Julia Tutorial on AI for Mathematical Biology

Pengfei Song

2024-02-06

Table of contents

Welcome **4**

1 Julia Basics **5**
1.1 Why Julia? . . . 5
1.2 Installation: Julia+VSCode . . . 5
1.2.1 Recommended . . . 5
1.2.2 Alternatives . . . 6
1.2.3 Editor . . . 6
1.3 Packages and Environment Management . . . 6
1.4 Other Julia Courses and Materials . . . 7

2 Neural Networks **8**
2.1 IMPORTANT: Activate Julia environment first . . . 8
2.2 Multilayer neural network as a function . . . 8
2.3 IMPORTANT: destructure . . . 9
2.4 Using DNN to learn $\sin(x)$ . . . 9
2.4.1 Question: why this DNN learn the $\sin(x)$ data but fails to predict $\sin(x)$? How to improve the performance? . . . 11
2.5 Optimization . . . 11

3 Differential Equations **13**
3.0.1 Solving SIR Model . . . 13
3.1 IMPORTANT: Activate Julia environment first . . . 13
3.1.1 Learn Julia in AI era . . . 13
3.1.2 Data visulization . . . 14
3.2 Solving SIR model with neural networks embedded . . . 15

4 Couple of differential equations and deep learning **17**
4.1 IMPORTANT: Activate Julia environment first . . . 17
4.2 Loading packages and setting random seeds . . . 17
4.3 Generating test data from logistic model . . . 18
4.4 Define neural ODE . . . 19
4.5 Define Loss functions and Callbacks . . . 20
4.6 Train the DNN embedded in differential equations . . . 20
4.7 Output and data visulization . . . 21
4.7.1 Question: modify the codes, change `Lux.jl` back to `Flux.jl`? . . . 23

5 Symbolic Regression **24**
5.1 IMPORTANT: Activate Julia environment first 25
5.2 Project: predicting the peak time 26

Welcome

1 Julia Basics

1.1 Why Julia?

Slow language is not suitable for this task — learning neural networks embedded in differential equations.

In short - Solving two language problems: `C++` is fast but difficult; `Python` is easy but slow. `Julia` is fast and easy. - Language for Mathematics: writing `Julia` is just like writing mathematics. - Similar syntax as `Matlab`; Simple as `Python`; Fast as `C++`.

More advantages and disadvantages can be seen in Why Julia? · Julia for Optimization and Learning

1.2 Installation: Julia+VSCode

`Julia` + `VScode`

1.2.1 Recommended

We recommend to install Julia via `juliaup`. We are using the latest, *stable* version of Julia (which at the time of this writing is `v1.10`). Once you have installed `juliaup` you can get any Julia version you want via:

```
juliaup add $JULIA_VERSION

# or more concretely:
juliaup add 1.10

# but please, just use the latest, stable version
```

Now you should be able to start Julia and be greeted with the following:

```
$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.10.0 (2023-12-25)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia>
```

1.2.2 Alternatives

Julia can also be installed from the official website download page. The appropriate version is the **64-bits** version for the Windows operating system in most cases. In case of difficulties, see platform-specific instructions.

1.2.3 Editor

There is no one way to install/develop and run Julia, which may be strange for users coming from MATLAB, but for users of general purpose languages such as Python, C++ this is quite common. Most of the Julia programmers to date are using

- Visual Studio Code,
- and the corresponding Julia extension.

This setup is described in a comprehensive step-by-step guide in *Julia for Optimization & Learning*.

For other editors, we refer to Julia IDE

1.3 Packages and Environment Management

`Julia` manages packages and environments like `Rust`. Very convenient! Strongly recommendation: go through this document Working with Environment · Pkg.jl quickly.

To set up the same environment as me, you can follow the guides in Song921012/Julia_Tutorial_on_AI4MathBic

1.4 Other Julia Courses and Materials

- Official documentation
- Slack channel: Community
- Important:Cheatsheet for differences between Julia and Matlab and Python
- Cheatsheet of basic functions
- Cheatsheet of advanced functions
- Think Julia: How to Think Like a Computer Scientist
- From Zero to Julia!
- Recommended: Julia for Optimization and Learning
- Scientific Programming in Julia
- Julia Data Science
- Statistics with Julia: Fundamentals for Data Science, Machine Learning and Artificial Intelligence

2 Neural Networks

Neural network is no mystery, it is just a function with powerful learning ability.

At this time, you don't need to care much details about its complicated structure.

In Julia, two deep learning packages are commonly used: - `Flux.jl`:Welcome · Flux - `Lux.jl`:LuxDL Docs

Here, we use `Flux.jl` as an example. More details can be seen in packages official documents.

