Predator-Prey time series prediction

For this project we are going to need *a lot* of dependencies...

Don't be surprised if loading them all takes +10 minutes. You can observe the progress in the `Status` window at the bottom-right.

Table of Contents


```
begin
    using Flux, DataFrames, GLMakie, Images, FileIO, ImageShow,
    DifferentialEquations, ModelingToolkit, PlutoUI, Optimization,
    ComponentArrays, ForwardDiff, Lux, Random, DiffEqFlux
    using Statistics: mean; using CSV: read
    import OptimizationPolyalgorithms.PolyOpt
    import OptimizationOptimisers.ADAMW
    import Optimisers.Adam
    import Optim.BFGS
    PlutoUI.TableOfContents()
end
```

By default, the notebook wil not be running the cells with the trainings - they can also take a relatively long time 🐢.

If you wish to commence all the training processes, just click on the next checkbox.

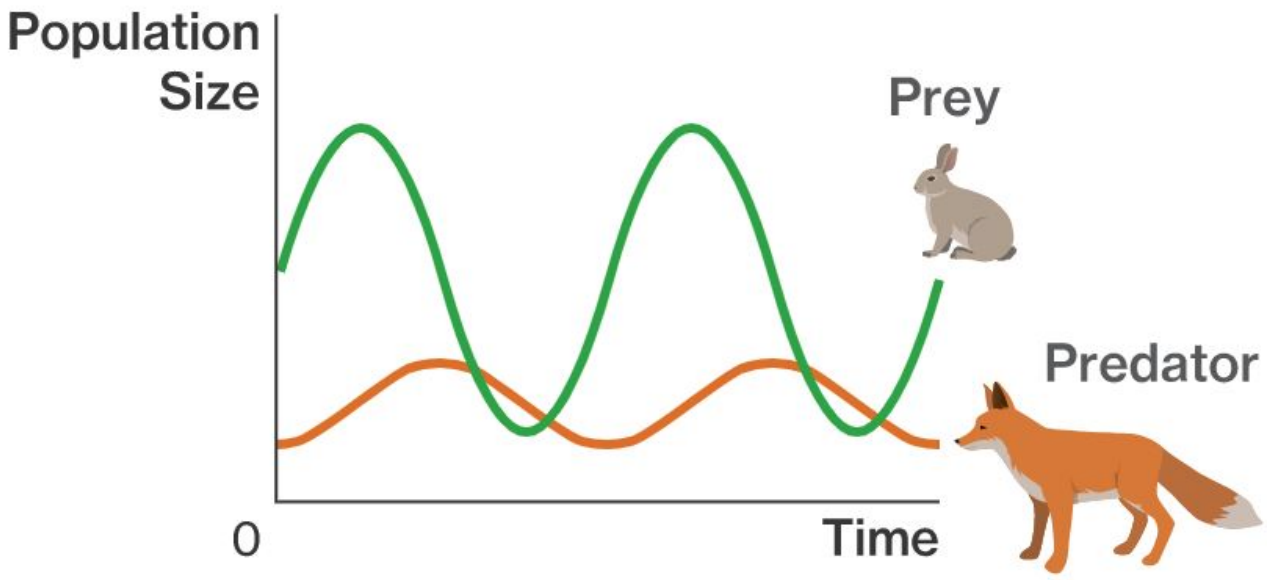
```
1 @bind dotrain PlutoUI.CheckBox(default=false)
```

Introduction

The present project consists in shocasing differents models for time series prediction.

The time series that we want to predict comes from Predator-Prey model. Each series represents the population of a particular species at a certain moment, with one of the species being the predator 🐆 and the other being the prey 🐇.

Therefore, they deal with the general loss-win interactions and hence may have applications outside of ecosystems.




```
begin
    image_path_0 = "./images/predator-prey.png"
    image_0 = load(image_path_0)
    image_0
end
```

The proposed models to make the predictions of this time series are:

- Basic full machine learning model: MLP (Multilayer Perceptron)
- Informed model (Neural ODEs model, based on Lotka-Volterra equations)

Basically the idea is to implement and compare these two different models. The first is a full ML model, which learns only from the data using a MLP model. The second is a more informed model, which makes use of the knowledge we have of the dynammics of these kind of time series, that is, modeling them with an Ordinary Differential Equation (ODE).

Multi Layer Perceptron

```
1 md"# Multi Layer Perceptron"
```

Dataset preparation

Load the dataset

Hare & Lynx dataset, a time series dataset of the population of two species; Hudson Bay company Lynx-Hare dataset from Leigh 1968, parsed from paper copy http://katalog.ub.uni-heidelberg.de/titel/66489211

```
rawdata =
```

	year	hare	lynx
1	1847	21000	49000
2	1848	12000	21000
3	1849	24000	9000
4	1850	50000	7000
5	1851	80000	5000
6	1852	80000	5000
7	1853	90000	11000
8	1854	69000	22000
9	1855	80000	33000
10	1856	93000	33000
more			
57	1903	45000	50000

```
1 rawdata = read("../datasets/Leigh1968_harelynx.csv", DataFrame)
```

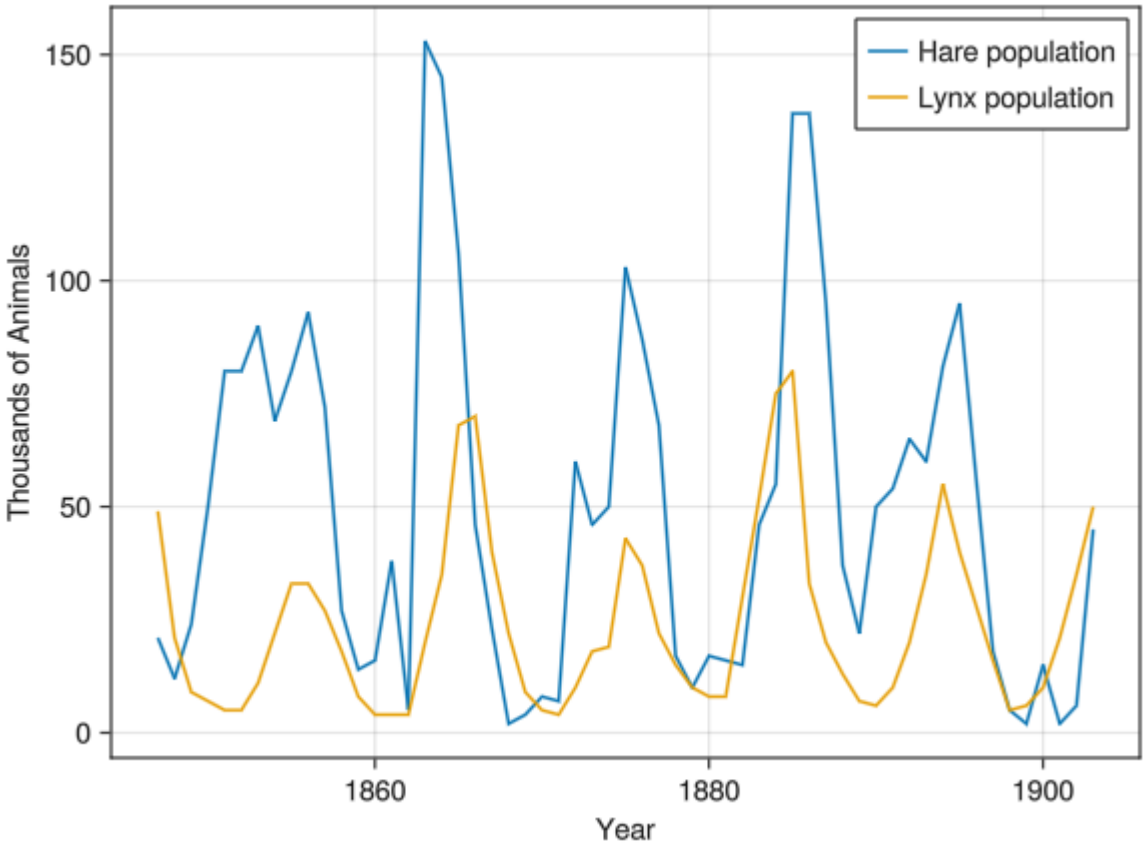
Separate in train and test subsets

There is data of 57 years, therefore we split the data in the train and test subsests in the following manner: |train| = 45 first years and |test| = 12 last years. Also, for simplicity, we will be measuring the populations in 'thousands of animals'.

