Lorenz_analysis

March 26, 2024

```
[9]: # import numpy, scipy, and matplotlib
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

import torch
%matplotlib widget

import os
import tempfile

# torch random seed
torch.manual_seed(0)
```

```
[9]: <torch._C.Generator at 0x7fa9948d5590>
```

1 Lorenz analysis

In this notebook we compare the performance of the different model on the lorenz dataset.

1.1 Useful func

```
[10]: def lorenz(t, Y):
    """
    This function returns the right-hand side of the Lorenz system of ordinary␣
 ↪differential equations.
    Parameters:
    t (float): time
    Y (array): state vector [x, y, z]

    Returns:
    array: [dxdt, dydt, dzdt]
    """

    x, y, z = Y
    dxdt = 10 * (y - x)
    dydt = x * (28 - z) - y
    dzdt = x * y - 8/3 * z

    return [dxdt, dydt, dzdt]


def seq_lorenz(Y):
    """
    This function returns the right-hand side of the Lorenz system of ordinary␣
  ↪differential equations.
    Parameters:
    Y (array): state vector [x, y, z]

    Returns:
    array: [dxdt, dydt, dzdt]
    """

    x, y, z = Y[:, 0], Y[:, 1], Y[:, 2]
    dxdt = 10 * (y - x)
    dydt = x * (28 - z) - y
    dzdt = x * y - 8/3 * z

    return np.array([dxdt, dydt, dzdt]).T
```

