PSE: Apprentissage Profond: Neural ODE pour Systèmes Dynamiques

Spiral and ladscape exploring

Hannah Plath et Thiago Maffei

Prof. Alexandre Allauzen

Importing libraries

```
#######################################################
# Useful imports, preliminary commands, etc

# Numpy and maths
import math
import numpy as np
from scipy.linalg import expm

# Matplotlib and figures-related commands
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
sns.color_palette("bright")
import matplotlib.cm as cm
```

Generating Data

The classes below generate the data from which the model will try to learn.

```
class SpiralData():

  def __init__( self, A = np.array([[-0.1, -1.], [1., -0.1]]), a0 = np.array([0.6,
    self.time = np.linspace(0,tf, n_points)
    self.data = np.zeros((n_points, 2))
    for i in range(n_points):
      self.data[i] = expm(self.time[i]*A)@a0

  def getData(self):
    return self.data, self.time

  def scatterGraph(self):
    plt.scatter(self.data[0, 0],self.data[0,1],marker="o",c="r")
    plt.scatter(self.data[1:, 0],self.data[1:,1],marker="o")
```

```
      plt.grid()

  class LorenzData():

    def __init__(self, rho = 28, sigma = 10, beta = 8/3, init_values = [0.01, 0, 0],
      self.time = np.linspace(0,tf, n_points)
      self.data = np.zeros((n_points, 3))
      self.data[0] = init_values
      x = init_values[0]
      y = init_values[1]
      z = init_values[2]

      for n in range(n_points - 1):
        dx = sigma*(y - x)*dt
        dy = (x*(rho - z) - y)*dt
        dz = (x*y - beta*z)*dt

        x += dx
        y += dy
        z += dz

        self.data[n+1] = [x,y,z]

    def getData(self):
      return self.data, self.time

    def scatterGraph(self, size = (12,10)):
        fig = plt.figure(figsize=size)
        ax = plt.axes(projection='3d')
        ax.plot3D(self.data[:,0], self.data[:,1], self.data[:,2], 'blue')
        ax.scatter3D(self.data[:,0], self.data[:,1], self.data[:,2], color = 'red');
```

Neural ODE

We determine a ODE solver to "solve" the neural network (do the feedforward propagation).

```
def ode_solve(x0, t0, t1, f, n_steps = 1):
    """
    Simplest Euler ODE initial value solver
    """
    eps = (t1 - t0)/n_steps
    t = t0
    h = x0
    for i_step in range(n_steps):
        h = h+ eps*f(h)
        t = t + eps
    return h
```

```
u = nn.Linear(2,2)
Params2Vec(u.parameters())
```

```
indices = th.tensor([0, 1, 2, 3])
th.index_select(Params2Vec(u.parameters()), 0, indices)
```

We define the Neural ODE class.

```
import torch as th
import torch.nn as nn
from torch.nn  import functional as F

class ODE_Function(nn.Module):

  def unfold_ode(self, z0,times, nsteps=1):
  """
    Applies the ODE solver to the network
  """
        bs, *z_shape = z0.size()
        time_len = times.size(0)
        zs = th.zeros(time_len, bs, *z_shape).to(z0)
        zs[0] = z0
        for i_t in range(time_len - 1):
          z0 = ode_solve(z0, times[i_t], times[i_t+1], self.network, nsteps)
          zs[i_t+1] = z0
        return zs
```

We define two types of neural network:

- Linear
- Non Linear

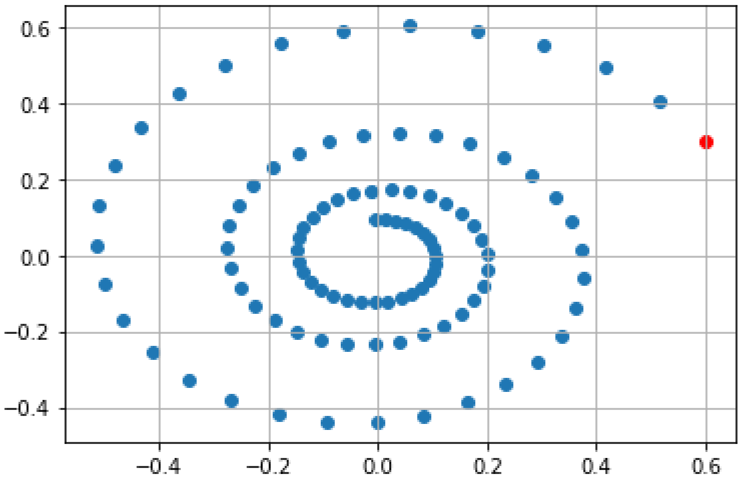
```
class LinearODE(ODE_Function):

  def __init__(self, dim = 2, W = None, bias = None):
    super(LinearODE, self).__init__()
    if bias is not None:
      self.network = nn.Linear(dim,dim)
      self.network.bias = nn.Parameter(bias)
    else:
      self.network = nn.Linear(dim,dim,bias = False)
    # if we need to setup the init. value of W
    if W is not None :
      self.network.weight = nn.Parameter(W)

  def forward(self, z):
    return self.newtwork(z)

class NonLinearODE(ODE_Function):

  def __init__(self, dim):
    super(NonLinearODE, self).__init__()
    # modules with parameters
    self.network = nn.Sequential(
        nn.Linear(dim,2*dim),
        nn.ReLU(),
        nn.Linear(2*dim, dim)
        )

  def forward (self, z):
    return self.network(z)
```

We define a class that contains all the steps we need to perform in order to test the model in different data. This is important because it automates the testing process and ensures that all tests are performed in the same way (reproducibility).

```
class Experience():

  def training(self, model, data, time, optimizer, nepoch, nsteps):

    self.data = th.Tensor(data)
    z0 = self.data[0]
    self.time = th.from_numpy(time).to(z0)
    self.nepoch = nepoch

    self.model = model

    with th.no_grad():
      zs = self.model.unfold_ode(z0, self.time)

    wsize = 1
    nprint = 10
    indices = np.array(range(self.time.shape[0]-wsize),dtype=np.int)
    self.losses = th.zeros(self.nepoch)

    for e in range(self.nepoch):
      np.random.shuffle(indices)
      ave_loss = th.zeros(1)

      for i in indices:
        x = self.data[i]
        out = self.model.unfold_ode(x,self.time[i:i+wsize+1],nsteps)[-1]
        loss = F.mse_loss(out, self.data[i+wsize].detach())
        ave_loss += loss
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

      self.losses[e] = ave_loss.item()
      if (e%nprint) == 0:
        print(e, ave_loss)

  def getLearnedModel(self):
    return self.model

  def predict(self, data, times, nstep):
    data = th.Tensor(data)
    z0 = data[0]
    times = th.from_numpy(times).to(z0)

    with th.no_grad():
      zpred = self.model.unfold_ode(z0,times,nstep)

    return zpred

  def plotLosses(self, size = (12, 9)):

    plt.plot(self.losses, range(self.nepoch))
    plt.title('Evolution of loss per epoch')
    plt.figure(figsize = size)

  def getParams(self):
      return self.model.network.weight
```

Spiral

```
#generate data
s = SpiralData()
spiral_data, spiral_time = s.getData()
s.scatterGraph()
```

