Advanced Deep Learning - Lab 5

Simon Brandeis, Emmanuel Jehanno
Chloé Portier

12th March 2020

1 Introduction

In this lab, the task is to imitate a given dynamical system, namely, Rössler attractor. The Rössler attractor is actually chaotic, which means that any small change in the initial point $w + dw$ will lead to a completely different trajectory after some time.

Rössler attractor is also ergodic, which means that any sufficiently long trajectory will go through all the possible states w of the system. Thus, learning from any single very long trajectory is theoretically similar to learning from many shorter trajectories taken at random locations ("random" not as in "uniform over R^3" but according to the probability of the dynamical system to get there, which is not given [but might be estimated]).

2 Training of our model

2.1 Approach chosen

We decided to choose the third approach : state fully known, continuous system. Our training set has multiple inputs : four successive positions (w_t,w_{t+1},w_{t+2} and w_{t+3}) and three successive velocities ($\dot{w}_t$,$\dot{w_{t+1}}$ and $\dot{w_{t+2}}$).

```
### Build the data and the target trajectory
Niter = int(1e5)
delta_t = 1e-3 # or 1e-2
ROSSLER_MAP = RosslerMap(delta_t=delta_t)
INIT = np.array([-5.75, -1.6,  0.02])
traj, t = ROSSLER_MAP.full_traj(Niter, INIT)

traj_t = torch.from_numpy(traj[:-3])
traj_tp1 = torch.from_numpy(traj[1:-2])
traj_tp2 = torch.from_numpy(traj[2:-1])
traj_tp3 = torch.from_numpy(traj[3:])
dot_traj_t = temporal_derivative(traj_t, traj_tp1, delta_t=delta_t)
dot_traj_tp1 = temporal_derivative(traj_tp1, traj_tp2, delta_t=delta_t)
dot_traj_tp2 = temporal_derivative(traj_tp2, traj_tp3, delta_t=delta_t)

training_set = TensorDataset(traj_t, traj_tp1, traj_tp2, traj_tp3,
                             dot_traj_t, dot_traj_tp1, dot_traj_tp2)
```

Thanks to this information, our model will be able to predict a velocity $\dot{w}_t$ for a given state w_t.

$$\dot{w}_t = NN(w_t) \tag{1}$$

2.2 Choice of the loss

Our goal here is to be as close as possible to the trajectory of the attractor. Because we want the model to *behave* like the attractor, we chose a loss that evaluates how the predicted trajectory differs from the real one on several timesteps.

$$L = \frac{1}{|batch|} \sum_{k=0}^{|batch|} (\sum_{i=1}^{3} \frac{1}{4-i} ||w_{t_k+i} - \hat{w}_{t_k+i}||_2 + ||\dot{w}_{t_k} - \hat{\dot{w}}_{t_k}||_1) \tag{2}$$

We chose the L_1 loss to penalize error on the predicted velocity. This loss penalizes equally errors on the predictions. We chose this loss to be more robust to regions of the trajectory that have very peculiar velocities.

We chose the L_2 loss to penalize errors on the position $w_t + i$. This loss penalizes quadratically large errors on the positions, which is useful in our case: errors on position grow exponentially along the trajectory. We compute the L_2 loss on several points of the predicted trajectory $\hat{w}_t$ and the real trajectory w_t. Reasons for this choice are the following:

- **Numerical reasons:** Because the timestep used to generate the trajectories is very small, errors on the position keep small between instants t and $t+1$. Comparing positions on further timesteps makes the error bigger and thus it is easier to optimize numerically.
- **Intuitive reasons:** If the model does not behave like the attractor, it will diverge from its trajectory relatively quickly. Looking at the predicted positions on a larger span of time injects information while training so the model can learn how it should behave in time, not only on a given point in time.

We also chose to weight the error on predictions: predictions further in time have a larger weight. The choice of the weights is arbitrary. Below is the code we use to implement the loss.

```
from torch import nn

# Predicted trajectory, computed using the model's predicted velocity
x_pred=[xtp1_pred, xtp2_pred, xtp3_pred]
# True trajectory of the R ssler attractor
x_true=[xtp1, xtp2, xtp3]
# Real velocity of the R ssler attractor at position xt
dot_xt_true=dot_xt
# Model's prediction for the velocity at position xt
dot_xt_pred=dot_xt_pred

def my_custom_loss(x_pred, x_true, dot_xt_pred, dot_xt_true, delta_t=delta_t):
    loss = nn.functional.l1_loss(dot_xt_true, dot_xt_pred)
    for i in range(len(x_true)):
        loss += 1/(len(x_true) - i)*nn.functional.mse_loss(x_true[i], x_pred[i])

    return loss
```

