ATCD Homework 2

Bennet Weiß

June 2023

1 Introduction

In this exercise, you will explore the solution method of a neural differential equation, given by

$$\frac{d\mathbf{z}}{dt} = \mathbf{f}(\mathbf{z}, t, \theta) \tag{1}$$

over the time interval t_0, t_1. Here $\mathbf{z} \in \mathbb{R}^n$ and f is a neural network with parameter vector (weights) θ. You will also study an application of an NDE to data from a simple linear differential equation. The integral form of 1 is given by

$$\mathbf{z}(t) = \mathbf{z}(t_0) + \int_{t_0}^{t_1} \mathbf{f}(\mathbf{z}, \theta, t)dt. \tag{2}$$

The derivations in this homework are based on [1].

2 a)

The Loss function L, based on the mean squared error can be defined as

$$\begin{aligned} L &= \frac{1}{K}\sum_{k=1}^{K} \|\mathbf{z}(t) - \hat{\mathbf{z}}(t)\|^2 \\ &= \frac{1}{K}\sum_{k=1}^{K} \|\mathbf{z}(t) - \mathbf{z}(t_0) - \int_{t_0}^{t_1} \mathbf{f}(\mathbf{z}, \theta, t)dt\|^2 \end{aligned} \tag{3}$$

where $\mathbf{z}(t_k), k = 1, ..., K$ is data of the solution, $\hat{\mathbf{z}}(t)$ are the predictions of the NDE and $\|\cdot\|$ is the Euclidean norm. In the last step, eq. (2) was used.

3 b)

θ are the weights that need to be optimized in order to reduce the loss function L. The derivative of L with respect to θ is required to update the weights using for instance the gradient descent method. Here, θ is corrected in the direction that decreases L, where η is the learning rate, a hyperparameter of the network:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta\frac{\partial L}{\partial \theta}. \tag{4}$$

As can be seen from eq. (3), L is an explicit function of $\mathbf{z}$ and only implicitly depends on θ. Consequently, using the chain rule, the derivative of L with respect to $\mathbf{z}$ is required to calculate $\partial L/\partial\theta$ in eq. (4). As the system is continuous in time, the starting end ending time t_0 and t_1, respectively can also be optimized. Consequently, the derivative of the Loss function L with respect to these times is also required for optimization.

4 c)

Let $\mathbf{a}(t) = \partial L/\partial \mathbf{z}(t)$ be the so-called adjoint, which is necessary to optimize L as described before. Using the chain rule, this quantity can be rewritten to

$$\mathbf{a}(t) = \frac{\partial L}{\partial \mathbf{z}(t)} = \frac{\partial L}{\partial \mathbf{z}(t+\epsilon)} \frac{\partial \mathbf{z}(t+\epsilon)}{\partial \mathbf{z}(t)}, \tag{5}$$

where we directly used the chain rule. The first factor can be directly identified as $\mathbf{a}(t+\epsilon)$, while the second factor can be approached by using eq. (2). The latter equation can be rewritten to

$$\mathbf{z}(t+\epsilon) = \mathbf{z}(t) + \int_t^{t+\epsilon} \mathbf{f}(\mathbf{z}(t), \theta, t) dt = T_\epsilon(\mathbf{z}(t), t). \tag{6}$$

Here, T symbolizes the transformation after a time step ϵ. Using a Taylor series of T around $\mathbf{z}(t)$, this equation can be simplified to

$$\mathbf{z}(t+\epsilon) = \mathbf{z}(t) + \epsilon \mathbf{f}(\mathbf{z}(t), \theta, t) + \mathcal{O}(\epsilon^2). \tag{7}$$

Essentially, this is nothing else as taking f to be constant within the integral interval which is especially reasonable for small ϵ. Differentiating $\mathbf{a}(t)$ with respect to t by using the limit definition of the derivative and using equations (5) and (7) we arrive at the following statement:

$$\begin{aligned}
\frac{d\mathbf{a}}{dt} &= \lim_{\epsilon \to 0} \frac{\mathbf{a}(t+\epsilon) - \mathbf{a}(t)}{\epsilon} \\
&= \lim_{\epsilon \to 0} \frac{\mathbf{a}(t+\epsilon) - \frac{\partial L}{\partial \mathbf{z}(t+\epsilon)} \frac{\partial \mathbf{z}(t+\epsilon)}{\partial \mathbf{z}(t)}}{\epsilon} \\
&= \lim_{\epsilon \to 0} \frac{\mathbf{a}(t+\epsilon) - \mathbf{a}(t+\epsilon) \frac{\partial}{\partial \mathbf{z}(t)} \left(\mathbf{z}(t) + \epsilon \mathbf{f}(\mathbf{z}(t), \theta, t) + \mathcal{O}(\epsilon^2) \right)}{\epsilon} \\
&= \lim_{\epsilon \to 0} \frac{\mathbf{a}(t+\epsilon) - \mathbf{a}(t+\epsilon) \left(\mathbf{I} + \epsilon \frac{\partial \mathbf{f}(\mathbf{z}(t), \theta, t)}{\partial \mathbf{z}(t)} + \mathcal{O}(\epsilon^2) \right)}{\epsilon} \\
&= -\lim_{\epsilon \to 0} \mathbf{a}(t+\epsilon) \left(\frac{\partial \mathbf{f}(\mathbf{z}(t), \theta, t)}{\partial \mathbf{z}(t)} + \mathcal{O}(\epsilon) \right),
\end{aligned}$$

