"

Training neural ODE

Neurales ODE in Anlehnung an eine einfache differentielle Gleichung (ODE). Wobei die differenzelle Änderungen in einem System beschreibt. Durch Eingabe der Startwerte kann so der tatsächliche Graph abgebildert werden. Anstelle eines Solvers, wo einzelne Parameter des ODE angepasst werden, soll ein neurales Netzwerk trainiert werden, dass die ODE abbildet.

Training eines NN (Dense).

1. 2->16, Aktivierungsfunktion tanh,
2. 16->2, Aktivierungsfunktion linear.

Eingabe in das Training: Zeitpunkte zu denen Gemessene Punkte mit trainierten verglichen werden sollen.

Die zwei Knoten in Eingabe und Ausgabe Layer entsprechen den Temperaturen an den gegeben Temperatursensoren des tclab, während die erste Heizquelle zyklisch aufgeheizt wird. Das heißt, beide Sensoren werden zyklisch erhitzt. Sensor 1, stärker als Sensor 2. Sensor 2 ist im Aufheizen und Abkühlen im Vergleich zu Sensor 1 leicht verzögert.

Es soll ein neurales ODE mit dem Netzwerk angelernt werden. Das Netz soll die Änderungen des Verlauf des Temperaturen Vorhersagen.

Für die Vorhersage wird daher die Starttemperatur benötigt.

```
begin
    using CSV
    using DiffEqFlux
    using Flux: train!, Chain, Dense
    using Flux
    using DataInterpolations
    using DataFrames
    using OrdinaryDiffEq
    using Plots
    using PlutoUI
    using BenchmarkTools
end
```

Table of Contents

Training neural ODE

```
celsius2kelvin (generic function wi
```

```
celsius2kelvin(c) = c + 273.15
```

```
prepare_dataframe (generic function with 1 method)
```

```
function prepare_dataframe(data_frame_path)

    _df = DataFrame(CSV.File(data_frame_path))

    start_times_stat = unique(_df, :run)

    _df.start_time =  start_times_stat[trunc.(Int, _df.run).+1, :time]
    _df.duration = _df.time - _df.start_time
    _df.T1 = _df.T1 .|> celsius2kelvin
    _df.T2 = _df.T2 .|> celsius2kelvin
    _df.Column1 = _df.Column1 .+ 1
    _df.run = _df.run .+ 1

    return _df
end
```

	Column1	T1	T2	Q1	time	run	start_time	duration
1	1	294.693	295.499	100.0	1.68496e9	1.0	1.68496e9	0.0
2	2	294.693	295.338	100.0	1.68496e9	1.0	1.68496e9	1.35462
3	3	294.693	295.466	100.0	1.68496e9	1.0	1.68496e9	2.5741
4	4	294.693	295.466	100.0	1.68496e9	1.0	1.68496e9	3.794
5	5	294.693	295.466	100.0	1.68496e9	1.0	1.68496e9	5.01308
6	6	294.693	295.563	100.0	1.68496e9	1.0	1.68496e9	6.23228
7	7	295.015	295.563	100.0	1.68496e9	1.0	1.68496e9	7.45149
8	8	295.015	295.531	100.0	1.68496e9	1.0	1.68496e9	8.67103
9	9	295.338	295.563	100.0	1.68496e9	1.0	1.68496e9	9.8905
10	10	295.338	295.531	100.0	1.68496e9	1.0	1.68496e9	11.1097
	more							
8640	8640	298.238	299.527	0.0	1.68497e9	12.0	1.68496e9	868.37

```
begin
    heating_df_path = joinpath(@__DIR__, "measurements_heating_and_cooling.csv")
    heating_df = prepare_dataframe(heating_df_path)
end
```

[295.499, 295.338, 295.466, 295.466, 295.466, 295.563, 295.563, 295.531, 295.563, 295.

```
begin
    # define data for the ODE

    end_point = 3600;
    tspan = (0.0, end_point);
    tsteps = range(tspan[1], tspan[2], length=120);

    # seconds since start
    t = heating_df[1:end_point, :time] .- heating_df.start_time[1];
    u = heating_df[1:end_point, :Q1];
    T₁ = heating_df[1:end_point, :T1];
    T₂ = heating_df[1:end_point, :T2];
end
```

([294.693, 300.397, 310.288, 319.212, 326.302, 332.257, 336.856, 340.909, 343.979,