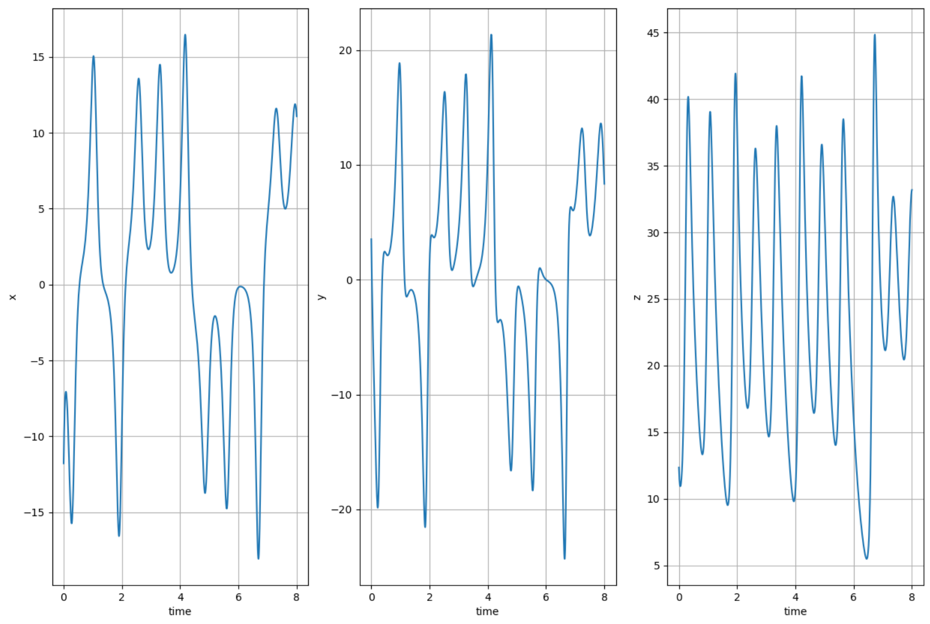
[11]:
```
def plot_components(X, Y):
    """
    This function plots the components of the state vector Y as a function of␣
  ↪time X.
    """
    plt.figure(figsize=(15, 10))
    plt.subplot(131)
    plt.plot(X, Y[:,0], label='x')
    plt.ylabel('x')
    plt.xlabel('time')
    plt.grid()

    # do all the same for y
    plt.subplot(132)
    plt.plot(X, Y[:,1], label='y')
    plt.ylabel('y')
    plt.xlabel('time')
    plt.grid()

    # do all the same for z
    plt.subplot(133)
    plt.plot(X, Y[:,2], label='z')
    plt.ylabel('z')
    plt.xlabel('time')
    plt.grid()



def plot_compare_components(X, Y, Y_pred):
    """
    This function plots the components of the state vector Y and Y_pred as a␣
 ↪function of time X.
    """
    plt.figure(figsize=(15, 10))
    plt.subplot(131)
    plt.plot(X, Y[:,0], label='x')
    plt.plot(X, Y_pred[:,0], label='x_pred')
    plt.ylabel('x')
    plt.xlabel('time')
    plt.legend()
    plt.grid()

    # do all the same for y
    plt.subplot(132)
    plt.plot(X, Y[:,1], label='y')
    plt.plot(X, Y_pred[:,1], label='y_pred')
    plt.ylabel('y')
    plt.xlabel('time')
    plt.legend()
    plt.grid()

    # do all the same for z
    plt.subplot(133)
    plt.plot(X, Y[:,2], label='z')
    plt.plot(X, Y_pred[:,2], label='z_pred')
    plt.ylabel('z')
    plt.xlabel('time')
    plt.legend()
    plt.grid()


def show_history(history, name=None):
    """
    This function plots the loss and learning rate as a function of epoch.
    """
    history = np.array(history)
    fig, ax = plt.subplots(figsize=(15, 10))

    # plot the loss
    ax.plot(history[:, 0], label='loss')
    ax.legend(loc='upper left')
    ax.set_yscale('log')
    ax.set_xlabel('epoch')
    ax.set_ylabel('loss')
    plt.grid()

    # plot the learning rate
    ax2 = ax.twinx()
    ax2.plot(history[:, -1], label='lr', color='r')
    ax2.set_yscale('log')
    ax2.set_ylabel('lr')
    # legend to the right
    ax2.legend(loc='upper right')
    plt.grid()
    plt.title('history' + name)

    # history to list
    history = history.tolist()



# def a function called ode_loss
def lorenz_loss_ode(model, X):
    """
    This function calculates the loss of the ode for the Lorenz system
    as a function of the input tensor X (time).
    Parameters:
    model (torch.nn.Module): the model
    X (torch.tensor): the input tensor
    """

    X.requires_grad = True
    Y_pred = model(X)

    # get the derivatives