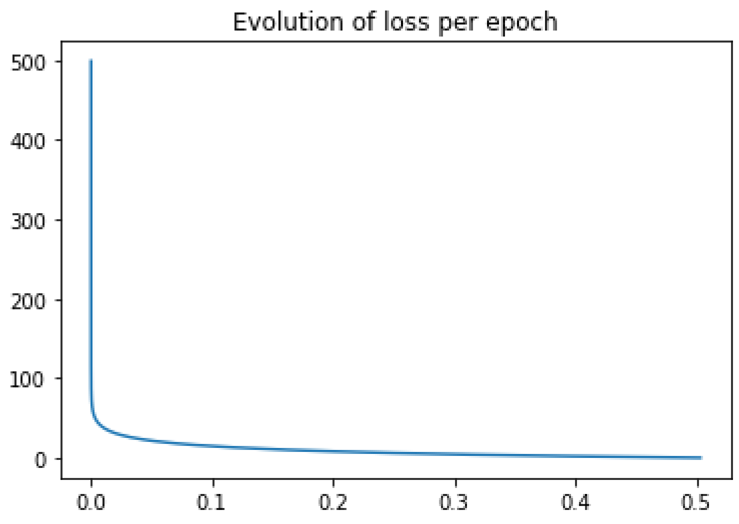


```
#define the parameters of the training
nepochs = 500
nsteps = 5
#initialize the parameters the network need to learn
np.random.seed(1)
w = th.rand(2,2)
b = th.rand(2)

#----------------------------------
#we define our models
```

```
#without bias
model = LinearODE(2,w)

#with bias

model_bias = LinearODE(2,w,b)

#run the experience for model without bias
optimizer = th.optim.SGD(model.parameters(),lr=1e-2,momentum=0.95)
e = Experience()
e.training(model, spiral_data, spiral_time, optimizer, nepochs, nsteps)


#run the experience for model with bias
optimizer = th.optim.SGD(model_bias.parameters(),lr=1e-2,momentum=0.95)
e_bias = Experience()
e_bias.training(model_bias, spiral_data, spiral_time, optimizer, nepochs, nsteps)


zpred = e.predict(spiral_data, spiral_time, nstep=5)
zpred_bias = e_bias.predict(spiral_data, spiral_time, nstep=5)


#final parameters matrix for both models
print(e.getParams())
print(e_bias.getParams())
```

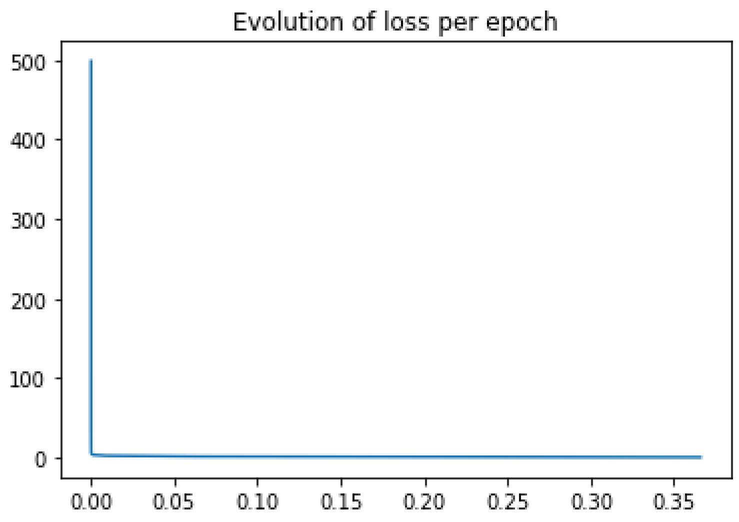
```
Parameter containing:
tensor([[-0.1199, -0.9957],
        [ 0.9957, -0.1199]], requires_grad=True)
Parameter containing:
tensor([[-0.1199, -0.9957],
        [ 0.9957, -0.1199]], requires_grad=True)
```

```
#evolution of loss for model without bias
e.plotLosses()
```



```
<Figure size 864x648 with 0 Axes>
```

