NEURAL ODE'S AND TURBULENCE

CRISTON HYETT AND MISHA CHERTKOV

Abstract

The seminal work by Chen et. al. displayed a new methodology of Neural ODE's that can effectively and efficiently model dynamical systems[**?**]. Here, we explore this methodology and its ability to learn so-called Tetrad Dynamics[**?**]. On top of this springboard, we also investigate the effect of integratting neural ODE's with other methodologies; physics-informed neural networks (PINNs) and so-called Universal ODEs (a neural ODE with some modeling knowledge built in). We show the ability to reproduce some results from [**?**] in a fluid mechanics context, and lay the groundwork to move into the learning of reduced models for turbulent flow.

1. Introduction

As discussed in the Neural ODE paper[**?**], one can think of recurrent/residual neural networks as a Forward Euler discretization of continuous dynamics, where the state at hidden layer $i+1$ is defined by

$$x_{i+1} = x_i + f(x_i, \theta_i)$$

Extending this idea to when the step-size tends to zero, and the number of hidden layers tends to infinity, we recover the idea of a differential equation,

$$\frac{dx}{dt} = f(x, \theta)$$

where the state at a hidden "layer" t is given by

$$x(t) = \int_{t_0}^{t} f(x(t), \theta) dt$$

and output of the network is defined by solving the differential equation. In our work, the right hand side $f(x, \theta)$, will be a dense neural network (architecture described in section 4). We are interested in this neural ODE methodology primarily for two reasons:

i) Efficiency in calculation. A result by Pontriagin[**?**] demonstrates the ability to perform sensitivity analysis in the same cost as solving the defined ODE. In addition, the transition into ODEs allows us to leverage the decades of development of efficient and stable ODE solvers, providing guarantees of accuracy and computational cost.

ii) Ability for the learned NN to encode not just information about the data in question, but importantly, its structure. This refocus from learning the data to learning the ODE governing it opens doors to more stable extrapolation of dynamical systems, and - as we'll see in the case of universal ODE's - interpretability of the network.

In this paper, we apply Neural ODE's to tetrad dynamics, a phenomenlogical model of turbulence that attempts describe the statistical geometry of turbulent flows([**?**]). Here we will be restricted to learning the "pure" model, as a stepping stone to future work in data-based model discovery of reduced models of turbulence.

2. Neural ODEs

Since the Neural ODE publication by Chen, et.al., the work has recieved considerable attention. Much of science is concerned with spatio-temporal data, exhibited by the ubiquity of ordinary and partial differential equations. Combining this seemingly natural representation of time-series data with the efficiency improvements mentioned above, as well as the additional structure encoded into the trained network, results in a

promising methodology for scientific study. We will spend this section discussing in more detail the stated reasons this methodology seemed to be a good fit for our problem.

For context, let us consider an example neural ode, that is where the predictions $y(t)$ are given by the solution of the ODE

$$\frac{dy}{dt} = f(x, \theta) \tag{1}$$

and where f is a neural network of some architecture. Our goal will be to optimize a loss function $L : \mathbb{R}^n \to \mathbb{R}$ (perhaps MSE between prediction and training data).

It is known that a computationally efficient way to perform sensitivity analysis of a feed-forward network is through what is called "automatic differentiation", where the network has a corresponding system of equations which consumes the intermediate calculations of the network, and calculates the gradient with respect to the parameters. Thus, for efficient computation, one must have these intermediate results on-hand, requiring the storage of these intermediate calculations on the forward pass. For very deep networks, this can result in the need for large stores of memory, resulting in a computational bottleneck. Neural ODE's however, are able to perform sensitivity analysis using by solving an adjoint ODE[**?**]. This allows us to calculate the gradient of the neural ODE w.r.t. the parameters in a computationally efficient manner.

The phenomena of better fitting to noisy, unevenly spaced data, as well as better extrapolation is really a question of what underlies the data. In many scientific applications, one believes that there is at least a component of the dynamics that are governed by differential equations (here we're concerned with ODEs, but with a bit more work these ideas can be extended to PDEs and SDEs, although the details of senstivity analysis seem to be delicate), so that fitting the derivative instead of the noisy data itself unsurprisingly has a much better chance of finding the local minimum in the functional space.

