DiffEqFlux.jl：当微分方程遇见神经网络和GPU

马英博

自我介绍

数值微分方程:JuliaDiffEq

稠密数值线性代数:RecursiveFactorization.jl

高性能计算:JuliaBLAS.jl和AsmMacro.jl

DiffEqFlux.jl

DifferentialEquations.jl + Flux.jl = O(1)空间复杂度的神经微分方程反向传播 + GPU 支持 + 刚性/非刚性微分方程求解器

什么是微分方程

微分方程描述变化率和量之间的关系

$$u(t)'=f(u(t),p,t)$$

状态

参数

（不关于时间）

时间

洛特卡－沃尔泰拉方程

$$\frac{dx}{dt} = x(\alpha - \beta y)$$

$$\frac{dy}{dt} = -y(\gamma - \delta x)$$

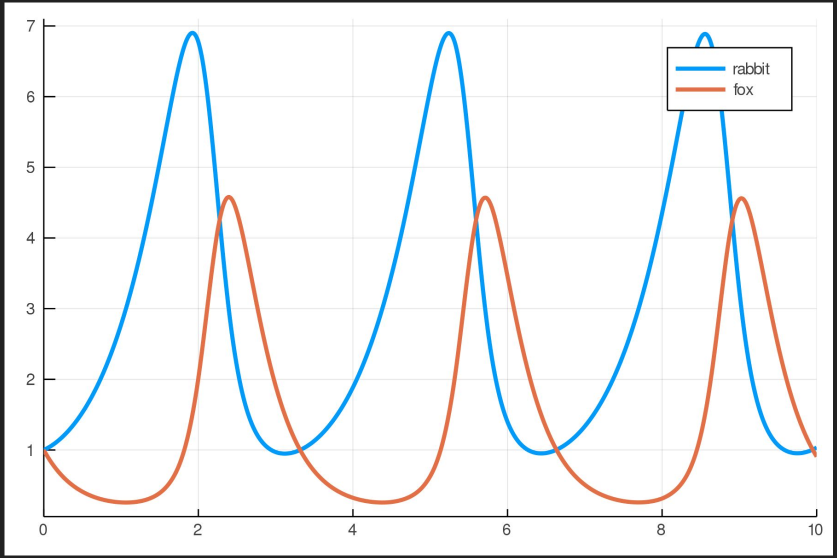
- y 是掠食者（如狼）的数量；
- x 是猎物（如兔子）的数量；
- dy/dt 与 dx/dt 表示上述两族群相互对抗的时间之变化；
- t 表示时间；
- α, β, γ 与 δ 表示与两物种互动有关的系数，皆为正实数。

写成Julia代码

```julia
using ParameterizedFunctions, OrdinaryDiffEq

lotka_volterra = @ode_def begin
  d🐇 = α*🐇 - β*🐇*🦊
  d🦊 = -δ*🦊 + γ*🐇*🦊
end α β δ γ

u0 = [1.0, 1.0]; p = (1.5,1.0,3.0,1.0); tspan = (0, 10.0)
sol = solve(lotka_volterra, u0, tspan, p)
```



GPU+微分方程

```julia
using OrdinaryDiffEq, CuArrays
u0 = rand(Float32, 100000); tspan = (0.0f0, 100.0f0);

f(du, u, p, t) = @. du = sin(cos(u)) - exp(u)
cpu_prob = ODEProblem(f, u0, tspan)
gpu_prob = ODEProblem(f, cu(u0), tspan)
```

把在CPU的数组放到GPU上

```julia
julia> @time solve(cpu_prob, Tsit5());
  0.608194 seconds (711 allocations: 113.710 MiB, 0.70% gc time)

julia> @time solve(gpu_prob, Tsit5());
  0.118043 seconds (34.89 k allocations: 1.485 MiB)
```

残差神经网络 vs 常微分方程

残差神经网络 --- 学习变化量比学习整个系统更简单

$$u_{t+1} = u_t + nn(u_t, p)$$

欧拉法解常微分方程

$$u_{t+1} = u_t + \Delta t \cdot f(u_t, p)$$

连续版本的残差神经网络就是常微分方程

u(t)'=神经网络(u(t),p,t)

