1. Callback.jl

#This could be cleaned up in the future, but works for now.

#Plots the initial solution, before training, based on the initialized kinetic parameters

#Every n\_plot, plots the current trained solution as well as the loss profiles.

#All plots are in the /results/figs/ folder.

#\*\*Note that "plotting the solution" means plotting one randomly picked dataset. Not all datasets are re-plotted every n\_plot epochs.\*\*

function plot\_sol(i\_exp, HR1,HR2, HR3, exp\_data, Tlist, cap, sol0 = nothing)

    beta = l\_exp\_info[i\_exp]

    T0=100+273.15

    sol=HR1+HR2+HR3

    ind = length(Tlist)

    plt = plot(

        Tlist,

        exp\_data[:, 2],

        seriestype = :scatter,

        label = "Exp",

    )

    plot!(

        plt,

        Tlist,

        HR1,

        lw = 3,

        legend = :left,

        label = "CRNN R1",

    )

    plot!(

        plt,

        Tlist,

        HR2,

        lw = 3,

        legend = :left,

        label = "CRNN R2",

    )

    plot!(

        plt,

        Tlist,

        HR3,

        lw = 3,

        legend = :left,

        label = "CRNN R3",

    )

    plot!(

        plt,

        Tlist,

        sol,

        lw = 3,

        legend = :left,

        label = "CRNN sum",

    )

    xlabel!(plt, "Time [min]")

    ylabel!(plt, "HRR")

    title!(plt, cap)

    exp\_cond = string(

        @sprintf(

            "T0 = %.1f K \n beta = %.2f K/min",

            T0,

            beta,

        )

    )

    annotate!(plt, exp\_data[end, 1] / 60.0 \* 0.85, 0.4, exp\_cond)

    p2 = plot(Tlist, sol, lw = 2, legend = :right, label = "heat release")

    xlabel!(p2, "Time [min]")

    ylabel!(p2, "W/g")

    plt = plot(plt, p2, framestyle = :box, layout = @layout [a; b])

    plot!(plt, size = (800, 800))

    return plt

end

cbi = function (p, i\_exp)

    exp\_data = l\_exp\_data[i\_exp]

    sol = pred\_n\_ode(p, i\_exp, exp\_data)[1]

    times = pred\_n\_ode(p, i\_exp, exp\_data)[2]

    raw\_sol=pred\_n\_ode(p, i\_exp, exp\_data)[3]

    HR1=HRR\_getter(times, raw\_sol[:, :])[:, 1]\*w\_delH[1]

    HR2=HRR\_getter(times, raw\_sol[:, :])[:, 2]\*w\_delH[2]

    HR3=HRR\_getter(times, raw\_sol[:, :])[:, 3]\*w\_delH[3]

    Tlist = similar(times)

    T0=100+273.15

    beta = l\_exp\_info[i\_exp, 1]

    for (i, t) in enumerate(times)

        Tlist[i] = getsampletemp(t, T0, beta)

    end

    value = l\_exp[i\_exp]

    plt = plot\_sol(i\_exp, HR1,HR2, HR3, exp\_data, Tlist, "exp\_$value")

    png(plt, string(fig\_path, "/conditions/pred\_exp\_$value"))

    return false

end

function plot\_loss(l\_loss\_train, l\_loss\_val; yscale = :log10)

    plt\_loss = plot(l\_loss\_train, yscale = yscale, label = "train")

    plot!(plt\_loss, l\_loss\_val, yscale = yscale, label = "val")

    plt\_grad = plot(list\_grad, yscale = yscale, label = "grad\_norm")

    xlabel!(plt\_loss, "Epoch")

    ylabel!(plt\_loss, "Loss")

    xlabel!(plt\_grad, "Epoch")

    ylabel!(plt\_grad, "Gradient Norm")

    plt\_all = plot([plt\_loss, plt\_grad]..., legend = :top, framestyle=:box)

    plot!(

        plt\_all,

        size=(1000, 450),

        xtickfontsize = 11,

        ytickfontsize = 11,

        xguidefontsize = 12,

        yguidefontsize = 12,

    )

    png(plt\_all, string(fig\_path, "/loss\_grad"))

end

l\_loss\_train = []

l\_loss\_val = []

list\_grad = []

iter = 1

cb = function (p, loss\_train, loss\_val, g\_norm)

    global l\_loss\_train, l\_loss\_val, list\_grad, iter

    if !isempty(l\_loss\_train)

        if loss\_train<minimum(l\_loss\_train)

            global p\_opt=deepcopy(p)

        end

    end

    push!(l\_loss\_train, loss\_train)

    push!(l\_loss\_val, loss\_val)

    push!(list\_grad, g\_norm)

    if iter % n\_plot == 0 || iter==1

        display\_p(p)

        if @isdefined p\_opt

            @printf("parameters of lowest yet loss:")

            display\_p(p\_opt)

        end

        list\_exp = randperm(n\_exp)[1]

