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
```

2) CRNN_cathode.jl

```
3) ##main driver file. Running this trains and plots the CRNN.
4)
5) include("header.jl")
6) include("dataset.jl")
7) include("network.jl")
8) include("callback.jl")
9) epochs = ProgressBar(iter:n_epoch);
10)loss_epoch = zeros(Float64, n_exp);
11)grad_norm = zeros(Float64, n_exp);
12)
13)for epoch in epochs #loop epochs
14)     global p
15)     for i_exp in randperm(n_exp) #loop heating rate datasets
16)         if i_exp in l_val #ignore validation data
17)             continue
18)         end
19)         grad = ForwardDiff.gradient(x -> loss_neuralode(x, i_exp), p)
20)         grad_norm[i_exp] = norm(grad, 2)
21)         if grad_norm[i_exp] > grad_max
22)             grad = grad ./ grad_norm[i_exp] .* grad_max
23)         end
24)         update!(opt, p, grad) #update parameters using ForwardDiff
    gradient
25)     end
26)     for i_exp = 1:n_exp
```

```
27)         loss_epoch[i_exp] = loss_neuralode(p, i_exp) #save raw loss
    value for plotting
28)     end
29)     loss_train = mean(loss_epoch[l_train])
30)     loss_val = mean(loss_epoch[l_val])
31)     grad_mean = mean(grad_norm[l_train])
32)     set_description(
33)         epochs,
34)         string(
35)             @sprintf(
36)                 "Loss train: %.2e val: %.2e grad: %.2e",
37)                 loss_train,
38)                 loss_val,
39)                 grad_mean,
40)             )
41)         ),
42)     )
43)     cb(p, loss_train, loss_val, grad_mean) #plotting script
44)end
45)
46)conf["loss_train"] = minimum(l_loss_train)
47)conf["loss_val"] = minimum(l_loss_val)
48)YAML.write_file(config_path, conf)
49)
50)for i_exp in randperm(n_exp)
51)     cbi(p, i_exp)
52)end
53)
```

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	B	C	D	E	F	G	H	I	J	K
	119.2911	0.000845									
	129.3677	0.001923									
	138.6824	0.002438									
	150.3378	0.002439									
	163.0068	0.003794									
	174.1655	0.003385									
	180.3266	0.004512									
	186.6605	0.006052									
	192.9944	0.006332									
0	199.3282	0.007241									
1	205.6621	0.007941									
2	211.996	0.008361									
3	218.3298	0.010741									
4	224.5358	0.014411									
5	233.5311	0.022092									
6	239.6347	0.029566									
7	248.2718	0.03956									
8	254.6057	0.045709									
9	260.9395	0.050701									
0	267.2734	0.053606									
1	273.6073	0.05581									

San