2.1 IMPORTANT: Activate Julia environment first

```
using Pkg
Pkg.activate(".")
```

```
  Activating project at `~/Desktop/MyProjects/Julia_Tutorial_on_AI4MathBiology`
```

2.2 Multilayer neural network as a function

You only need to regard neural network as a function $f(x,p)$, x input, p parameters.

```
using Flux
dnn_model = Chain(Dense(1, 10, swish), Dense(10, 100,swish),Dense(100, 10, swish), Dense(10,
# Remark: In Deep Learning, there is no one dimensional scalar but one dimensional vector
dnn_model([1.0f0])
```

```
1-element Vector{Float32}:
 -0.040261105
```

```
dnn_model2 = Chain(Dense(2, 10, swish), Dense(10, 100,swish),Dense(100, 10, swish), Dense(10
# Remark: f0 here means we use Float32
dnn_model2([1.0f0,2.0f0])[1]
```

```
0.100376524f0
```

Note here that we didn't set parameters for this deep learning architecture, however, it has an input. Why?

The reason is that in `Flux.jl`, a DNN model has parameters pre defined. Sometimes, it is very inconvenient. Thus, we need to destructure the neural network, so that we can change the parameter of the neural network.

2.3 IMPORTANT: destructure

```
parameter, structure=Flux.destructure(dnn_model2) # parameters and structures
newpara = parameter*0.1 .+ 0.3
# DNN with new parameters
f(x,p)=structure(p)(x)# x input, p parameter
println("DNN with new parameter:", f([1.0f0,2.0f0],newpara))
```

```
DNN with new parameter:Float32[286.11746, 278.07166]
```

2.4 Using DNN to learn $\sin(x)$

```
using Flux
using Flux:train!,params
# 1. Gennerate Data and Define DNN
x = Array(-2 :0.01:2 )'
data = sin.(x)
#dnn_model = Chain(Dense(1, 1), x-> cos.(x))
#dnn_model = Chain(Dense(1, 128, relu), Dense(128, 1), x-> cos.(x))
dnn_model = Chain(Dense(1, 10, swish), Dense(10, 100,swish),Dense(100, 10, swish), Dense(10,
# dnn_model = Chain(Dense(1, 32, relu), Dense(32, 1, tanh))


# 2. Define Loss Functions
loss2(a,b) = Flux.Losses.mae(dnn_model(x), data)

# Train the model
opt=ADAM(0.02)
```

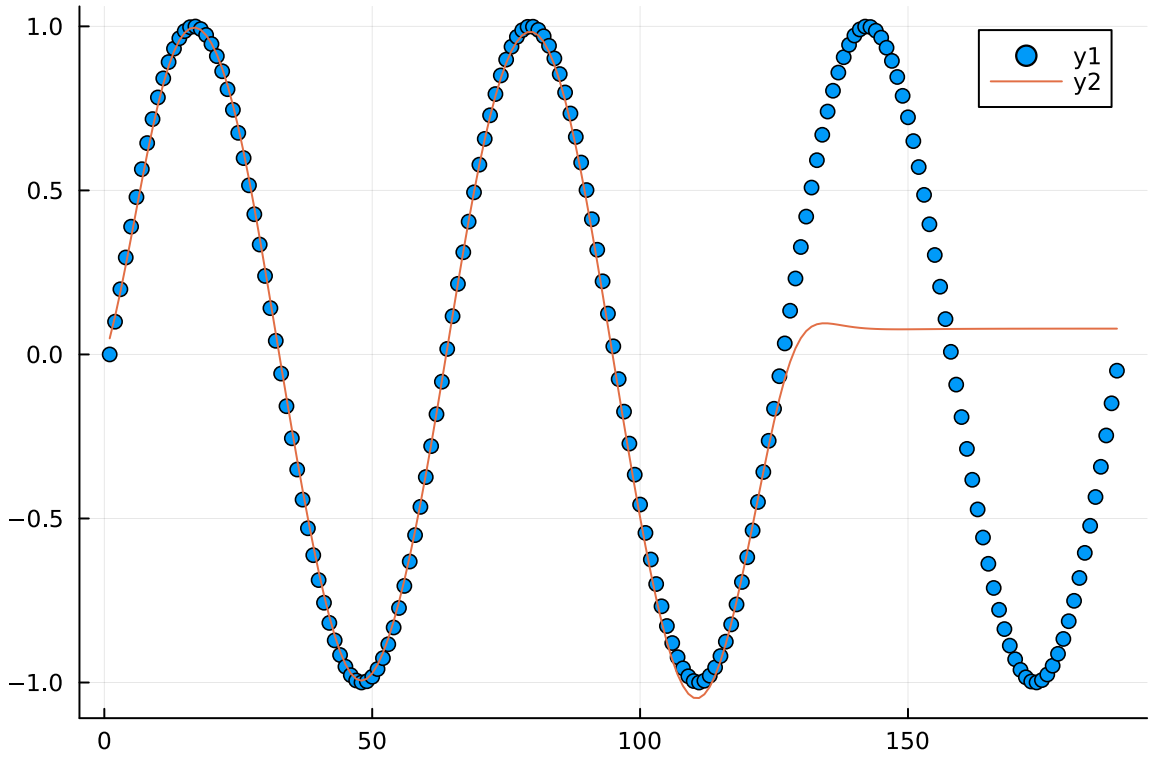
```
println(loss2(x, data))
train!(loss2, Flux.params(dnn_model), [(x,data)], opt)
println(loss2(x, data))



for epoch in 1:5000
    train!(loss2, params(dnn_model), [(x,data)], opt)
    if epoch%500==0
    println(loss2(x, data))
    end
end
```

```
0.6178579239993328
0.8133888838007648
0.029455694189493377
0.019722234455541977
0.02941249289718608
0.044564484807402126
0.017978833994311296
0.018942334715419555
0.04541769339893456
0.019092272589503845
0.020229457337176477
0.02070247009650008
```