3. Tetrad Dynamics

The eventual goal of our effort is to use DNS data and NODEs to learn reduced models for hydrodynamic turbulence, and thus allow us to learn and hypothesize models governing the statistical geometry of turbulent flows. As a first step in this direction, we attempt to learn the Tetrad Model, put forward by Chertkov et.al. This model considers the time evolution of a coherent volume of fluid, Γ, parameterized by four points, or a triad of vectors upon elimination of the center of mass. We can combine these vectors to obtain the inertial tensor ρ. As we are concerned with a volume of fluid the velocity gradient tensor we are interested in is the course-grained (on the scale of Γ), $\hat{M}$. Then we can describe the evolution of the tetrad according to the course-grained velocity gradient tensor

$$\frac{d}{dt}\hat{\rho} = \hat{M}^T\hat{\rho} \tag{2}$$

Finally, the evolution of $\hat{M}$ is given by

$$\frac{d}{dt}\hat{M} = -\hat{M}^2 + \hat{\Pi} \cdot \mathrm{trace}(\hat{M}^2) \tag{3}$$

where $\hat{\Pi}$ is a measure of local anisotropy,

$$\hat{\Pi} = \frac{\left(\hat{\rho}^{-1}\right)^T \left(\hat{\rho}^{-1}\right)}{\mathrm{trace}\left[\left(\hat{\rho}^{-1}\right)\left(\hat{\rho}^{-1}\right)^\dagger\right]} \tag{4}$$

While there are plenty of things I don't yet understand (regarding equation 3 in particular) I'll discuss briefly why this is a good model to start with in our attempt to learn the statistical geometry of hydrodynamic turbulence. First, it is a plausible model, with significant analysis to suggest it exhibits desirable physical properties. Further, the inclusion of this tetrad geometry defined by a given scale, allows one to probe different inertial ranges of interest.

4. Learning Tetrad Dynamics

In this section I'll discuss our attempts, challenges and successes, as a means of informing future endeavors in this direction. One of the first decisions we made was to begin with the study of the so-called Vielfosse

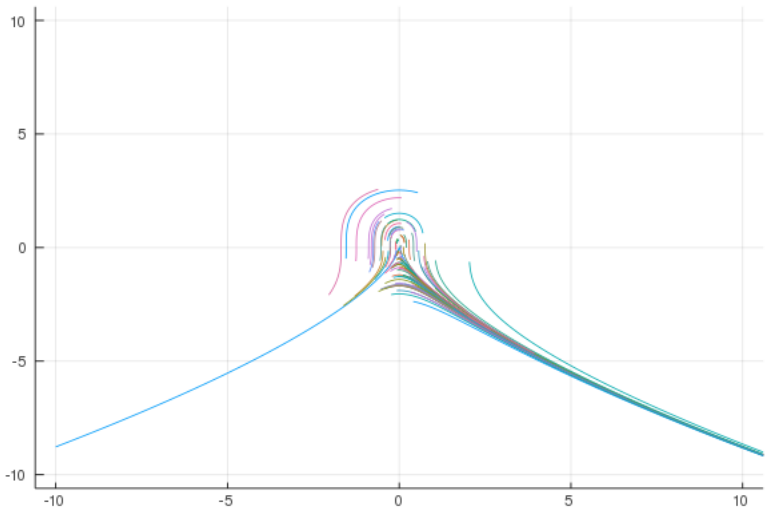

FIGURE 1. The lines show trajectories of invariants Q, R of the velocity gradient tensor.

dynamics[**?**], a simpler system intended to model the evolution of the local velocity gradient $\hat{m}$, given by

$$\frac{d}{dt}\hat{m} = -\hat{m}^2 + \frac{\text{trace}(\hat{m}^2)}{3}\hat{I} \tag{5}$$

This system is not as interesting physically, as it has been shown to lead to finite-time singularities. However, it does retain much of the form of the interested tetrad dynamics, and it has easily identifiable dynamics in order to debug the code. In particular, in figure (1) we plot the evolution of invariants of the velocity gradient tensor[**?**]

$$Q = -\frac{1}{2}\text{trace}\left(\hat{m}^2\right) \qquad R = -\frac{1}{3}\text{trace}\left(\hat{m}^3\right) \tag{6}$$

Again, as a first step, we attempt to learn the "pure" model, and as shown in listing(7.1.1), we generate data by solving this ODE forward and saving at some set of points (recall that when moving to DNS data, this training data will be extracted by coarse-graining the real DNS data). We then construct a 5-layer dense network with 50 nodes per layer. We use the Tsit5() solver to propogate through the network, (as well as solve the pure ODE) as it is quite fast and accurate. I found through experimentation that the choice of solver had enormous impact on performance, (profiling the code suggested most of the cycles were spent in the calculation of the adjoint equation, a result I still don't understand). The remaining details are handled largely in the background of the packages, though it should be noted I used the ADAM optimization routine (a variant of SGD) for training the network, and the mean-squared error for the loss function, though we will discuss more interesting possible choices below.

