Subpace Inference for Neural ODE's

Manu Francis

07-02-2021

---

1 Introduction

Ordinary Differential Equations(ODE) and Machine Learning(ML) are used to model any non linear systems. Consider a system with input `x` and output `y`, the system representation using machine learning will be as below:

$$y = ML(x)$$

The training process in machine learning optimises the parameters of Neural Network (NN) and the trained model will be able to predict output for a given input. This machine learning modelling can be used for non linear system by referring universal approximation (UA) theorem. According to UA, with enough layers or parameters ML representation can approximate any nonlinear function sufficiently close, but the machine learning models requires large set of training data for the proper modelling.

Another method for non-linear system modelling is the by using ordinary differential equations. For ODE base representation, knowledge about the structure of the system is necessary. For example, the birth rate of a prey in a habitat is depends on current population.

$$prey_{birthrate} = \alpha * prey_{population}$$

The α is a learnable parameter and the solution of the ODE is $prey_{population}e^{t\alpha}$ and simply it is an exponential relation. This embedding the model structure using ODE's doen't require any training data. Therefore, ODE is helpful for modelling which avoids the data hungry of machinelearning models.

The both modelling methods have its own advantages and disadvantages. Neural ODE$'$s are introduced to model a system by combining machine learning and ODE$'$s together. This method is trying to model ODE representation of a system by using machine learning method instead of

$$y = ML(x)$$

neural ODE's trying to model as

$$y^{'} = ML(x)$$

DiffEqFlux.jl package helps to implement Neural ODE's in Julia.

1.0.1 Example of Neural ODE

Lokta Voltera ODE's is used for the study and it is represented as:

$$x' = \alpha x + \beta xy$$
$$y' = -\delta y + \gamma xy$$

This ODE is solved using DifferentialEquations.jl package and the result is as shown below:

```
using DifferentialEquations
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
u0 = [1.0,1.0]
tspan = (0.0,10.0)
p = [1.5,1.0,3.0,1.0]
prob = ODEProblem(lotka_volterra,u0,tspan,p)
sol = solve(prob)
using Plots
plot(sol)
```

Sometimes we won′t have exact knowledge of complete structure of non linear system to model using ODE′s This case we use Neural ODE′s to model the non linear system and to solve simply like training of Neural Network.

The neural ODE representation of spiral ODE is discussed below:

```
using DiffEqFlux
using Flux
using Flux: Data.DataLoader
using Flux: @epochs
using DifferentialEquations
using Plots
u0 = Float32[2.; 0.]
datasize = 30
tspan = (0.0f0,1.5f0)

function trueODEfunc(du,u,p,t)
    true_A = [-0.1 2.0; -2.0 -0.1]
    du .= ((u.^3)'true_A)'
end
t = range(tspan[1],tspan[2],length=datasize)
prob = ODEProblem(trueODEfunc,u0,tspan)
ode_data = Array(solve(prob,Tsit5(),saveat=t))
dudt = Chain(x -> x.^3,
            Dense(2,50,tanh),
            Dense(50,2))
n_ode = NeuralODE(dudt,tspan,Tsit5(),saveat=t,reltol=1e-7,abstol=1e-9)
ps = Flux.params(n_ode)
```

The prediction of ODE solution with randomly initialized network is shown below:

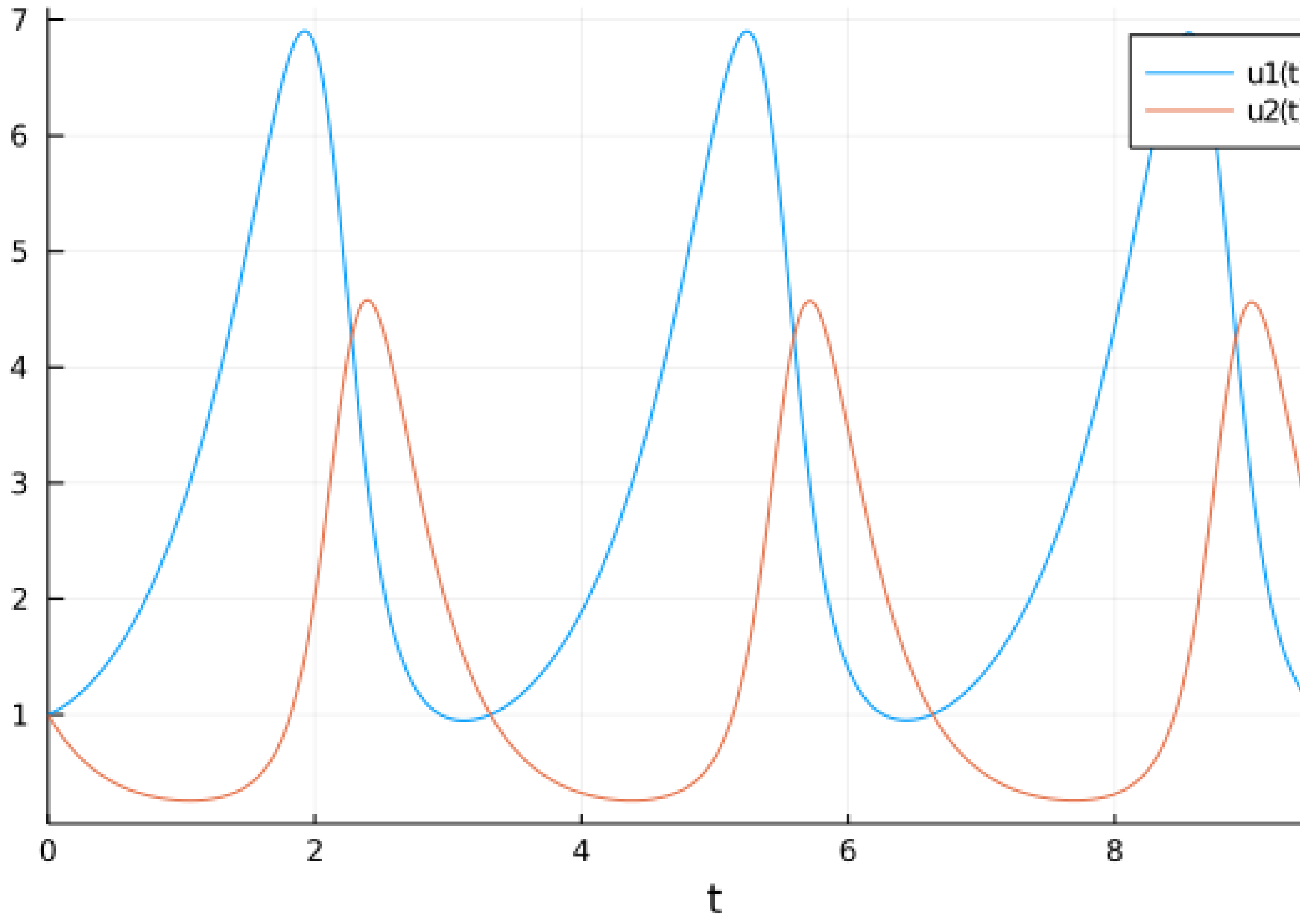

Figure 1: Lokta - Volterra ODE solution

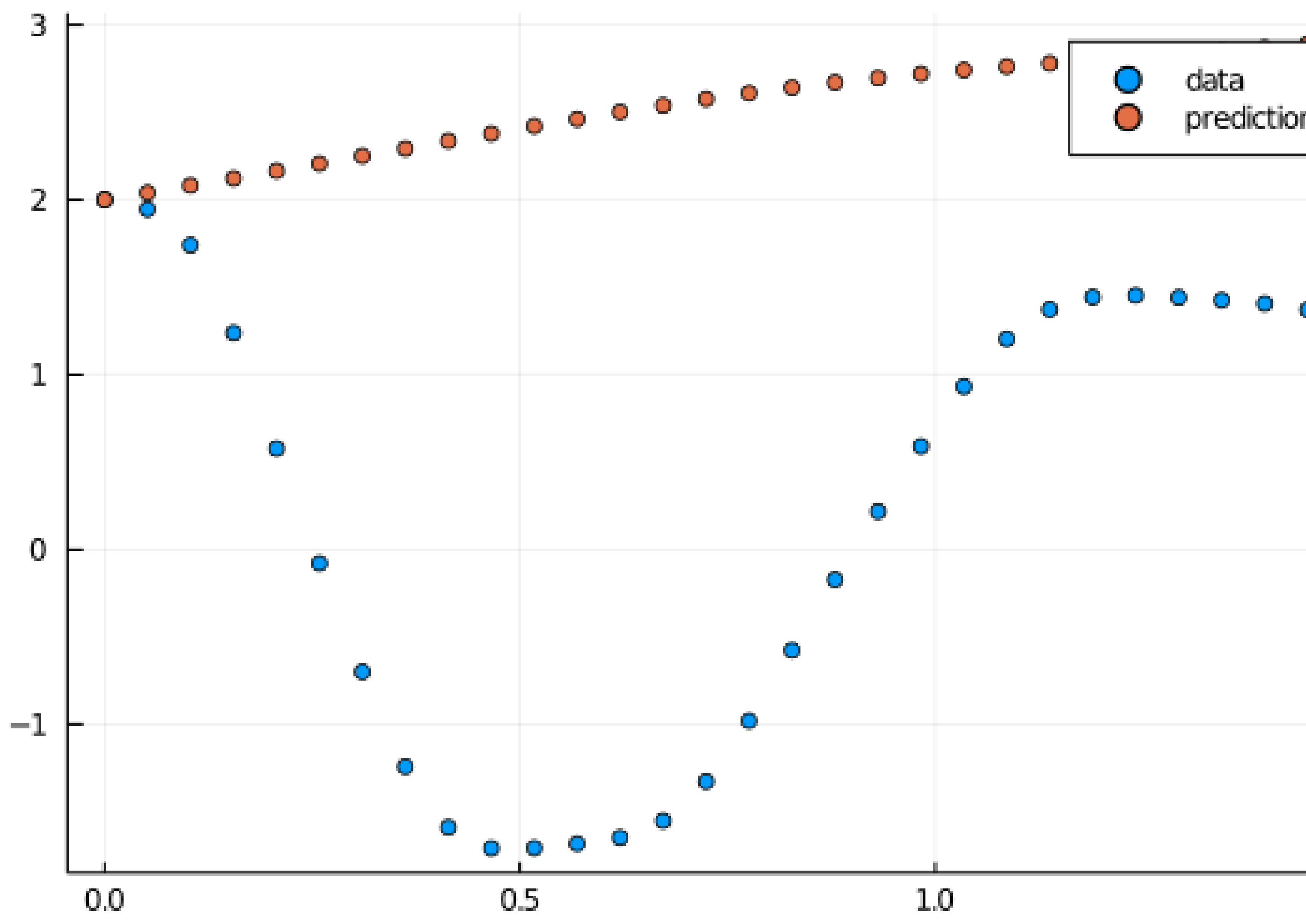

Figure 2: Neural ODE solution before training

```
pred = n_ode(u0) # Get the prediction using the correct initial condition
scatter(t,ode_data[1,:],label="data")
scatter!(t,pred[1,:],label="prediction")
```

Now we can start training the Neural ODE.

```
function predict_n_ode()
  n_ode(u0)
end
loss_n_ode() = sum(abs2,ode_data .- predict_n_ode())
data = Iterators.repeated((), 1000)
