
end
```

式の微分からアルゴリズムの微分へ

✓ 数値微分なら中身がなんでも結果さえあれば微分できた

```
function numerical_derivative(f, x::Number; h=1e-8)
    return (f(x + h) - f(x)) / h
end

coef = 1
numerical_derivative(linear_regression_error, coef) # -27.99999474559445

lr = 0.01
for i in 1:100
    coef -= lr * numerical_derivative(linear_regression_error, coef)
end
```

↔ 数式微分は、中身の式の構造を見る必要がある

式の微分からアルゴリズムの微分へ

AST を見てみると...

```
quote
    function linear_regression_error(coef)
        pred = x * coef
        error = 0
        for i in eachindex(y)
            error += (y[i] - pred[i])^2
        end
        return error
    end
end ▷ dump
```



```
Expr
head: Symbol block
args: Array{Any}((2,))
 1: LineNumberNode
  line: Int64 3
  file: Symbol In[42]
 2: Expr
head: Symbol function
args: Array{Any}((2,))
 1: Expr
  head: Symbol call
  args: Array{Any}((2,))
    1: Symbol linear_regression_error
    2: Symbol coef
 2: Expr
  head: Symbol block
  args: Array{Any}((9,))
    1: LineNumberNode
    line: Int64 3
    file: Symbol In[42]
    2: LineNumberNode
    line: Int64 4
    file: Symbol In[42]
    3: Expr
    head: Symbol =
    args: Array{Any}((2,))
      1: Symbol pred
      2: Expr
    4: LineNumberNode
    line: Int64 5
    file: Symbol In[42]
    5: Expr
    head: Symbol =
    args: Array{Any}((2,))
      1: Symbol error
      2: Int64 0
    6: LineNumberNode
    line: Int64 6
    file: Symbol In[42]
    7: Expr
    head: Symbol for
    args: Array{Any}((2,))
      1: Expr
      2: Expr
    8: LineNumberNode
    line: Int64 9
    file: Symbol In[42]
    9: Expr
    head: Symbol return
    args: Array{Any}((1,))
      1: Symbol error
```

- ✓ 再代入、ループ、条件分岐など含んだプログラムに対して、直接変換を頑張ることで「陽に書かれた形の」導関数のプログラムを得るのは難しいタスク

- 自動微分

```
function automatic_derivative(f::Function, x::Number)::Number
    g = automatic_operation(f, x)
    return g
end
```

自動微分の方針

数値微分と同様、

(関数, 値) から直接、値を出力する

自動微分の特徴

- ✓ (入力 \downarrow 出力) が高次元でも高速に計算できる！

自動微分の特徴

数値微分 … $f : \mathbb{R}^n \rightarrow \mathbb{R}$ の微分は n 回の f の評価が必要



自動微分 なら 1 回 + 1 回で可能！

※ 単に2回と書かなかった理由はこの後のスライドを参照してください

自動微分の特徴

- ✓ 応用で非常に有用で広く使われる

PyTorch TensorFlow

Chainer

Zygote

[自動微分のアイデア]

連鎖律(Chaine Rule)...

$$\frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx}$$

を使う

自動微分のアイデア

$f(x) = 2x^2 + 3x + 1$ を例に考える

これを分解する...

$$y_1 = x$$

$$y_2 = y_1^2$$

$$y_3 = 2y_2$$

$$y_4 = 3y_1$$

$$y_5 = y_3 + y_4$$

$$y_6 = y_5 + 1$$

としたら、 $f(x) = y_6$ とする

自動微分のアイデア

すると...

$$\frac{dy_6}{dx} = \underbrace{\frac{dy_6}{dy_5}}_{\text{ }} \cdot \frac{dy_5}{dx}$$

これは行けるはず！

自動微分のアイデア

$$y_6 = y_5 + 1$$

⇒ これは $f(x) = x + 1$ の微分ができるなら極めて容易に微分できる

同様に、全ての微分を展開していくと…

$$\frac{\partial y_6}{\partial x} = \frac{\partial y_6}{\partial y_5} \left(\frac{\partial y_5}{\partial y_3} \frac{\partial y_3}{\partial y_2} \frac{\partial y_2}{\partial y_1} + \frac{\partial y_5}{\partial y_4} \frac{\partial y_4}{\partial y_1} \right)$$

自動微分のアイデア

$$y_1 = x$$

$$y_2 = y_1^2$$

$$y_3 = 2y_2$$

$$y_4 = 3y_1$$

$$y_5 = y_3 + y_4$$

$$y_6 = y_5 + 1$$

... $\frac{\partial y_i}{\partial y_j}$ ($i \geq j$) は簡単な計算！

自動微分のアイデア

$$\frac{\partial y_6}{\partial x} = \frac{\partial y_6}{\partial y_5} \left(\frac{\partial y_5}{\partial y_3} \frac{\partial y_3}{\partial y_2} \frac{\partial y_2}{\partial y_1} + \frac{\partial y_5}{\partial y_4} \frac{\partial y_4}{\partial y_1} \right)$$

$$= 1 \cdot (1 \cdot 2 \cdot 2x + 1 \cdot 3)$$

$$= 4x + 3$$



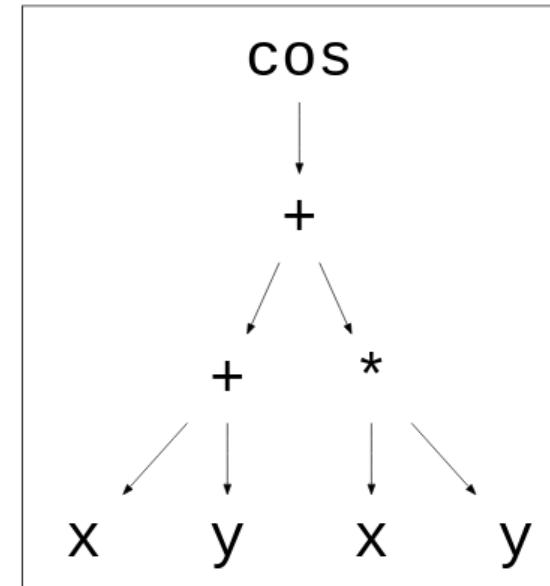
全体が計算できた！

自動微分の基本的な構造