把神经网络放到微分方程里面

Flux神经网络

```
dudt = Chain(Dense(2,50,tanh),
             Dense(50,50,tanh),
             Dense(50,2))
```

把常微分方程的u'=f(u,p,t)换成神经网络

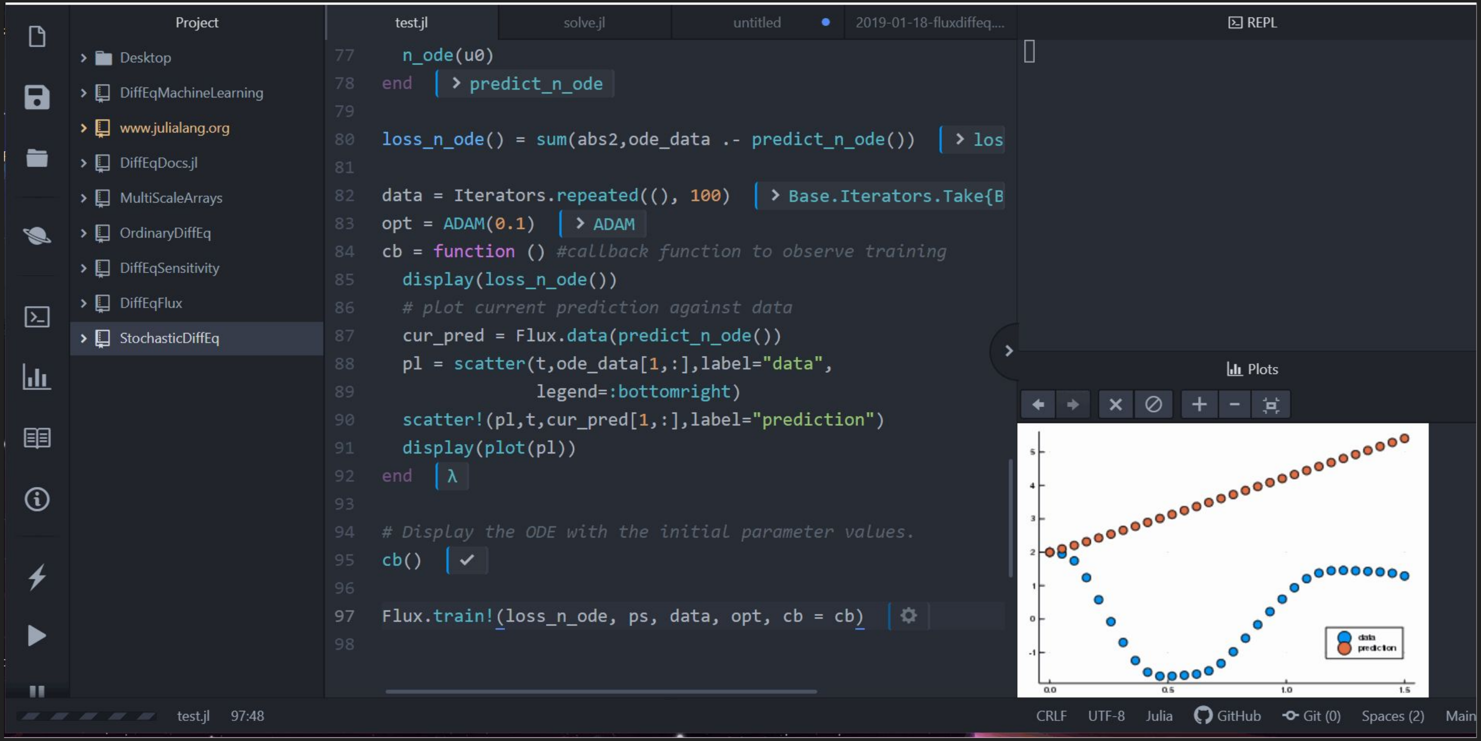
```
x->neural_ode(gpu(dudt),gpu(x),tspan,Tsit5(),saveat=0.1)
```