        @printf(

            "Min Loss train: %.2e val: %.2e",

            minimum(l\_loss\_train),

            minimum(l\_loss\_val)

        )

        println("\n update plot ", l\_exp[list\_exp], "\n")

        for i\_exp in list\_exp

            cbi(p, i\_exp)

        end

        plot\_loss(l\_loss\_train, l\_loss\_val; yscale = :log10)

        if @isdefined p\_opt

            @save string(ckpt\_path, "/mymodel.bson") p opt l\_loss\_train l\_loss\_val list\_grad iter p\_opt

        end

    end

    iter += 1

end

if is\_restart

    @load string(ckpt\_path, "/mymodel.bson") p opt l\_loss\_train l\_loss\_val list\_grad iter

    iter += 1

end

1. CRNN\_cathode.jl
2. ##main driver file. Running this trains and plots the CRNN.
3. include("header.jl")
4. include("dataset.jl")
5. include("network.jl")
6. include("callback.jl")
7. epochs = ProgressBar(iter:n\_epoch);
8. loss\_epoch = zeros(Float64, n\_exp);
9. grad\_norm = zeros(Float64, n\_exp);
10. for epoch in epochs #loop epochs
11. global p
12. for i\_exp in randperm(n\_exp) #loop heating rate datasets
13. if i\_exp in l\_val #ignore validation data
14. continue
15. end
16. grad = ForwardDiff.gradient(x -> loss\_neuralode(x, i\_exp), p)
17. grad\_norm[i\_exp] = norm(grad, 2)
18. if grad\_norm[i\_exp] > grad\_max
19. grad = grad ./ grad\_norm[i\_exp] .\* grad\_max
20. end
21. update!(opt, p, grad) #update parameters using ForwardDiff gradient
22. end
23. for i\_exp = 1:n\_exp
24. loss\_epoch[i\_exp] = loss\_neuralode(p, i\_exp) #save raw loss value for plotting
25. end
26. loss\_train = mean(loss\_epoch[l\_train])
27. loss\_val = mean(loss\_epoch[l\_val])
28. grad\_mean = mean(grad\_norm[l\_train])
29. set\_description(
30. epochs,
31. string(
32. @sprintf(
33. "Loss train: %.2e val: %.2e grad: %.2e",
34. loss\_train,
35. loss\_val,
36. grad\_mean,
37. )
38. ),
39. )
40. cb(p, loss\_train, loss\_val, grad\_mean) #plotting script
41. end
42. conf["loss\_train"] = minimum(l\_loss\_train)
43. conf["loss\_val"] = minimum(l\_loss\_val)
44. YAML.write\_file(config\_path, conf)
45. for i\_exp in randperm(n\_exp)
46. cbi(p, i\_exp)
47. end

3) Cathode.jl

include("header.jl")

function load\_exp(filename)

    exp\_data = readdlm(filename,',', Float64) #[t, HRR]

    index = indexin(unique(exp\_data[:, 1]), exp\_data[:, 1])

    exp\_data = exp\_data[index, :]

    return exp\_data

end

l\_exp\_data = [];

l\_exp\_info = zeros(Float64, length(l\_exp), 1);

heating\_rates=[2, 5, 10, 15, 20]

for (i\_exp, value) in enumerate(l\_exp)

    filename = string("exp\_data/cath\_", string(cathode\_num),"\_", string(heating\_rates[value]), ".csv")

    exp\_data = Float64.(load\_exp(filename))

    temps=exp\_data[:, 1]

    times=(temps.-100).\*60/heating\_rates[value] #convert temperatures into times for ODE solving

    exp\_data[:, 1]=times

    push!(l\_exp\_data, exp\_data)

end

l\_exp\_info[:, 1] = heating\_rates;

4) header.jl

using Random, Plots

using Zygote, ForwardDiff

using OrdinaryDiffEq, DiffEqSensitivity

using DiffEqCallbacks

using LinearAlgebra

using Statistics

using ProgressBars, Printf

using Flux

using Flux.Optimise: update!

