DiffEqFlux.jl 介紹

Outline

- Introduction to DiffEqFlux.jl
- Ordinal differential equations
- ODE in Flux framework
- Demo Neural ODE

Introduction to DiffEqFlux.jl

DiffEqFlux.jl 是由 DifferentialEquations.jl 的作者 Chris Rackauckas 以及 Flux.jl 的作者 Mike Innes 聯手合作的作品。

All the following materials comes from [this blog (https://julialang.org/blog/2019/01/fluxdiffeq)](https://julialang.org/blog/2019/01/fluxdiffeq).

Ordinal differential equations

In [ ]:
```
using DifferentialEquations
```

In [ ]:
```
function lotka_volterra(du,u,p,t)
  x, y = u
  α, β, δ, γ = p
  du[1] = dx = α*x - β*x*y
  du[2] = dy = -δ*y + γ*x*y
end
```

In [ ]:
```
u0 = [1.0,1.0]
tspan = (0.0, 10.0)
p = [1.5,1.0,3.0,1.0];
```

In [ ]:
```
prob = ODEProblem(lotka_volterra, u0, tspan, p)
sol = solve(prob)
```

In [ ]:
```
using Plots
```

In [ ]:
```
plot(sol)
```

make u0 and tspans to be functions of p

In [ ]:
```
u0_f(p, t0) = [p[2], p[4]]
tspan_f(p) = (0.0, 10*p[4])
p = [1.5, 1.0, 3.0, 1.0]
prob = ODEProblem(lotka_volterra, u0_f, tspan_f, p)
```

ODE in Flux framework

Solving problem by Flux

In [ ]:
```
using Flux, DiffEqFlux
```

In [ ]:
```
p = [1.5, 1.0, 3.0, 1.0]
prob = ODEProblem(lotka_volterra, u0, tspan, p)
```

In [ ]:
```
diffeq_rd(p, prob, Tsit5(), saveat=0.1)
```

Use ODE in Flux

Initial Parameters

In [ ]:
```
p = param([2.2, 1.0, 2.0, 0.4])
params = Flux.Params([p])
```

Wrap problem as 1-layer network

In [ ]:
```
predict_rd() = diffeq_rd(p, prob, Tsit5(), saveat=0.1)[1, :]
```

Loss function

In [ ]:
```
sin_data = [sin(2x) for x = 1:101];
```

In [ ]:
```
loss_rd() = sum(abs2, predict_rd() .- sin_data)
```

Prepare dummy data

```
In [ ]: data = Iterators.repeated((), 100)
```

Optimizer

```
In [ ]: opt = ADAM(0.1)
```

Callback function

```
In [ ]: function cb()
            display(loss_rd())
            # using `remake` to re-create our `prob` with current parameters `p`
            display(plot(solve(remake(prob, p=Flux.data(p)), Tsit5(), saveat=0.1), ylim=(0, 6)))
        end
```

```
In [ ]: cb()
```

Training

```
In [ ]: Flux.train!(loss_rd, params, data, opt, cb=cb)
```

Flux layer

```
m = Chain(
  Dense(28^2, 32, relu),
  Dense(32, 10),
  softmax)
```

Stick ODE layer into MLP

```
m = Chain(
  Dense(28^2, 32, relu),
  # this would require an ODE of 32 parameters
  p -> diffeq_rd(p, prob, Tsit5(), saveat=0.1)[1, :],
  Dense(32, 10),
  softmax)
```

Stick ODE layer into CNN

```
m = Chain(
  Conv((2,2), 1=>16, relu),
  x -> maxpool(x, (2,2)),
  Conv((2,2), 16=>8, relu),
  x -> maxpool(x, (2,2)),
  x -> reshape(x, :, size(x, 4)),
  x -> diffeq_rd(p, prob, Tsit5(), saveat=0.1, u0=x)[1, :],
  Dense(288, 10), softmax) |> gpu
```

Neural ODE layer

Generate some data

In [ ]:
```
function trueODEfunc(du, u, p, t)
    true_A = [-0.1 2.0; -2.0 -0.1]
    du .= ((u.^3)'true_A)'
end
```

In [ ]:
```
u0 = Float32[2.; 0.]
datasize = 30
tspan = (0.0f0, 1.5f0)
```

In [ ]:
```
t = range(tspan[1], tspan[2], length=datasize)
prob = ODEProblem(trueODEfunc, u0, tspan)
ode_data = Array(solve(prob, Tsit5(), saveat=t))
```

Approximate derivative `dudt` with neural ODE layer

In [ ]:
```
dudt = Chain(x -> x.^3,
             Dense(2, 50, tanh),
             Dense(50, 2))

n_ode(x) = neural_ode(dudt, x, tspan, Tsit5(), saveat=t, reltol=1e-7,
abstol=1e-9)
ps = Flux.params(dudt)
```

In [ ]:
```
pred = n_ode(u0) # Get the prediction using the correct initial condit
ion
scatter(t, ode_data[1, :], label="data")
scatter!(t, Flux.data(pred[1, :]), label="prediction")
```

Model and loss function

In [ ]:
```
predict_n_ode() = n_ode(u0)
loss_n_ode() = sum(abs2, ode_data .- predict_n_ode())
```

Optimizer and callback

In [ ]:
```
data = Iterators.repeated((), 1000)
opt = ADAM(0.1)
function cb2()
    display(loss_n_ode())
    # plot current prediction against data
    cur_pred = Flux.data(predict_n_ode())
    pl = scatter(t, ode_data[1, :], label="data")
    scatter!(pl, t, cur_pred[1, :], label="prediction")
    display(plot(pl))
end
```

In [ ]:
```
cb2()
```

In [ ]:
```
Flux.train!(loss_n_ode, ps, data, opt, cb=cb2)
```

For example, if your data is unevenly spaced at time points t, just pass in saveat=t and the ODE solver takes care of it.

DiffEqFlux API (https://github.com/JuliaDiffEq/DiffEqFlux.jl#api-documentation)

Thank you for attention