Advanced Topics in Climate Dynamics
Homework 2

L. Vulto
4685784
l.vulto@students.uu.nl

June 22, 2023

(a) **Suppose we have data $\vec{z}(t_k)$, $k = 1, ..., K$ of the solution. Formulate the loss function L, based on the mean squared error, used for fitting the weights of the NDE.**

We can define z_k as:

$$\mathbf{z}(t_k) = \mathbf{z}(t_0) + \int_{t_0}^{t_k} \mathbf{f}(\mathbf{z}, \theta, t)dt \tag{1}$$

Where $t \in [0, 1]$. Then, to find the loss function, we can use the definition on the mean squared error to find

$$L = \frac{1}{K}\sum_{k=1}^{K}|z(t_k) - z_k|^2$$

where z_k is calculated via (1).

(b) **Argue that when to optimise the loss function L, we need derivatives of L to $\vec{z}$, θ, t_0 and t_1.**

$\frac{\partial L}{\partial z}$ tells us how the loss changes as the predicted value of $\vec{z}$ changes. Since z is a direct parameter of **f**, the loss update after we change $\vec{z}$ tells us how good the predicted $\vec{Z}$ is.

To optimise the loss function, the first step is to determine how the gradient of the loss depends on the hidden state z(t) at each instant. This quantity is $\frac{\partial L}{\partial z(t)}$. Like stated in the exercise, the parameter θ are the weights of the neural network. We need the gradient of the loss function with respect to the weights to minimise the loss function because like this we can update the weights. Namely, by moving in the direction of the negative gradient, we can reduce the loss. To calculate the gradient of we need the derivative of L to θ

These derivatives provide information about how the loss function changes as we vary the values of these variables, allowing us to update them in a way that minimizes the loss.

The derivatives of L with respect to t_0 and t_1 tell us how the loss changes as we vary the initial and final times of the NDE respectively.

The NDE is solved over a time interval from t_0 to t_1, the initial and final its solution depends on both the initial state z_0 and the initial time t_0. They behave in the relation according to (1)

Changing the initial time can change the solution of the NDE, which in turn can change the predicted output and affect the value of the loss function. Therefore we need the derivative of L with respect to z_0 and z_1 as well.

(c) **Show that:** $\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)^T \frac{\partial f(\mathbf{z}(t),t,\theta)}{\partial \mathbf{z}(t)}$

c, d and e have been solved after consulting the appendix of paper (Chen et al., 2018). For this proof, **the vectors will now be row vectors, and not column vectors**. This effectively makes $\mathbf{a}^T$ equal to $\mathbf{a}(t)$. We start with $\mathbf{a}(t) = \partial L/\partial \mathbf{z}(t)$. If we apply the chain rule, we find

$$\mathbf{a}(t) = \frac{\partial L}{\partial \mathbf{z}(t)} = \frac{\partial L}{\partial \mathbf{z}(t+\varepsilon)} \frac{\partial \mathbf{z}(t+\varepsilon)}{\partial \mathbf{z}(t)} \quad \text{or} \quad \mathbf{a}(t) = \mathbf{a}(t+\varepsilon) \frac{\partial T_\varepsilon(\mathbf{z}(t),t)}{\partial \mathbf{z}(t)}$$

If we look at the definition of a derivative, we find:

$$\begin{aligned} \frac{d\mathbf{a}(t)}{dt} &= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t)}{\varepsilon} \\ &= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon) \frac{\partial}{\partial \mathbf{z}(t)} T_\varepsilon(\mathbf{z}(t))}{\varepsilon} \\ &= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon) \frac{\partial}{\partial \mathbf{z}(t)} \left(\mathbf{z}(t) + \varepsilon f(\mathbf{z}(t),t,\theta)\right)}{\varepsilon} \end{aligned} \tag{2}$$

Here we made use of a taylor expansion around $\mathbf{z}(t)$. I This has been kept as a Taylor expansion to the first order for simplicity's sake.

$$= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon) \left(I + \varepsilon \frac{\partial f(\mathbf{z}(t),t,\theta)}{\partial \mathbf{z}(t)}\right)}{\varepsilon} \tag{3}$$

Here, I is the identity matrix. This gets here since $\frac{\partial z(t)}{\partial z(t)} = I$.