	hare	lynx
1	21.0	49.0
2	12.0	21.0
3	24.0	9.0
4	50.0	7.0
5	80.0	5.0
6	80.0	5.0
7	90.0	11.0
8	69.0	22.0
9	80.0	33.0
10	93.0	33.0
	more	
57	45.0	50.0

```
begin
    # Create new data frames, with numbers in Float32 format
    df_train = mapcols(x -> Float32.(x .÷ 1000), rawdata[1:45,[:hare, :lynx]])
    df_test  = mapcols(x -> Float32.(x .÷ 1000), rawdata[46:end,[:hare, :lynx]])
    df = vcat(df_train, df_test)
end
```

Visualization of the entire dataset




```
begin
    # Line chart of the data with Makie
    dataset_figure_lines= Figure()
    ax_1 = GLMakie.Axis(dataset_figure_lines[1,1],
        xlabel="Year",
        ylabel="Thousands of Animals")
    GLMakie.lines!(rawdata.year, df.hare, label="Hare population")
    GLMakie.lines!(rawdata.year, df.lynx, label="Lynx population")
    axislegend(ax_1; position=:rt)
    dataset_figure_lines
end
```

The MLP Model

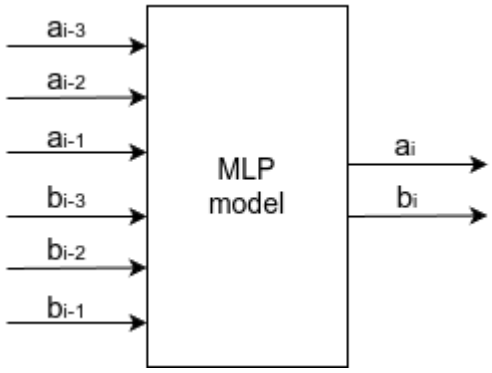
The built model was developed following the MLP time series forecasting tutorial. Based on this we developed a *Multiple Parallel Series / Multivariate forecasting MLP model*. The idea of it is to train a Multi-Layer Perceptron (MLP) for, given multiple parallel time series (say two in our case: `a` and `b`): [(a1,b1), (a2,b2), (a3,b3),..., (an, bn)], to be able to predict the next value for each of the time series: (an+1, bn+1).

For doing this we can follow two different approaches:

- (1) Predict with 1 single MLP. *~Assuming a and b are dependent~*
- (2) Predict each output series with a different MLP (1 per series). *~Assuming a and b are independent~*

In particular, we are going to follow the 1st approach, since we understand that this way the

model learns that both time series `a` and `b` are relationed, and that the outputs, although different, depend on a joint input [(a1,b1), (a2,b2), ...,(aw,bw)].



Window-based

Another important aspect that we do in our model is to follow a window based approach. That is, the input of the MLP is a windowed time series of size `w`: `[(a1,b1), (a2,b2), ..., (aw,bw)]`.

This way, we make that a certain prediction only depends on the last `w` values of the time series. Since *does the number of hares in 1860 matter when trying to predict the number of hares in 1900?* (See the nature of the time series in the `Visualization of the entire dataset` subsection).

E.g. w=3, input:[(a1,b1), (a2,b2), (a3,b3)] and the MLP model will produce as output (a'4,b'4).

Therefore, the challenge here is to select a proper window size `w`.

Preparing the training data

We create customised train datasets in the windowed input format. That is, since what we have is a time series like hares = [h1, h2, h3, ..., hn], to be able to train the model following the window approach we need to format this training data so that we have all the corresponding slices of size `w` in this series and its corresponding target output value to train/fit the model.

Example:

- `w` = 3
- hares = [4, 2, 1, 5, 7, 6]
- lynxes = [2, 3, 4, 3, 1, 3]

For each time series we will build all the corresponding input slices of size `w` with the corresponding target output (the next value of that slice):

- hares: [4,2,1]:5, [2,1,5]:7, [1,5,7]:6
- lynxes: [2,3,4]:3, [3,4,3]:1, [4,3,1]:3

As inputs we will provide $[h_{i-3}, h_{i-2}, h_{i-1}, l_{i-3}, l_{i-2}, l_{i-1}]$. As output, similarly we will expect to get two values $[h_i, l_i]$.

```
- input_1 = [4,2,1]+[2,3,4] = [4,2,1,2,3,4] output_1 = [5]+[3] = [5,3]
- input_2 = [2,1,5]+[3,4,3] = [2,1,5,3,4,3] output_2 = [7]+[1] = [7,1]
- input_3 = [1,5,7]+[4,3,1] = [1,5,7,4,3,1] output_2 = [6]+[3] = [6,3]
```

Therefore, the MLP will learn its weights and biases based on this windowed approach training, so that the MLP model will try to adjust to the output value of each time series whenever the input is like the given one.

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

```
# Function to split the training data in window-based manner
# window size w = lookback
function create_dataset(data, lookback)
    X, Y = [], []
    for i in lookback+1:length(data)
        push!(X, data[i-lookback:i-1])
        push!(Y, data[i])
    end
    return hcat(X...), vcat(Y...)'
end
```

Looking at the nature of the time series in the `Visualization of the entire dataset` subsection, we select 3~5 as window size. Greater or smaller window sizes could also be selected.

```
lookback = 5
```

```
lookback = 5  # Window size
```

```
(5×40 Matrix{Float32}:
 21.0  12.0  24.0  50.0  80.0  80.0  90.0  69.0  …   15.0   46.0   55.0  137.0  137

(5×40 Matrix{Float32}:                                                          , 1:
 49.0  21.0   9.0   7.0   5.0   5.0  11.0  …   8.0  30.0  52.0  75.0  80.0  33.0   !
```

```
X_b, Y_b = create_dataset(df_train.lynx, lookback)
```

In Flux, by default, each column is treated as a separate data point in matrix inputs, so the target data is also a matrix with the same setup.

```
X_ab =
10×40 Matrix{Float32}:
 21.0  12.0  24.0  50.0  80.0  80.0  90.0  69.0  …   15.0   46.0   55.0  137.0  137.0
 12.0  24.0  50.0  80.0  80.0  90.0  69.0  80.0      46.0   55.0  137.0  137.0   95.0
 24.0  50.0  80.0  80.0  90.0  69.0  80.0  93.0      55.0  137.0  137.0   95.0   37.0
 50.0  80.0  80.0  90.0  69.0  80.0  93.0  72.0     137.0  137.0   95.0   37.0   22.0
 80.0  80.0  90.0  69.0  80.0  93.0  72.0  27.0     137.0   95.0   37.0   22.0   50.0
 49.0  21.0   9.0   7.0   5.0   5.0  11.0  22.0  …   30.0   52.0   75.0   80.0   33.0
 21.0   9.0   7.0   5.0   5.0  11.0  22.0  33.0      52.0   75.0   80.0   33.0   20.0
  9.0   7.0   5.0   5.0  11.0  22.0  33.0  33.0      75.0   80.0   33.0   20.0   13.0
  7.0   5.0   5.0  11.0  22.0  33.0  33.0  27.0      80.0   33.0   20.0   13.0    7.0
  5.0   5.0  11.0  22.0  33.0  33.0  27.0  18.0      33.0   20.0   13.0    7.0    6.0
```

```
# as before but now concatenating the b series corresponding column input for
  each of the columns -> vertical concatenation
X_ab = vcat(X_a, X_b)
```

```
Y_ab =
2×40 Matrix{Float32}:
 80.0  90.0  69.0  80.0  93.0  72.0  27.0  …  137.0  95.0  37.0  22.0  50.0  54.0
  5.0  11.0  22.0  33.0  33.0  27.0  18.0      33.0  20.0  13.0   7.0   6.0  10.0
```

```
1 Y_ab = vcat(Y_a, Y_b)
```

Training the model

train!(loss, params, data, opt; cb)

Flux train function: For each datapoint `d` in data, compute the gradient of `loss` with respect to params through backpropagation and call the optimizer `opt`. If d is a tuple of arguments to loss call loss(d...), else call loss(d). A callback is given with the keyword argument `cb`.