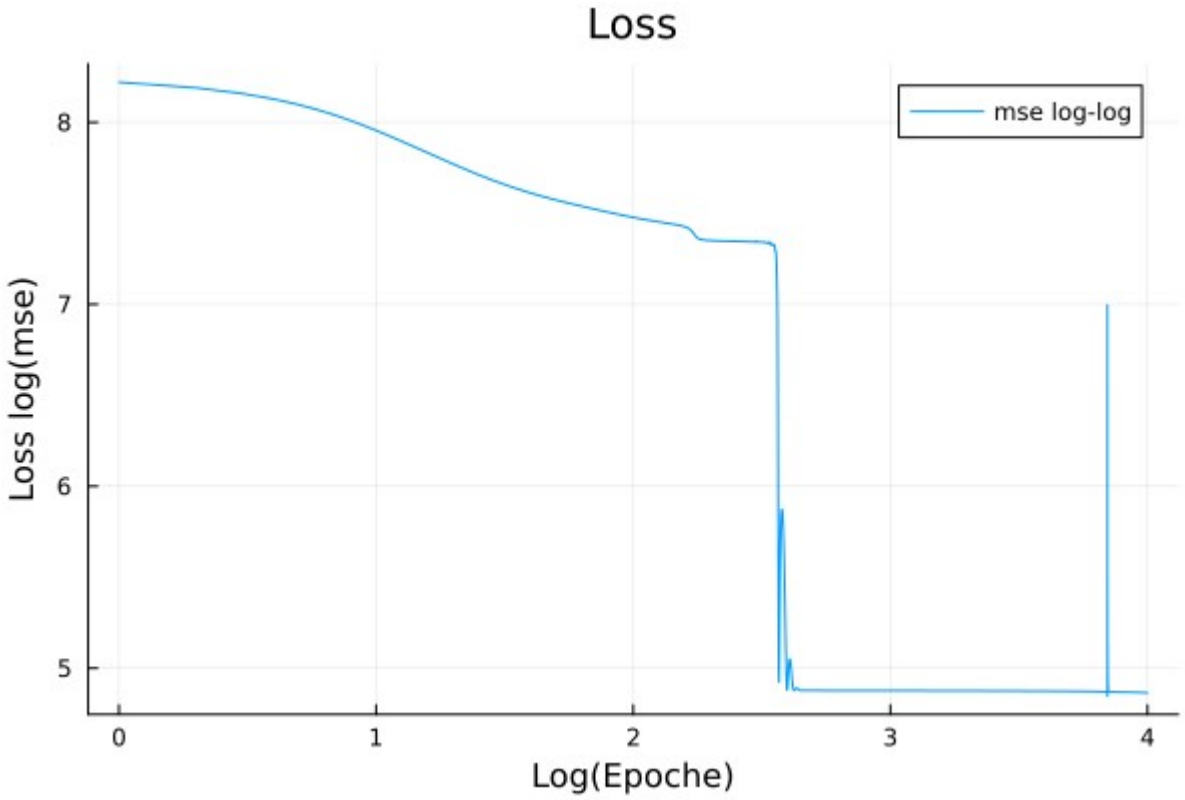
```
begin
    # Interpoliere the u, y
    u_t = ConstantInterpolation(u, t);
    T₁_t = LinearInterpolation(T₁, t);
    T₂_t = LinearInterpolation(T₂, t);
    T0 = [T₁_t(0),T₂_t(0)]

    ode_data = Array(T₁_t(tsteps)), Array(T₂_t(tsteps));
end
```

loss_n_ode (generic function with 1 method)