$$\begin{aligned} &= \lim_{\varepsilon \to 0^+} \frac{-\varepsilon \mathbf{a}(t+\varepsilon) \frac{\partial f(\mathbf{z}(t),t,\theta)}{\partial \mathbf{z}(t)}}{\varepsilon} \\ &= \lim_{\varepsilon \to 0^+} -\mathbf{a}(t+\varepsilon) \frac{\partial f(\mathbf{z}(t),t,\theta)}{\partial \mathbf{z}(t)} \\ &= -\mathbf{a}(t) \frac{\partial f(\mathbf{z}(t),t,\theta)}{\partial \mathbf{z}(t)} \end{aligned} \tag{4}$$

QED

Or, if we do want to have column vectors, this is the same as

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)^T \frac{\partial f(\mathbf{z}(t),t,\theta)}{\partial \mathbf{z}(t)}$$

(d) **Describe a numerical solution procedure to determine $\mathbf{a}(t_0)$.** If we treat the ODE solver as a black box, we can compute the gradients using the adjoint sensitivity method. Namely, in the code, this is done by solving 'ODEAdjoint' back in time. ODEAdjoint effectively solves the ODE in (4). First, we have to evaluate $\frac{d\mathbf{z}}{dt} = \mathbf{f}(\mathbf{z}(\mathbf{t}), t, \theta)$ forwards in time. This is done from t_0 untill t_1, and then we know $\mathbf{z}(\mathbf{t}) \forall t$. If we then also evaluate the loss at the final time we thus have $\mathbf{a}\,(t_1)$ as well. The adjoint ODE is used to compute the gradient of the loss function with respect to the initial state of the system and the parameters of the neural network. If we can solve for (4), and we have the initial

condition for the adjoint differential equation $\mathbf{a}(t_1) = \frac{dL}{d\mathbf{z}(t_1)}$ we can give a solution for $\mathbf{a}(t_0)$ in the form of

$$\mathbf{a}(t_0) = \mathbf{a}(t_1) + \int_{t_1}^{t_0} \frac{d\mathbf{a}(t)}{dt} dt = \mathbf{a}(t_1) - \int_{t_1}^{t_0} \mathbf{a}(t)^T \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} \tag{5}$$

This leaves us with the gradient of the loss function for the initial state, which ultimately can be used to find better θ.

(e) **To determine $\partial L/\partial\theta$, the augmented matrix vector $\vec{x} = (\vec{z}, \theta, t)$ is used and the augmented equations are written as**

$$\frac{d\mathbf{x}}{dt} = \mathbf{g}(\mathbf{z}, \theta, t) \tag{6}$$

where $\mathbf{g} = (\mathbf{f}, \mathbf{0}, 1)^T$. We also have $\mathbf{b} = (\mathbf{a}, \mathbf{a}_\theta, \mathbf{a_t})$ where $\mathbf{a}_\theta = \partial L/\partial\theta$ and $\mathbf{a_t} = \partial L/\partial t$.

To start, we want to generalise (4). If we think of $\theta(t)$ and t as states with constant differential equations, we find $\frac{\partial\theta(t)}{\partial t} = \mathbf{0}$ $\frac{dt(t)}{dt} = 1$ If we combine these with z to an augmented state, we find

$$\frac{d}{dt}\begin{bmatrix} \mathbf{z} \\ \theta \\ t \end{bmatrix}(t) = f_{aug}([\mathbf{z}, \theta, t]) := \begin{bmatrix} f([\mathbf{z}, \theta, t]) \\ \mathbf{0} \\ 1 \end{bmatrix}, \mathbf{a}_{aug} = \begin{bmatrix} \mathbf{a} \\ \mathbf{a}_\theta \\ \mathbf{a}_t \end{bmatrix},$$

Or, with our introduced variables:

$$\frac{d}{dt}\begin{bmatrix} \mathbf{z} \\ \theta \\ t \end{bmatrix}(t) = f_{aug}([\mathbf{z}, \theta, t]) = \mathbf{g}$$

$$\mathbf{a_{aug}} = \mathbf{b}^T$$

The Jacobian has the form

$$\frac{\partial f_{aug}}{\partial[\mathbf{z}, \theta, t]} = \begin{bmatrix} \frac{\partial f}{\partial z} & \frac{\partial f}{\partial \theta} & \frac{\partial f}{\partial t} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