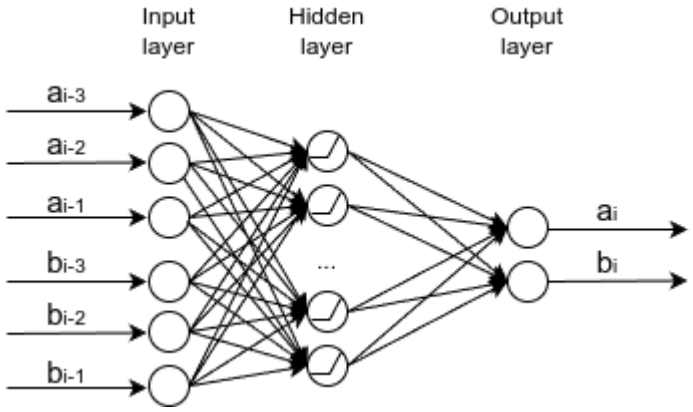
- `loss`: loss function -> to evaluate how well the model is performing and do the adjust of the weights and biases accordingly.
- `params(model)`: where the parameters of the model to be adjusted to minimize the loss (weights and biases) are expressed.
- `data`: training data, with inputs and target outputs, for each of the inputs:(`X_ab`, `Y_ab`)
- `optimizer`: GD (Gradient Descent), SGD (Stochastic Gradient Descent), ADAM (ADaptive Moment Estimation)...

Model setup

1. MLP structure

Chain -> to stack layers, Dense -> fully connected neural network layers

- 1st layer: Dense(`lookback * 2`, `hiddenNeurons`, `relu`) – it is the hidden layer
  - Input size: `lookback * 2` (window size = lookback), so `#lookback` inputs per time series, 2 in this case
  - Output size: `hiddenNeurons` -> hidden layer has `#hiddenNeurons` neurons
  - Activation function: ReLU - introduces non linearity to the model
- Output layer: Dense(`hiddenNeurons`,2)
  - Input size: `hiddenNeurons`, to match the outputs of all the neurons in the hidden layer (`hiddenNeurons`)
  - Output size: 2, since we are forecasting a single future value for each of the two time series




```
hiddenNeurons = 15
```

```
# Number of neurons in the hidden layer
hiddenNeurons = 15
```

```
model_ab = Chain(
             Dense(10 => 15, relu),                # 165 parameters
             Dense(15 => 2),                       # 32 parameters
           )                   # Total: 4 arrays, 197 parameters, 1.020 KiB.
```

```
# model
model_ab = Flux.Chain(
    Flux.Dense(lookback*2 => hiddenNeurons, relu; init =
    Flux.glorot_uniform(MersenneTwister(1000))),
    Flux.Dense(hiddenNeurons => 2, init =
    Flux.glorot_uniform(MersenneTwister(1000)))
  )
```

```
ps_ab =
Params([Float32[0.15183224 0.1697517 … -0.31367162 0.048938707; -0.14633945 -0.2614147
```

```
ps_ab = Flux.params(model_ab)
```

2. Definition of the loss function

MSE between the model predicted values ($\hat{y}1$, $\hat{y}2$ = model(x)) and the target values: `y1, y2`.

```
loss_ab(x,y) = Flux.Losses.mse(model_ab(x), y)
```

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

```
function loss_ab(x,y)
    pred = model_ab(x)
    return mean((pred .- y).^2)
end
```

3. Optimizer

- initially: simple Gradient Descent
- More advanced optimizers: ADAM (ADaptive Moment Estimation)

```
optimizer_ab =  Adam(0.01, (0.9, 0.999), 1.0e-8, IdDict())
```

```
#optimizer_ab = Flux.Descent(0.01)
optimizer_ab = Flux.ADAM(0.01)
```

```
data_pair_ab =
  [(10×40 Matrix{Float32}:
     21.0  12.0  24.0  50.0  80.0  80.0  90.0  69.0  …   15.0   46.0   55.0  137.0  13
```

```
1 data_pair_ab = [(X_ab,Y_ab)]
```

Training loop

```
begin
    # TOTRY: Variate the # of epochs
    epochs = 500
    for epoch in 1:epochs
        Flux.train!(loss_ab, ps_ab, data_pair_ab, optimizer_ab)
        println("Epoch $epoch, Loss: $(loss_ab(X_ab,Y_ab))")
    end

end
```

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

Testing the model

We are going to build the predictions of the test values, for 🐇 and 🐆, with the constructed model and then check the error obtained in terms of the difference with the real test values for those years.

The predictions obtained by the model are going to be constructed in the `pred_test_a` and `pred_test_b` vectors for the hares and lynxes, respectively.

The construction of the prediction vectors will **not** be done using the real test data values, instead we will make use of the self-predicted values by the model.

For instance, if we assume the window size `lookback=3`, we will construct the test years predictions of a certain time series like:

- `[x_43, x_44, x_45] -> x_46'` (*x_43,x_44,x_45 are real values from the train dataset*)
- `[x_44, x_45, x_46'] -> x_47'` (*however, x_46' is not the real test value, but the one that has just been predicted by the model in the previous step*).
- `[x_45, x_46', x_47'] -> x_48'`
- `[x_46', x_47', x_48'] -> x_49'`
- ...
- `[x_54', x_55', x_56'] -> x_57'`

Once we have this predicted $|\hat{test}|$ output set, we compare it against the real $|test|$ dataset to obtain the error and measure the performance.