where $\mathbf{I}$ is the identity matrix with required dimensions.

Lastly, the limit can be executed, errors $\mathcal{O}(\epsilon)$ linear in ϵ will vanish and we arrive at the following result:

$$\frac{d\mathbf{a}}{dt} = -\mathbf{a}(t) \frac{\partial \mathbf{f}(\mathbf{z}(t), \theta, t)}{\partial \mathbf{z}(t)}. \tag{8}$$

If $\mathbf{a}$ is a row vector and $\frac{\partial \mathbf{f}}{\partial \mathbf{z}}$ is a Jacobian matrix then $\mathbf{a} \frac{\partial \mathbf{f}}{\partial \mathbf{z}}$ is also a row vector. If $\mathbf{a}$ is a column vector, eq. (8) would not be valid because of a dimension mismatch. To fix this, the column vector $\mathbf{a}$ can be transposed to $\mathbf{a}^T$ so that the complete equation reads

$$\frac{d\mathbf{a}}{dt} = -\mathbf{a}(t)^T \frac{\partial \mathbf{f}(\mathbf{z}(t), \theta, t)}{\partial \mathbf{z}(t)}, \tag{9}$$

which is the identity that was to be shown.

5 d)

In order to numerically determine $\mathbf{a}(t_0)$, one typically employs a method known as adjoint sensitivity analysis. This method involves solving a system of ordinary differential equations (ODEs) backwards in time. The first step is the forward pass, where the original ODE $\frac{d\mathbf{z}(t)}{dt} = \mathbf{f}(\mathbf{z}(t), t, \theta)$ is solved forward in time from t_0 to t_1 to obtain the state variable $\mathbf{z}(t)$ at each time point. This way, also $\mathbf{f}$ is known at every point t. There are different integration methods to realize this, the easiest being the Euler method. Following this, the loss function L is evaluated at the final time t_1 using the computed state variable $\mathbf{z}(t_1)$, yielding $L(\mathbf{z}(t_1))$. The gradient of the loss function with

respect to the state variable at the final time, $\frac{\partial L}{\partial \mathbf{z}(t_1)}$, is then computed and set as the initial value of the adjoint state $\mathbf{a}(t_1)$. The specific form of this derivative depends on the loss function L. This leads to the backward pass, where the adjoint equation $\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)^T \frac{\partial \mathbf{f}(\mathbf{z}(t),t,\theta)}{\partial \mathbf{z}(t)}$ is solved backwards in time from t_1 to t_0 to obtain the adjoint state $\mathbf{a}(t)$ at each time point. Here, for every time step, we can make use of the known $\mathbf{f}$ and $\mathbf{z}(\mathbf{t})$. Once again, the numerical integration can be achieved by different methods methods such like the Euler method. The desired output is the value of the adjoint state at the initial time, $\mathbf{a}(t_0)$. This procedure allows for the computation of the gradient of the loss function with respect to the initial state $\mathbf{z}(t_0)$, which can ultimately be used for the updating the weights θ, as described in section 4.

6 e)

To determine $\partial L/\partial\theta$, it is convenient to introduce the augmented vector $\mathbf{x} = (\mathbf{z}, \theta, t)$ with the augmented equations

$$\frac{d\mathbf{x}}{dt} = (\mathbf{f}, \mathbf{0}, 1)^T =: \mathbf{g}(\mathbf{z}, \theta, t). \tag{10}$$

