

$$(X, O) = e^{-\frac{x^2}{2\sigma^2}}$$

$$x(X, O) = -\frac{x}{\sigma^2} G(X, O) = -\frac{x}{\sigma^2} e^{-\frac{x^2}{2\sigma^2}}$$

$$xx(X, O) = \frac{x^2 - \sigma^2}{\sigma^4} G(X, O) = \frac{x^2 - \sigma^2}{\sigma^4} e^{-\frac{x^2}{2\sigma^2}}$$

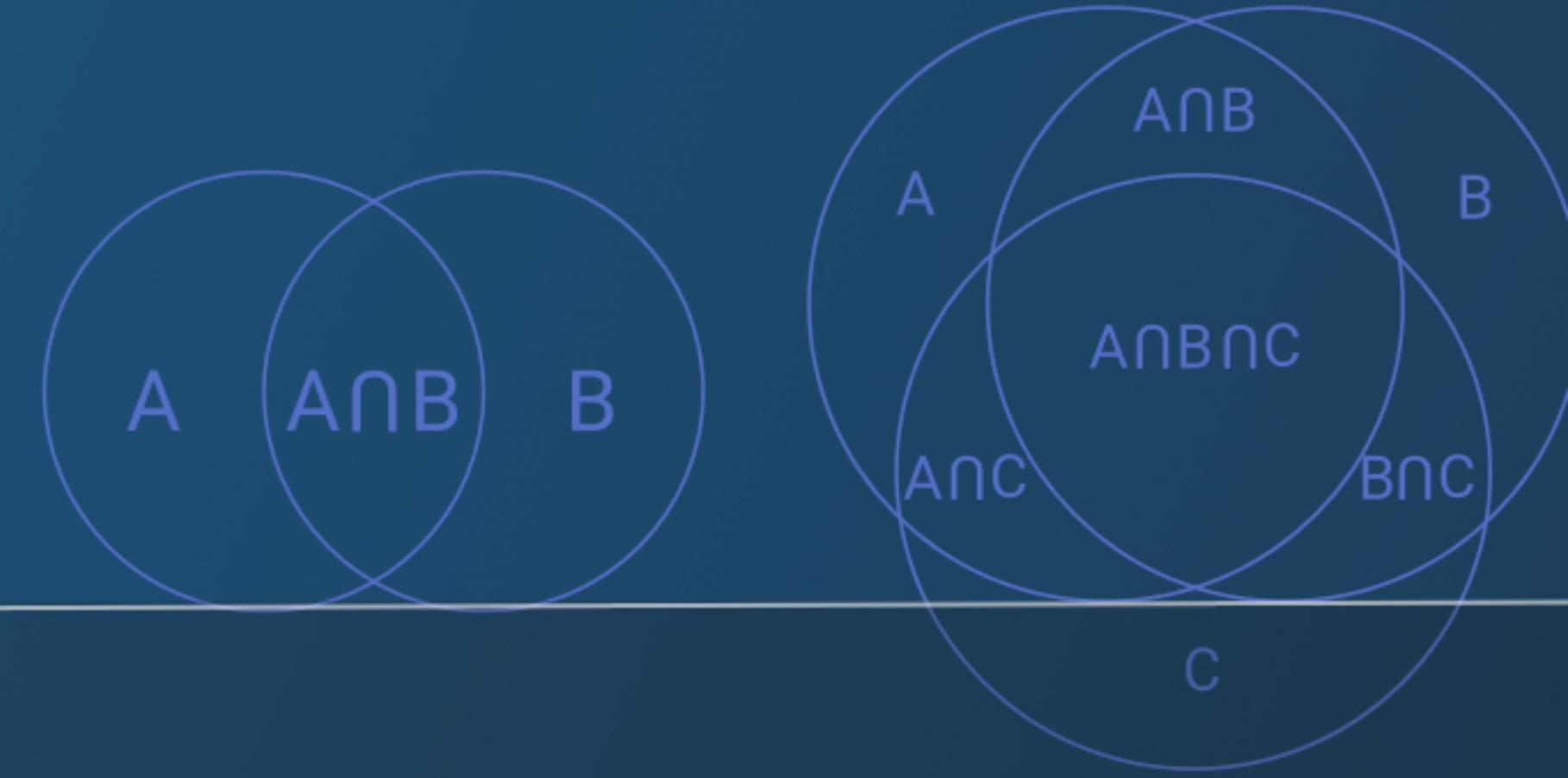
$$xxx(X, O) = -\frac{x^3 - x\sigma^2}{\sigma^6} G(X, O) = -\frac{x^3 - x\sigma^2}{\sigma^6} e^{-\frac{x^2}{2\sigma^2}}$$