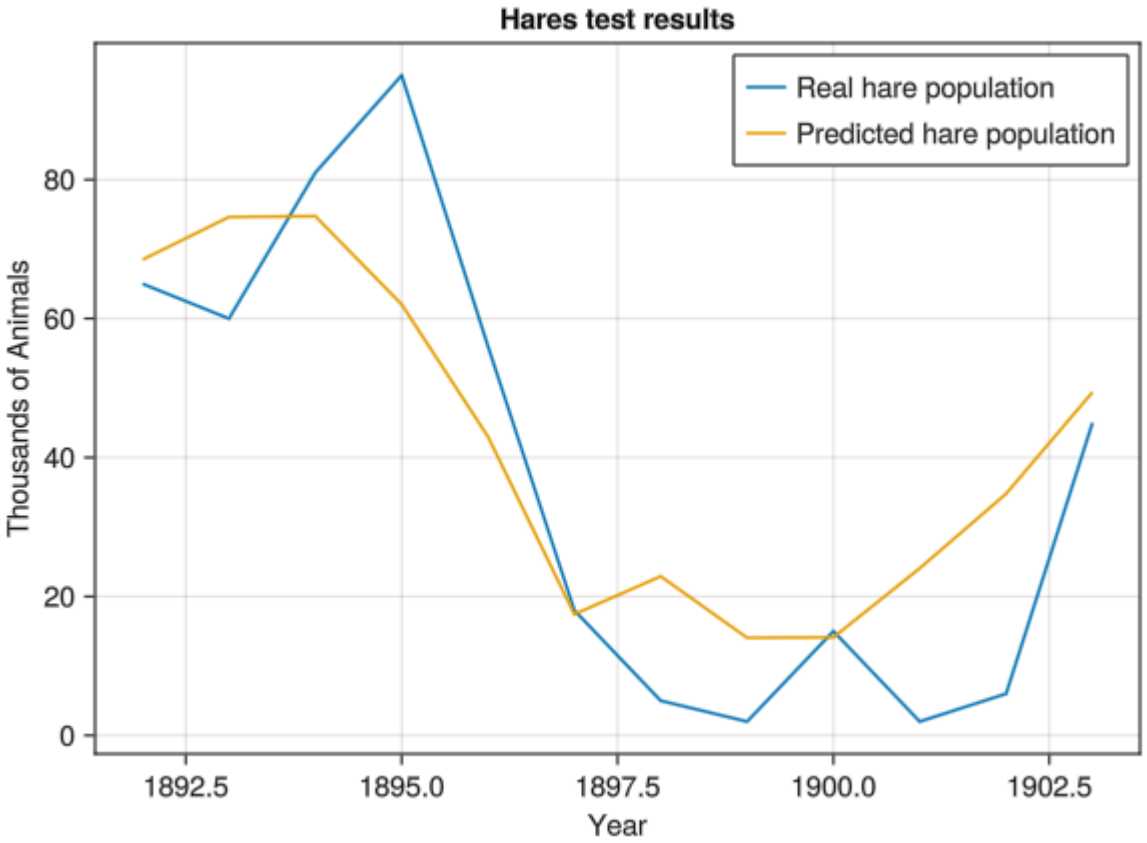
```
begin
    test_data_a = df_test.hare
    test_data_b = df_test.lynx
    pred_test_a = Float32[]
    pred_test_b = Float32[]
    input_vec_a = Float32[]
    input_vec_b = Float32[]
    # fill until the w (lookback) size, initially all from the last w training data elements
    for i in length(df_train.hare)-lookback+1:length(df_train.hare)
        push!(input_vec_a, df_train.hare[i])
        push!(input_vec_b, df_train.lynx[i])
    end
end
```

```
for i in 1:length(test_data_a)
    # put together the two vectors to do the prediction
    input_vec_ab = vcat(input_vec_a, input_vec_b)
    output_ab = model_ab(input_vec_ab)
    # The first predicted value is for the a series and the second for the b series
    push!(pred_test_a, output_ab[1])
    push!(pred_test_b, output_ab[2])
    # prepare the new following input vector (size of this vector = lookback / window size)
    for w in 1:lookback-1
        input_vec_a[w] = input_vec_a[w+1]
        input_vec_b[w] = input_vec_b[w+1]
    end
    input_vec_a[lookback] = output_ab[1]
    input_vec_b[lookback] = output_ab[2]
end
```

Joint error - MSE(hare) + MSE(lynx) = 587.1561f0

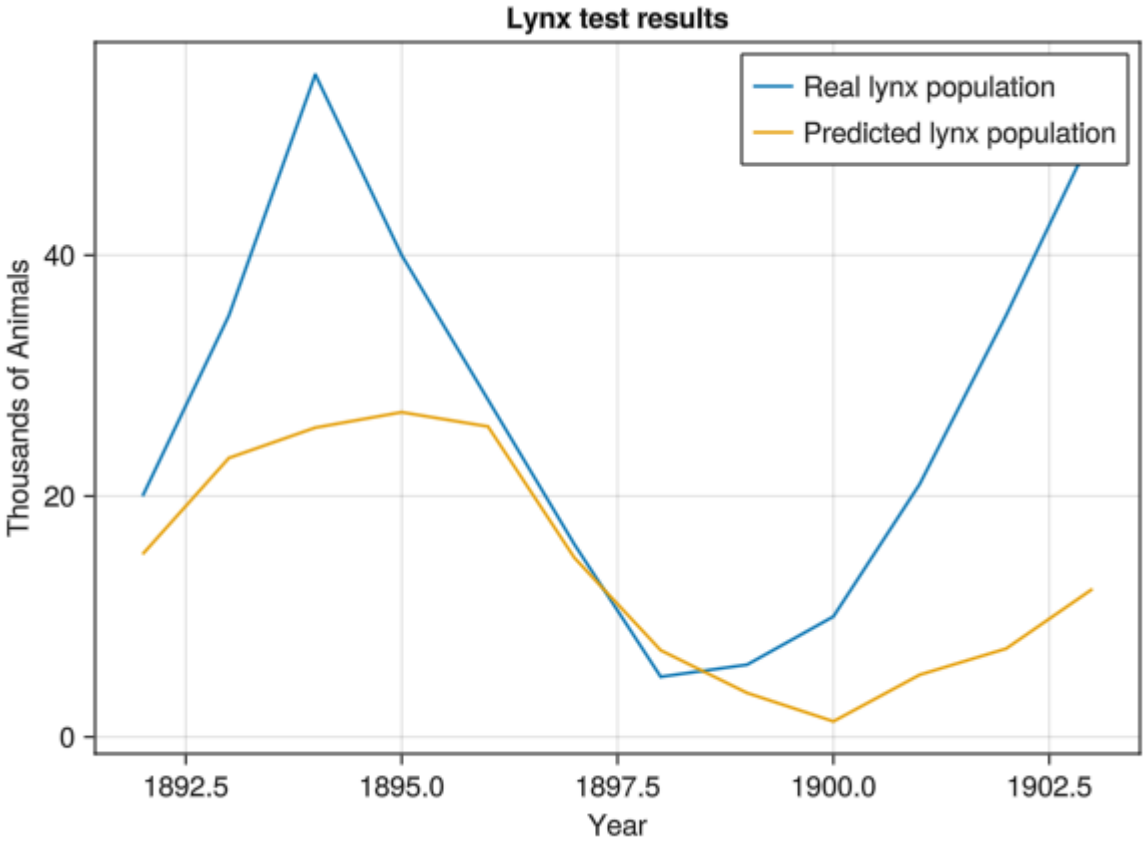
Hares

MSE = 276.86502f0



Lynx

MSE = 310.29114f0



#epochs	Optimizer (learning rate)	Window size	hiddenNeurons	MSE
1000	Gradient_Descent(0.01)	3	15	1347 + 266 = 1613 (BAD - GD gives a straight line as prediction!)
100	ADAM(0.01)	3	15	1098 + 547 = 1645 (underfitting - flat shape curve)
1000	ADAM(0.01)	3	15	1384 + 193 = 1577
500	ADAM(0.01)	3	15	895 + 464 = 1359
500	ADAM(0.01)	10	150	5997 + 2557 = 8555 (overfitted - error ~ 0 for train dataset)
500	ADAM(0.01)	10	50	7178 + 4435 = 11613 (overfitted - error ~ 10^-2 for train dataset)
500	ADAM(0.01)	5	50	1353 + 486 = 1839 (overfitted - error ~ 10^-2 for train dataset)
500	ADAM(0.01)	5	15	276 + 310 = 587
100	ADAM(0.01)	5	15	759 + 617 = 1377
200	ADAM(0.01)	3	15	1110 + 561 = 1672
200	ADAM(0.01)	3	5	972 + 448 = 1420 (quite flat shape of the curve)
200	ADAM(0.01)	3	25	771 + 538 = 1310
500	ADAM(0.01)	3	25	406 + 431 = 837

As described in our prediction model there are many tunnable paramters. The tunning process to select a combination that provides a good enough model in terms of the prediction of the test dataset is described in the previous table. From the experiments done we derived the following conclusions in relation with the different parameters:

- **Optimizer**: ADAM (ADaptive Moment Estimation) is preferrable over GD (Gradient Descent), since for the last many more epochs of training were needed to produce similar MSEs, also GD tended to produce straight lines as prediction, which of course are far to match the periodicity seen in our data.
- **Window size**  : 3~5 was the preferred size, since higher window sizes tended to show overfitting in the training dataset, and the model was not able to generalize for the prediction of the testing values. Intuituively, small window sizes are better due to the nature of our time series periodicity (see it in `Visualization of the entire dataset` subsection).
- **HiddenNeurons**: In relation with the neurons on the hidden layer of the MLP, it was observed that a high number tended to produce overfitting on the training data whereas a low number tended to produce underfitting reflected in a quite flat shape of the prediction curve. Therefore an "intermediate" number (e.g. 15~30) was selected.
- **`#epochs`** : The behavior in this case was similar as the described for the `hiddenNeurons` , meaning that a high number of epochs tended to produce an overfitted model and a low number of epochs tended to produce an underfitted model. Therefore, the preferred

number of epochs was selected to be ~500.

So far, with these experiments we managed to obtain a total MSE (`MSE(hares) + MSE(lynx)`) of `587`, with: `#epochs = 500`, `optimizer = ADAM(0.01)`, `window = 5` and `hiddenNeurons = 15`.

Lokta-Volterra system

We wish to model the evolution of our twin time series, $x \sim$ 🐇 and $y \sim$ 🐆, as a Lokta-Volterra dynamic system, which is defined as follows:

$$\begin{cases} x'(t) = \alpha x(t) - \beta x(t) y(t) \\ y'(t) = -\gamma y(t) + \delta x(t) y(t) \end{cases}$$

```
lotka_volterra! (generic function with 1 method)
```

```
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
```

For our case, we may impose that the initial time $t = 0$ corresponds to the first year in our train dataset, namely 1847. The final time will be $t = 44$, corresponding to the year 1891.

Therefore, the initial conditions for the system are $x(0) = 21$ and $y(0) = 49$.

```
[1.07771, 0.0340597, 1.04049, 0.028716]
```

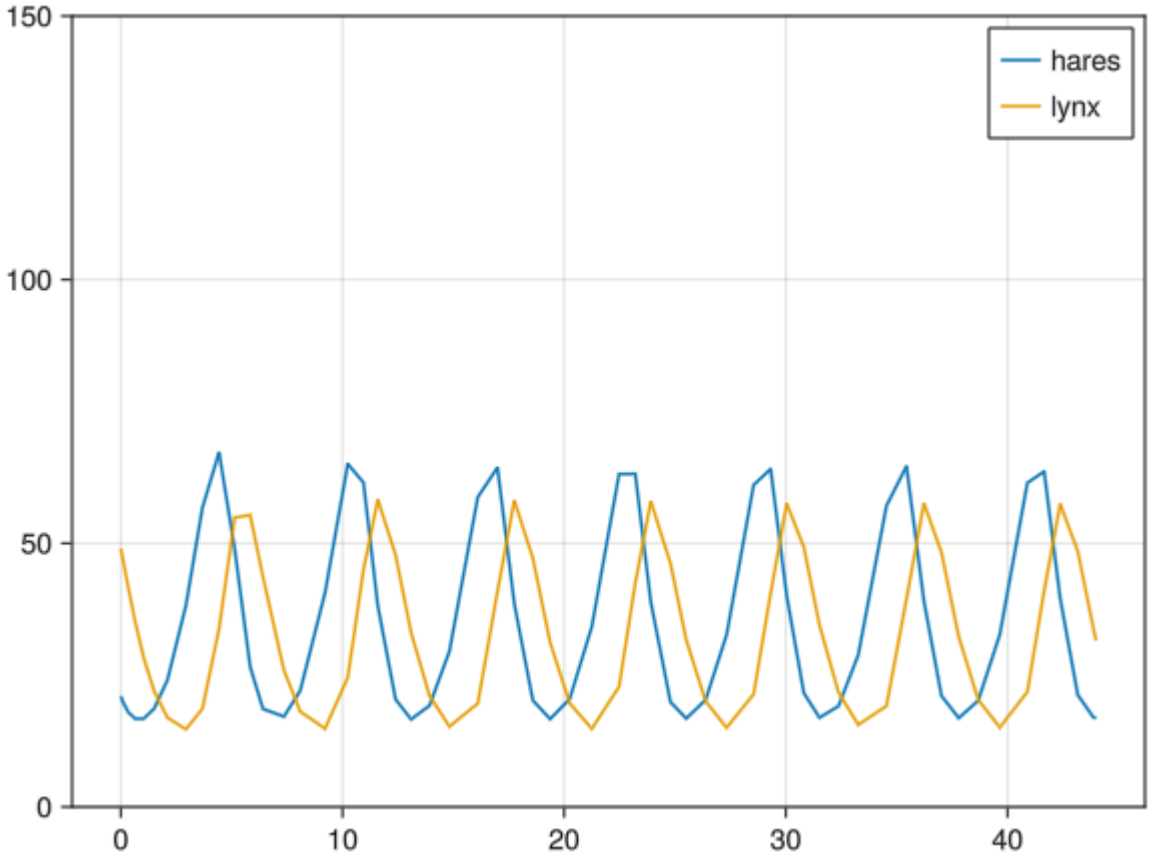
```
begin
    # Simulation interval
    tspan = Float32.((0.0, size(df_train)[1]-1))

    # Initial condition
    u0 = [df_train.hare[1], df_train.lynx[1]]

    # Initial values for the parameters
    p = [ 1.0777069248324076, 0.034059726645187034, 1.040486193025437 ,
    0.028715996648942803] # [α, β, δ, γ]
end
```

Fun interactivity

With Pluto.jl, we can get some interactivity to test out how the values for the parameters affect the final solution!

Click and drag on the values of the parameters...

α= **1.1** β= **0.0** δ= **1.0** γ=a **0.0**



Do note that at certain values, there will be times when either population get very close to zero. This can lead to the differential equation being numerically unstable (aka *stiff*).

In the future, we will default to using ODE solvers designed for treating stifness.

Direct Optimiziation

Our goal is to estimate the 'best' parameters for the Lokta-Volterra system, such that when we solve the differential equations we get trajectories as close as possible to our data.

Model setup

We define an ODE problem with the setup provided by the DifferentialEquations package, with respect to an initial guess.

```
pguess =   [1.0, 0.1, 1.0, 0.1]
```

```
1 pguess = [1.0, 0.1, 1.0 ,0.1]
```

```
ode_problem = ODEProblem with uType Vector{Float32} and tType Float32. In-place: true
              timespan: (0.0f0, 44.0f0)
              u0: 2-element Vector{Float32}:
               21.0
               49.0
```

```
1 # Setup the ODE problem
2 ode_problem = ODEProblem(lotka_volterra!, u0, tspan, pguess)
```

Then, we need to define a criteria for *fitness*, a **loss function**.

$$\mathcal{L}(\alpha, \beta, \delta, \gamma) = \sum_{i=1}^{57} \|(\hat{x}_i, \hat{y}_i) - (x(t_i), y(t_i))\|_2^2$$

Where $\hat{x}, \hat{y}$ represent our date points and $x(t), y(t)$ are the solutions to the Lokta-Volterra systems with the given parameters.

Note that at every call of `direct_loss`, we solve the differential equation. We use the `Rosenbrock23` solver, which should work well for stiff ODEs.