Data visulization

```
using Plots
println(loss2(x, data))
y = Array(-2 :0.1:4 )'
scatter(sin.(y)')
plot!(dnn_model(y)')
```

```
0.02070247009650008
```



2.4.1 Question: why this DNN learn the $\sin(x)$ data but fails to predict $\sin(x)$? How to improve the performance?

2.5 Optimization

Training a deep learning model is an optimization problem. In Julia, there is a unified optimization package with many popular optimizers, such as LBFS, BFGS, ADAM, SGD, evolution algorithms. For more details, one can seen - Optimization.jl: A Unified Optimization Package · Optimization.jl

At this time, you only need to understand the following example:

$$\min_x (x_1 - p_1)^2 + p_2 * (x_2 - x_1^2)^2$$

where p is parameter you can pre defined.

```
# Import the package and define the problem to optimize
using Optimization
rosenbrock(u, p) = (p[1] - u[1])^2 + p[2] * (u[2] - u[1]^2)^2
```

```
u0 = zeros(2)
p = [1.0, 100.0]

prob = OptimizationProblem(rosenbrock, u0, p)

# Import a solver package and solve the optimization problem
using OptimizationOptimJL
sol = solve(prob, NelderMead())
```

```
retcode: Success
u: 2-element Vector{Float64}:
 0.9999634355313174
 0.9999315506115275
```

Import a different solver package and solve the optimization problem a different way

```
using OptimizationBBO
prob = OptimizationProblem(rosenbrock, u0, p, lb = [-1.0, -1.0], ub = [1.0, 1.0])
sol = solve(prob, BBO_adaptive_de_rand_1_bin_radiuslimited())
```

```
retcode: Failure
u: 2-element Vector{Float64}:
 0.9999999999999876
 0.9999999999999785
```

3 Differential Equations

3.0.1 Solving SIR Model

More details can be seen in

Getting Started with Differential Equations in Julia · DifferentialEquations.jl

SciML/DifferentialEquations.jl: Multi-language suite for high-performance solvers of differential equations and scientific machine learning (SciML) components. Ordinary differential equations (ODEs), stochastic differential equations (SDEs), delay differential equations (DDEs), differential-algebraic equations (DAEs), and more in Julia.

3.1 IMPORTANT: Activate Julia environment first

```
using Pkg
Pkg.activate(".")
```

```
  Activating project at `~/Desktop/MyProjects/Julia_Tutorial_on_AI4MathBiology`
```

3.1.1 Learn Julia in AI era

Actually, you can also use Cursor - The AI-first Code Editor (VSCode with ChatGPT embedded.)

```
using DifferentialEquations
function sir!(du,u,p,t)
    S,I,R = u
     ,  = p
    du[1] = - *S*I
    du[2] =  *S*I- *I
    du[3] =  *I
end
parms = [0.1,0.05]
```