Julia 程式語言學習馬拉松

Day 31



cupay 陪跑專家 : Andy Tu



可微分程式設計





重要知識點



- 了解如何使用 Zygote 計算函數微分
- 了解如何使用 Zygote 計算自定義資料結構及其運算的微分



Zygote.jl



- 深度學習的核心就是 backpropagation，而深度學習框架的核心就是去計算函數的梯度/微分及 computational graph。
- 自動微分是讓電腦計算微分式的一種有效率的方式，它避免了讓程式設計師將微分式寫死在程式當中，可以依據使用者自訂的函數自動地推論出微分式。有計算上的效率及準確性。
- Zygote.jl 是在 Julia 語言中相當知名的自動微分套件，它將自動微分的特性嵌入語言，讓 Julia 成為了支援 differentiable programming 的語言。



為什麼我們需要自動微分？



- 在深度學習及機器學習領域中，函數的微分被大量使用。主要用以解函數的最佳化問題。
- 不過往往一個問題被設計成特定形式，也需要將微分形式固定即可。在深度學習中，有多樣的函數被研究人員使用，其對應的微分式就需要隨著函數變化。
- 將微分式寫死是不明智的決定，我們需要讓電腦自動推論微分式。



簡單函數的微分



- 我們就來使用 Zygote 所提供的可微分程式設計。
- 我們先來嘗試簡單的多項式函數。
- 我們先定義相對應的函式，例如： $f(x) = x^2 + 2x + 1$

```
julia> using Zygote  
julia> f(x) = x^2 + 2x + 1  
julia> f(5)
```



簡單函數的微分



- Zygote 提供最基礎的梯度計算是使用 gradient，第一個參數要給定要微分的函數，第二則是給定要代入的資料點。
- gradient 會回傳一個數組，第一個值就是微分值。
- 或是以更像數學的方式寫，可以使用 $f'(5)$ ，即可得到微分值。

```
julia> gradient(f, 5)
(12,)
julia> f'(5)
12
```



三角函數微分

- 嘗試任意三角函數的微分。
- 可以注意到我們並沒有事先給定任何微分規則。
- \sin 的微分為 \cos ，代入 π (180 度)，取得的值為 -1。

```
julia> g(x) = sin(x)
julia> g'(π)
-1.0
```



指數及 log 函數微分



```
julia> gradient(log, 2)
(0.5,)
julia> gradient(exp, 2)
(7.38905609893065,)
```



複雜函數微分

- 嘗試相對複雜的函數。

```
julia> f(x) = log(1 + exp(5*sin(x) + 5x + 2))
julia> f'(1)
4
julia> g(a, b) = a*b
julia> gradient(g, 3, 4)
(4, 3)
```



自定義函式微分



- 我們來嘗試自定義的函式微分，當中含有遞迴的結構。

```
julia> function foo(x, n)
    if n == 0
        return x
    end
    return foo(x*n, n-1)
end
julia> gradient(x->foo(x, 4), 5)
(24,)
```



語言支援的微分系統



- Zygote 套件的一個相當重要的特性就是自動微分是嵌入到語言當中的語言特性，這代表微分是語言的第一級支援功能。
- Zygote 是個 source-to-source 的程式碼翻譯器，他會將使用者撰寫的程式碼進一步轉換成低階的 LLVM 中間碼，經由 LLVM 相關函式庫的優化，可以推論出精簡的函數微分形式，再進一步進行編譯。



將微分式轉為 LLVM 中間碼



- 我們來看看函數轉換成 LLVM 中間碼的樣子。

```
julia> f(x) = 5*sin(x)

julia> @code_llvm f'(\pi)

define double @"julia_#30_17697"() {
top:
    ret double -5.000000e+00
}
```



對資料結構中的函數微分



- 即使是在字典當中的函式一樣能夠處理。

```
julia> d = Dict(:a=>sin, :b=>cos)
julia> gradient(d[:a], 5)
(0.28366218546322625,)
```



對不同資料結構微分



- 我們常常自定義一些資料結構，如果有在資料結構上定義運算一樣能夠支援。

```
import Base: +, -  
struct Point  
    x::Float64  
    y::Float64  
end  
a::Point + b::Point = Point(a.x + b.x, a.y  
+ b.y)
```



對不同資料結構微分



- 需要注意的是，計算梯度的函數的回傳值必定要是個純量，不可以是其他資料結構或是陣列，不然會有錯誤。
- 所以為了避免這樣的問題，在示範中取出了計算後的 Point 的 x 值，其不影響梯度的計算。

```
julia> a = Point(1, 2)
julia> b = Point(3, 4)
julia> gradient(x->(x+b).x, a)
((x = 1.0, y = nothing),)
```

知識點 回顧

- 了解如何使用 Zygote 計算函數微分
- 了解如何使用 Zygote 計算自定義資料結構及其運算的微分



推薦閱讀

- [Zygote.jl 官方文件](#)





解題時間

請跳出 PDF 至官網 Sample Code
& 作業開始解題