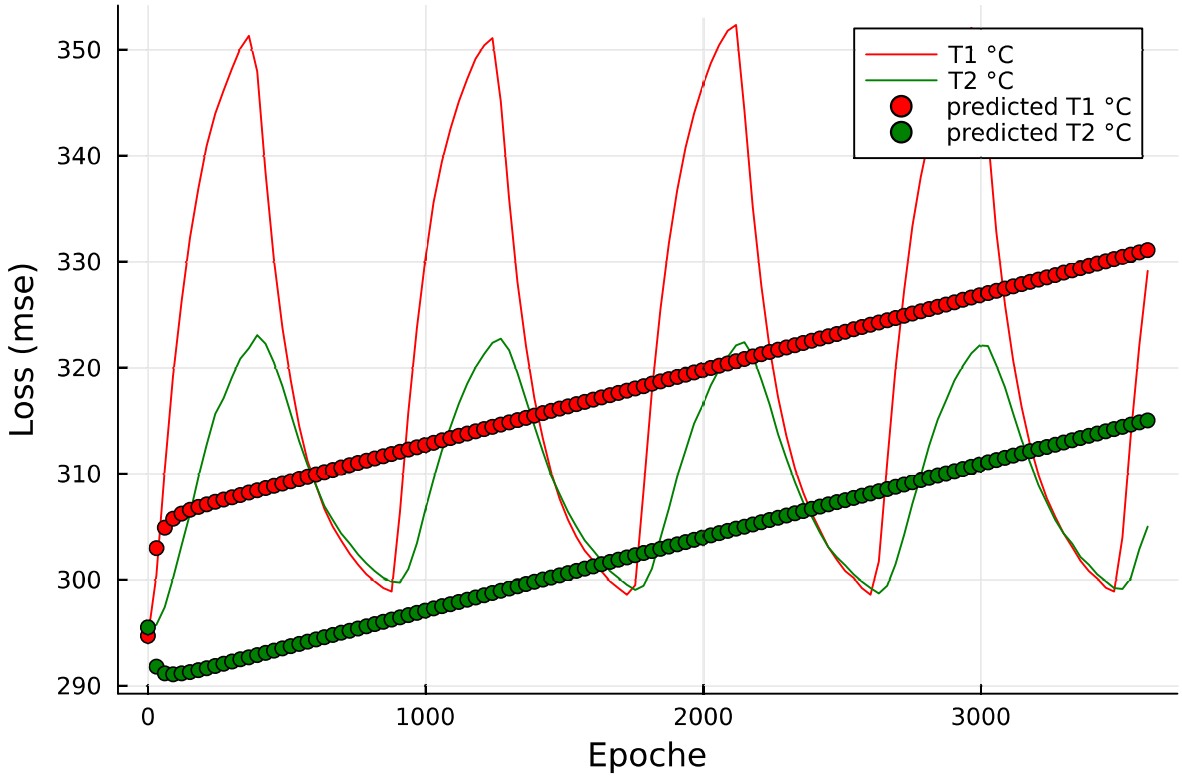
```
begin
    # durch Interpolation langsam im Training
    dudt = Flux.Chain(# x -> u_t.(x),  # wie Verallgemeinderung auf andere u_t
                      Flux.Dense(2=>64, tanh),
                      Flux.Dense(64=>2)) |> f64

    n_ode = NeuralODE(dudt, tspan, DP5(), saveat=tsteps, reltol=1e-7, abstol=1e-9)

    y_predict_before = n_ode(T0)
    ps = Flux.params(n_ode.p)

    function predict_n_ode()
          n_ode(T0)
    end

    loss_n_ode() = sum(abs2,ode_data[1] .- n_ode(T0)'[:, 1]) +
    sum(abs2,ode_data[2] .- n_ode(T0)'[:, 2])


end
```

```
begin

    η = 1e-4
    opt_n_ode = ADAM(η)
    epoches = 10_000

    loss_vector = Vector{Float64}()
    callback() = push!(loss_vector, loss_n_ode())

    data = Iterators.repeated((), epoches)

    Flux.train!(loss_n_ode, ps, data, opt_n_ode, cb=callback)
end
```

100%




```
plot(1:epoches .|> log10, loss_vector .|> log10, label="mse log-log",
title="Loss", xlabel = "Log(Epoche)", ylabel = "Loss log(mse)",)
```




```
begin
    y_predict_after = n_ode(T0)

    # plot result
    p_node = plot(tsteps, ode_data[1], label="T1 °C", lc=:red, xlabel = "Epoche",
    ylabel = "Loss (mse)",)
    plot!(p_node, tsteps, ode_data[2], label="T2 °C", lc=:green)
    scatter!(p_node, tsteps, y_predict_after[1, :], label="predicted T1 °C",
    mc=:red)
    scatter!(p_node, tsteps, y_predict_after[2, :], label="predicted T2 °C",
    mc=:green)
end
```

Beim Training mit zwei Temperaturkurven ist der Verlauf des Losses glatter als, wenn nur ein Sensor angelernt wird. Der Loss nimmt bis zu 1000 Zyklen ab und stackniert hiernach.

Die Oszilation der Temperaturen beim Heizen und Abkühlen konnte nicht nachgebildet werden.

Die Der Prozentsatz der Intensität der Heizung wird, testweise mit Interpolation als erste Layer hinzugefügt. Hierfür wurde eine ConstantInterpolation gewählt, da diese leicht schneller ist, als die Linear und bei den Wertebereich [0, 100] geringere Fehler machen sollte.