    dx_dt_pred = torch.autograd.grad(Y_pred[:,0], X, grad_outputs=torch.
 ↪ones_like(Y_pred[:,0]), create_graph=True)[0]
    dy_dt_pred = torch.autograd.grad(Y_pred[:,1], X, grad_outputs=torch.
 ↪ones_like(Y_pred[:,1]), create_graph=True)[0]
    dz_dt_pred = torch.autograd.grad(Y_pred[:,2], X, grad_outputs=torch.
 ↪ones_like(Y_pred[:,2]), create_graph=True)[0]

    # get true derivatives, using the lorenz parameter
    dx_dt_ode = 20 * (Y_pred[:,1] - Y_pred[:,0])
    dy_dt_ode = Y_pred[:,0] * (28 - Y_pred[:,2]) - Y_pred[:,1]
    dz_dt_ode = Y_pred[:,0] * Y_pred[:,1] - 8/3 * Y_pred[:,2]
    # loss ode
    loss_ode = (dx_dt_pred[:,0]- dx_dt_ode)**2 + (dy_dt_pred[:,0]-␣
 ↪dy_dt_ode)**2 + (dz_dt_pred[:,0]- dz_dt_ode)**2

    return loss_ode



def plot_propagation(X, Y_true, Y_pred, Y0_index):
    """
    This function compare predicted and true solution of the lorenz system,
    in addition there is the solution (with solve_ivp) propagated from the␣
 ↪predicted state at the Y0_index.
    Parameters:
    X (array): time
    Y_true (array): true state
    Y_pred (array): predicted state
    Y0_index (int): index of the predicted state to be used as initial condition
    """

    # get the predicted state for the maximum index
    Y0 = Y_pred[Y0_index]

    # evove with the lorenz function, use scipy ivp_solve
    sol = sp.integrate.solve_ivp(lorenz, [X[Y0_index], 1], Y0,␣
 ↪t_eval=X[Y0_index:])

    # plot the solution
    plt.figure(figsize=(15, 10))
    plt.subplot(131)
    plt.plot(sol.t, sol.y[0], label='x_prop', marker='o')
    # plot the predicted solution
    plt.plot(X, Y_pred[:,0], label='x_pred')
    # plot x real
    plt.plot(X, Y_true[:,0], label='x_real')
    # red dot in the maximum index
    plt.plot(X[Y0_index], Y_pred[Y0_index,0], 'ro')
    plt.ylabel('x')
    plt.xlabel('t')
    plt.grid()
    plt.legend()

    # do all the same for y
    plt.subplot(132)
    plt.plot(sol.t, sol.y[1], label='y_prop', marker='o')
    plt.plot(X, Y_pred[:,1], label='y_pred')
    plt.plot(X, Y_true[:,1], label='y_real')
    plt.plot(X[Y0_index], Y_pred[Y0_index,1], 'ro')
    plt.ylabel('y')
    plt.xlabel('t')
    plt.grid()

    # do all the same for z
    plt.subplot(133)
    plt.plot(sol.t, sol.y[2], label='z_prop', marker='o')
    plt.plot(X, Y_pred[:,2], label='z_pred')
    plt.plot(X, Y_true[:,2], label='z_real')
    plt.plot(X[Y0_index], Y_pred[Y0_index,2], 'ro')
    plt.ylabel('z')
    plt.xlabel('t')
    plt.grid()
```