We also introduce $\mathbf{b} = (\mathbf{a}, \mathbf{a}_\theta, \mathbf{a}_t)$, where $\mathbf{a}_\theta = \partial L/\partial\theta$ and $\mathbf{a}_t = \partial L/\partial t$.
Let us look at the Jacobean of $\mathbf{g}$:

$$\frac{\partial \mathbf{g}}{\partial \mathbf{b}} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial \mathbf{z}} & \frac{\partial \mathbf{f}}{\partial \theta} & \frac{\partial \mathbf{f}}{\partial t} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}, \tag{11}$$

where $\mathbf{0}$ symbolizes a matrix filled with zeros of appropriate dimensions.
As a next step, we make use of eq. (9) and plug in $\mathbf{b}$ for $\mathbf{a}$ and replace the derivative of $\mathbf{f}$ with respect to $\mathbf{z}$ with a derivative of $\mathbf{g}$ with respect to $\mathbf{x}$. We can do this as $\mathbf{x}$ follows eq. (10), which has the same structure as eq. (1). This results in the following relation:

$$\frac{d\mathbf{b}(t)}{dt} = -\mathbf{b}(t)^T \frac{\partial \mathbf{g}(\mathbf{x})}{\partial \mathbf{x}}(t) = -\left[\mathbf{a}^T \frac{\partial \mathbf{f}}{\partial \mathbf{z}}, \mathbf{a}^T \frac{\partial \mathbf{f}}{\partial \theta}, \mathbf{a}^T \frac{\partial \mathbf{f}}{\partial t}\right]. \tag{12}$$

While we indeed see that the first element corresponds to eq. (9), we can also identify

$$\frac{d\mathbf{a}_\theta}{dt} = -\mathbf{a}^T \frac{\partial \mathbf{f}}{\partial \theta}. \tag{13}$$

Integrating this equation backwards from t_1 to t_0 results in the required equation

$$\mathbf{a}_\theta(t_0) = -\int_{t_1}^{t_0} \mathbf{a}^T \frac{\partial \mathbf{f}(\mathbf{z}, \theta, t)}{\partial \theta} dt, \tag{14}$$

while $\mathbf{a}_\theta(t_1)$ was set to $\mathbf{0}$.

7 f)

To plot the decrease of the loss as a function of epochs, we need to alter the *conduct_experiment* function in the code such that it saves its loss for each epoch during training. The new function created this way is called *ce_loss* and returns an array containing the loss for each epoch during training.

```
def ce_loss(ode_true, ode_trained, n_steps, name, plot_freq=10, K=200):
    # Create data
    z0 = Variable(torch.Tensor([[0.6, 0.3]]))

    losses = []

    t_max = 6.29*5
    n_points = K

    index_np = np.arange(0, n_points, 1, dtype=int)
    index_np = np.hstack([index_np[:, None]])
    times_np = np.linspace(0, t_max, num=n_points)
```

```
    times_np = np.hstack([times_np[:, None]])

    times = torch.from_numpy(times_np[:, :, None]).to(z0)
    obs = ode_true(z0, times, return_whole_sequence=True).detach()
    obs = obs + torch.randn_like(obs) * 0.01

    # Get trajectory of random timespan
    min_delta_time = 1.0
    max_delta_time = 5.0
    max_points_num = 32
    def create_batch():
        t0 = np.random.uniform(0, t_max - max_delta_time)
        t1 = t0 + np.random.uniform(min_delta_time, max_delta_time)

        idx = sorted(np.random.permutation(index_np[(times_np > t0) & (times_np
            < t1)])[:max_points_num])

        obs_ = obs[idx]
        ts_ = times[idx]
        return obs_, ts_

    # Train Neural ODE
    optimizer = torch.optim.Adam(ode_trained.parameters(), lr=0.01)
    for i in range(n_steps):
        obs_, ts_ = create_batch()

        z_ = ode_trained(obs_[0], ts_, return_whole_sequence=True)
        loss = F.mse_loss(z_, obs_.detach())
        losses.append(loss.detach().numpy())

        optimizer.zero_grad()
        loss.backward(retain_graph=True)
        optimizer.step()

    return np.squeeze(losses)
```

Listing 1: Definition of *ce_loss*.

This function has also been changed such that it can take K as an argument. The parameter K represents the number of points making up the spiral which should be regressed. Increasing the value of K will result in a higher density of points, providing more fine-grained observations and batches during training. However, it will also increase the computational cost for each epoch as more points need to be evaluated and processed.