If we use this with (4) we obtain

$$\frac{d\mathbf{a}_{aug}(t)}{dt} = -(\mathbf{b})\frac{\partial f_{aug}}{\partial[\mathbf{z}, \theta, t]}(t) = -\begin{bmatrix} \mathbf{a}\frac{\partial f}{\partial \mathbf{z}} & \mathbf{a}\frac{\partial f}{\partial \theta} & \mathbf{a}\frac{\partial f}{\partial t} \end{bmatrix}(t) \tag{7}$$

In (7), the first element is the adjoint differential equation from (4), but the second element can be used to obtain the total gradient wrt the parameters if we integrate over the full interval and setting $\mathbf{a}_\theta(\mathbf{t_N}) = 0$ This yields:

$$\frac{dL}{d\theta} = \mathbf{a}_\theta(t_0) = -\int_{t_N}^{t_0} \mathbf{a}(t)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \theta} dt \tag{8}$$

We pointed out the similarity between the adjoint method and backpropagation (eq. 38). Similarly to backpropagation, ODE for the adjoint state needs to be solved backwards in time. We specify the constraint on the last time point, which is simply the gradient of the loss wrt the last time point, and can obtain the gradients with respect to the hidden state at any time, including the initial value

(f) **In the notebook, first data from the linear differential equation**

$$\frac{d\mathbf{z}}{dt} = \begin{pmatrix} -0.1 & -1.0 \\ 1.0 & -0.1 \end{pmatrix} \mathbf{z}$$

is used to train 'weights' from a 2×2 matrix. Determine how the loss function decreases versus the number of epochs for three different values of the number of data points K = 50, 100, 200. The mean of the loss function after 10 runs per number of data points can be seen in Figure 1.