[34]:
```
class PINN_LearningSchedule:
    # write documentation
    """
    This class implements the learning schedule for the PINN model.
    """

    def __init__(self, experiment_folder, experiment_name,
                 precision=None, n_points_checkpoint=1, model=None):
        """
        This function initializes the class.
        Checks if the experiment folder exists, if not, creates it.

        Parameters:
        experiment_folder (str): path to the experiment folder
        experiment_name (str): name of the experiment
        precision (float): precision hyperparameter
        n_points_checkpoint (int): number of points to save the model
        model (torch.nn.Module): model to train
        """
        self.experiment_folder = experiment_folder
        self.experiment_name = experiment_name

        self.n_points_checkpoint = n_points_checkpoint
        self.precision = precision

        self.X = None
        self.Y = None

        # if experiment folder does not exist, create it
        if not os.path.exists(experiment_folder):
            os.makedirs(experiment_folder)
            self.experiment_folder = experiment_folder
            self.model = model

            self.n_points = 1
            self.history =[]
            self.epochs = 0

            # print start new experiment
            print('Start new experiment: ', experiment_folder)

        # if experiment folder exists, load the last model
        else:
            self.experiment_folder = experiment_folder

            # get the last model
            list_files = [f for f in os.listdir(experiment_folder) if f.
↪endswith('.pt')]
            number_of_points = [f.split('_')[-1] for f in list_files]
            number_of_points = [f.split('.')[0] for f in number_of_points]
            number_of_points = [int(f) for f in number_of_points]
            max_index = number_of_points.index(max(number_of_points))
            self.n_points = max(number_of_points)

            # load the last model and history
            self.model = torch.load(os.path.join(experiment_folder,␣
↪list_files[max_index]))
            self.history = np.load(os.path.join(experiment_folder,␣
↪list_files[max_index].split('.')[0] + '_history.npy'))
            self.epochs = self.history[-1, 0]
            self.history = self.history.tolist()
            # get epochs

            # print load existing experiment, gives the number of points
            print('Load existing experiment: ', experiment_folder, ' with ',␣
↪self.n_points, ' points', ' and ', self.epochs, ' epochs')


    def load_data(self, X, Y):
        """
        This function loads the data to the class.
        """
        self.X = X
        self.Y = Y


    def train(self, max_lr, min_lr, patience=700, factor=0.7):
        """
        This function trains the model using the learning schedule.
        """

        optimizer = torch.optim.Adam(self.model.parameters(), lr=max_lr)
        scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,␣
↪'min', factor=factor, patience=patience, min_lr=min_lr)
        loss_fn = torch.nn.MSELoss()

        while True:
            while True:

                # cut the data to n_points
                X_temp = self.X[:self.n_points]
                Y_temp = self.Y[:self.n_points]

                # convert to torch tensor
                X_temp = torch.tensor(X_temp, dtype=torch.float32).view(-1, 1)
                Y_temp = torch.tensor(Y_temp, dtype=torch.float32)



               ␣
↪#-------------------------------------------------------------------------------------------
                # PINN training
                # Train the model using the PINN loss
                # for one epoch
               ␣
↪#-------------------------------------------------------------------------------------------

                # train the model
                optimizer.zero_grad()
                X_temp.requires_grad = True
                Y_pred = self.model(X_temp)

                # get the derivatives
                dx_dt_pred = torch.autograd.grad(Y_pred[:,0], X_temp,␣
↪grad_outputs=torch.ones_like(Y_pred[:,0]), create_graph=True)[0]
                dy_dt_pred = torch.autograd.grad(Y_pred[:,1], X_temp,␣
↪grad_outputs=torch.ones_like(Y_pred[:,1]), create_graph=True)[0]
                dz_dt_pred = torch.autograd.grad(Y_pred[:,2], X_temp,␣
↪grad_outputs=torch.ones_like(Y_pred[:,2]), create_graph=True)[0]

                # get true derivatives, using the lorenz parameter
                dx_dt_ode = 10 * (Y_pred[:,1] - Y_pred[:,0])
                dy_dt_ode = Y_pred[:,0] * (28 - Y_pred[:,2]) - Y_pred[:,1]
                dz_dt_ode = Y_pred[:,0] * Y_pred[:,1] - 8/3 * Y_pred[:,2]
                # loss ode
                loss_ode = loss_fn(dx_dt_pred[:,0], dx_dt_ode) +␣
↪loss_fn(dy_dt_pred[:,0], dy_dt_ode) + loss_fn(dz_dt_pred[:,0], dz_dt_ode)

                # add loss ic
                loss_ic = torch.mean((Y_pred[0] - Y_temp[0])**2)

                loss = 20*loss_ode + loss_ic

                loss.backward()
                optimizer.step()
                scheduler.step(loss)
               ␣
↪#------------------------------------------------------------------------------------------


                self.epochs += 1


                # save history
                if self.epochs % 1000 == 0:
                    self.history.append([self.epochs, loss.item(), loss_ode.
↪detach().numpy(), loss_ic.detach().numpy(), optimizer.param_groups[0]["lr"],␣
↪self.n_points])
                    print(f'N_points, {self.n_points}, Epoch {self.epochs},␣
↪Loss {loss.item()}, lr {optimizer.param_groups[0]["lr"]}')

                # check loss, if less than precision, break and n_points + 1
                if loss < self.precision:
                    break

            # save the model at n_points_checkpoint
            if self.n_points % self.n_points_checkpoint == 0:
                date = datetime.datetime.now().strftime("%Y%m%d%H%M%S")
                torch.save(self.model, os.path.join(self.experiment_folder,␣
↪f'{self.experiment_name}_{date}_{self.n_points}.pt'))
                # print model saved
                print('Model saved at ', os.path.join(self.experiment_folder,␣
↪f'{self.experiment_name}_{date}_{self.n_points}.pt'))
                # save history
                np.save(os.path.join(self.experiment_folder, f'{self.
↪experiment_name}_{date}_{self.n_points}_history.npy'), self.history)

            # if num_pint equal total number of points, break
            if self.n_points == len(self.X):
                # print end of experiment
                print('End of experiment, reached the end of the data')
                break

            # increase n_points
            self.n_points = self.n_points + 1

            # restart lr scheduler
            optimizer = torch.optim.Adam(self.model.parameters(), lr=max_lr)
            scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,␣
↪'min', factor=factor, patience=patience, min_lr=min_lr)

    def get_history(self):
        return self.history
```