To check how the loss function behaves for different K, the following code has been set up. It calls *ce_loss* for different K=65, 100, 200 and saves the value of the loss function for every epoch step. Setting K to a value of 50 resulted in an error that could not be resolved within reasonable time. A slight increase to K=65 resolved the problem while still being a good representative for a low number of points. Therefore, the difference in the behaviour of the loss function between K=50 and K=65 is expected to be marginal. To make sure the results are representative, the experiment is repeated multiple times (10) and the loss at each K and epoch is averaged. Each time, the training was performed for 4000 epochs. This large number is hoped to give enough time for the loss function to converge. Due to limited computation power, it is not possible to increase this figure further, as the training already took one hour to process. The resulting plot is shown in in Figure 1. Here, next to the average loss of each epoch as indicated by the data points, a running mean of 100 epochs with equal weight is applied and plotted to show the more long term trend of the loss during training. This is necessary especially because the variance of the loss function is very high, which will also be discussed in the following section. The code calling *ce_loss* multiple times for the different K is shown below.

```
ode_true = NeuralODE(SpiralFunctionExample())

n_runs = 10
K_arr = [65, 100, 200]
k_losses = []
```

```
6
7 # loop through all K values
8 for K in K_arr:
9     av_losses = []
10    # run multiple times for one K value
11    for n in range(n_runs):
12        print(f"Progress: K={K}, n={n}", end="\r")
13        ode_trained = NeuralODE(RandomLinearODEF())
14        loss = ce_loss(ode_true, ode_trained, 4000, "comp", plot_freq=1000, K=K)
15        loss_val = np.mean(loss[-100:-1])
16        av_losses.append(loss)
17    k_losses.append(av_losses)
```

Listing 2: Calling *ce_loss*.

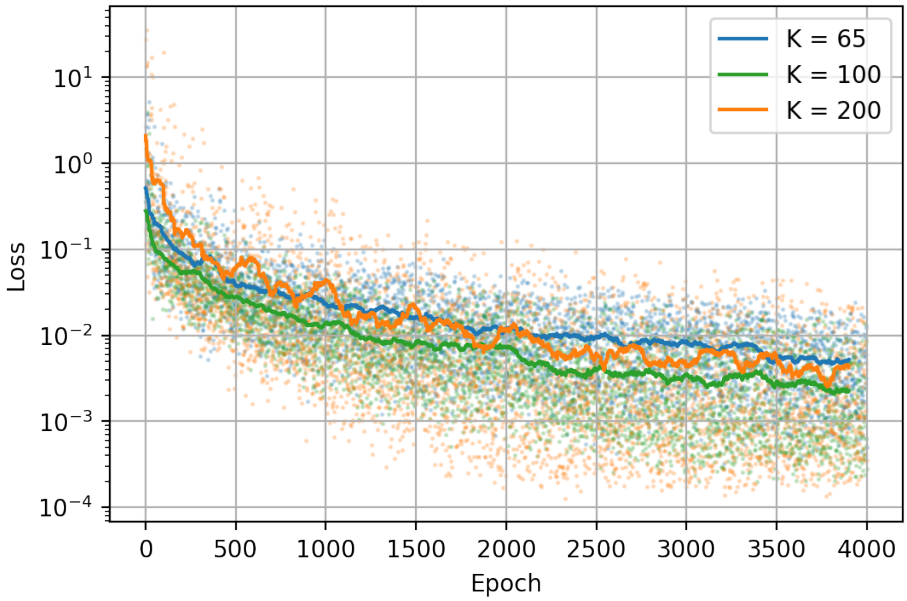

Figure 1: Evolution of the averaged loss function during training for different numbers of points K. A running mean over 100 epochs is overlaid to show the overarching trend.

Looking at Figure 1, there is no significant difference between the number of points K visible. For all K, the trend of the average loss function is a rapid decrease for early epochs which is slowed down with more epochs. The loss function with the highest amount of points K=200 is almost constantly found in between the loss function for K=65 and K=100 with a lower number of points. Therefore, it cannot be concluded from this plot that a higher number of points increases the speed of convergence for the loss function. On the one hand, this might have been a reasonably sounding assumption as more available points make it easier to regress. On the other hand, the loss is normalized by the number of points. This should cancel out the most obvious influence of K which is that the sum in the loss function becomes longer.

In every training run, the speed of convergence of the loss function is heavily dependent on the random initialization. If the latter is really bad, the loss function might converge slower or not converge at all within the limited number of epochs. This can be seen in Figure 2, where the development of the loss function is plotted seperately for each run and all K. It can be seen, that for all K, some runs seem to converge extremely slowly so that they are much worse than their counterparts towards the end of the training. At the same time, other runs converge to a similar level but at a considerably later epoch.