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

```
1  begin
2      true_values = transpose(Matrix(df_train))
3      function direct_loss(newp)
4          newprob = remake(ode_problem, p = newp)
5          sol = solve(newprob, Rosenbrock23(autodiff=false), saveat = 1)
6
7          loss = try sum(abs2, sol .- true_values)
8          catch e
9                  return Inf, sol
10         end
11         return loss, sol
12     end
13 end
```

We now define some utility functions.

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

```
function plot_trajectories(y, pred)
    n = size(y, 2)
    m = size(pred,2)
    years = rawdata.year
    fig_node = Figure()
    ax3 = GLMakie.Axis(fig_node[1,1], xlabel="Year", ylabel="Thousands of
    Animals")
    GLMakie.ylims!(ax3,0,150)

    GLMakie.lines!(ax3, years[1:m], pred[1,:], label="x(t)")
    GLMakie.lines!(ax3, years[1:m], pred[2,:], label="y(t)")
    GLMakie.scatter!(ax3, years[1:n], y[1, :];
    label = "hare", color=:green)
    GLMakie.scatter!(ax3, years[1:n], y[2, :];
    label = "lynx", color=:red2)
    axislegend(ax3; position=:rt)
    display(fig_node)
    return fig_node
end
```

```
callback = #11 (generic function with 1 method)
```

```
callback = function (p, l, sol)
    println(l)
    # Tell Optimization.solve to not halt the optimization. If return true, then
    # optimization stops.
    return false
end
```

Training the model

```
OptimizationProblem. In-place: true
u0: 4-element Vector{Float64}:
 1.0
 0.1
 1.0
 0.1
```

```
begin
    optf = Optimization.OptimizationFunction((x, p) -> direct_loss(x),
    Optimization.AutoForwardDiff())
    optprob = Optimization.OptimizationProblem(optf, pguess)
end
```

```
optimized_params =  [1.0, 0.1, 1.0, 0.1]
```

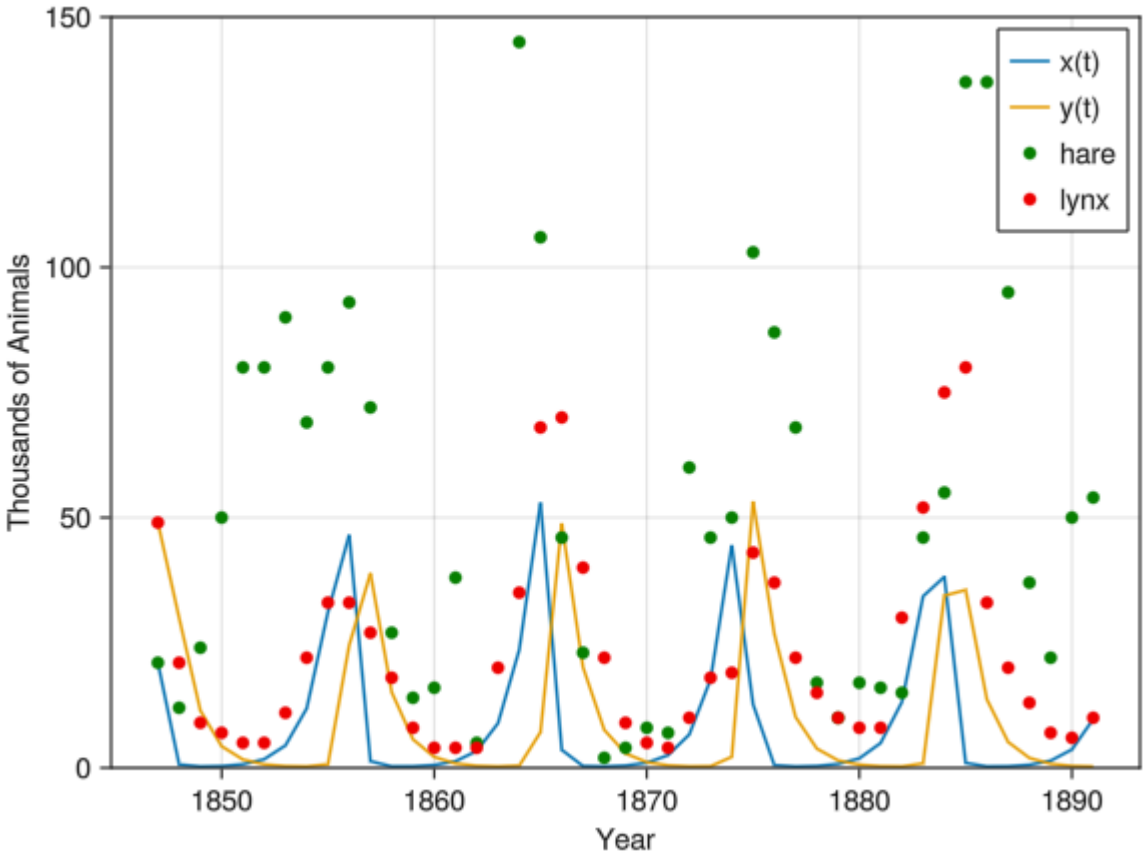
```
optimized_params = if dotrain
    Optimization.solve(optprob, PolyOpt(), callback = callback, maxiters = 500)
    else  pguess end
```

```
[1.0, 0.1, 1.0, 0.1]
```

```
dotrain ? optimized_params.u : optimized_params
```

When the nº of iterations is too large, the salver always ends up printing warnings. The resulting

"optimized" parameters yield the following trajectories:




```
begin
        newprob = remake(ode_problem, p = optimized_params)
        sol = solve(newprob, Rosenbrock23(autodiff=false), saveat = 1)
        pred  = mapreduce(permutedims, vcat, sol.u)'
        plot_trajectories(true_values, pred )
end
```

Neural ODE

For this section, we are going to attempt learning the dynammics of our time series, without assuming it has the structure of a Lokta-Volterra system. Instead, we will model the derivatives of the hare and lynx populations with a Neural Network.

$$\begin{cases} x'(t) = NN_1(x, y, \theta) \\ y'(t) = NN_2(x, y, \theta) \end{cases}$$

This setup is known as a "Neural Ordinary Differential Equation" (Neural ODE or NODE).

Model setup

We will use `DiffEqFlux.jl` to declare our NODEs, and the NN framework `Lux.jl` instead of `Flux.jl` as it is the prefered choice for NeuralODEs according to the SCIML documentation.

We start by defining a custom wrapper function `neural_ode` that will initialize our NeuralODE and its parameters, algong with other utility functions.

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

```
function neural_ode(t, data_dim)
    f = Lux.Chain(
        Lux.Dense(data_dim, 5, Lux.tanh),
        Lux.Dense(5, data_dim)
    )

    node = NeuralODE(
        f, extrema(t), Tsit5(),
        saveat=t,
        abstol=1e-9, reltol=1e-9
    )
    θ, state = Lux.setup(rng, f)

    return node, ComponentArray(θ), state
end
```

```
predict (generic function with 3 methods)
```

```
predict(y0, t, p=nothing, state=nothing) = begin
    node, p_random, state_random = neural_ode(t, length(y0))
    if p === nothing p = p_random end
    if state === nothing state = state_random end
    ŷ = Array(node(y0, p, state)[1])
end
```