1.2 Load data

[12]:
```
# import data
# data are generated by "src/DHOscillator_data_gen.py"
data = np.load('../data/Lorenz_data.npy')
# Y is the state, X is the time, Y is made of x, y, z
X = data[:,0]
Y = data[:,1:]
```

[13]:
```
# plot components
plot_components(X, Y)
```



1.3 Define the model

This is inspired by the tuned model on DHO

[14]:
```
# Model class
class FFNN(torch.nn.Module):
    def __init__(self, n_layers, n_neurons):
        super(FFNN, self).__init__()
        layers = []
        for i in range(n_layers):
            if i == 0:
                layers.append(torch.nn.Linear(1, n_neurons))
            else:
                layers.append(torch.nn.Linear(n_neurons, n_neurons))
            layers.append(torch.nn.Tanh())
        layers.append(torch.nn.Linear(n_neurons, 3))
        self.model = torch.nn.Sequential(*layers)
    def forward(self, x):
        return self.model(x)
```

[15]:
```
n_layers = 4
n_neurons = 28
n_epochs = 10000
# create model
short_model = FFNN(n_layers, n_neurons)
```

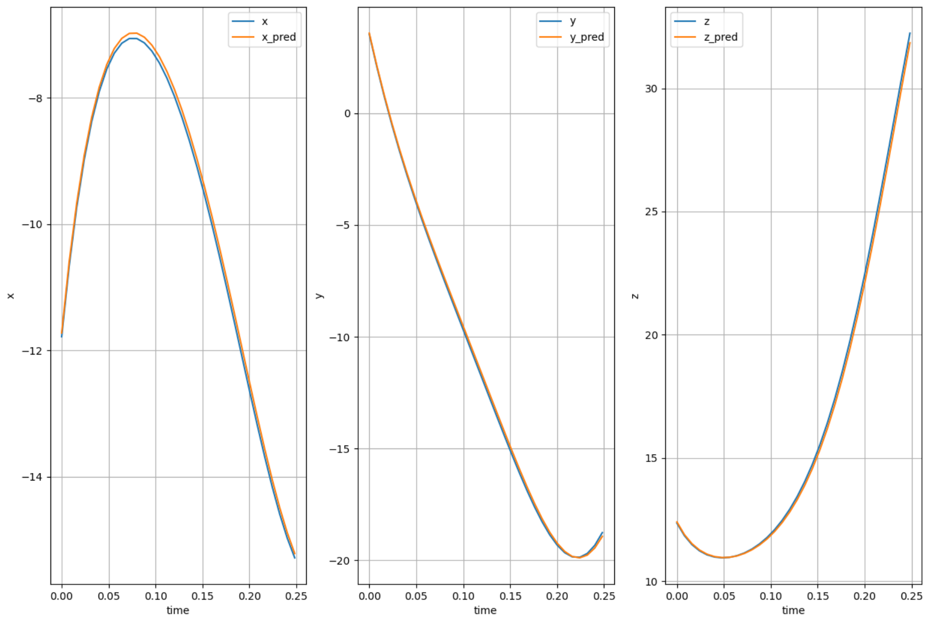
1.4 Short and long model

Here we present the results of the short and long model, we start with the short model.