opt = ADAM(0.1)
cb = function () #callback function to observe training
  display(loss_n_ode())
  # plot current prediction against data
  cur_pred = predict_n_ode()
  pl = scatter(t,ode_data[1,:],label="data")
  scatter!(pl,t,cur_pred[1,:],label="prediction")
  display(plot(pl))
end

# Display the ODE with the initial parameter values.
cb()
```

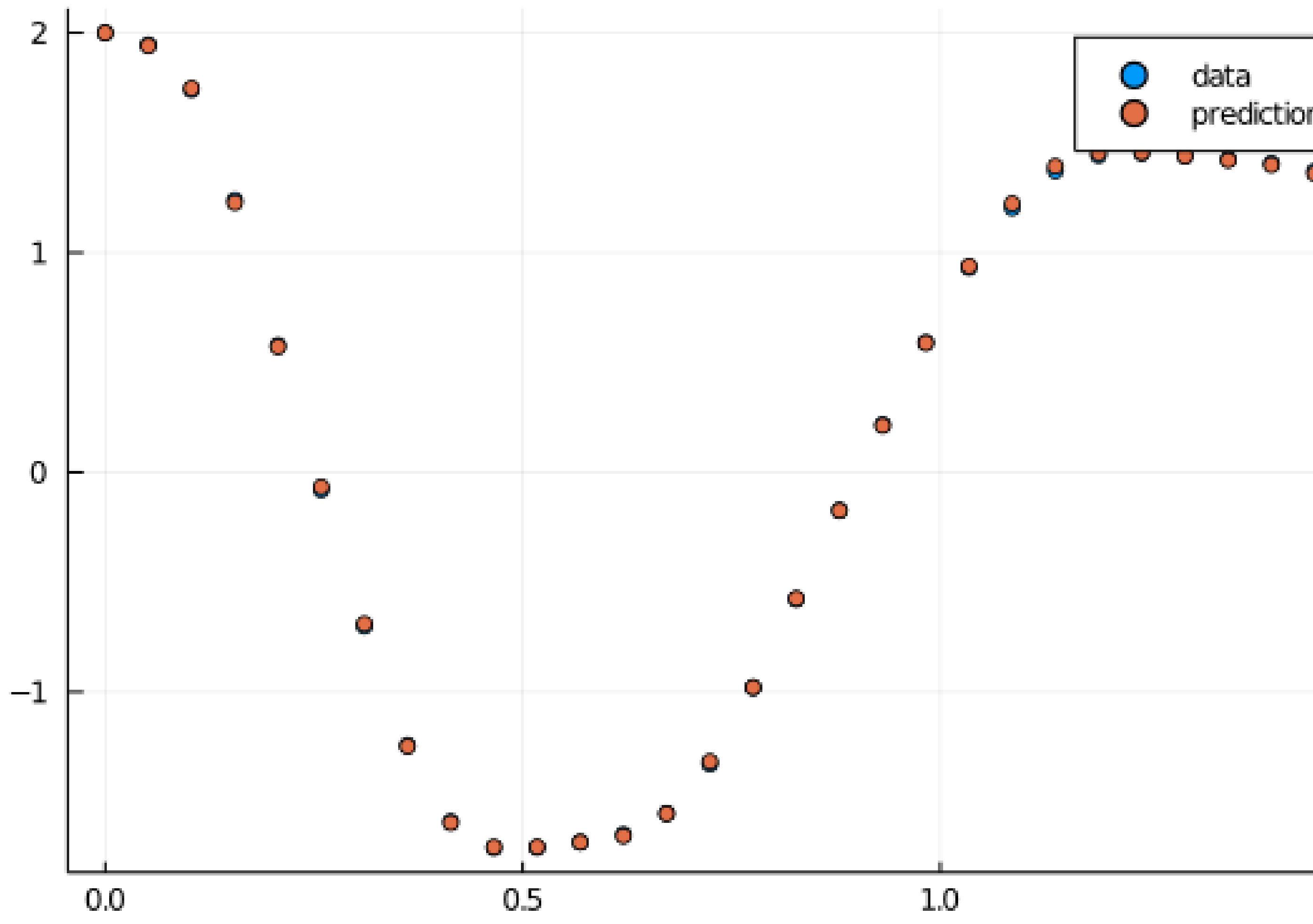

Figure 3: Neural ODE solution after training

```
Flux.train!(loss_n_ode, ps, data, opt, cb = cb)
```