Testing out the model

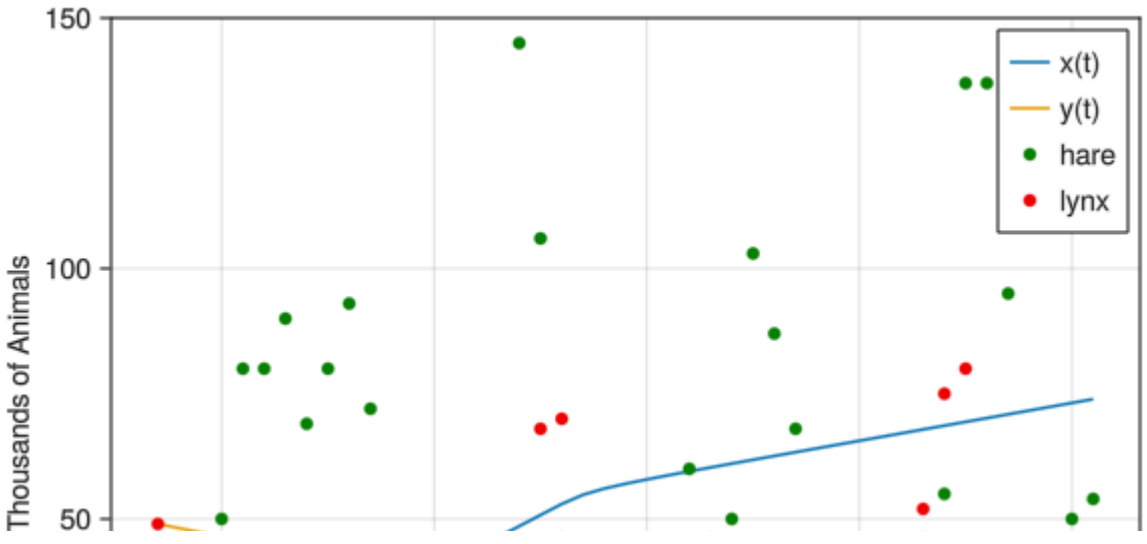
The parameters of the Neural Network are initialized randomly, therefore the initial trajectories will not make much sense. The following code shows how this model is used.

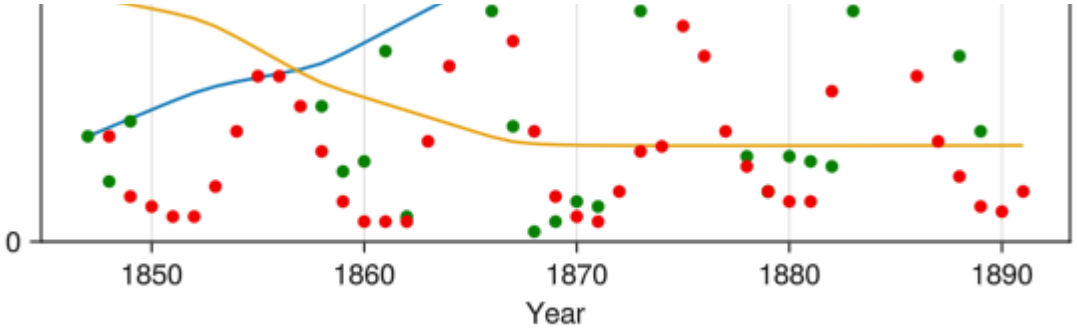
```
MersenneTwister(1000)
```

```
begin
    u_train = transpose(Matrix(df_train))
    t_train = Float32.(Array(0.0:Float32(size(df_train)[1]-1)))
    train_years = rawdata.year[1:45]
    rng = Random.MersenneTwister(1000)
end
```

```
prediction =
2×45 Matrix{Float32}:
 21.0  22.7704  24.5401  26.3057  28.048   …  71.6467  72.402   73.1573  73.9125
 49.0  48.1606  47.32    46.4732  45.5898     19.1799  19.1846  19.1892  19.1939
```

```
prediction = predict(u0, t_train)
```


150
100
50
Thousands of Animals
x(t)
y(t)
hare
lynx


0
1850
1860
1870
1880
1890
Year

```
plot_trajectories(u_train, prediction)
```

Training the model

For this setup, we will use a `callback` function to print the values of the loss function at every epoch of the training method `solve`.

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

```
function train(node, p, st, y, maxiters, lr)
    pred(p) = Array(node(u0, p, st)[1])
    loss(p) = sum(abs2, pred(p) .- y)/length(u_train)
    callback(p, l) = begin
        display(l)
        return false
    end
    adtype = Optimization.AutoZygote()
    optf = OptimizationFunction((p, _ ) -> loss(p), adtype)
    optprob = OptimizationProblem(optf, p)
    res = solve(optprob, Adam(lr), maxiters=maxiters, callback=callback)
    #res = solve(optprob, BFGS(), maxiters=maxiters, callback=callback)
    return res.u, st, pred(p)
end

```

Using only the first 5 data points (goes wrong)

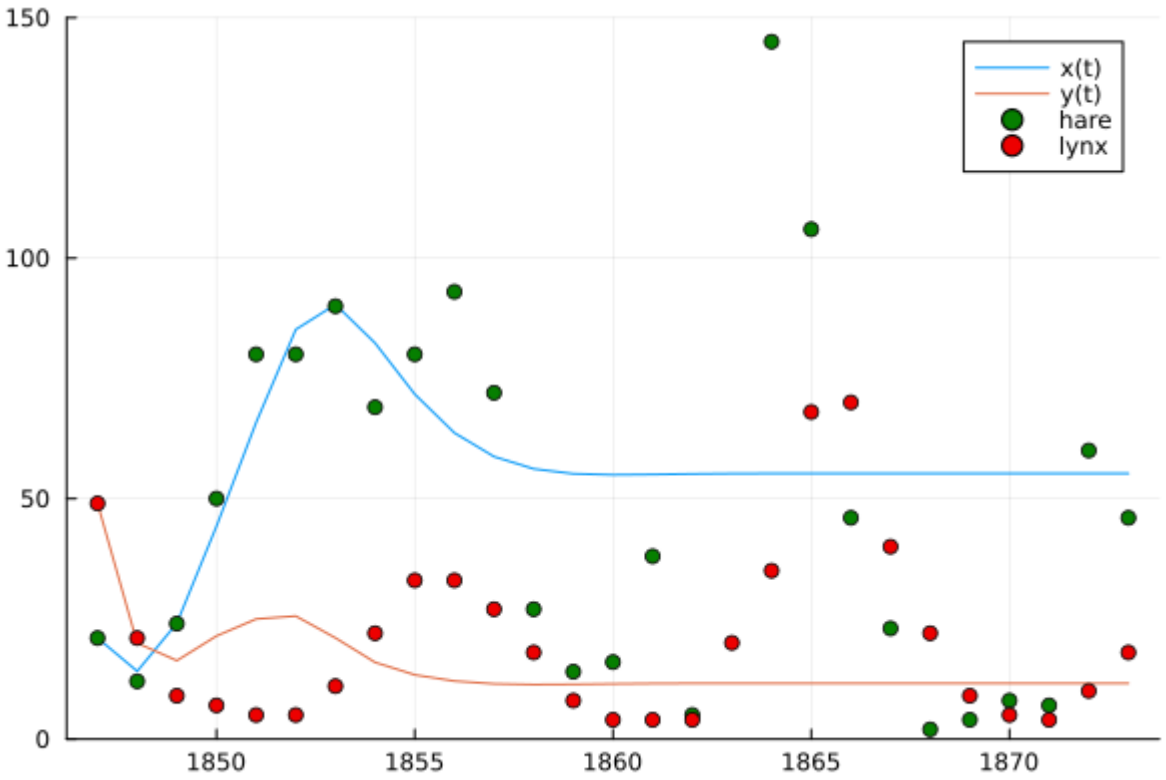
☐

Activate the Button above to launch only this training.

```
if train_small_node
    sample_size = 10
    node_small, θ, st = neural_ode(t_train[1:sample_size], 2)
    println("Loss values")
    θ, st, _ = train( node_small, θ, st, true_values[:, 1:sample_size], 50, 1e-2)

    plot_trajectories(true_values, predict(u0, t_train, θ, st))
else
    md"Activate the Button above to launch only this training."
end
```

Our implementation seems to be faulty at some point, as it should not take this long. Further exploration of SCIML librararies would be needed.