To rule out that bad initializations significantly deteriorate the averaging procedure, a threshold has been implemented. If the loss function has not decreased sufficiently towards the end of the training, this result is discarded and the training is repeated. A threshold of $loss\_thresh = 5e - 4$ that has to be met on average for the last 100 epochs has been proven to be a sufficient barrier as results with a loss function converged below that typically resemble the spiral well. The result, where only training rounds that meet this criterion have been

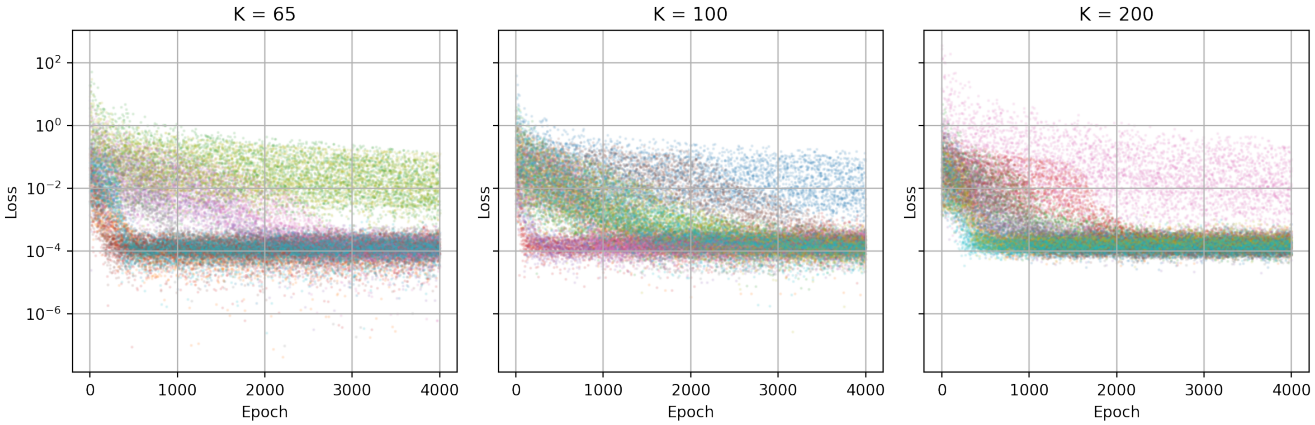

Figure 2: Evolution of the loss function during training for different numbers of points K. The different colors denote one of the 10 runs that were performed for each K.

averaged, is plotted in figure 3. The overall trend is, unsurprisingly, that the average loss converges faster across all K then without the threshold. There is, however, still no clear pattern visible, as it is now the loss of K=65 that is sandwiched by the other experiments.

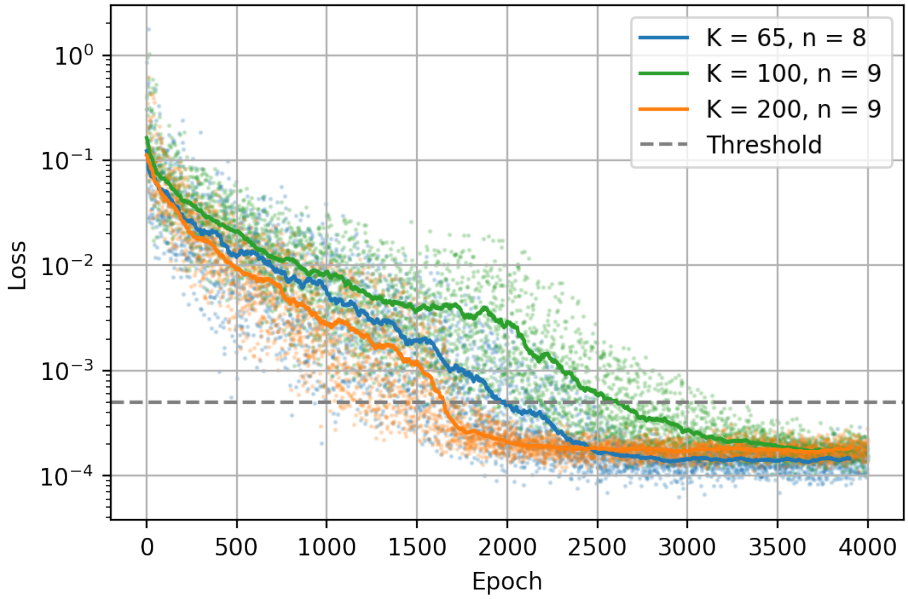

Figure 3: Evolution of the averaged loss function during training for different numbers of points K. The different colors denote one of the 10 runs that were performed for each K. A threshold of $loss\_thresh = 5e - 4$ helps to discard very bad training rounds before averaging. n denotes the number of runs averaged for each K.

The previous measure ensured that the final loss function after 4000 epochs was sufficiently low. However, this does not capture unusually slow converging runs that ultimately manage to converge below the threshold. They might also be a result of unfortunate initialization and thus not very representative to show the loss function behaviour. Consequently, the last effort is to capture bad initializations that show relatively high loss functions of above 1 at the beginning of the training, as shown by the code below. In fact, this threshold is enough to also dismiss the runs that were excluded by the threshold before. This could be seen by setting $loss\_thresh = 1$ which still yielded the same result (not shown here). Now, we can be sure that all unusually slowly converging runs are disregarded.
The resulting graph is shown in Figure 4 and is again characterized by an overall increase of the speed of convergence by the loss function. Unfortunately, there is still no clear trend with respect to the different values of K visible.