[23]:
```
# select only time < 0.25
X_sub = X[X<0.25]
Y_sub = Y[X<0.25]

# to torch
X_sub_t = torch.tensor(X_sub).float().view(-1, 1)
Y_sub_t = torch.tensor(Y_sub).float()
```

[24]:
```
# load ../model/short_model_lorenz.pt
short_model = torch.load('../models/short_model_lorenz.pt')
```

[25]:
```
# plot compare
Y_pred = short_model(X_sub_t)
Y_pred = Y_pred.detach().numpy()
plot_compare_components(X_sub, Y_sub, Y_pred)
```

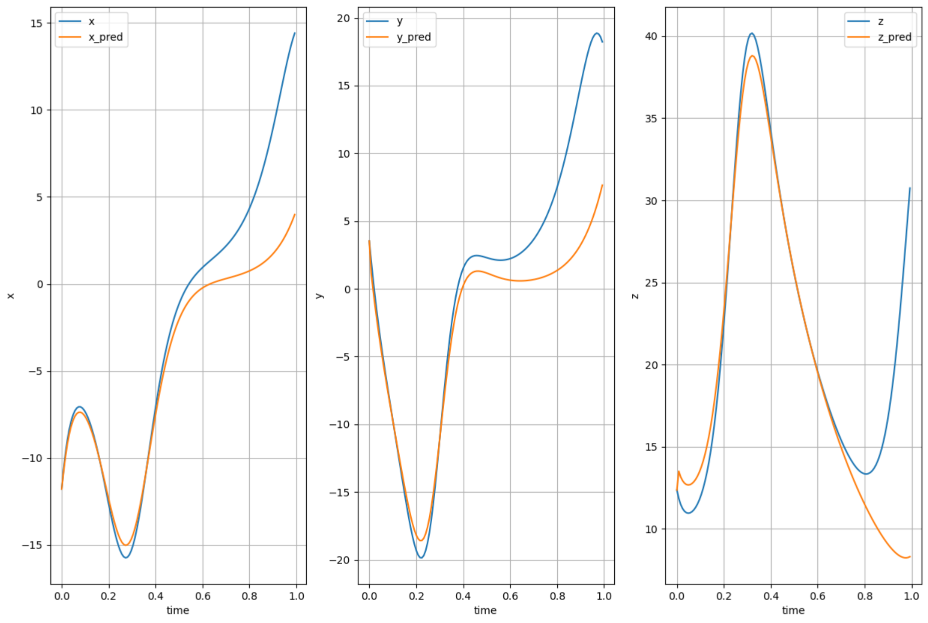


[26]:
```
# get mse
mse = torch.nn.MSELoss()
loss = mse(short_model(X_sub_t), Y_sub_t)
# print short mse
print('short model mse:', loss.item())
```

```
short model mse: 0.025902362540364265
```

[27]:
```
# select only time < 0.25
X_sub = X[X<1]
Y_sub = Y[X<1]

# to torch
X_sub_t = torch.tensor(X_sub).float().view(-1, 1)
Y_sub_t = torch.tensor(Y_sub).float()
```

[28]:
```
# load ../model/long_model_lorenz.pt
long_model = torch.load('../models/long_model_lorenz.pt')
# plot compare
Y_pred = long_model(X_sub_t)
Y_pred = Y_pred.detach().numpy()
plot_compare_components(X_sub, Y_sub, Y_pred)
```



```
[29]: # get and print mse
      loss = mse(long_model(X_sub_t), Y_sub_t)
      print('long model mse:', loss.item())
```