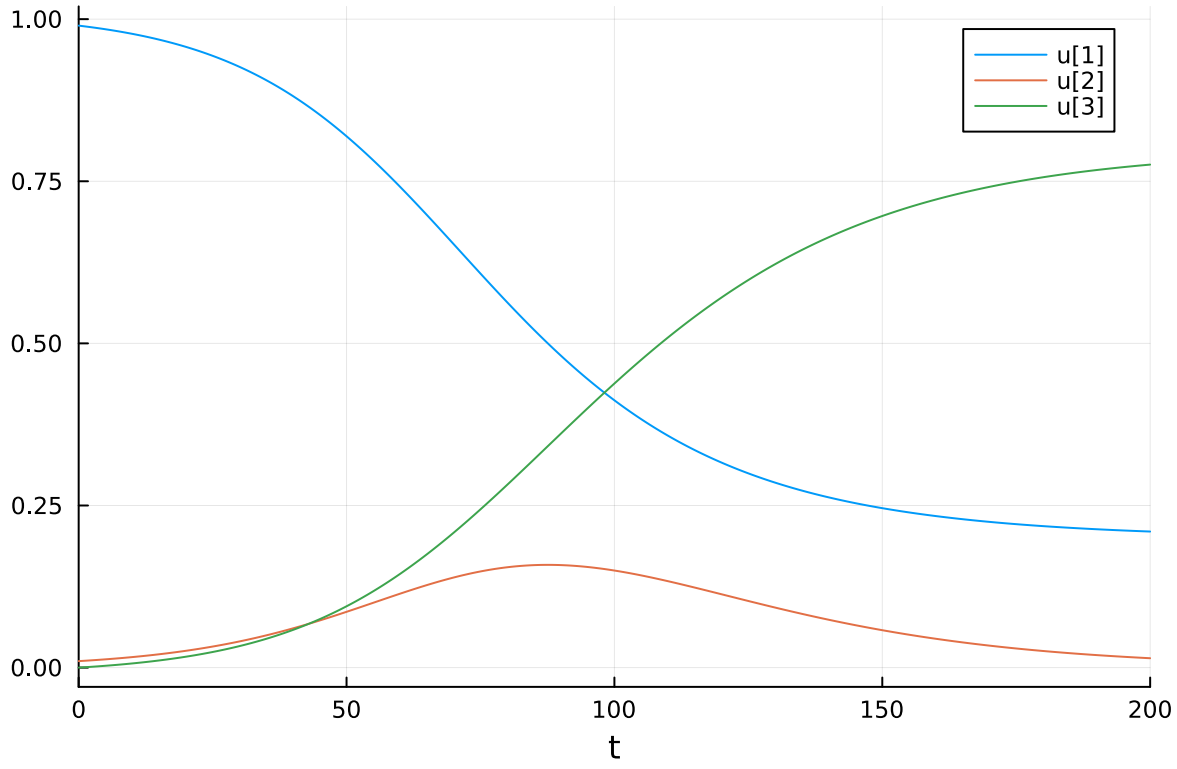
```
initialvalue = [0.99,0.01,0.0]
tspan = (0.0,200.0)
sir_prob = ODEProblem(sir!,initialvalue,tspan,parms)
sir_sol = solve(sir_prob,saveat = 0.1);
```

3.1.2 Data visulization

In Julia, we use `Plots.jl` or `Makie.jl` or `TidierPlots.jl`(similar to `ggplot2`)

- Plots.jl
- Makie.jl
- TidierOrg/TidierPlots.jl: Tidier data visualization in Julia, modeled after the ggplot2 R package.

```
using Plots
plot(sir_sol)
```

