Report

This is a report on machine learning for physical model. For simplicity, this report will only focus on undamped harmonic oscillator(abbr. udho).

The report consists of four main parts: physical model, neural ODE, structured neural ODE, parameters estimation.

Physical Model

Physical models are some ODEs that describe dynamic systems. The ODEs are used to generate the data that the machine learning models attempt to fit.

Undamped harmonic oscillator is a classical hamiltonian system, in which there is no energy dissipation.

The total energy of the system can be represented by the Hamiltonian

$$H(q,p) = q\frac{1}{2c}q + p\frac{1}{2m}p$$

,

where c is the spring compliance, q is the displacement of the spring, m is the mass, p is the momentum. The state of the system (q,p) moves along the symplectic gradient

$$\dot{q} = \frac{\partial H}{\partial p}, \quad \dot{p} = -\frac{\partial H}{\partial q}$$

.

In order to plot the physical model, first you need to use some packages.

```
begin
    ## using package
    using DifferentialEquations    ### for ODE Problem
    using Plots    ### for plot
end
```

ODE function

Define ODEs to describe the dynamic system udho. $u = [q, p]$ is the state of the system and $du = [\dot{q}, \dot{p}]$ is the gradient of the state. Parameters are given in a set $params = [m, c]$.

```
md"""
### ODE function
Define ODEs to describe the dynamic system udho. $u=[q,p]$ is the state of the system and $du=[q̇,ṗ]$ is the gradient of the state. Parameters are given in a set $params=[m,c]$.
"""
```

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

```
## define ODE function
function ODEfunc_udho(du,u,params,t) ### du=[q̇,ṗ] u=[q,p], params=[m,c]
  ## conversion
  q, p = u ### u[1] = q, u[2] = p
  m, c = params ### params[1] = m, params[2] = c
  ## ODEs
  du[1] = p/m
  du[2] = -q/c
end
```

Initialization

Set initial condition

```
md"""
### Initialization
Set initial condition
"""
```

```
u0 = [1.0, 1.0]
```

```
u0 = [1.0; 1.0]
```

Set initial parameters

```
init_params = [1.5, 1.0]
```

```
init_params = [1.5, 1.0]
```

Set timespan and timesteps

```
tspan = (0.0, 20.0)
```

```
tspan = (0.0, 20.0) ### start, stop
```

```
tsteps = 0.0:0.020020020020020002:20.0
```

```
tsteps = range(tspan[1], tspan[2], length = 1000) ### start, stop, size of dataset
```

ODE problem

In order to solve the ODE problem, the package DifferentialEquations.jl is used. A function called ODEProblem() in DifferentialEquations.jl constructs a ODE problem.

```
md"""
### ODE problem
In order to solve the ODE problem, the package DifferentialEquations.jl is used.
A function called ODEProblem() in DifferentialEquations.jl constructs a ODE
problem.
"""
```

```
prob = ODEProblem with uType Vector{Float64} and tType Float64. In-place: true
       timespan: (0.0, 20.0)
       u0: 2-element Vector{Float64}:
        1.0
        1.0
```

```
prob = ODEProblem(ODEfunc_udho, u0, tspan, init_params) ### ODE function, initial condition, timespan, initial parameters
```

Then another function called solve() solves the ODE problem. Value1 and value2 are q and p.

sol =

	timestamp	value1	value2
1	0.0	1.0	1.0
2	0.02002	1.01321	0.979847
3	0.04004	1.02615	0.959433
4	0.0600601	1.03882	0.938762
5	0.0800801	1.05121	0.91784
6	0.1001	1.06332	0.896673
7	0.12012	1.07515	0.875267
8	0.14014	1.08668	0.853626
9	0.16016	1.09793	0.831758
10	0.18018	1.10889	0.809667

more

```
sol = solve(prob, Tsit5(), saveat = tsteps) ### ODE problem, algorithm, timesteps
```

Data handling

For convenient, convert the result into array form.

```
md"""
### Data handling
For convenient, convert the result into array form.
"""
```

```
ode_data =
2×1000 Matrix{Float64}:
 1.0  1.01321   1.02615   1.03882   1.05121  …  -1.28558   -1.28741   -1.28889
 1.0  0.979847  0.959433  0.938762  0.91784      -0.149732  -0.123975  -0.0981859
```

```
ode_data = Array(sol)
```