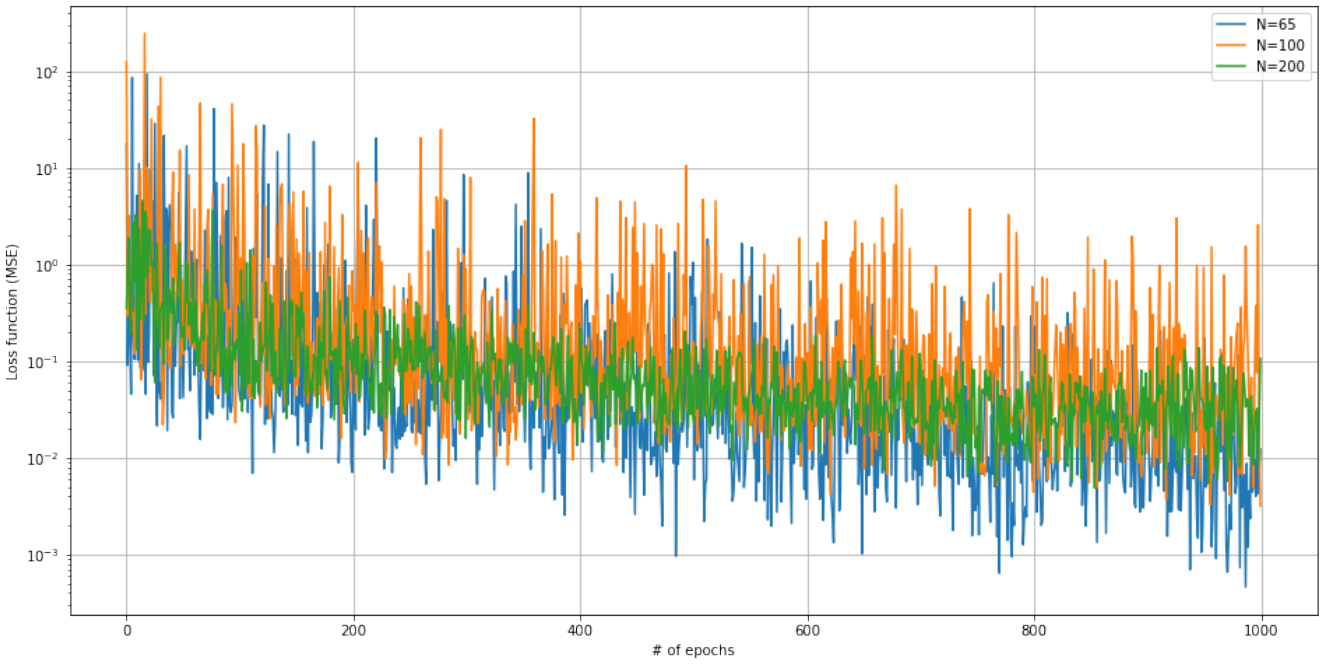

Figure 1: Mean of 10 runs per number of data points.

The random weights very much influence the starting points of the NODE. Sometimes, the initial conditions are in order of 10^{14}. This results in loss functions in order of 10^{2}. These solutions sometimes do not converge very nicely. Because this is not the best solution for the NODE, I also looked at what happened if I set a limit on the loss function at 10^{1}. If the loss function exceeds this limit, the calculation isn't taken into account when calculating the loss function. This can be seen in Figure 2.

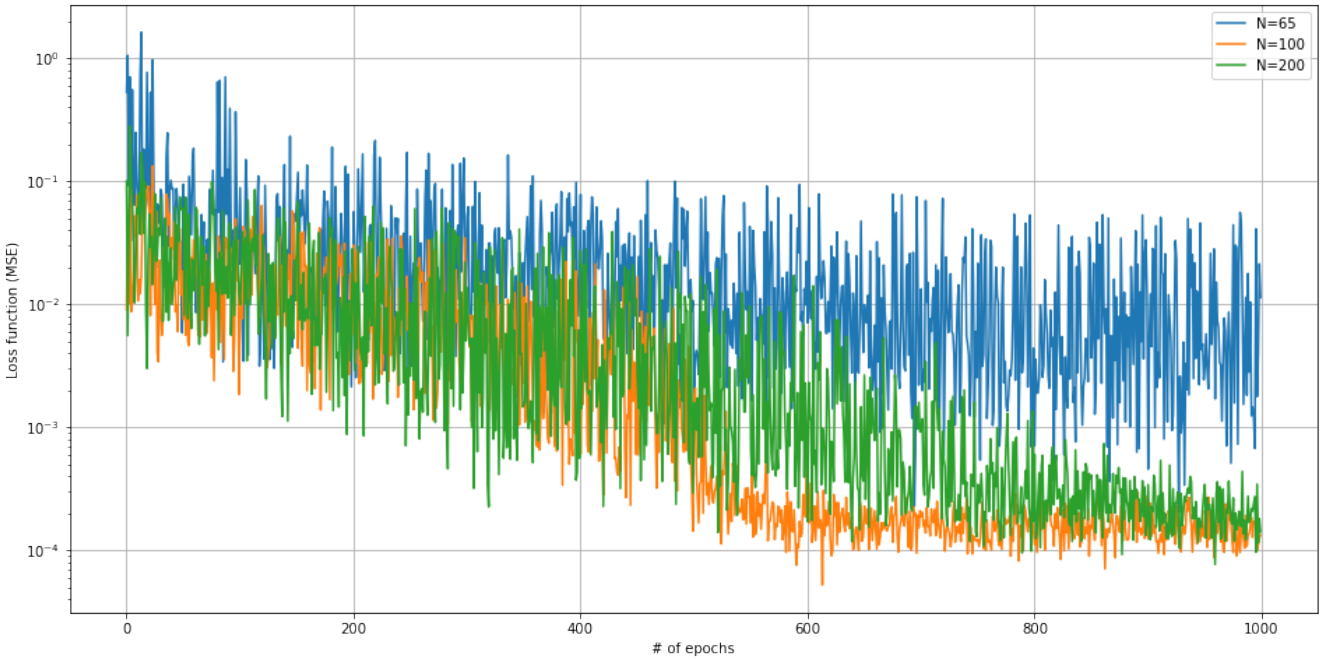

Figure 2: Mean of 5 runs per number of data points, constrained on Maximum loss.

In Figure 2 it looks like $N = 100$ and $N = 200$ are converging to a lower loss than $N = 65$, but the results are not too robust. This can be because the loss function still strongly depends on the initial random weights chosen in the code.