The solution from trained ODE is as below:

1.0.2 Importance of subspace inference in Neural ODE?

The training of neural ODE′s provides slightly various parameters at different iterations. So, there will be an uncertainty in the solution using different neural ODE′s. Bayesian inference methods help to identify the uncertainties of the neural ODE parameters by using Markov Chain Monte Carlo (MCMC) or variational Inference (VI) samples. This Bayesian inference methods will be expensive when the number of parameters in the Neural ODE increases.

Subspace Inference method is introduced to reduce time to calculate the uncertainties in neural ODE′s or Neural networks. This method constructs a subspace of large Neural ODE parameter space and generates Bayesian inference on them. After sampling, this smaller subspace will be transformed to Neural ODE parameter space.

SubspaceInference.jl is a Julia package developed for uncertainty analysis for Neural Networks and Neural ODE′s.

The subspace inference analysis using using SubspaceInference.jl is discussed with spiral

ODE.

```
using Pkg
Pkg.add("https://github.com/efmanu/SubspaceInference.jl")
```

Before defining the ODE, we have to use some packages for inference

```
using Flux, DiffEqFlux;
using BSON: @save;
using BSON: @load;
using Zygote;
using SubspaceInference;
using DifferentialEquations;
using PyPlot;
using Flux: Data.DataLoader;
using Flux: @epochs;
using Distributions;
```

We can define spiral ODE using DifferentialEquations.jl

```
#initial conditions and time span
len = 100

u0 = Array{Float64}(undef,2,len)
u0 .= [2.; 0.]
datasize = 30
tspan = (0.0,1.5)

function trueODEfunc(du,u,p,t)
    true_A = [-0.1 2.0; -2.0 -0.1]
    du .= ((u.^3)'true_A)'
end

t = range(tspan[1],tspan[2],length=datasize)
```

The ODE is solved to generate the training data. In this example, ODE output variables and the solutions is for `30` data points. To train the neural ODE, this ODE will be solved for `len` number of times and a noise of `Normal(0.0, 0.1)` will be added to the solution.

```
ode_data = Array{Float64}(undef, 2*datasize, len)
for i in 1:len
        prob = ODEProblem(trueODEfunc,u0[:,i],tspan)
        ode_data[:,i] = reshape(Array(solve(prob,Tsit5(),saveat=t))', :, 1)
end
ode_data_bkp = ode_data
ode_data += rand(Normal(0.0,0.1), 2*datasize,len);

(fig, f_axes) = PyPlot.subplots(ncols=1, nrows=1)
for i in 1:len
        f_axes.scatter(t,vec(ode_data[1:1:datasize,i]), c="red", alpha=0.3, marker="*",
label ="data with noise")
end
f_axes.plot(t,vec(ode_data_bkp[1:1:datasize,1]), c="red", marker=".", label = "data")
fig.show();
```

The Figure 4 illustrates the different solutions for spiral ODE.

The subspace inference methods use pretrained neural ODE and it is set up and trained as below:

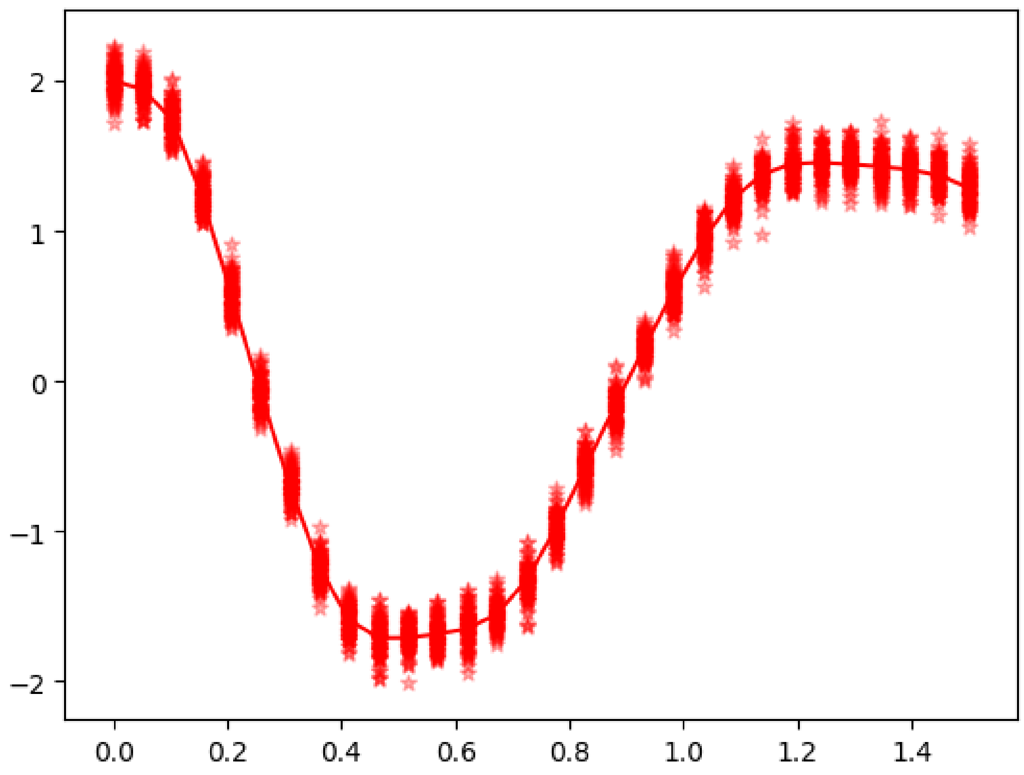

Figure 4: Solution of ODE with noise

```
dudt = Chain(x -> x.^3, Dense(2,15,tanh),
             Dense(15,2))
n_ode = NeuralODE(dudt,tspan,Tsit5(),saveat=t,
        reltol=1e-7,abstol=1e-9);

ps = Flux.params(n_ode);

sqnorm(x) = sum(abs2, x)
L1(x, y) = sum(abs2, n_ode(vec(x)) .-
        reshape(y[:,1], :,2)')+sum(sqnorm, Flux.params(n_ode))/100
#call back
cb = function () #callback function to observe training
  @show L1(u0[:,1], ode_data_bkp[:,1])
end

#optiizer
opt = ADAM(0.1);

#format data
X = u0 #input
Y =ode_data #output

data =  DataLoader(X,Y);

@epochs 4 Flux.train!(L1, ps, data, opt);
cb();
```

The solution of ODE with pretrained network is shown in the Figure 5:

```
(fig, f_axes) = PyPlot.subplots(ncols=1, nrows=1)
pred = n_ode(vec(u0[:,1])) # Get the prediction using the correct initial condition
```

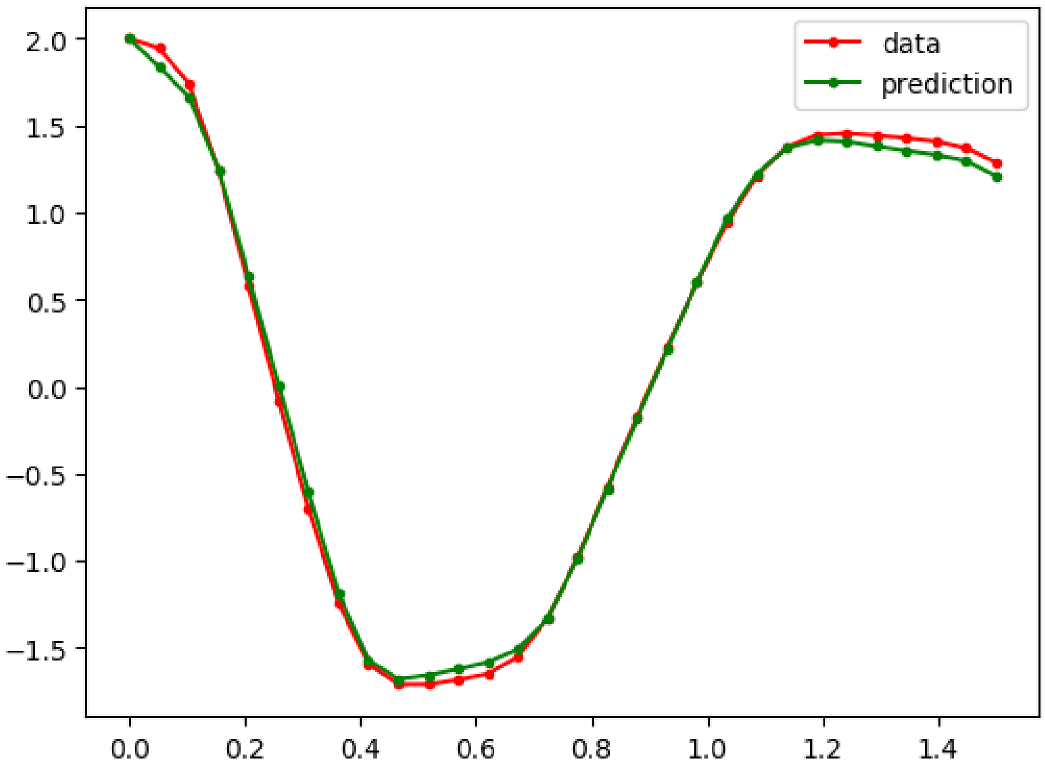

Figure 5: Predicted solution

```
f_axes.plot(t,vec(ode_data_bkp[1:datasize,1]), c="red", marker=".", label = "data")
f_axes.plot(t,vec(pred[1,:]), c="green", marker=".", label ="prediction")
f_axes.legend()
fig.show()
```