For all K, the loss converges already after around 1000 epochs and a shows a constant behaviour thereafter. Additionally, from this analysis, it cannot be said that the number of badly converging runs (that were discarded) has a dependence on K. This can be checked by looking at the number of included runs n in Figure 3 and 4 which can be 10 at maximum. There is no simple relationship between n and K deductible.

```
loss_thresh = 5e-4
high_thresh = 1

plt.figure(dpi=200)

for idx, losses in enumerate(k_losses):
    av_bool = []
    for single_loss in losses:
        if (np.mean(single_loss[-100:-1]) < loss_thresh) and
            (np.max(single_loss[0:10]) < high_thresh):
            av_bool.append(True)
        else:
            av_bool.append(False)

    av_loss = np.mean(np.array(losses)[av_bool], axis=0)
    running_mean = np.convolve(av_loss, np.ones(100) / 100, mode='valid')

    plt.plot(av_loss, "o", alpha=.2, markersize=1, color=colors[idx], zorder=1)
    plt.plot(running_mean, color=colors[idx], zorder=2, label=f"K =
        {K_arr[idx]}")

plt.axhline(loss_thresh, label="Threshold", ls="dashed", color="tab:gray")
plt.yscale("log")
plt.legend()
plt.grid(True)
plt.ylabel("Loss")
plt.xlabel("Epoch")
```

Listing 3: Creating threshold figures.

The conclusion from this analysis is thus that there is either no persistent trend in the loss function behaviour for different K, or that the variance of the individual runs is too high to accurately capture it with only 10 runs per value K. Due to the high variance of the system as well as the extreme dependence on initial conditions, a higher number of runs is required to analyze the K dependence with more statistics. For the scope of this project, however, this is too expensive to run.

8 g)

Now, we want to solve a more complicated ODE. To use a FNN for $\mathbf{f}$, we have to create a FNN first. Luckily, this has already been done in Project 1 and can therefore be copied from there. The ODEF with FNN (NNODEF) is then inheriting from ODEF and uses the FNN with 1 hidden layer and 16 neurons. While these hyperparamters of the network were set, the activation function can be discussed. In testing, ELU seemed to perform better than RELU which is why the first was ultimately chosen. An extensive quantitative testing of different activation functions was not performed, however.

```
# Define the FNN architecture: taken from previous project
class FNN(nn.Module):
    def __init__(self, input_size, output_size, n_hidden, n_neurons_h, A):
        super(FNN, self).__init__()
        self.hidden_layers = nn.ModuleList() # empty nn.ModuleList to store
            hidden layers
        self.hidden_layers.append(nn.Linear(input_size, n_neurons_h)) # create
            first hidden layer
        for i in range(n_hidden-1):
            self.hidden_layers.append(nn.Linear(n_neurons_h, n_neurons_h)) #
                create the rest of the hidden layers
```