2.3 Network architecture

For our model, we chose a simple feed forward network:

```
Feedforward(
  (layer_1): Linear(in_features=3, out_features=16, bias=True)
  (layer_2): Linear(in_features=16, out_features=16, bias=True)
  (layer_3): Linear(in_features=16, out_features=16, bias=True)
  (out): Linear(in_features=16, out_features=3, bias=True)
)
```

We define the following training function (which is completely **data-driven**):

```
def train(num_epochs, batch_size, criterion, optimizer, model, dataset, delta_t,
    display=True):
    train_error = []
    train_loader = DataLoader(dataset, batch_size, shuffle=True)
    model.train()
    for epoch in range(num_epochs):
        epoch_average_loss = 0.0
        for (xt, xtp1, xtp2, xtp3, dot_xt, dot_xtp1, dot_xtp2) in train_loader:
            # Predict velocity at position xt
            dot_xt_pred = model(xt.float())

            # Predict the next steps of the trajectory
            xtp1_pred = xt + dot_xt_pred * delta_t
            dot_xtp1_pred = model(xtp1_pred.float())
```

```
            xtp2_pred = xtp1 + dot_xtp1_pred * delta_t
            dot_xtp2_pred = model(xtp2_pred.float())
            xtp3_pred = xtp2 + dot_xtp2_pred * delta_t

            # Compute the loss
            loss = criterion(x_pred=[xtp1_pred, xtp2_pred, xtp3_pred],
                             x_true=[xtp1, xtp2, xtp3],
                             dot_xt_true=dot_xt,
                             dot_xt_pred=dot_xt_pred)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            epoch_average_loss += loss.item() * batch_size / len(dataset)
        train_error.append(epoch_average_loss)
        if display:
            print('Epoch [{}/{}], Loss: {:.4f}'
                  .format(epoch + 1, num_epochs, epoch_average_loss))
    return train_error
```

3 Evaluation of our model

To assess the quality of our model, we generated from a random starting point a trajectory and compared our estimated trajectory to the real trajectory. Figure 1 represents the predicted and real trajectories in the 3d space, and shows that our model can actually replicate the overall behaviour of the attractor.

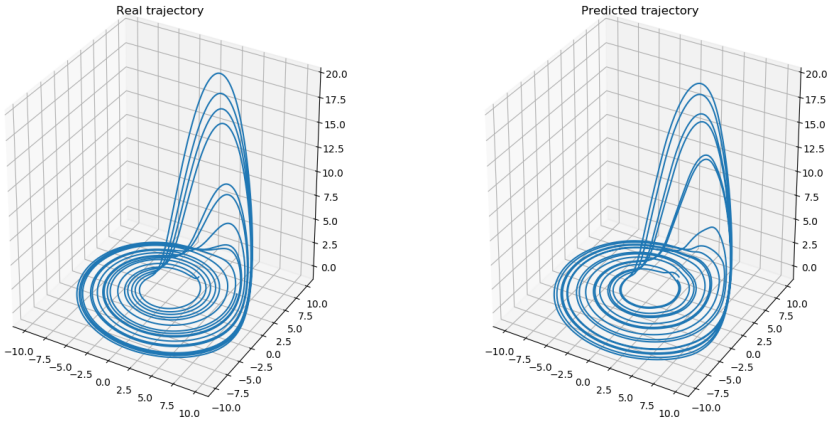
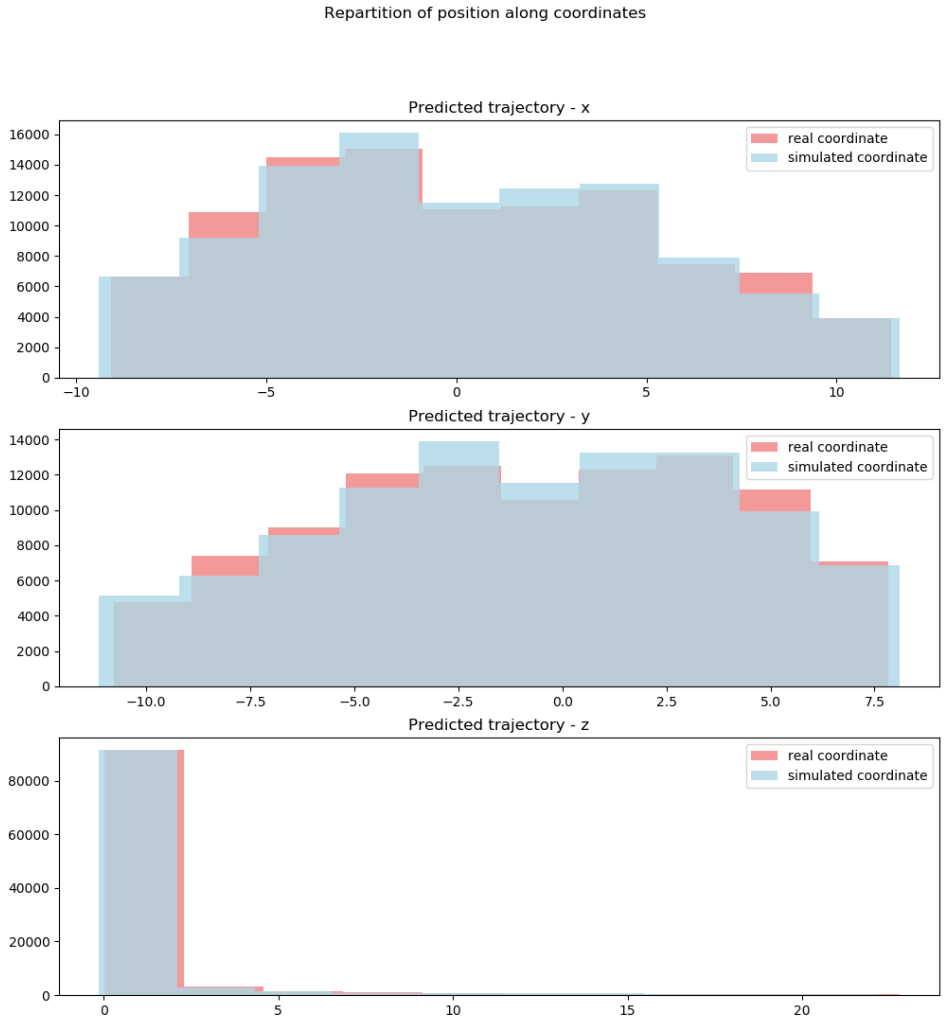