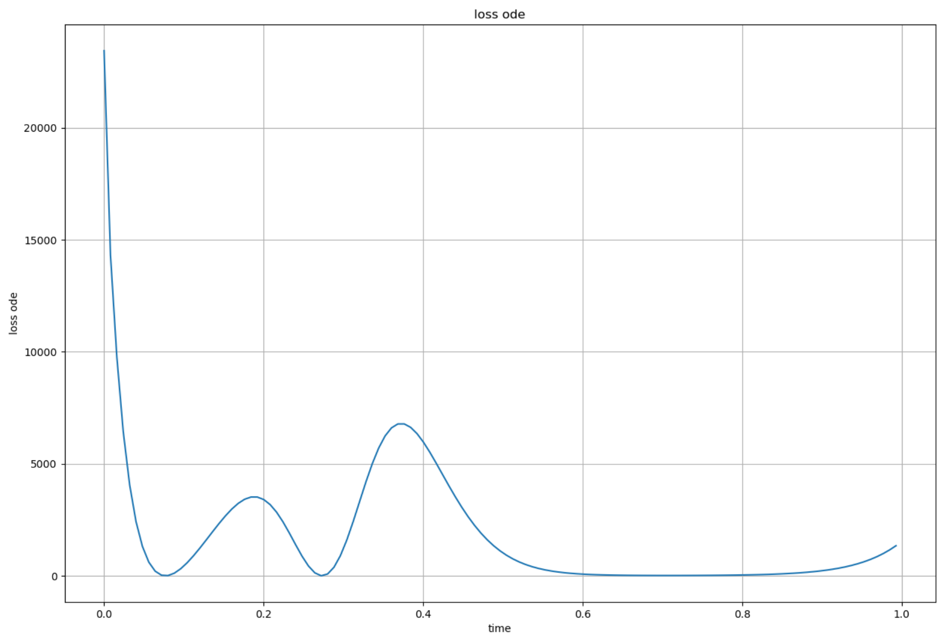
```
long model mse: 21.551982879638672
```

We can see that short model have a mse 3 order of magnitute lower than long model even if the training was performed for 3x epochs. This indicate, togheter with the plateu of the loss history, that is not because the long model did not converged.

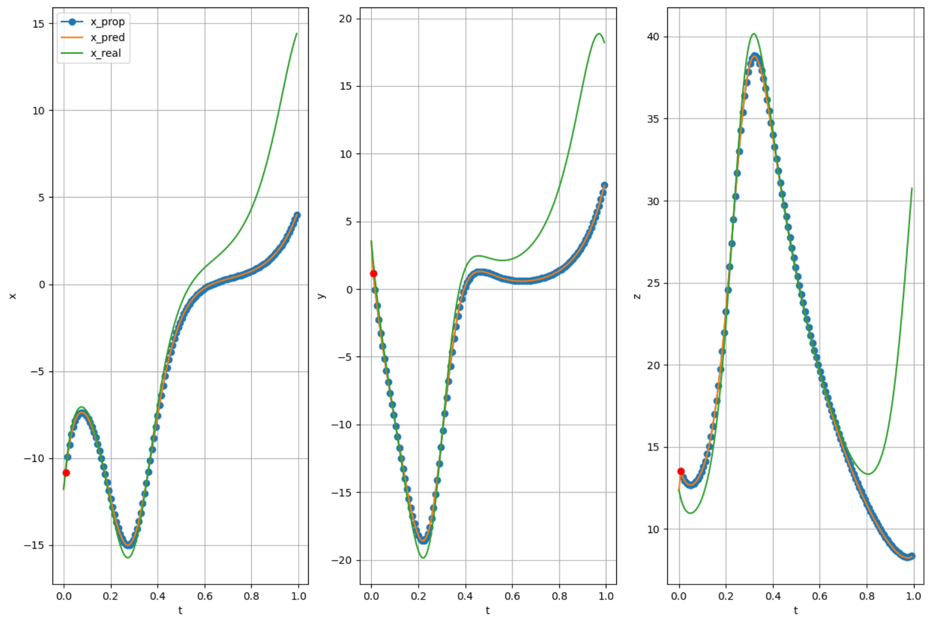
We try to convice ourself looking at the loss_ode in time.

```
[31]: # get loss ode
      loss_ode = lorenz_loss_ode(long_model, X_sub_t)
      # plot loss ode
      plt.figure(figsize=(15, 10))
      plt.plot(X_sub, loss_ode.detach().numpy())
      plt.ylabel('loss ode')
      plt.xlabel('time')
      plt.grid()
      plt.title('loss ode')
```

```
[31]: Text(0.5, 1.0, 'loss ode')
```



```
[32]: # propagate
      plot_propagation(X_sub, Y_sub, Y_pred, 1)
```



In the last plot we show that propagating with RK the state predicted in the points where the error is higer, the numerical solution and the predicted one is very similar. This is a good indication that the model actually converged but on another solution, as the IC are different. This ispired us to produce the Learning Schedule algorithm for training the PINN on chaotic system.