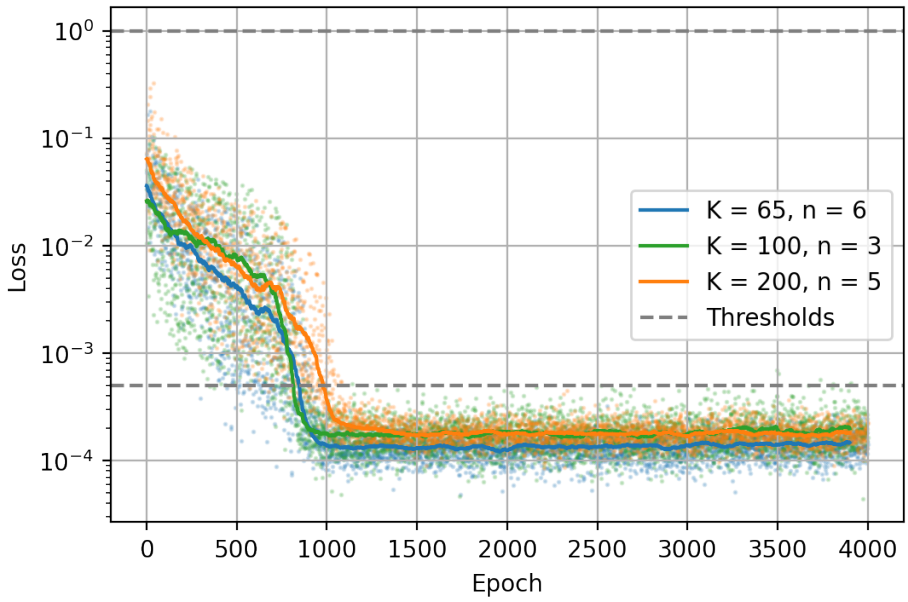

Figure 4: Evolution of the averaged loss function during training for different numbers of points K. The different colors denote one of the 10 runs that were performed for each K. Thresholds of $loss\_thresh = 5e - 4$ and $high\_thresh = 1$ help to discard very bad training rounds before averaging. n denotes the number of runs averaged for each K.

```
        self.output_layer = nn.Linear(n_neurons_h, output_size) # create output
            layer
        self.activation = A

    def forward(self, x):
        '''
        input tensor x is used to perform forward pass through the FNN to create
            output tensor
        '''
        for layer in self.hidden_layers:
            x = self.activation(layer(x)) # go through all hidden layers and
                create input for next hiddenlayer
        x = self.output_layer(x) # produce output tensor # uses softmax for 2
            outputs, sigmoid for 1 output
        return x

# define ODEF with NN
class NNODEF(ODEF):
    def __init__(self):
        super(NNODEF, self).__init__()
        self.hidden_size = 16

        # Define the FNN architecture
        self.fnn = FNN(2, 2, 1, self.hidden_size, nn.ReLU())

    def forward(self, z, t):
        # Implement the forward pass of the FNN
        out = self.fnn(z)
        return out
print(square_root(16))
```

Listing 4: Create ODEF with FNN.

To stop the training not after a set number of epochs as previously, but after the loss function has decreased

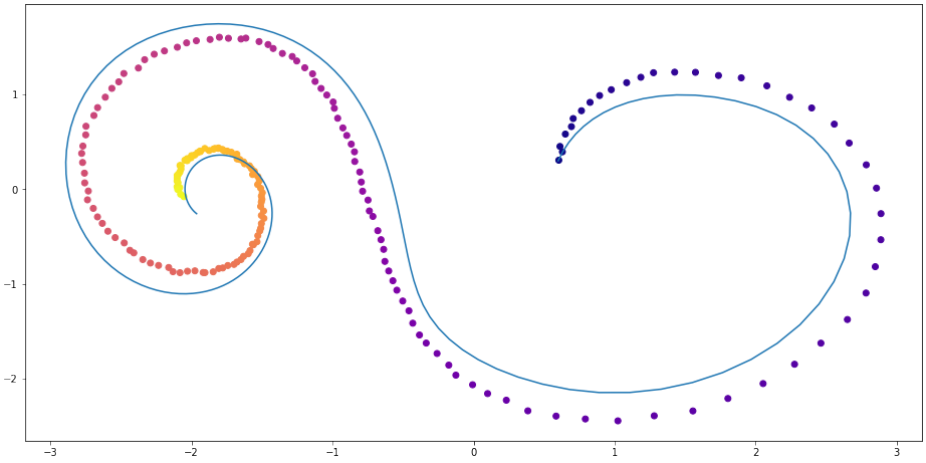

Figure 5: Trajectory after loss has sufficiently decreased.

sufficiently, we have to alter the function *ce_loss* once again. This new function is called *ce_loss_stop* and its code can be found below this paragraph. It takes the variable *max_loss* as an input, stops the training once the loss has undercut *max_loss* for a number of consecutive epochs, set by *counter_thres*. The result for *max_loss* = 5e-4 and *counter_thres* = 3 can be seen in Figure 5. These values were chosen because for *max_loss* to high or *counter_thres* too low, no solution was found within a reasonable number of epochs. Conversely, for *max_loss* too low and textitcounter_thres too high, the resulting solution is usually not satisfying. It can be seen that the network does not achieve a perfect representation of the spiral. However, it should be noted that the overall shape of the spiral is represented well by the blue line. To improve the performance, other network types should be tested.

```
def ce_loss_stop(ode_true, ode_trained, n_steps, name, plot_freq=10, K=200,
    max_loss=1e-4):
    # Create data
    z0 = Variable(torch.Tensor([[0.6, 0.3]]))

    losses = []

    t_max = 6.29*5
    n_points = K

    index_np = np.arange(0, n_points, 1, dtype=int)
    index_np = np.hstack([index_np[:, None]])
    times_np = np.linspace(0, t_max, num=n_points)
    times_np = np.hstack([times_np[:, None]])

    times = torch.from_numpy(times_np[:, :, None]).to(z0)
    obs = ode_true(z0, times, return_whole_sequence=True).detach()
    obs = obs + torch.randn_like(obs) * 0.01

    # Get trajectory of random timespan
    min_delta_time = 1.0
    max_delta_time = 5.0
    max_points_num = 32
    def create_batch():
        t0 = np.random.uniform(0, t_max - max_delta_time)
        t1 = t0 + np.random.uniform(min_delta_time, max_delta_time)

        idx = sorted(np.random.permutation(index_np[(times_np > t0) & (times_np
            < t1)])[:max_points_num])

        obs_ = obs[idx]
        ts_ = times[idx]
```

```
        return obs_, ts_

    # Train Neural ODE
    optimizer = torch.optim.Adam(ode_trained.parameters(), lr=0.01)
    loss = 1
    for i in range(n_steps):
        obs_, ts_ = create_batch()

        z_ = ode_trained(obs_[0], ts_, return_whole_sequence=True)
        loss = F.mse_loss(z_, obs_.detach())
        losses.append(loss.detach().numpy())

        optimizer.zero_grad()
        loss.backward(retain_graph=True)
        optimizer.step()

        print(f"{loss}", end="\r")

        if loss < max_loss:
            break

    z_p = ode_trained(z0, times, return_whole_sequence=True)

    plot_trajectories(obs=[obs], times=[times], trajs=[z_p])
    clear_output(wait=True)
    return np.squeeze(losses)
```