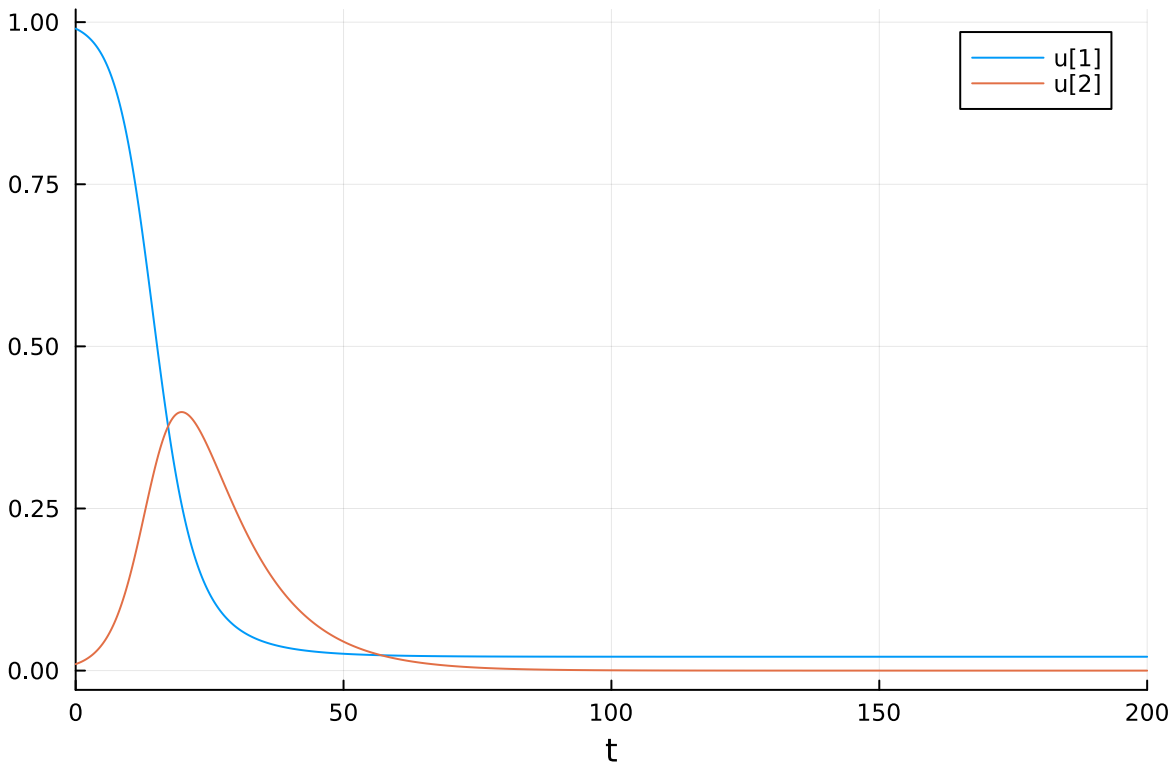


3.2 Solving SIR model with neural networks embedded

Solving the following differential equations with neural networks embedded

$$\begin{cases} \frac{\mathrm{d}S}{\mathrm{d}t} = -\mathrm{NN}(I)S, \\ \frac{\mathrm{d}I}{\mathrm{d}t} = \mathrm{NN}(I)S - \gamma I, \end{cases}$$

```
using DifferentialEquations
using Flux
using Plots
ann_node = Flux.Chain(Flux.Dense(1, 64, tanh), Flux.Dense(64, 1))
para, re = Flux.destructure(ann_node) # destructure
function SIR_nn(du,u,p,t)
    S, I = u
    du[1] =  - S*re(p)([I])[1]
    du[2] = S*re(p)([I])[1] - 0.1*I
end
initialvalue = Float32.([0.99,0.01])
tspan = (0.0f0,200.0f0)
prob_nn = ODEProblem(SIR_nn, initialvalue, tspan, para)
sol_nn=solve(prob_nn)
plot(sol_nn)
```


1.00
0.75
0.50
0.25
0.00
0
50
100
150
200
t
u[1]
u[2]

4 Couple of differential equations and deep learning

Now we will apply UDE

$$\begin{cases} I' = \gamma \text{NeuralNetwork}_\theta(t, I) I - \gamma I, \\ \mathcal{R}_t = \text{NeuralNetwork}_\theta(t, I), \end{cases}$$

to learn effective reproduction number from the data generated by logistic model

$$\begin{cases} I' = 0.2 \left(1 - \dfrac{I}{30}\right) I, \\ \mathcal{R}_t = 3 - \dfrac{I}{15}. \end{cases}$$

For detail on mathematics, one can see - Song P, Xiao Y. Estimating time-varying reproduction number by deep learning techniques[J]. J Appl Anal Comput, 2022, 12(3): 1077-1089.

4.1 IMPORTANT: Activate Julia environment first

```
using Pkg
Pkg.activate(".")
```

```
  Activating project at `~/Desktop/MyProjects/Julia_Tutorial_on_AI4MathBiology`
```

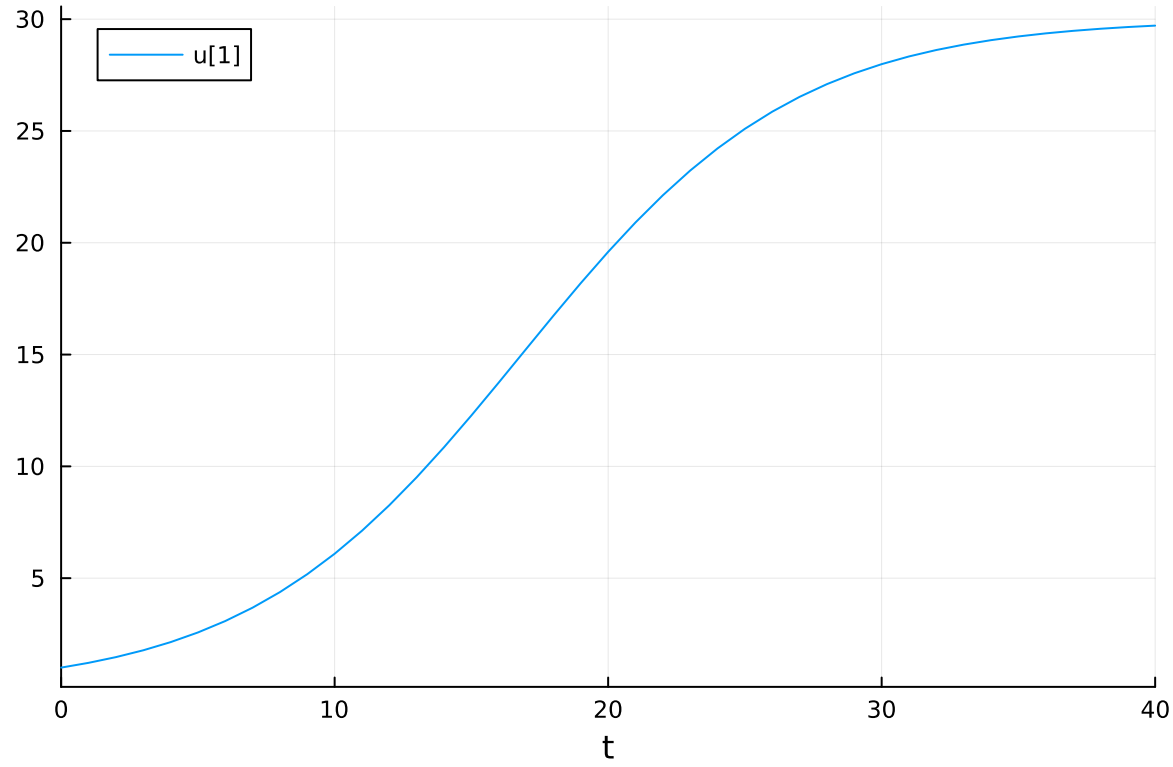
4.2 Loading packages and setting random seeds

```
##
using Lux, DiffEqFlux, DifferentialEquations, Optimization, OptimizationOptimJL, Random, Plo
using DataFrames
using CSV
```

```
using ComponentArrays
using OptimizationOptimisers
using Flux
using Plots
using LaTeXStrings
rng = Random.default_rng()
Random.seed!(1);
```