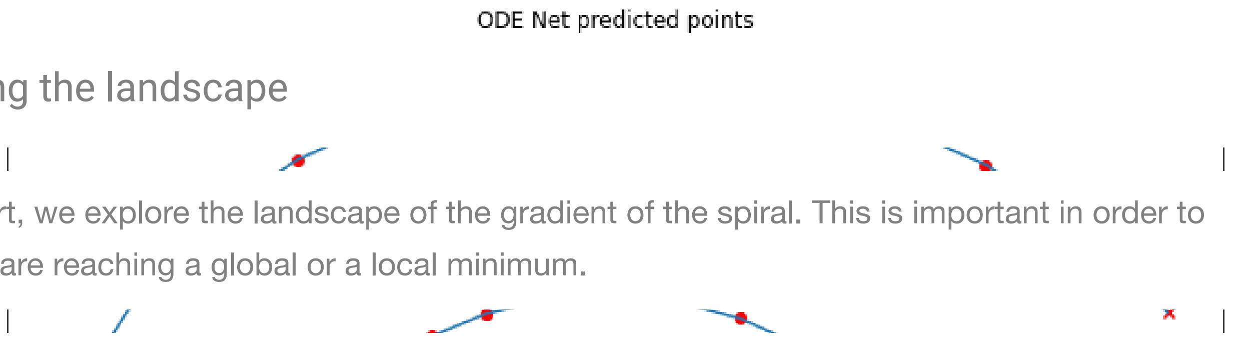
```
#evolution of loss for model with bias
e_bias.plotLosses()
```



```
<Figure size 864x648 with 0 Axes>
```

```
#we compare the prediction of our model with the real points

plt.figure(figsize=(12,9))
plt.title('ODE Net predicted points')

plt.scatter(spiral_data[0, 0],spiral_data[0,1], marker="x",c="r", label='First Poi
plt.scatter(spiral_data[1:, 0],spiral_data[1:,1],marker="o",c="r", label='Real Poi
plt.plot(zpred[:, 0], zpred[:, 1], lw=1.5, label='ODE Net Fit')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f9f0ff0c8d0>
```

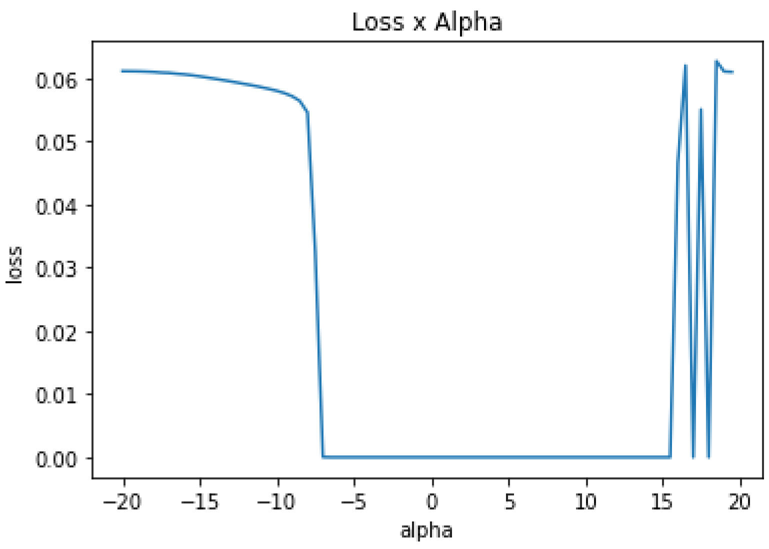


Exploring the landscape

In this part, we explore the landscape of the gradient of the spiral. This is important in order to see if we are reaching a global or a local minimum.