Pick out the first and second row.

```
x_axis_ode_data =
 [1.0, 1.01321, 1.02615, 1.03882, 1.05121, 1.06332, 1.07515, 1.08668, 1.09793, 1.10889,
```

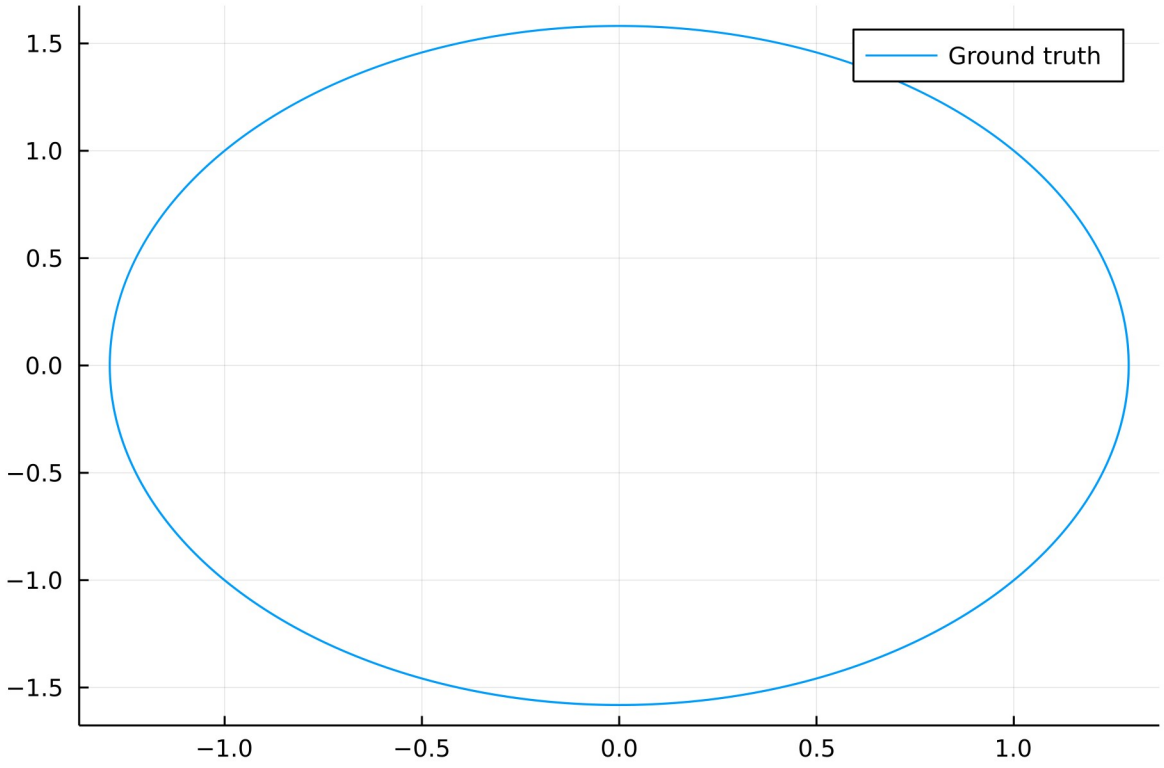
```
x_axis_ode_data = ode_data[1,:]
```

```
y_axis_ode_data =
 [1.0, 0.979847, 0.959433, 0.938762, 0.91784, 0.896673, 0.875267, 0.853626, 0.831758, 0
```

```
y_axis_ode_data = ode_data[2,:]
```

Plot it out, with q as x axis and p as y axis.

```
plt =
```




```
plt = plot(x_axis_ode_data, y_axis_ode_data, label="Ground truth")
```

Neural ODE

The model generated by neural ODE is considered as the baseline model. The baseline model is used to benchmark the machine learning results.

In hamiltonian dynamics, the states of a system (q, p) are represented as points in the phase space. Let $x = [q, p]$, $dx/dt = RHS(x)$ where the RHS could be replaced with a neural network. In this case, regardless of the system, the neural network always tries to approximate the system without any physics priors.