每隔0.1个单位时间保存一次输出

在GPU上做运算

Tsit5算法

把微分方程放到神经网络里面



梯度计算

对模型做微分（传统方法）

对程序做微分（可微分编程∂P）

对模型做微分

顾名思义，对模型做微分就是把描述模型的公式微分。

```
 d/dx √x

= d/dx x^(1/2)

= 1/2 * x^(-1/2)

= 1/(2*√x)
```

```
julia> deriv1(x) = 1/(2*√x)
deriv1 (generic function with 1
method)

julia> deriv1(4)
0.25
```

对程序做微分 可微分编程

```julia
function mysqrt(x, tol=5eps(float(x)))
    tol = abs(tol)
    guess = one(x)
    displacement = 2tol
    while displacement > tol
        newguess = 1/2*(guess + x/guess)
        displacement = abs(newguess - guess)
        guess = newguess
    end
    return guess
end
```

ForwardDiff.jl并不知道mysqrt所想最终表示的意思。它把整个程序做了微分。

```julia
julia> mysqrt(4)
2.0

julia> using ForwardDiff: derivative

julia> derivative(mysqrt, 4)
0.2500002594080783

julia> derivative(x->mysqrt(x, 1e-16), 4)
0.25
```

下午Mike会讲更多可微分编程

微分方程的敏感度分析

对程序进行微分（机器学习领域的常见做法）➡️ 自动微分

对逼近的微分

对模型进行微分（传统做法）➡️ 伴随敏感性分析

对微分的逼近

微分方程的敏感性分析 正向自动微分 对逼近的微分

```julia
using ForwardDiff
ForwardDiff.gradient([1,2,3,4.0]) do p
    dual_u0 = convert.(eltype(p), u0)
    prob = ODEProblem(lotka_volterra,
                      dual_u0, tspan, p)
    sol = solve(prob, Tsit5())
    sum(Array(sol)) # cost function
end
```

注意微分方程的初始条件要
和参数的类型一样

时间复杂度是O(N)

空间复杂度是O(1)

微分方程的敏感性分析 逆向自动微分 对逼近的微分

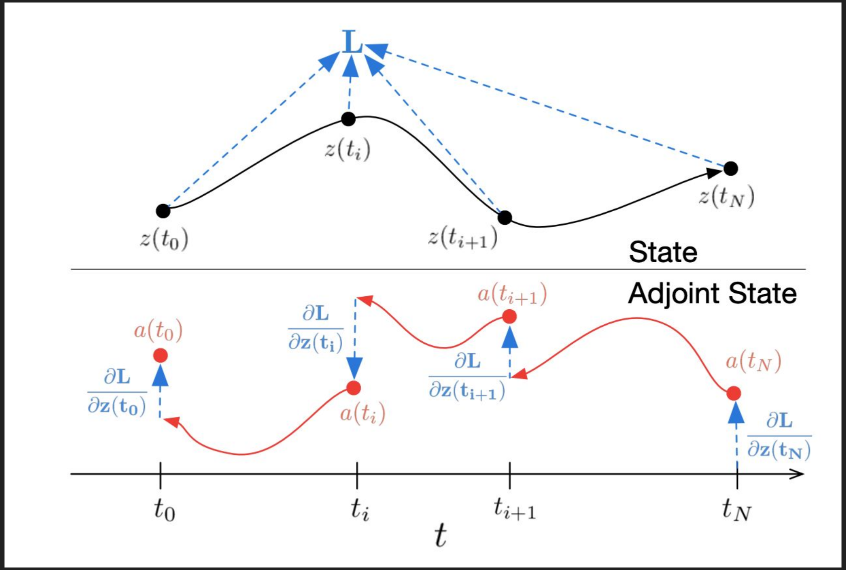
```
julia> using Flux

julia> Flux.Tracker.gradient([1,2,3,4.0]) do p
           dual_u0 = convert.(eltype(p), u0)
           prob = ODEProblem(lotka_volterra, dual_u0,
                             tspan, p)
           sol = solve(prob, Tsit5())
           sum(Array(sol)) # cost function
       end
([13.454529896823164, -10.291900686697643, 4.251656188686566,
-4.074835024391901] (tracked),)
```