We obtained fairly nice results, reproducing the dynamics of the invariants as shown in figure(2). These dynamics are evolved over an extremely short timespan - reflective of how the network was trained. Looking at figure(1), we see that the invariants can evolve quite differently depending on the location of the initial position in the Q-R plane. In particular, the evolution above and below the asymptotic solution are drastically different, and the relative "velocities" of trajectories vary throughout the plane. Thus, to give our network the best chance of success, we exposed it to a wide range of initial conditions, (in the results displayed, 200 randomly chosen matrices $\hat{m}_0$), but to keep computational time limited, we only evolved these trajectories for a very short period of time.

In the Neural ODE paper, the authors discussed the so-called "latent ODE", present in the network after learning. Here we were able to recreate this phenomena, as we saw qualitative agreement of invariant trajectories evolved 30 times longer than the trained data, even when testing against data never seen by the network. This result is a very strong indicator of the power of learning the underlying ODE for scientific modeling, it is shown in figure(3).

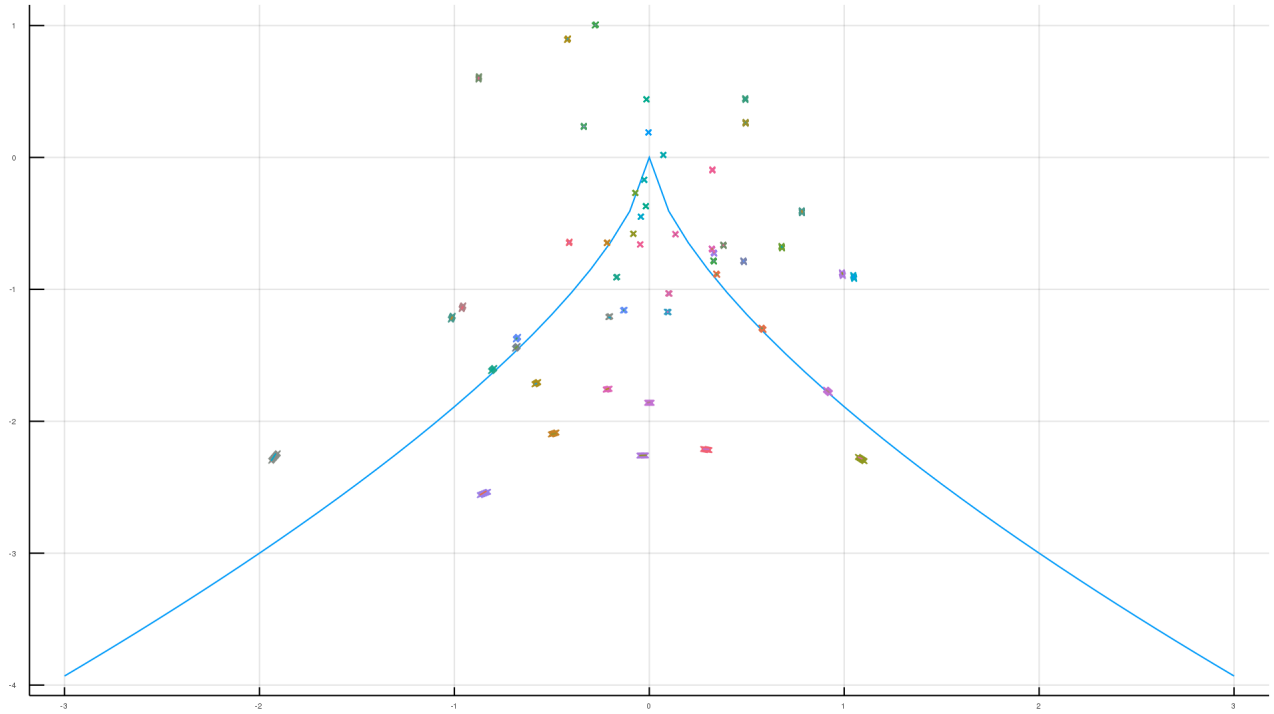

FIGURE 2. The 'X's show the true trajectories of invariants Q, R of the velocity gradient tensor, while the overlaid solid lines show the NODE predicted dynamics.

5. Conclusion

In this work we applied the novel methodology of Neural ODEs to the old problem of a statistical description of the geometry of hydrodynamic turbulence. We were able to learn a relatively simple pure model, showing the expressive and predictive power of neural ODEs, while laying the groundwork for future work in this arena.