To speed things up, we run a this model on a strong computer in a single Julia script file outside of this notebook (NewNODE.jl, can be found in the repository). The results obtained were quite surprising, as the orbits seem to converge to an attractor fixed point.


150
100
50
0
1850
1855
1860
1865
1870
x(t)
y(t)
hare
lynx

Conclusions

The MLP model was successfull in the series forecast, and quite easy to implement compared to the ODE models. However, we noted that the results were somewhat *erratic*, in the sense that they were very dependent on the randomized initialization of the parameters.

From the beggining, we were aware that it is impossible to find values for the parameters (α , β, δ, γ) producing Lokta-Volterra ODE perfectly fitting our data. What we did not expect is that the potential *stiffness* of ODE would end up breaking the Optimization scheme.

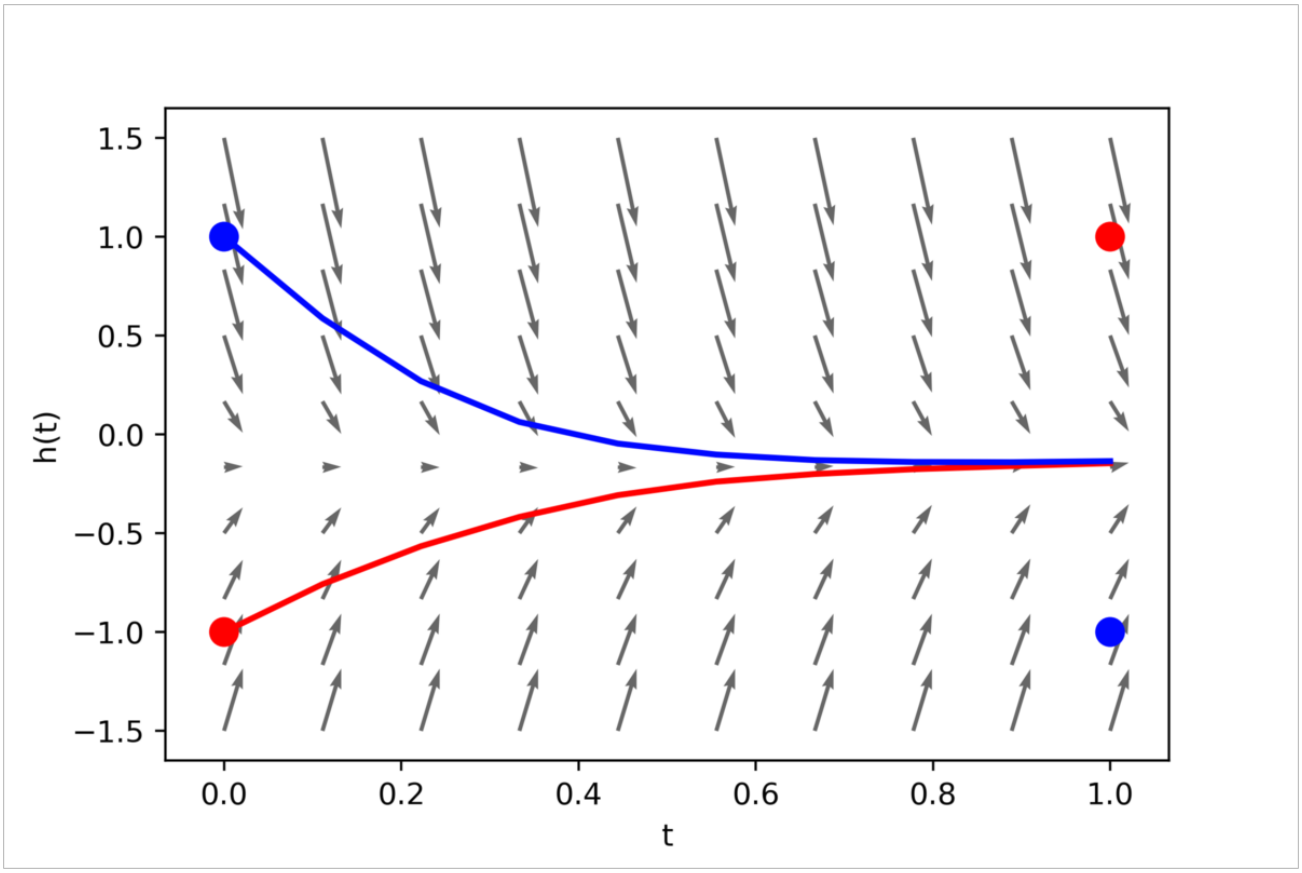
Our NeuralODE performed rather disappointingly. Our implementations was very slow, compared to the other methods, and still we were not able to find a convincing result.

Future work

We believe that the issue comes from the fact that NeuralODE do not (by default) hold the same Universal Approximation Properties as Neural Networks. This has been studied by several authors, the latest work to our knowledge is by Teshima et. al..

Augmentation

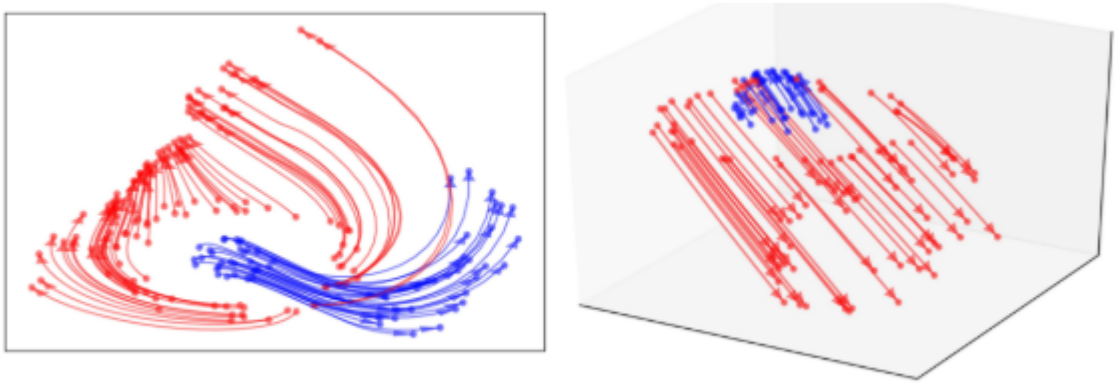
A clear intuition on why that is the case can be found on the image below, representing a one-dimensional NeuralODE h'(t) = f(h(t), t). It is impossible to find a continuous vector field f that would make any pair of trajectories intersect.




```
load("./images/node-non-universality.png")
```

The simplest solution to this problem was proposed by Dupont et. al.. One can *augment* the dimension of the state, giving enough space to the trajectories to cross without intersecting. This idea is actually very similar to kernel feature spaces.

The image belows shows an example of this for a classification task, where NeuralODEs try to find trajectories separating two levels of data points.



Neural ODE Augmented Neural ODE

We will be implementing that in the future.

Hybrid Model

We plan on implementing a model that combines the NeuralODE with the direct ODE parameter optimization. In this case, the Neural Network will represent the dynammics of the difference between a pure Lokta-Volterra ODE and our 'noisy' dataset.

$$\begin{cases} x'(t) = \alpha x(t) - \beta x(t) y(t) + NN_1(x, y, \theta) \\ y'(t) = -\gamma y(t) + \delta x(t) y(t) + NN_2(x, y, \theta) \end{cases}$$

Our training scheme will try to learn both $\alpha, \beta, \gamma, \delta$ and the Neural Network parameters θ.

We believe that model will be able to approximate the trajectory, as it has already been studied by Rackauckas et.al in their paper "Universal Differential Equations for Machine Learning".

We have already tried to implement it, but we faced some issues. Our goal is to fix the bugs and present the model for the following delivery.