4.3 Generating test data from logistic model

```
function model2(du, u, p, t)
    r,   = p
    du .= r .* u .* (1 .- u ./  )
end
u_0 = [1.0]
p_data = [0.2, 30]
tspan_data = (0.0, 30)
prob_data = ODEProblem(model2, u_0, tspan_data, p_data)
data_solve = solve(prob_data, Tsit5(), abstol=1e-12, reltol=1e-12, saveat=1)
data_withoutnois = Array(data_solve)
data = data_withoutnois #+ Float32(2e-1)*randn(eltype(data_withoutnois), size(data_withoutnoi
tspan_predict = (0.0, 40)
prob_predict = ODEProblem(model2, u_0, tspan_predict, p_data)
test_data = solve(prob_predict, Tsit5(), abstol=1e-12, reltol=1e-12, saveat=1)
plot(test_data)
```



4.4 Define neural ODE

```
ann_node = Lux.Chain(Lux.Dense(1, 10, tanh), Lux.Dense(10, 1))
p, st = Lux.setup(rng, ann_node)
function model2_nn(du, u, p, t)
    du[1] = 0.1 * ann_node([t], p, st)[1][1] * u[1] - 0.1 * u[1]
end
prob_nn = ODEProblem(model2_nn, u_0, tspan_data, ComponentArray(p))
function train( )
    Array(concrete_solve(prob_nn, Tsit5(), u_0,  , saveat=1,
        abstol=1e-6, reltol=1e-6))#,sensealg = InterpolatingAdjoint(autojacvec=ReverseDiffVJ
end
println(train(p))
```

```
[1.0 0.9034894787099034 0.816044774397625 0.738899351409905 0.6708651886632545 0.610311506668
```

4.5 Define Loss functions and Callbacks

```
function loss( )
    pred = train( )
    sum(abs2, (data .- pred)), pred # + 1e-5*sum(sum.(abs, params(ann)))
end

const losses = []
callback( , l, pred) = begin
    push!(losses, l)
    if length(losses) % 100 == 0
        println(losses[end])
    end
    false
end

pinit = ComponentArray(p)
println(loss(p))
callback(pinit, loss(pinit)...)
```

```
(8216.907972450199, [1.0 0.9034894787099034 0.816044774397625 0.738899351409905 0.670865188
```

```
WARNING: redefinition of constant Main.losses. This may fail, cause incorrect answers, or pr
```

```
false
```

4.6 Train the DNN embedded in differential equations

```
##
adtype = Optimization.AutoZygote()

optf = Optimization.OptimizationFunction((x, p) -> loss(x), adtype)
optprob = Optimization.OptimizationProblem(optf, pinit)

result_neuralode = Optimization.solve(optprob,
    OptimizationOptimisers.ADAM(0.01),
    callback=callback,
    maxiters=3000)
```