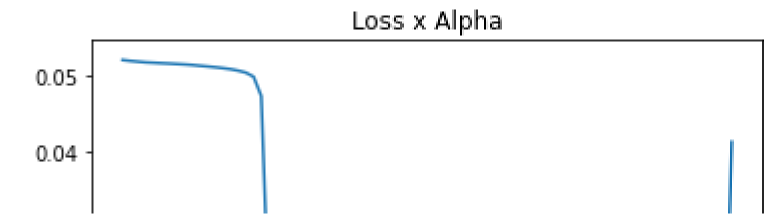
```
def changeTheta(**kwargs):
"""
  Changes the parameters the start parameters of the model
"""
  theta = kwargs.get('theta')
  alpha = kwargs.get('alpha')
  column = kwargs.get('column')

  if(column == None):
    return (theta + alpha).reshape((2,2))
  elif(column == 0):
    array = theta.detach().numpy()
    array[0] += alpha
    array[2] += alpha
  elif(column == 1):
    array = theta.detach().numpy()
    array[1] += alpha
    array[3] += alpha

  return (th.Tensor(array)).reshape((2,2))


from torch.nn.utils import (
  parameters_to_vector as Params2Vec,
  vector_to_parameters as Vec2Params
)

def exploreLandscape(init, final, step, w , column = None):
"""
  Explore the gradient landscape
"""
  losses = []
  a = np.arange(init, final, step)

  for alpha in a:
    m = LinearODE(2, w)
    theta_init = Params2Vec(m.parameters())
    ntheta = changeTheta(theta = theta_init, alpha = alpha, column = column)
    Vec2Params(ntheta,  m.parameters())
    optimizer = th.optim.SGD(m.parameters(),lr=1e-2,momentum=0.95)
    exp = Experience()
    exp.training(m, spiral_data, spiral_time, optimizer, nepoch = 500, nsteps = 5)
    prediction = exp.predict(spiral_data, spiral_time, nstep=5)
    mse_loss = nn.MSELoss()
```

```
      loss = mse_loss(prediction, th.from_numpy(spiral_data))
      print("for alpha ", alpha, "loss equals ", loss)
      losses.append(loss)

    return losses, a

  def plot_loss(alpha,losses):
    plt.plot(alpha, losses)
    plt.ylabel('loss')
    plt.xlabel('alpha')
    plt.title('Loss x Alpha')
    plt.figure(figsize = (12,9))


  #initialize the parameters of model
  w = th.rand(2,2)
  #experiment changing all the values of the parameters
  losses, alpha = exploreLandscape(-20,20,0.5, w)


  plot_loss(alpha,losses)
```



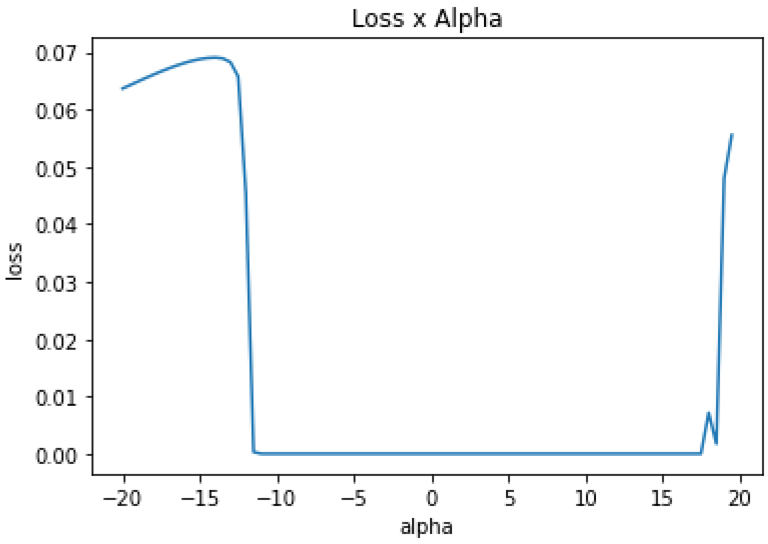
```
    <Figure size 864x648 with 0 Axes>
```

```
  #experiment changing only the first column of the matrix of parameters
  losses2, alpha2 = exploreLandscape(-20,20,0.5, w, 0)


  plot_loss(alpha2,losses2)
```



```
#experiment changing only the second column of the matrix of parameters
losses3, alpha3 = exploreLandscape(-20, 20, 0.5, w, 1)
```

```
plot_loss(alpha3,losses3)
```



```
<Figure size 864x648 with 0 Axes>
```

As for tin the first experiment, there was a lot of instability when alpha increased, we decided to explore the landscape in that direction.