1.0.3 Subspace Inference for Neural ODE

We need to modify the loss function for subspace construction because this algorithm updates weight parameters every time and calculate the loss.

```
L1(m, x, y) = sum(abs2, m(vec(x)) .- reshape(y[:,1], :,2)')+sum(sqnorm,
Flux.params(m))/100;
```

The subspace inference is generated for subspace size of `3` with `100` iterations as below. This algorithm generates uncertainties using MH algorithm with subspace with proposal distribution of `0.1`. During inference, the posterior samples of subspace is generated by considering the prior distribution of neural network parameters.

```
T = 1
M = 3
itr = 100
σ_z = 0.1 #proposal distribution

#do subspace inference
chn, lp, W_swa = SubspaceInference.subspace_inference(n_ode, L1, data, opt;
        σ_z = σ_z, itr =itr, T=T, M=M,  alg =:mh);
```

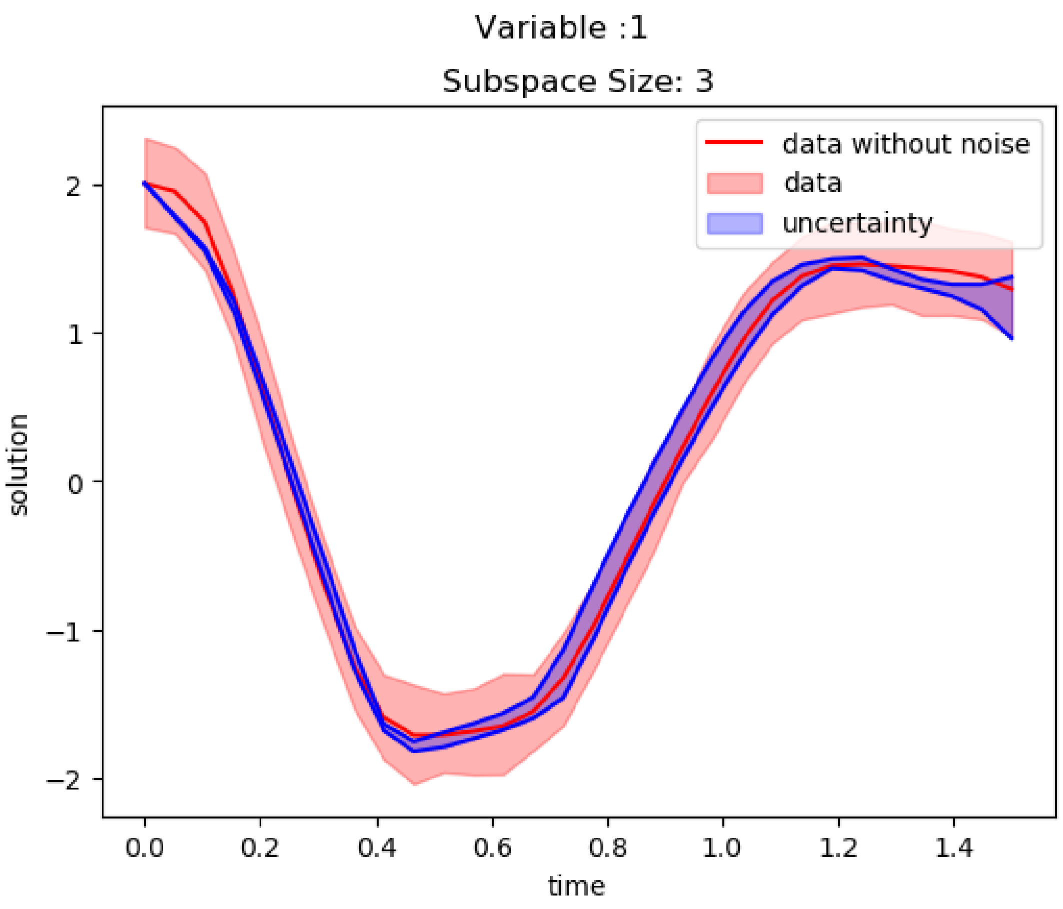

Figure 6: Uncertainty in var 1 solution

```
ns = length(chn)

trajectories = Array{Float64}(undef,2*datasize,ns)
for i in 1:ns
  new_model = SubspaceInference.model_re(n_ode, chn[i])
  out = new_model(u0[:,1])
  reshape(Array(out)',:,1)
  trajectories[:, i] = reshape(Array(out)',:,1)
end

all_trajectories = Dict()
all_trajectories[1] = trajectories
title = ["Subspace Size: $M"]

SubspaceInference.plot_node(t, all_trajectories, ode_data_bkp, ode_data, 2, datasize,
title)
```

The uncertainties in solution is plotted for two variables in Figure 6 and 7

The plot of variable 1 against variable 2 for subspace of 3 with 1.0 proposal distribution is Illustrated in the Figure 8. The Figure 9 discuss the uncertainties in predictions as well as in forecasting.

This package provides `NUTS, RWMH and MALA` algorithm based Bayesian inferences.