using Flux.Losses: mae, mse

using BSON: @save, @load

using DelimitedFiles

using YAML

ENV["GKSwstype"] = "100"

cd(dirname(@\_\_DIR\_\_))

conf = YAML.load\_file("./config.yaml")

expr\_name = conf["expr\_name"]

fig\_path = string("./results/", expr\_name, "/figs")

ckpt\_path = string("./results/", expr\_name, "/checkpoint")

config\_path = "./results/$expr\_name/config.yaml"

is\_restart = Bool(conf["is\_restart"])

ns = Int64(conf["ns"])

nr = Int64(conf["nr"])

lb = Float64(conf["lb"])

n\_epoch = Int64(conf["n\_epoch"])

n\_plot = Int64(conf["n\_plot"])

grad\_max = Float64(conf["grad\_max"])

maxiters = Int64(conf["maxiters"])

lr\_max = Float64(conf["lr\_max"])

lr\_min = Float64(conf["lr\_min"])

lr\_adam = Float64(conf["adam\_lr"])

lr\_decay = Float64(conf["lr\_decay"])

lr\_decay\_step = Int64(conf["lr\_decay\_step"])

w\_decay = Float64(conf["w\_decay"])

global cathode\_num=Int64(conf["cathode"])

llb = lb;

global p\_cutoff = -1.0

const l\_exp = 1:5

n\_exp = length(l\_exp)

l\_train = []

l\_val = []

for i = 1:n\_exp

    j = l\_exp[i]

    if !(j in [4])

        push!(l\_train, i)

    else

        push!(l\_val, i)

    end

end

opt = ADAMW(lr\_adam, (0.9, 0.999), w\_decay);

if !is\_restart

    if ispath(fig\_path)

        rm(fig\_path, recursive = true)

    end

    if ispath(ckpt\_path)

        rm(ckpt\_path, recursive = true)

    end

end

if ispath("./results") == false

    mkdir("./results")

end

if ispath("./results/$expr\_name") == false

    mkdir("./results/$expr\_name")

end

if ispath(fig\_path) == false

    mkdir(fig\_path)

    mkdir(string(fig\_path, "/conditions"))

end

if ispath(ckpt\_path) == false

    mkdir(ckpt\_path)

end

cp("./config.yaml", config\_path; force=true)

5) network.jl

#17 parameters total.

#3 rxn orders

#2 stoich coeffs (products only)

#3 Ea

#3 A

#3 b

#3 delta H

np = 17+1 #with slope at the end

p = randn(Float64, np) .\* 1.e-2;

p[1:3] .+= 1;  # A

#Ea initial condition. Roughly initialized to be in the right order (i.e. rxn 1 -> rxn 2 -> rxn 3)

Ea\_IC=[1.0, 1.1, 1.2]

p[4]+=Ea\_IC[1]

p[5]+=Ea\_IC[2]

p[6]+=Ea\_IC[3]

p[10] += 1; #delta H, roughly guessed based on how big the peaks in the data sort of look

p[11] += 0.2; #delta H

p[12] += 0.3; #delta H

p[13:15] .+= 1; #reaction order n

p[16:17] .+= 1; #stoich coeff nu

p[18]= 0.1; #slope, as per original CRNN code

function p2vec(p)

    #some clamps in place during debugging to make sure none of the parameters get too large or small

    #these don't appear to be necessary in the final runs

    slope = p[end] .\* 1.e1

    w\_A = p[1:3] .\* (slope \* 20.0) #logA

    w\_A = clamp.(w\_A, 0, 50)

    w\_out = p[16:17] #product stoich. coeffs

    w\_out=[1, w\_out[1], w\_out[2]]

    w\_out=clamp.(w\_out, 0.01, 5)

    w\_in\_order=p[13:15] #rxn orders

    w\_in\_order=clamp.(w\_in\_order, 0.01, 10)

    w\_in\_Ea = abs.(p[4:6]) #Ea

    w\_in\_Ea = clamp.(w\_in\_Ea, 0.0, 3)

    w\_in\_b = (p[7:9]) #non-exponential temp dependence, can be negative, no clamp

    w\_delH = abs.(p[10:12])\*100

    w\_delH=clamp.(w\_delH, 10, 300)

    return w\_in\_Ea, w\_in\_b, w\_out, w\_delH, w\_in\_order, w\_A

end

function display\_p(p)

    w\_in\_Ea, w\_in\_b, w\_out, w\_delH, w\_in\_order, w\_A = p2vec(p)

    println("\n species (column) reaction (row)")

    println("rxn ord | Ea | b | delH | lnA | stoich coeff")

    show(stdout, "text/plain", round.(hcat(w\_in\_order, w\_in\_Ea, w\_in\_b, w\_delH, w\_A, w\_out), digits = 2))

    println("\n")

end

function getsampletemp(t, T0, beta)

    if beta[1] < 100

        T = T0 .+ beta[1] / 60 \* t  # K/min to K/s

    end

    return T

end

const R = -1.0 / 8.314  # J/mol\*K

@inbounds function crnn!(du, u, p, t)