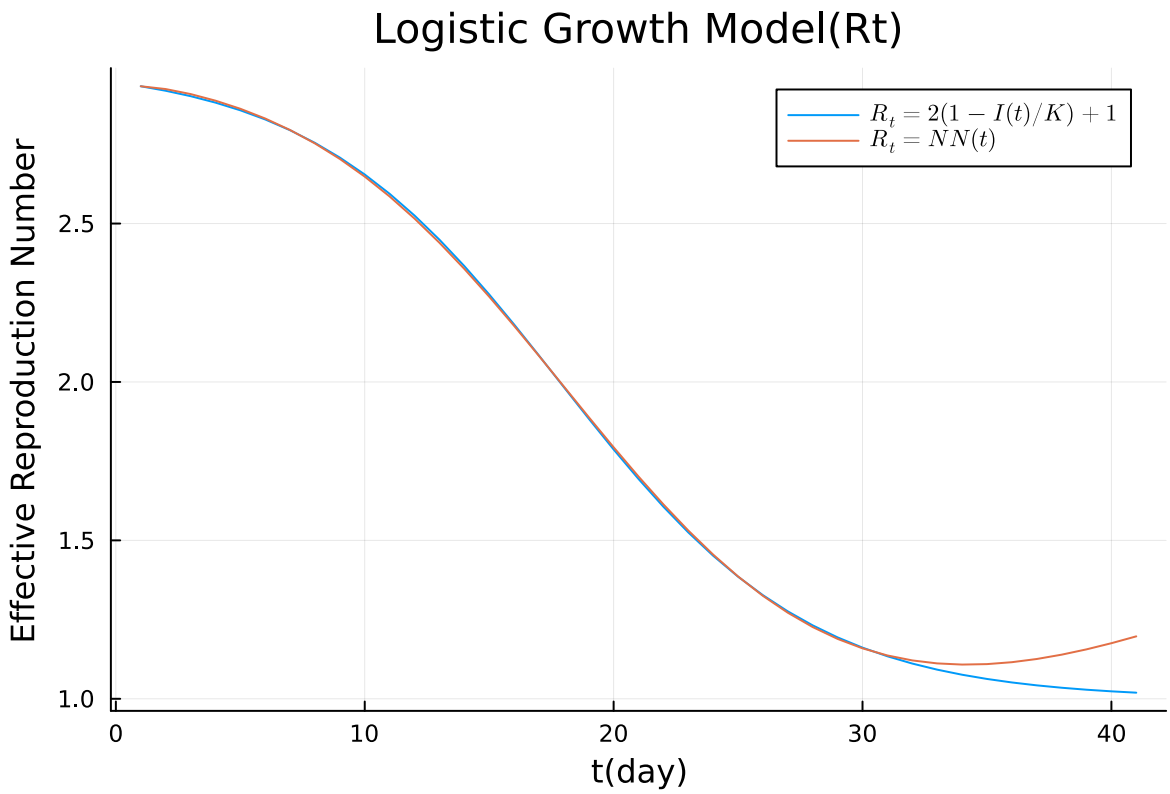
```
351.3882444818854
10.717546698923622
5.085851488814772
3.309161988025375
2.047732114123806
1.4779429126053438
1.1030499970359096
0.8671407377062605
0.89452740171059
1.1183121854553197
0.5708184017763677
0.6691065208401739
5.132183889809579
0.37482373305669753
0.34152053487660833
0.3957727120735596
0.45823548822468946
0.6362163059886281
0.5308457201849255
0.38457406727826243
0.3170416291552578
0.24499973482324744
0.2588192487633485
0.2888203122058308
0.2505142252862214
0.32557412313084977
0.21740930062101116
0.22007704827398192
0.27991144946026614
0.36140008308087385

retcode: Default
u: ComponentVector{Float32}(layer_1 = (weight = Float32[-0.07485764; -0.8849845; … ; 0.65817
```

4.7 Output and data visulization

```julia
pfinal = result_neuralode.u

println(pfinal)
prob_nn2 = ODEProblem(model2_nn, u_0, tspan_predict, pfinal)
```

```
s_nn = solve(prob_nn2, Tsit5(), saveat=1)

# I(t)
scatter(data_solve.t, data[1, :], label="Training Data")
plot!(test_data, label="Real Data")
plot!(s_nn, label="Neural Networks")
xlabel!("t(day)")
ylabel!("I(t)")
title!("Logistic Growth Model(I(t))")
#savefig("Figures/logisticIt.png")
# R(t)
f(x) = 2 * (1 - x / p_data[2]) + 1
plot((f.(test_data))', label=L"R_t = 2(1-I(t)/K)+1")
plot!((f.(s_nn))', label=L"R_t = NN(t)")
xlabel!("t(day)")
ylabel!("Effective Reproduction Number")
title!("Logistic Growth Model(Rt)")
#savefig("Figures/logisticRt.png")
```

```
(layer_1 = (weight = Float32[-0.07485764; -0.8849845; -0.91284543; -0.052147113; -0.96092546
```