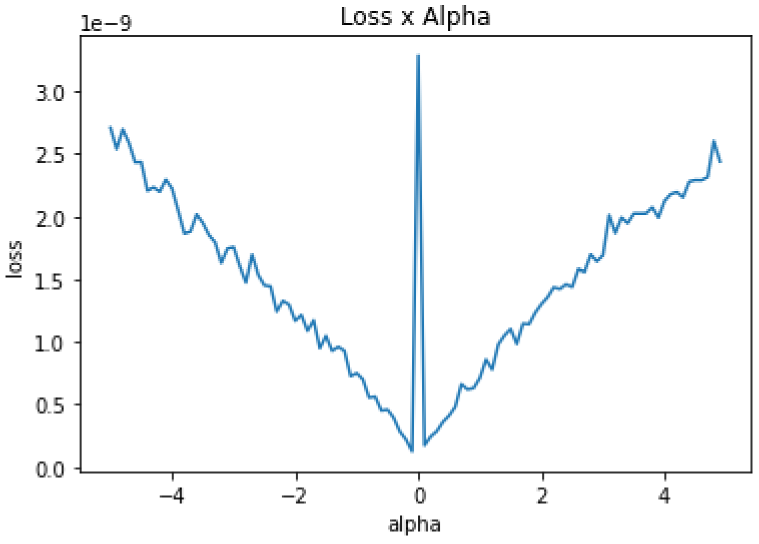
```
#experiment changing all parameters of the matrix
losses4, alpha4 = exploreLandscape(17, 23, 0.2, w)
```

```
plot_loss(alpha4, losses4)
```

We look at the landscape around zero, that is, around the minimum.

```
losses5, alpha5 = exploreLandscape(-5, 5, 0.1, w)
```

```
plot_loss(alpha5, losses5)
```



```
<Figure size 864x648 with 0 Axes>
```

Training with extreme initial condition

We initialize our model with parameters that are in the region of instability to verify if our model can still learn correctly.

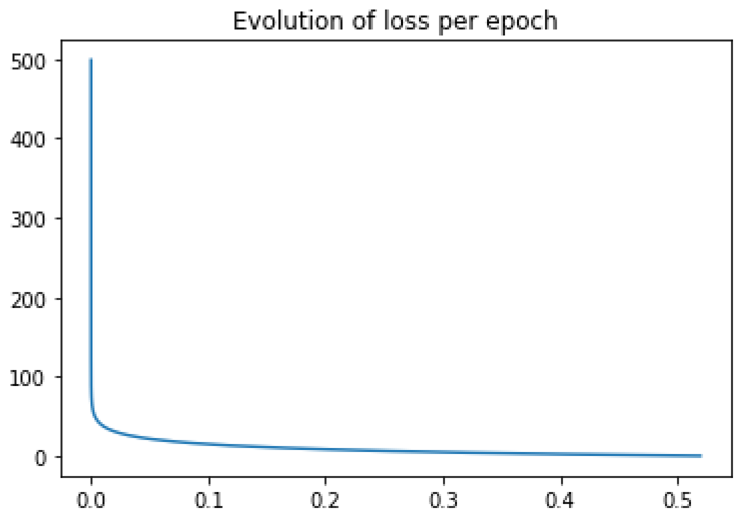
Negative initial value

```
nepochs = 500
nsteps = 5

np.random.seed(1)
w = th.rand(2,2)

extreme_model_1 = LinearODE(2, w)
theta_init = Params2Vec(extreme_model_1.parameters())
ntheta = changeTheta(theta = theta_init, alpha = -15)

optimizer = th.optim.SGD(extreme_model_1.parameters(),lr=1e-2,momentum=0.95)
e_extreme1 = Experience()
e_extreme1.training(extreme_model_1, spiral_data, spiral_time, optimizer, nepochs,
```

```
e_extreme1.getParams()
```

```
Parameter containing:
tensor([[-0.1199, -0.9957],
        [ 0.9957, -0.1199]], requires_grad=True)
```