6. Future Work

While most of this project centered around learning a simplified, pure ODE, it is important to view this effort in the broader context of learning statistical descriptions of turbulence. First, moving from learning this model to learning the full tetrad dynamics is a matter of applying more computational horsepower to a proven methodology. Once proven to learn this full tetrad model, one can move into learning real DNS data.

In the approach to real data, it is likely that our network will need additional assistance finding an accurate and general enough representation of the (now 18 dimensional) ODE. This can be provided using the ideas of physics-informed neural networks, placing physical constraints in the loss function as demonstrated by Raissi and collaborators[**?**]. An alternative, and exciting approach is to hypothesize some structure of the ODE, and allow the neural network to learn whatever else is needed to ensure agreement to the training data.

In the much broader picture, this work is the foundation of an exploration into data-driven model discovery and hypothesis testing using the power and expressability of neural networks.

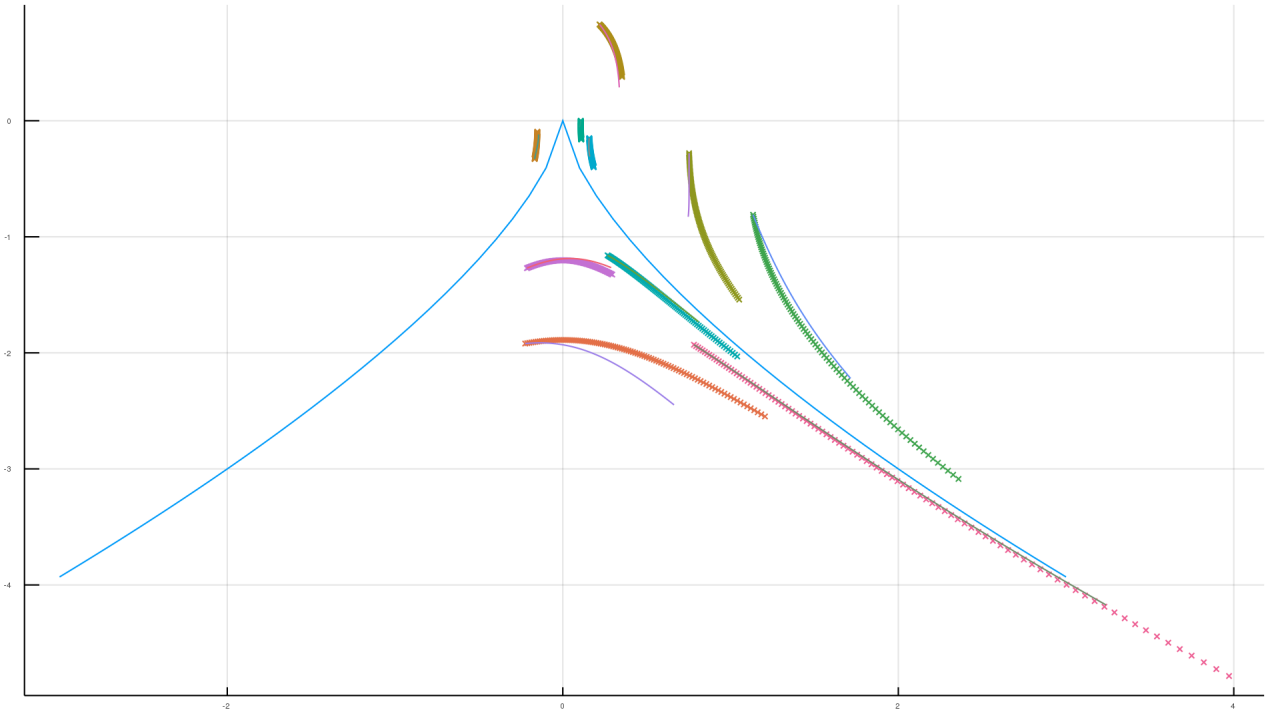

FIGURE 3. The 'X's show the true trajectories of invariants Q, R of the velocity gradient tensor, while the overlaid solid lines show the NODE predicted dynamics. The test data is randomly chosen, so the NODE has never seen these initial conditions, and we evolve 30 times longer than the trained timespan. The qualitative agreement shows the predictive power of neural ode's.