```
function f(x, y)
    t1 = x + y
    t2 = x * y
    return cos(t1 + t2)
end

x = 3
y = 2
z = f(x, y)
```

入手
→



計算
→

$$\nabla f(3, 2)$$

計算グラフ

自動微分の用語

Prefix たち

- Forward = Bottom Up = BU = Tangent Linear = 前進型
- Backward = Reverse = Top Down = TD = Adjoint = 後退型 = 高速自動微分

自動微分の勉強で参考になる文献

.....

1. 久保田光一, 伊里正夫 「アルゴリズムの自動微分と応用」 コロナ社 (1998)
 - i. 自動微分そのものについて扱ったおそらく唯一の和書です。 詳しいです。
 - ii. 形式的な定義から、計算グラフの縮小のアルゴリズムや実装例など実用まで触れられています。
 - iii. サンプルコードは、FORTRAN(😊), C++ でやはり全体的に古めの実装という感じです。
2. 斎藤康毅 「ゼロから作るDeep Learning ③」 O'Reilly Japan (2020)
 - i. トレースベースの Reverse AD を Python で実装します。
 - ii. Step by step で丁寧に進んでいくので、とてもおすすめです。
 - iii. 自動微分自体について扱った本ではないため、その辺りの説明は若干手薄かもしれません。
3. Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2015). Automatic differentiation in machine learning: A survey. ArXiv. /abs/1502.05767
 - i. 機械学習 x AD のサービスですが、機械学習に限らず AD の歴史やトピックを広く取り上げています。
 - ii. 少し内容が古くなっているかもしれません。
4. [Differentiation for Hackers](#)
 - i. Flux.jl や Zygote.jl の開発をしている Mike J Innes さんが書いた自動微分の解説です。 Juliaで動かしながら勉強できます。おすすめです。
5. Innes, M. (2018). Don't Unroll Adjoint: Differentiating SSA-Form Programs. ArXiv. /abs/1810.07951
 - i. Zygote.jl の論文です。かなりわかりやすいです。
6. [Zygote.jl のドキュメントの用語集](#)
 - i. 自動微分は必要になった応用の人人がやったり、コンパイラの人人がやったり、数学の人人がやったりで用語が乱立しまくっているのでこちらを参照して整理すると良いです
 - ii. 僕の知る限り、(若干のニュアンスがあるかもしれません) Reverse AD の別表現として以下があります。
Backward Mode AD = Reverse Mode AD = Fast Differentiation = Algoroihmic Differentiation = Adjoint Differentiation + その訳語たち, 微妙な表記揺れたち
7. [JuliaDiff](#)
 - i. Julia での微分についてまとまっています。ここにあるパッケージのどれかを用途に応じて選ぶのが良いです。
8. [Chainer のソースコード](#)
 - i. Chainer は Python 製の深層学習フレームワークですが、既存の巨大フレームワークと比較すると、裏も Python でとても読みやすいです。
 - ii. 気になる実装があつたら当たるのがおすすめです。また議論もいろいろと残っているのでそれを巡回するだけでとても勉強になります。