The code for Figure 2 is:

```
def loss_per_try(tries=3, points=np.array([65,100,200]),treshold
    =10):
    ode_true = NeuralODE(SpiralFunctionExample())
    tries = tries
    points = points
    mean_losses = []

    for vals in points:
        losses = []
        for i in range(tries):
            ode_trained = NeuralODE(RandomLinearODEF())
            loss = conduct_experiment(ode_true, ode_trained,
                1000, "comp", treshold=treshold,plot_freq=30,
                n_points=vals)
            losses.append(loss)
        mean_loss = np.mean(losses, axis=0)
        mean_losses.append(mean_loss)
    return mean_losses
```

For a more elaborate code, please consult the attached notebook.

(g) **Also a more complicated ODE is formulated in the notebook (TestODEF and you are welcome to enter your own). Use K = 200 and implement for f a FNN with 1 hidden layer and 16 neurons to solve the NDE in this case. Plot the trajectory which results after the loss function has sufficiently decreased.**

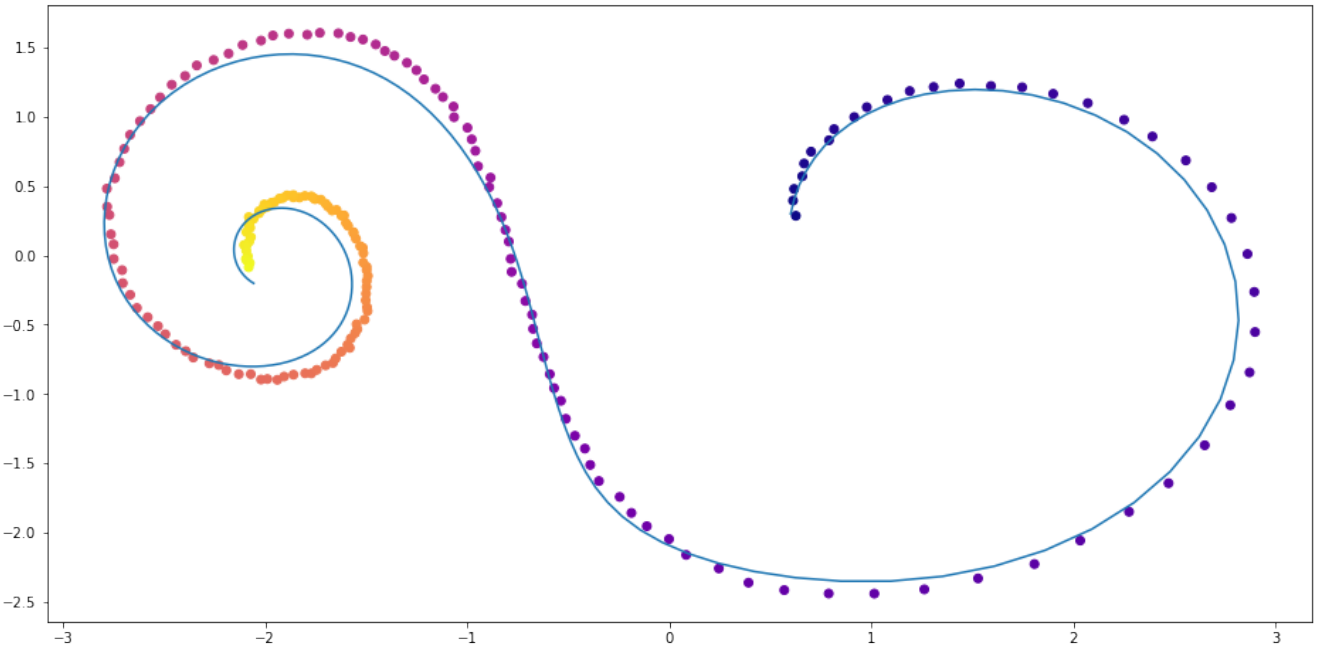

Figure 3: Solution to the function 'TestODEF' in the notebook.

In this figure the trajectory with parameters lr = 0.05, N=200. The number of epochs is dependent on when the loss function is smaller than 5×10^{-5}. The structure of **f** is a simple FNN with one hidden layer with 16 neurons.