1.5 PINN with learning schedule

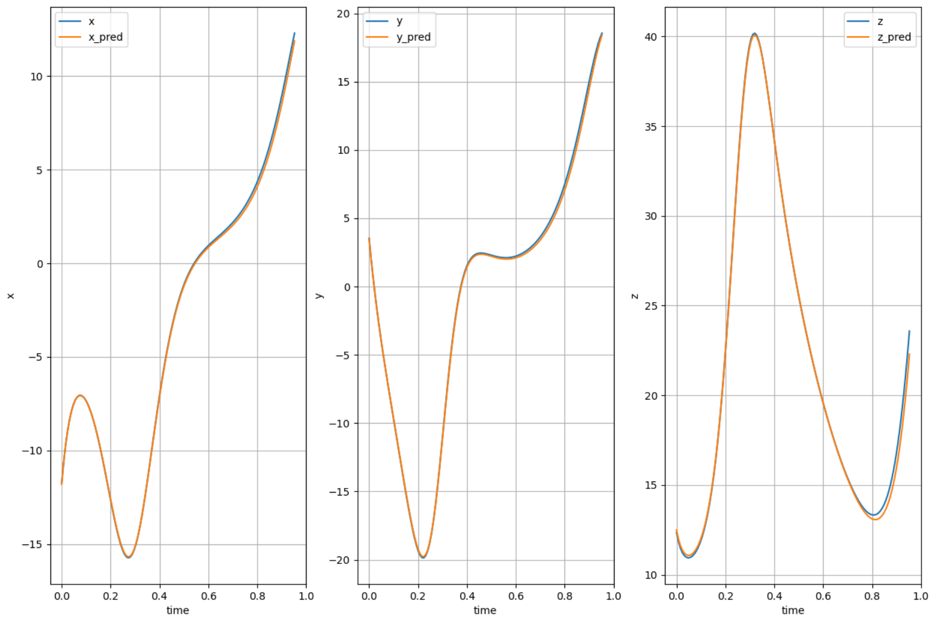
[44]:
```
# load experiment, folder name ../lorenz_PINN_shortbow

experiment_folder = '../models/lorenz_PINN_shortbow'
experiment_name = 'lorenz'

shortbow_PINN_LS = PINN_LearningSchedule(experiment_folder, experiment_name)
```

```
Load existing experiment:  ../models/lorenz_PINN_shortbow  with  120  points
and  26094000.0  epochs
```

[45]:
```
# get number of points
n_points = shortbow_PINN_LS.n_points
```

[46]:
```
# select only time < 0.25
X_sub = X[:n_points]
Y_sub = Y[:n_points]

# to torch
X_sub_t = torch.tensor(X_sub).float().view(-1, 1)
Y_sub_t = torch.tensor(Y_sub).float()
```

[47]:
```
# show compare
Y_pred = shortbow_PINN_LS.model(X_sub_t)
Y_pred = Y_pred.detach().numpy()
plot_compare_components(X_sub, Y_sub, Y_pred)
```



[48]:
```python
# get mse and print
loss = mse(shortbow_PINN_LS.model(X_sub_t), Y_sub_t)
print('shortbow PINN mse:', loss.item())
```

```
shortbow PINN mse: 0.06707832217216492
```

So the LS_model mse is nearly the same of the short one. With this example we demostrate that is possible to train a PINN on chaotic system with a learning schedule for longer time.