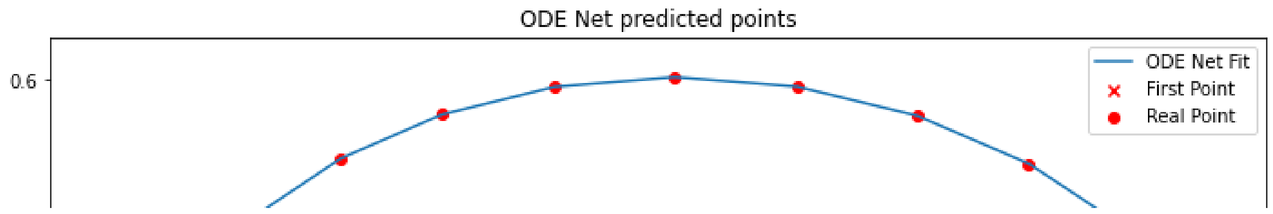
```
zpred_extreme_1 = e_extreme1.predict(spiral_data, spiral_time, nstep=5)
e_extreme1.plotLosses()
```



```
<Figure size 864x648 with 0 Axes>
```

```
plt.figure(figsize=(12,9))
plt.title('ODE Net predicted points')

plt.scatter(spiral_data[0, 0],spiral_data[0,1], marker="x",c="r", label='First Poi
plt.scatter(spiral_data[1:, 0],spiral_data[1:,1],marker="o",c="r", label='Real Poi
plt.plot(zpred_extreme_1[:, 0], zpred_extreme_1[:, 1], lw=1.5, label='ODE Net Fit'
plt.legend()
```

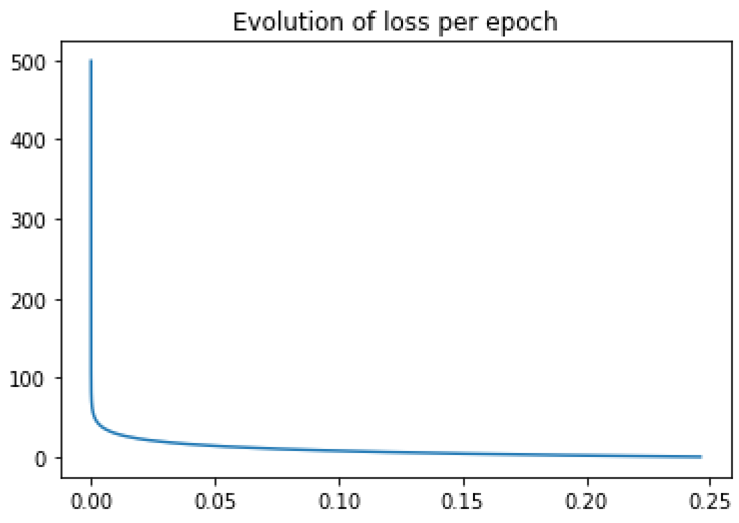
```
<matplotlib.legend.Legend at 0x7fed51075710>
```



Positive initial value

```
nepochs = 500
nsteps = 5

np.random.seed(1)
w = th.rand(2,2)

extreme_model_2 = LinearODE(2, w)
theta_init = Params2Vec(extreme_model_2.parameters())
ntheta = changeTheta(theta = theta_init, alpha = 18)

optimizer = th.optim.SGD(extreme_model_2.parameters(),lr=1e-2,momentum=0.95)
e_extreme2 = Experience()
e_extreme2.training(extreme_model_2, spiral_data, spiral_time, optimizer, nepochs,
```