时间复杂度是O(1)

空间复杂度是O(N)

微分方程的敏感性分析 伴随敏感性分析 对微分的逼近



Neural Ordinary Differential Equations (arXiv:1806.07366)

时间复杂度是O(1)

空间复杂度是O(1)

为了节省内存，假设常微分方程系统可逆，所以不把每一步都保存。

数学上正确做法是使用检查点（checkpointing scheme）。

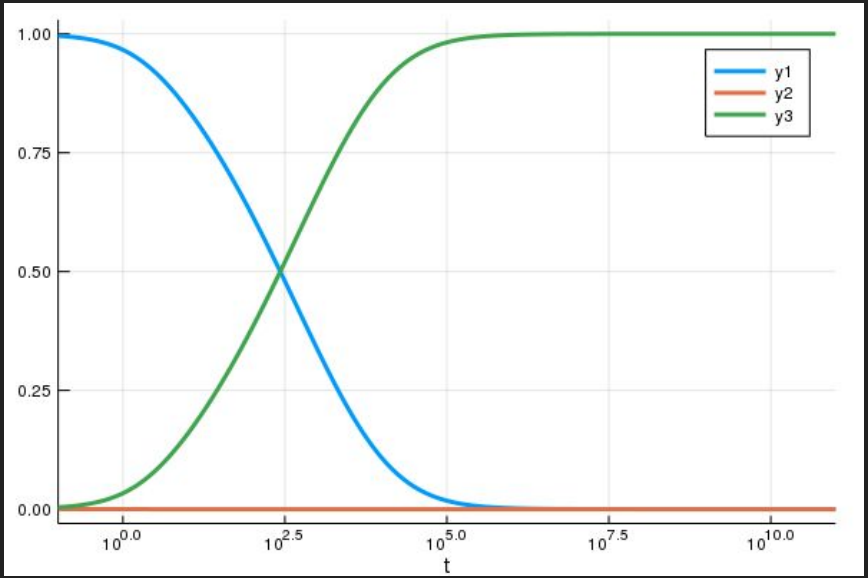
JuliaDiffEq的ODE求解器

有二百多种纯Julia的算法，其中优化的算法到目前为止有68种。包括

- 显式龙格—库塔法（ERK）
- 全隐式龙格—库塔法（FIRK）
- 单对角隐式龙格—库塔法（SDIRK）
- Rosenbrock法
- 并行外推算法
- 线性多步法
- 指数算法（Exponential integrator）
- 辛空间算法
- ...

刚性问题

```
rober = @ode_def Rober begin
  dy₁ = -k₁*y₁+k₃*y₂*y₃
  dy₂ =  k₁*y₁-k₂*y₂^2-k₃*y₂*y₃
  dy₃ =  k₂*y₂^2
end k₁ k₂ k₃
prob =
ODEProblem(rober,[1.0;0.0;0.0],(0.0,1e11),
(0.04,3e7,1e4))
```



不止ODE！(Demo)

JuliaDiffEq还提供除了ODE以外的求解器

- 偏微分方程（PDE）
- 随机微分方程（SDE）
- 随机偏微分方程（SPDE）
- 时滞微分方程（DDE）
- 时滞偏微分方程（DPDE）
- 微分代数方程（DAE）
- 偏微分代数方程（PDAE）

更多信息

Julia神经微分方程博客 https://julialang.org/blog/2019/04/fluxdiffeq-zh_tw

JuliaDiffEq的手册 http://docs.juliadiffeq.org/latest/

加入Julia Slack的 #diffeq-bridged 频道

感谢

Chris Rackauckas (@ChrisRackauckas)

David Widmann (@devmotion)

Mike Innes (@MikeInnes)

Jesse Bettencourt (@jessebett)

所有其他贡献者