4.7.1 Question: modify the codes, change `Lux.jl` back to `Flux.jl`?

5 Symbolic Regression

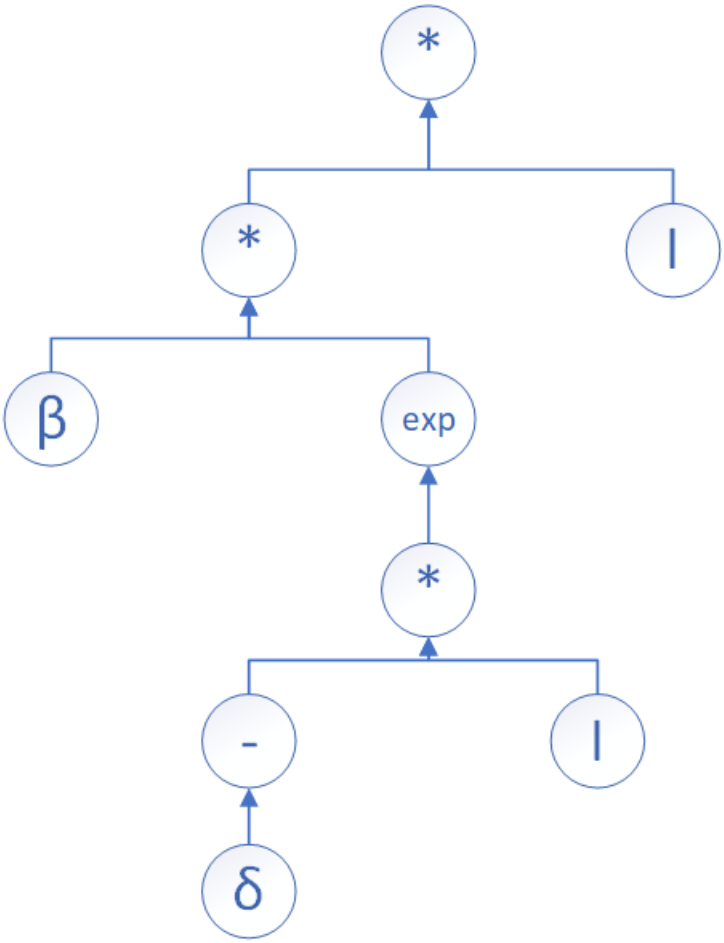
More details, we refer to MilesCranmer/SymbolicRegression.jl: Distributed High-Performance Symbolic Regression in Julia

Symbolic Regression (SR) is a type of regression analysis that searches the space of mathematical expressions to find the model that best fits a given dataset in terms of accuracy and simplicity. It utilizes binary trees to represent a function, and does not rely on a particular model as a starting point to the algorithm. Instead, initial expressions are formed by randomly combining mathematical building blocks such as - binary mathematical operators: $+, -, *, /$; - unary analytic functions: $\sin, \cos, \exp, \tanh, ...$; - constants; - state variables.

Usually, a subset of these primitives will be specified by the person operating it, but that's not a requirement of the technique. SR uses genetic programming, as well as more recently methods utilizing Bayesian methods and physics inspired AI to discover the equations. More details and benchmarks on symbolic regression methods can be seen in cavalab/srbench: A living benchmark framework for symbolic regression.

For example, we generate data from the following function which can be used to investigate the mass media impact on infectious disease:

$$f(I) = \beta \exp(-\delta I) I,$$



and the binary tree of the equation is shown in

The function can be discovered by using symbolic regression, where the codes implemented in Julia are as following.

5.1 IMPORTANT: Activate Julia environment first

```
using Pkg
Pkg.activate(".")
```

```
# Load packages
using SymbolicRegression
using SymbolicUtils
# Generating test data
```

```
I = collect(0:0.1:10)
f(x) = 0.2exp(-0.1*x)*x
Y = f.(I)


# choosing operations
options = SymbolicRegression.Options(
    binary_operators=(+, *, -),
    unary_operators=(exp,),
    npopulations=20
)


# equations searching
hallOfFame = EquationSearch(I', Y, niterations=150, options=options, parallelism=:multithrea
)
# output
dominating = calculate_pareto_frontier(I, Y, hallOfFame, options)
eqn = node_to_symbolic(dominating[end].tree, options)
```

```
(x1 * exp(-0.8866729615216796 - ((x1 - 5.823564614359717) * 0.10000000003526198))) * 0.27113
```

5.2 Project: predicting the peak time

Some immature ideas (may be wrong): - In Lectures 1&2 note, we know that

$$I(t_p) = I_0 + S_0\left(1 - \frac{1}{\mathcal{R}_0} - \frac{\ln \mathcal{R}_0}{\mathcal{R}_0}\right),$$

which implies the peak time

$$t_p = f(\mathcal{R}_0).$$

Q: Can we use symbolic regression to search f?

- For real case data, you can learn $\beta(t) = \text{NeuralNetwork}(t)$ first

$$\begin{cases} \dfrac{\mathrm{d}S}{\mathrm{d}t} = -\text{NeuralNetwork}(t)SI, \\ \dfrac{\mathrm{d}I}{\mathrm{d}t} = \text{NeuralNetwork}(t)SI - \gamma I, \end{cases}$$

then you get a new ODE. Solving this new ODE and setting NeuralNetwork$(t)S(t)I(t) - \gamma I(t) = 0$, you can solve the peak time t_p, and check whether t_p is minimum or maximum by checking the sign

$$\frac{\mathrm{d}^2 I}{\mathrm{dt}^2} = (\text{NeuralNetwork}(t)SI - \gamma I)' = (\text{NeuralNetwork}(t)'S + \text{NeuralNetwork}(t)S')I + (\text{NeuralNetwork}(t)S-$$

Q: - How to solve this fixed point problem NeuralNetwork$(t)S(t)I(t) - \gamma I(t) = 0$? - How to improve the generalization (prediction ability) of the neural network embedded in differential equations? - How to combine these two ideas?