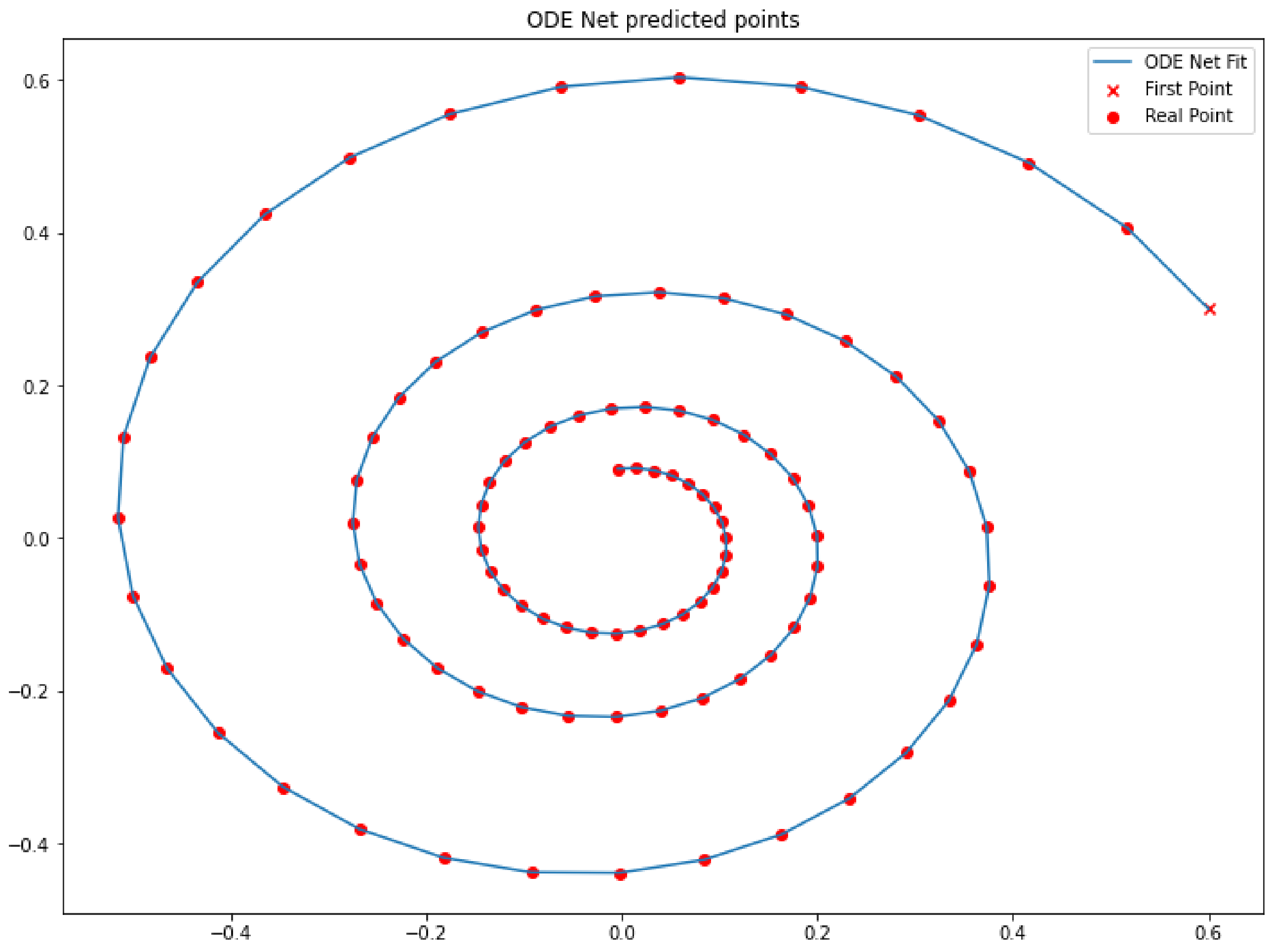
```
zpred_extreme_2 = e_extreme2.predict(spiral_data, spiral_time, nstep=5)
e_extreme2.plotLosses()
```



```
<Figure size 864x648 with 0 Axes>
```

```
plt.figure(figsize=(12,9))
plt.title('ODE Net predicted points')

plt.scatter(spiral_data[0, 0],spiral_data[0,1], marker="x",c="r", label='First Poi
plt.scatter(spiral_data[1:, 0],spiral_data[1:,1],marker="o",c="r", label='Real Poi
plt.plot(zpred_extreme_2[:, 0], zpred_extreme_2[:, 1], lw=1.5, label='ODE Net Fit'
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fed50268a10>
```


ODE Net predicted points
ODE Net Fit
First Point
Real Point
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.4
−0.2
0.0
0.2
0.4
0.6