    #given a current concentration, parameter vector, and time

    #return the three concentration gradients

    logX = @. log(clamp(u, lb, 10.0))

    T = getsampletemp(t, T0, beta)

    temp\_term= reshape(hcat(log(T), R/T)\*hcat(w\_in\_b, w\_in\_Ea\*10^5)', 3)

    rxn\_ord\_term=w\_in\_order.\*logX

    rxn\_rates= @. exp(temp\_term+rxn\_ord\_term+w\_A )

    du .=  -rxn\_rates #each reaction consumes the corresponding reactant

    #first and second reactions also produce c2 and c3:

    du[2]=du[2]+w\_out[2]\*rxn\_rates[1]

    du[3]=du[3]+w\_out[3]\*rxn\_rates[2]

end

function HRR\_getter(times, u\_outputs)

    #Take the concentration trajectories solved by crnn!(),

    #and compute the raw reaction rates, to multiply layer against dH to obtain the exothermic heat release.

    logX = @. log(clamp(u\_outputs, lb, 10.0))

    T = getsampletemp(times, T0, beta)

    temp\_term=@. log(T).\*w\_in\_b'+R/T\*(w\_in\_Ea\*10^5)'

    rxn\_ord\_term=transpose(w\_in\_order).\*transpose(logX)

    rxn\_rates= @. exp(temp\_term+rxn\_ord\_term.+w\_A' )

    return rxn\_rates

end

tspan = [0.0, 1.0];

u0 = zeros(ns);

u0[1] = 1.0; #start with unity normalized mass of c1 only: c2 and c3 are produced \*sequentially\* as products.

prob = ODEProblem(crnn!, u0, tspan, p, abstol = lb)

condition(u, t, integrator) = u[1] < lb \* 10.0

affect!(integrator) = terminate!(integrator)

\_cb = DiscreteCallback(condition, affect!)

alg = AutoTsit5(TRBDF2(autodiff = true));

sense = ForwardSensitivity(autojacvec = true)

function pred\_n\_ode(p, i\_exp, exp\_data)

    global beta = l\_exp\_info[i\_exp, :]

    global T0=100+273.15 #degrees K

    global w\_in\_Ea, w\_in\_b, w\_out, w\_delH, w\_in\_order, w\_A = p2vec(p)

    ts = @view(exp\_data[:, 1])

    tspan = [ts[1], ts[end]]

    #solve the species trajetory, which is independent of heat release (idealized DSC system):

    sol = solve(

        prob,

        alg,

        tspan = tspan,

        p = p,

        saveat = ts,

        sensealg = sense,

        maxiters = maxiters,

    )

    #post-processing: compute the heat release (actual desired solution) from the species trajectory:

    heat\_rel= HRR\_getter(ts, sol[:, :])\*w\_delH

    if sol.retcode == :Success

        nothing

    else

        @sprintf("solver failed beta: %.0f",  beta[1])

    end

    return heat\_rel, ts, sol

end

function loss\_neuralode(p, i\_exp)

    exp\_data = l\_exp\_data[i\_exp]

    pred = Array(pred\_n\_ode(p, i\_exp, exp\_data)[1]) #index=1 for times

    loss = mae(pred, @view(exp\_data[:, 2])) #index=2 for heat releases

    return loss

end

@time loss = loss\_neuralode(p, 1)

6) config.yaml

expr\_name: 4s8r-01

is\_restart: false

ns: 3 #c1, c2, c3

nr: 3 #each of above decomposes into next

lb: 1.e-8

n\_epoch: 1000000

n\_plot: 1000

grad\_max: 1.e2

maxiters: 5000000

adam\_lr: 1.e-3

lr\_max: 2.5e-5

lr\_min: 5.e-5

lr\_decay: 0.2

lr\_decay\_step: 500

w\_decay: 1.e-7

cathode: 1 #pick the cathode number (1 through 6)

EXPERIMENTAL DATA FOR EACH EXPT LOOKS LIKE THIS:  
A grid of white lines

Description automatically generated