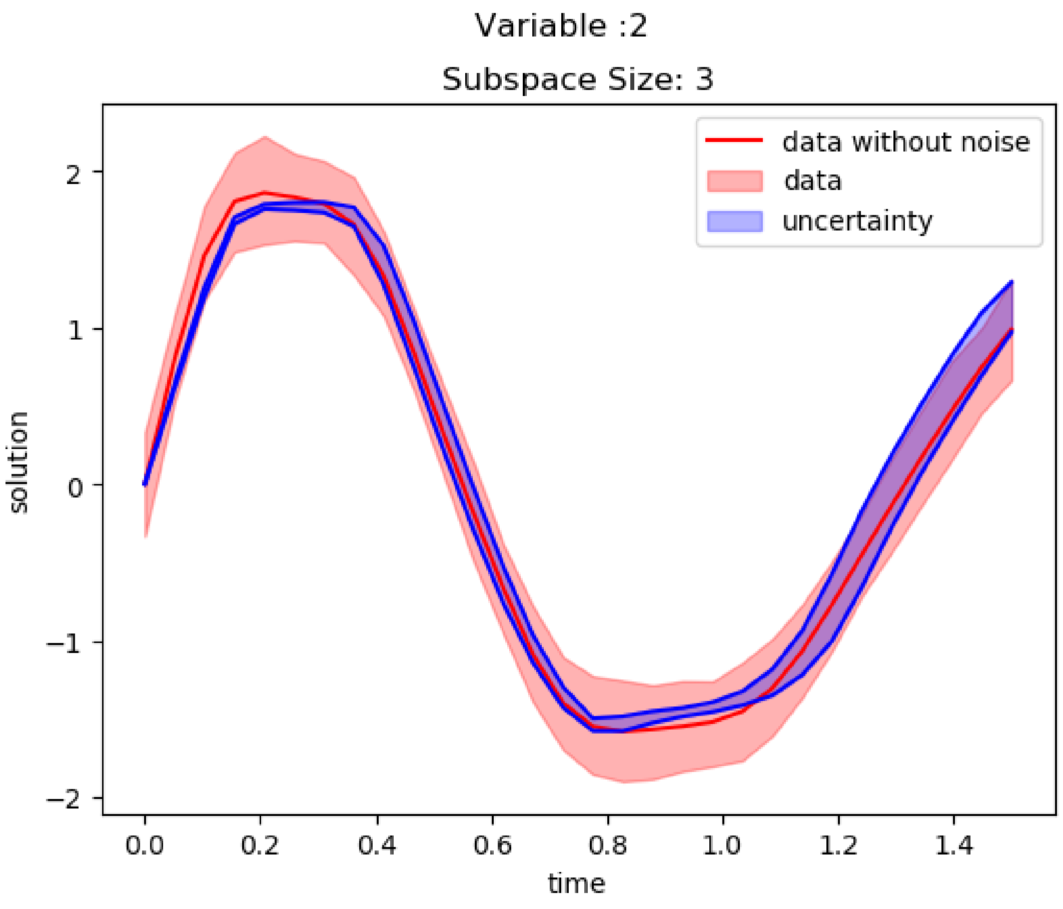

Figure 7: Uncertainty in var 2 solution

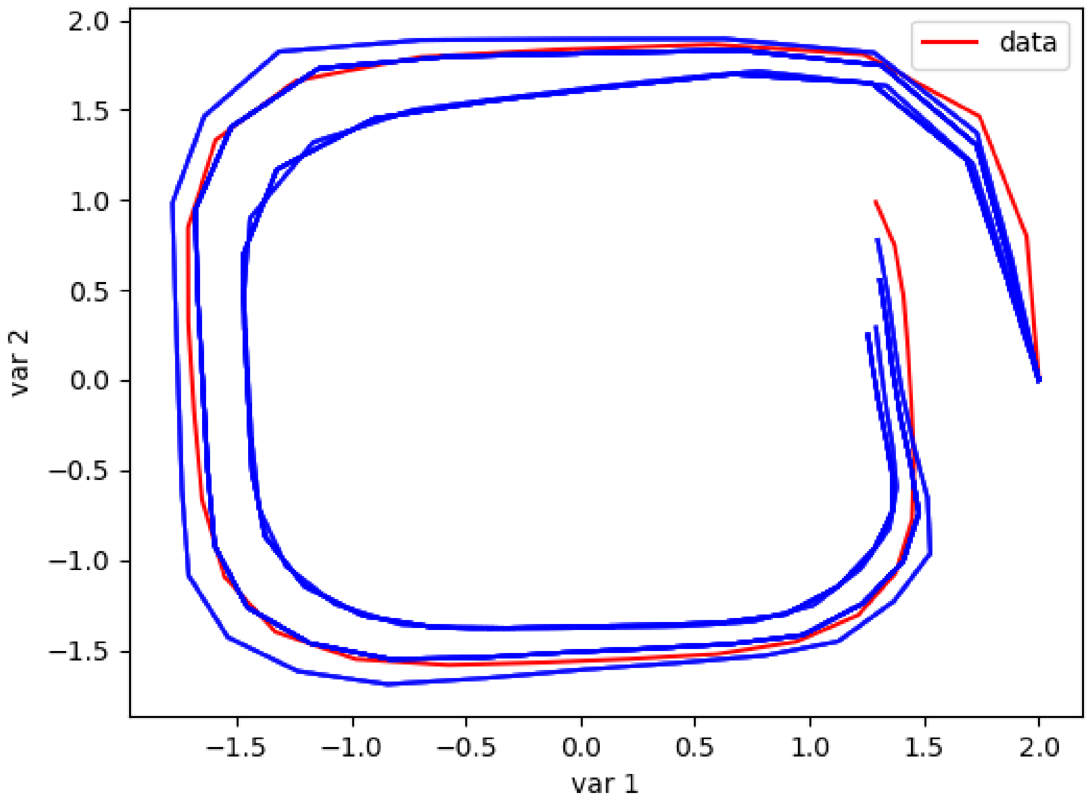

Figure 8: Variable 1 solution vs variable 2

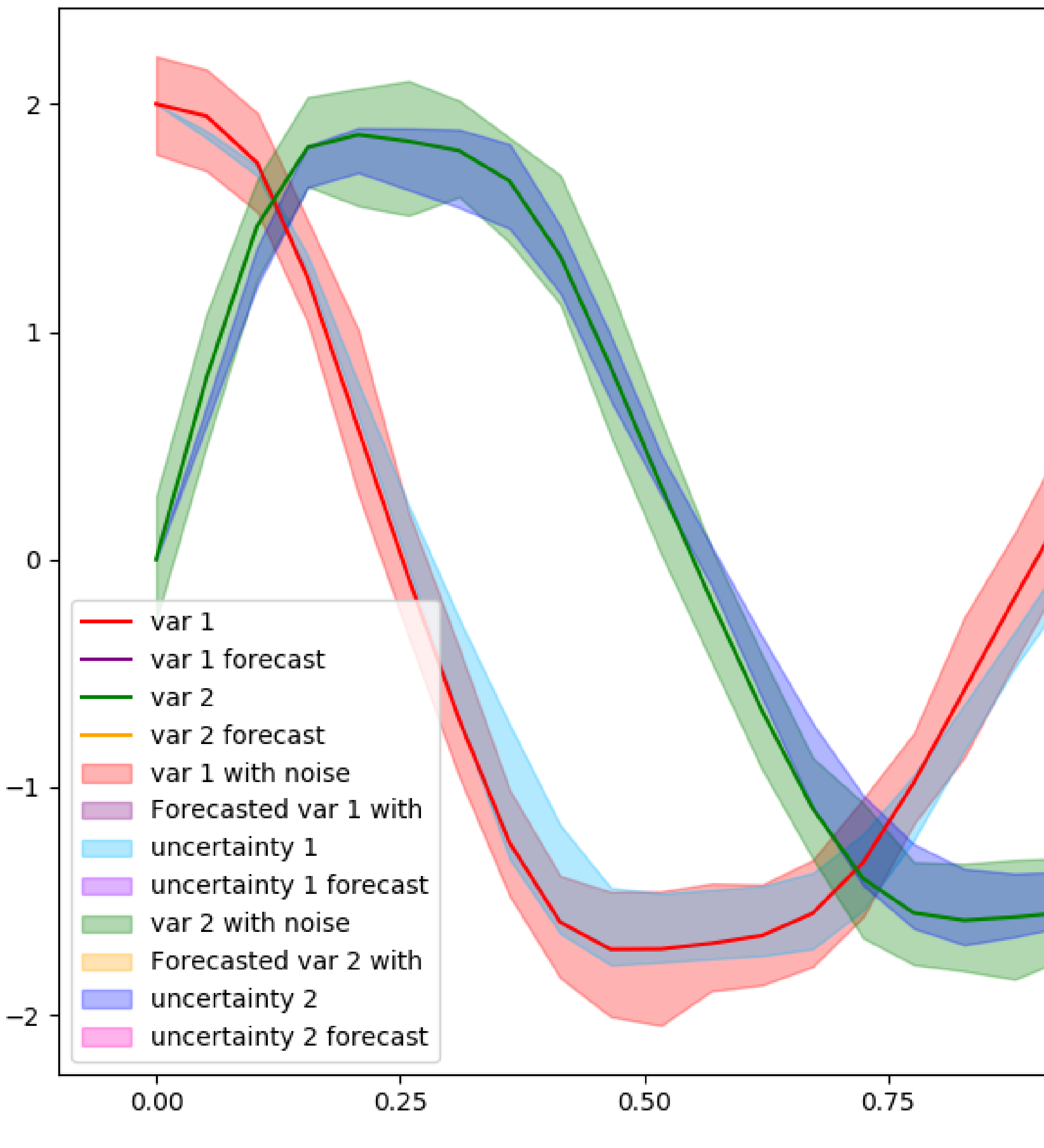

Figure 9: Prediction and Forecasting

1.0.4 Diffusion Map based subspace Construction

The subspace is constructed in above experiments by using dimensionality reduction techniques named Principle Component Analysis (PCA). This method is simple to implement however, it fails with many real-world datasets have non-linear characteristics. Therefore diffusion map is introduced in subspace construction to consider the non-linearity in dimensionality reduction.

1.0.5 Autoencoder based subspace construction

Similar to diffusion maps, autoencoder is also a dimensionality reduction method that accounts nonlinearity in neural ODE parameters. Moreover, this technique is based on machine learning.