We defined a simple class for this fnn and stopped after the loss function is lower than 5×10^{-5} we stop the experiment. The class is given by:

```
class NNODEF(ODEF):
    def __init__(self):
        super(NNODEF, self).__init__()
        self.input = nn.Linear(2,16)
        self.lin1 = nn.Linear(16,2,bias=True)#?
        self.a = nn.ELU()

    def forward(self,x,t):
        x = self.input(x)
        x = self.a(x)
        x = self.lin1(x)


        return x
```

Furthermore, the conduct experiment function is altered like this:

```
1
2       def conduct_experiment_g(..):
3           ...
4           optimizer = torch.optim.Adam(ode_trained.parameters(), lr
                =0.005)
5
6           if loss.detach().numpy() < 5e-5:
7                   break
```

To half the learning rate. This yielded in a quicker solution where the loss is less than 5×10^{-5}. This solution is shown in figure 3.

The Loss function is shown in Figure 4 with a minimum at 4.9×10^{-5}.

```
Minimum of the loss function is: 4.9211812e-05
```

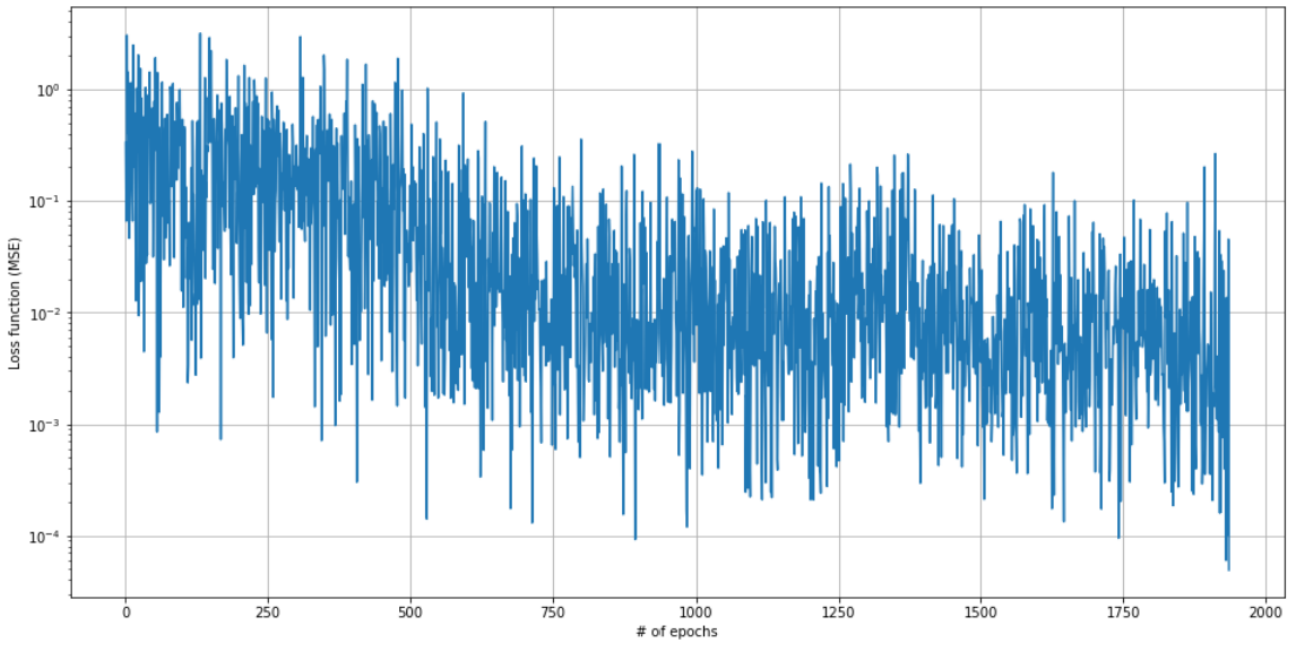

Figure 4: Loss function for Figure 3.

Appendix

References

Chen, R. T. Q., Rubanova, Y., Bettencourt, J., & Duvenaud, D. (2018). Neural Ordinary Differential Equations. *arXiv*.