Listing 5: Definition of *ce_loss_stop*

The following code gives insight in how the *ce_loss_stop* was called to create Figure 5. Evidently, also the loss as a function of epoch is saved. This is plotted in Figure 6. It can be seen that while the loss decreases in the first 700 epochs, it stays approximately constant thereafter. Note also the extremely high variance in the loss of almost 4 orders of magnitude. This shows, that *TestODEF* represents a trajectory that is extremely hard for the *NNODEF* to learn. Thus, for a satisfying result, the training has to be stopped at an advantageous time which is ensured by this algorithm. However, through this analysis, it becomes clear that implementing a FNN for $\mathbf{f}$ does not produce the aspired high quality fit as was the case for the simpler spiral.

```
min_count = 3
func = TestODEF(Tensor([[-0.1, -0.5], [0.5, -0.1]]), Tensor([[0.2, 1.], [-1,
    0.2]]), Tensor([[-1., 0.]]))
ode_true_com = NeuralODE(func)

func_lin = NNODEF()
ode_train_hl = NeuralODE(func_lin)

max_loss = 5e-4
loss = ce_loss_stop(ode_true_com, ode_train_hl, 5000, "comp", plot_freq=100,
    max_loss=max_loss, counter_thres=min_count)
```

Listing 6: Call of *ce_loss_stop* with *NNODEF*.

References

[1] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, David Duvenaud, Neural Ordinary Differential Equations, 2019, https://doi.org/10.48550/arXiv.1806.07366.

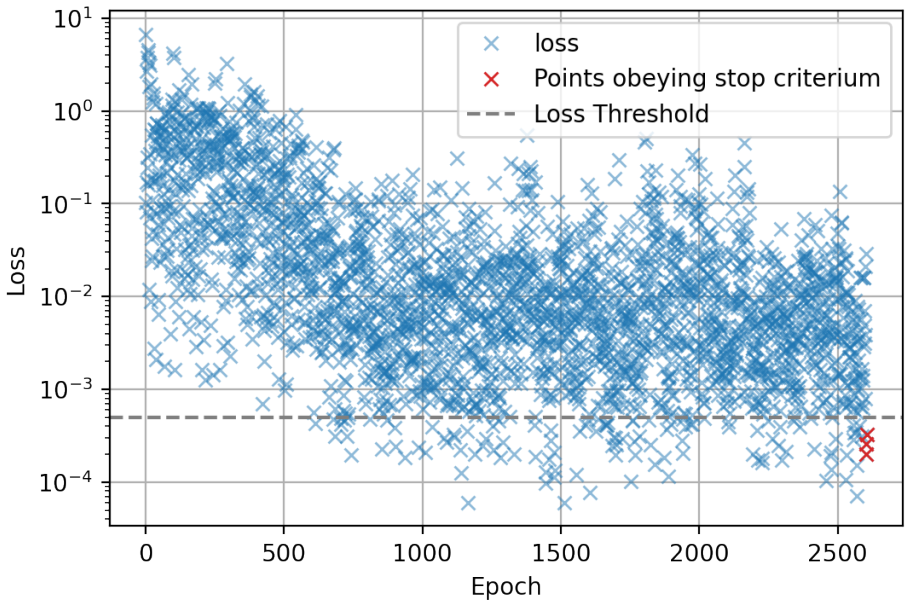

Figure 6: Loss as function of epochs for producing Figure 5. The points, where the set threshold, indicated by the gray dashed line, was undercut for three consecutive epochs, are marked in red. The last one of these points is the loss corresponding to the trajectory shown in Figure 5.