Use some packages

```
begin
    ## using package
    using DiffEqFlux    ### for NeuralODE and sciml_train
    using BenchmarkTools    ### for macro @btime
end
```

Neural network

Construct a neural network with FastChain() or Chain(). In FastChain() you can optionally guess a function to improve the performance. This option is only recommanded when you can guess a very precise function, otherwise it works even worse.

```
md"""
### Neural network
Construct a neural network with FastChain() or Chain(). In FastChain() you can optionally guess a function to improve the performance. This option is only recommanded when you can guess a very precise function, otherwise it works even
worse.
"""
```

```
NN =
(::DiffEqFlux.FastChain{Tuple{DiffEqFlux.FastDense{typeof(NNlib.tanh_fast), DiffEqFlux
```

```
## Make a neural network with a NeuralODE layer, where FastChain is a fast neural net structure for NeuralODEs
NN = FastChain(FastDense(2, 20, tanh), ### Multilayer perceptron for the part we don't know
                  FastDense(20, 10, tanh),
                  FastDense(10, 2))
```

Neural ODE problem

Construct a neural ODE problem by using NeuralODE().

```
md"""
### Neural ODE problem
Construct a neural ODE problem by using NeuralODE().
"""
```

```
prob_neuralode = NeuralODE()
```

```
prob_neuralode = NeuralODE(NN, tspan, Tsit5(), saveat = tsteps) ### neural network, timespan, algorithm, timesteps
```

To start the model training, we need two things. One is the loss function, another one is the parameters that we update in each train.

The parameters can directly obtained from the neural ODE problem, as they are already initialized when we first construct the neural ODE problem. Of course, alternatively, we can also initial it by hand: initial_params(NN).

Parameters

Get the parameters from the neural ODE problem.

```
md"""
To start the model training, we need two things. One is the loss function, another one is the parameters that we update in each train.

The parameters can directly obtained from the neural ODE problem, as they are already initialized when we first construct the neural ODE problem. Of course, alternatively, we can also initial it by hand: initial_params(NN).

### Parameters
Get the parameters from the neural ODE problem.
"""
```

```
neural_params =
```

```
[-0.163792, -0.430936, 0.288539, -0.0688391, -0.0929634, 0.200186, -0.388412, 0.15549
```

```
### get the parameters prob_neuralode.p in from neural network
neural_params = prob_neuralode.p
```

Loss function

Now we need to construct the loss function.

Firstly, give the neural ODE problem some information, i.e. the initial condition and neural parameters. This allows the neural network to return predicted data.

```
md"""
### Loss function
Now we need to construct the loss function.

Firstly, give the neural ODE problem some information, i.e. the initial condition and neural parameters. This allows the neural network to return predicted data.
"""
```

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

```
## Array of predictions from NeuralODE with parameters p starting at initial condition u0
function predict_neuralode(p)
    Array(prob_neuralode(u0, p)) ### initial condition, neural parameters
end
```

Secondly, compute L2 loss function.

```
md"""
Secondly, compute L2 loss function.
"""
```

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

```
## L2 loss function
function loss_neuralode(p)
      pred_data = predict_neuralode(p)
      loss = sum(abs2, ode_data .- pred_data) # Just sum of squared error, without mean.
      return loss, pred_data
  end
```

Training

Now we got the two things needed for training.

Before training, construct a callback function to plot the data in each train.

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

```
## Callback function to observe training
callback = function(p, loss, pred_data)
    ### plot original and prediction data
    println(loss)
    x_axis_pred_data = pred_data[1,:]
    y_axis_pred_data = pred_data[2,:]
    plt = plot(x_axis_ode_data, y_axis_ode_data, label="Ground truth")
    plot!(plt,x_axis_pred_data, y_axis_pred_data, label = "Prediction")
    display(plot(plt))
    return false
end
```

Train the model.

```
## In sciml_train, optimizer chain of ADAM -> BFGS is used by default
@btime DiffEqFlux.sciml_train(loss_neuralode, neural_params, cb = callback)  ### loss function, neural parameters, callback function
```

Structured Neural ODE

In previous part, the baseline model learned the state of the system $(\dot{q}, \dot{p})$. The whole of the RHS was seen as a neural network. However, this method requires batches of training data and very long training time. Moreover, the baseline model lacks interpretability since it is a pure black box model. Humans cannot intuitively understand how machine obtains these models. With these considerations some works use hamiltonian-based models rather than black box models.