7. APPENDIX

7.1. **Code.**

7.1.1. *Learning Vielfosse Dynamics.*

```
using DifferentialEquations, DiffEqSensitivity;
using Flux, DiffEqFlux;
using Test, OrdinaryDiffEq, CuArrays, Statistics;
using Optim, LinearAlgebra;
using Plots;


function vielfosseLearning(;displayPlots = true)
    #construct initial conditions:
       = Matrix{Float64}(I,3,3);
    numBatches = 200;
    M = rand((-1.0:0.1:1.0),(3,3));
    M[3,3] = -(M[1,1] + M[2,2]);
    u0 = reshape(M,(9,1));

    for i in 1:(numBatches-1)
        M = rand((-1.0:0.1:1.0),(3,3));
        M[3,3] = -(M[1,1] + M[2,2]);
```

```
    u0 = hcat(u0,reshape(M,(9,1)));
end

datasize = 10;
tspan = (0.0f0,0.01f0); #again, unsure what time-scales we're interested in
                        #here or how quickly the dynamics will evolve

function trueODEfunc(du,u,p,t)
    for i in 1:numBatches
        Msquared = reshape(u[:,i],(3,3))^2;
        du[:,i] = reshape(-Msquared + tr(Msquared)*(1.0/3.0)*   ,(9,1));
    end
end

solver = Tsit5();

t = range(tspan[1],tspan[2],length=datasize)
prob = ODEProblem(trueODEfunc,u0,tspan)
ode_data = Array(solve(prob,solver,saveat=t))
println("**********ODE DATA CALCULATED*******************");

dudt2 = FastChain(FastDense(9,50,tanh),
                  FastDense(50,50,tanh),
                  FastDense(50,50,tanh),
                  FastDense(50,50,tanh),
                  FastDense(50,50,tanh),
                  FastDense(50,9))

n_ode = NeuralODE(dudt2,tspan,solver,saveat=t)

function loss_n_ode(p)
    pred = n_ode(u0,p);
    loss = (1/datasize)*sum(abs2,ode_data .- pred)
    return loss;
end

res1 = DiffEqFlux.sciml_train(loss_n_ode, n_ode.p, ADAM(),
                              maxiters = 1000)
# cb(res1.minimizer,loss_n_ode(res1.minimizer)...)
#res2 = DiffEqFlux.sciml_train(loss_n_ode, res1.minimizer, LBFGS(), cb = cb, maxiters =

if (displayPlots)
    println("**********NODE TRAINED, PLOTTING*******************");

    closeall();
    x = -10:0.1:10
    p = plot(x, x->-((27.0/4.0)*x^2)^(1/3),legend=false);

    pred = n_ode(u0,res1.minimizer);
    Q_node = zeros(datasize);
```

```
R_node = zeros(datasize);
Q_true = zeros(datasize);
R_true = zeros(datasize);

for i in 1:size(u0)[2]
    for j in 1:datasize
        M_node = reshape(pred[:,i,j],(3,3));
        Q_node[j] = -tr(M_node^2)/2.0;
        R_node[j] = -tr(M_node^3)/3.0;

        M_true = reshape(ode_data[:,:,j][:,i],(3,3));
        Q_true[j] = -tr(M_true^2)/2.0;
        R_true[j] = -tr(M_true^3)/3.0;
    end
    nodeLabel = string("nnPred",i);
    scatter!(p,R_node,Q_node,marker=:x,label=nodeLabel);
    trueLabel = string("groundTruth",i);
    scatter!(p,R_true,Q_true,marker=:o,label=trueLabel);
end

display(p); #represents performance on data we trained on


closeall();
x = -10:0.1:10
q = plot(x, x->-((27.0/4.0)*x^2)^(1/3),legend=false);

Q_node = zeros(datasize);
R_node = zeros(datasize);
Q_true = zeros(datasize);
R_true = zeros(datasize);

numValidations = 20;

for i in 1:numValidations
    M = rand((-1.0:0.1:1.0),(3,3));
    M[3,3] = -(M[1,1] + M[2,2]);
    u0 = reshape(M,(9,1));
    numBatches = 1;
    prob = ODEProblem(trueODEfunc,u0,tspan);
    ode_data = Array(solve(prob,solver,saveat=t));

    pred = n_ode(u0,res1.minimizer);
    for j in 1:datasize
        M_node = reshape(pred[:,j],(3,3));
        Q_node[j] = -tr(M_node^2)/2.0;
        R_node[j] = -tr(M_node^3)/3.0;

        M_true = reshape(ode_data[:,:,j],(3,3));
        Q_true[j] = -tr(M_true^2)/2.0;
```

```
                R_true[j] = -tr(M_true^3)/3.0;
            end
            nodeLabel = string("nnPred",i);
            scatter!(q,R_node,Q_node,marker=:x,label=nodeLabel);
            trueLabel = string("groundTruth",i);
            scatter!(q,R_true,Q_true,marker=:o,label=trueLabel);
        end

        #display(p); #represents performance on data we DID NOT train on

    end

    return res1,n_ode,u0,p,q

end
```