Hamiltonian-based model also known as Hamiltonian Neural Networks(HNNs). The main purpose of HNNs is to endow neural networks with physics priors. In fact, if the system is not conserve, it has another upper-level name called physics-informed neural network, which is a method to solve ODEs with neural networks, regardless of energy conservation.

In this section, the learning object is the hamiltonian. To compute the loss function, the key is to compare the truth and the model's estimation of the gradient of the neural network $(\frac{\partial H}{\partial p}, \frac{\partial H}{\partial q})$.

However, DiffEqFlux.sciml_train() doesn't seem to support structured neural ODE.

Compute the gradient of the neural network.

```
dNN_dq =  [0.253466, 0.454336]
```

```
## ∂H/∂q
dNN_dq = gradient(u -> NN(u, initial_params(NN))[1], u0)[1]
```

```
dNN_dp =  [0.288385, 0.0443018]
```

```
## ∂H/∂p
dNN_dp = gradient(u -> NN(u, initial_params(NN))[2], u0)[1]
```

Parameters estimation

It is assumed that the ODEs of the system are completely known, except for some scalar parameters. In this case, the gradient descent algorithms can be applied in parameters estimation.

Use some packages

```
md"""
## Parameters estimation
It is assumed that the ODEs of the system are completely known, except for some
scalar parameters. In this case, the gradient descent algorithms can be applied
in parameters estimation.

Use some packages
"""
```

```
begin
    ## using package
    using DiffEqParamEstim    ### a package for simplified parameter estimation
    with DifferentialEquations.jl
    using RecursiveArrayTools    ### for VectorOfArray
    using LeastSquaresOptim    ### for LeastSquaresProblem and optimize
    using LinearAlgebra    ### for norm()
end
```

Data handling

In the part physical model, we already solved the ODE problem and got the result sol = solve(prob, Tsit5(), saveat = tsteps).

Convert timesteps into vector.

```
t =
  [0.0, 0.02002, 0.04004, 0.0600601, 0.0800801, 0.1001, 0.12012, 0.14014, 0.16016, 0.180
```

```
t = collect(range(0, stop=20, length=1000))
```

Randomize the data by adding some noise to the result.

```
randomized = VectorOfArray{Float64,2}:
             1000-element Vector{Vector{Float64}}:
              [1.0297792818807199, 1.0159872243432366]
              [1.008250718522352, 0.9752208036178968]
              [1.0108958704136712, 0.9740060715152989]
              [1.0275073822360252, 0.9426129239631921]
              [1.0868313194736854, 0.9438710959581514]
              [1.0600840636990427, 0.9313212973989856]
              [1.072758600798377, 0.852207712044719]
              ⋮
              [-1.2778895641189676, -0.2500342382798546]
              [-1.2528176117511172, -0.23592376450190317]
              [-1.2799038686033892, -0.18986908396827684]
              [-1.2542189616606974, -0.15965992869486748]
              [-1.2973060395047051, -0.14080918404429785]
              [-1.2805017438350161, -0.09255442995828767]
```

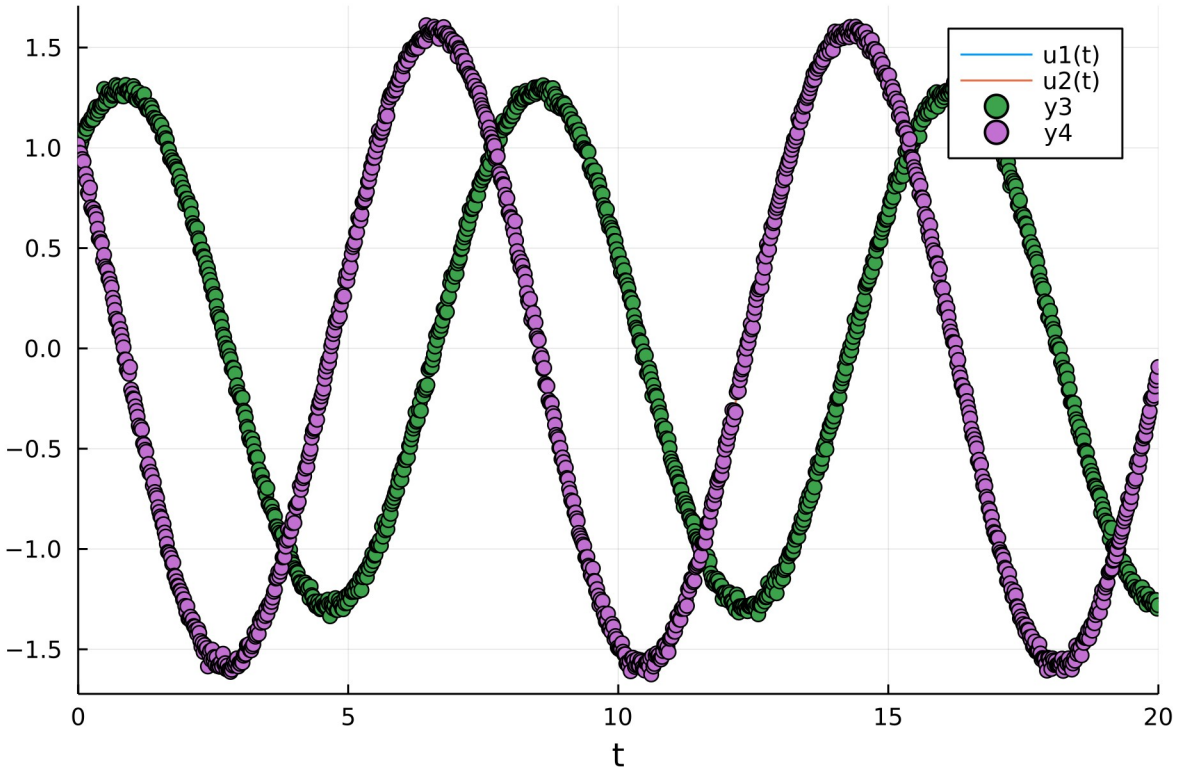
```
randomized = VectorOfArray([(sol(t[i]) + .02randn(2)) for i in 1:length(t)]) ##
sol(t[i]) means u(t) at the timepoint t=i
```

Convert the randomized data from a vector into an array.

```
data =
2×1000 Matrix{Float64}:
 1.02978  1.00825   1.0109    1.02751   1.08683   …  -1.25422  -1.29731   -1.2805
 1.01599  0.975221  0.974006  0.942613  0.943871     -0.15966  -0.140809  -0.0925544
```

```
data = convert(Array,randomized) ## convert the randomized data from a vector
into an array
```

Plot it out.




```
begin
    plot(sol)
    scatter!(t,data')
end
```

Estimation

Build a loss function so that we can optimize it by using LeastSquaresOptim.jl in the following.

```
loss_function = #49 (generic function with 1 method)
```

```
## build a LeastSquaresOptim object
loss_function = build_lsoptim_objective(prob, t, data, Tsit5()) ### ODE problem, timesteps(vector form), randomized data(array form), algorithm
```

Guess parameters

```
guess_params =   Float64[
                    1:  1.5
                    2:  0.8
               ]
```

```
## guess an initial set of parameters
guess_params = [1.5, 0.8]
```

Use optimize() to calculate the gradient of the loss function and try to find the minimum.

```
res = Results of Optimization Algorithm
       * Algorithm: Dogleg
       * Minimizer: [1.5013458348140098,0.9991910786128674]
       * Sum of squares at Minimum: 0.801292
       * Iterations: 8
       * Convergence: true
       * |x - x'| < 1.0e-08: false
       * |f(x) - f(x')| / |f(x)| < 1.0e-08: true
       * |g(x)| < 1.0e-08: false
       * Function Calls: 9
       * Gradient Calls: 6
       * Multiplication Calls: 44
```

```
## using LeastSquaresOptim.optimize!
res = optimize!(LeastSquaresProblem(x = guess_params, f! = loss_function,
                output_length = length(t)*length(prob.u0)),
                LeastSquaresOptim.Dogleg(LeastSquaresOptim.LSMR()))
```

Pick out the minimum.

```
est_params =   [1.50135, 0.999191]
```

```
est_params = res.minimizer
```

Print result.

```
println("Ground truth: $(init_params)\nEstimated parameters: $(round.(est_params,
sigdigits=5))\nError: $(round(norm(est_params - init_params) ./ norm(init_params)
.* 100, sigdigits=3))%")
```

```
Ground truth: [1.5, 1.0]
Estimated parameters: [1.5013, 0.99919]
Error: 0.0871%
```