Figure 1: Trajectories in the 3D space

3.1 Histograms and trajectories

For each coordinate, we plotted in Figure **??** histograms that show the distribution of the particle in the space for the real trajectory and the simulated trajectory. We observe that the probabilities to reach any location are well predicted by our model since both histograms are very similar. The prediction performances along every axis appear to be equivalent : we predict as well the position in the x axis as we do in the y axis. Moreover, it seems that the z axis isn't very interesting here since the trajectory is very coplanar, but we think that this visualization with histograms is in fact not a good one.

Moreover, we also plotted in Figure 2 a visualisation of each trajectory on each axis. We see more differences in the movement amplitude but trends are the same. On this visualization, we observe some peaks in the z axis corresponding to the non-coplanar part of the trajectory which is the specificity of the Rössler attractor. Interestingly, the observed peaks happen sometimes with a delay (as well as the predictions along x and y). Since the system is chaotic, very small differences infer in huge variations of



the trajectory later. The predictions being not perfectly precise, this can infer in great exponential shifts in the later trajectory. Indeed the first peak is very precise but the other ones present a delay.

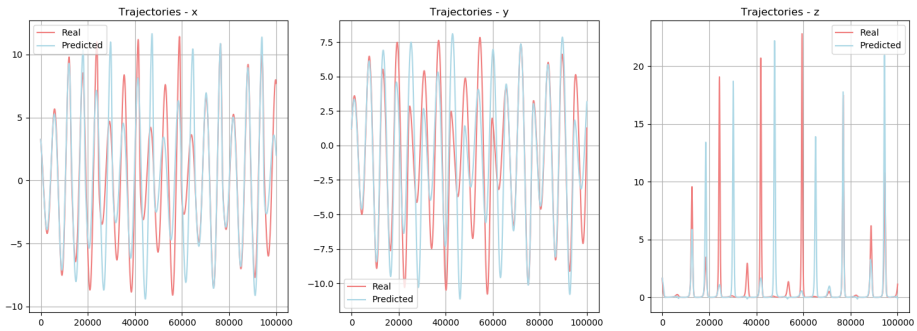


Figure 2: Trajectories along coordinates

3.2 Auto-correlations

To confirm what we started to see on trajectories visualisations, we decided to show auto-correlations on each axis. The graphs are the auto-correlation between the first half of the trajectory with the second half of the same trajectory. We can highlight that on each axis (mostly x and y) trajectories have the same auto-correlation. This shows a consistent way of behave in a time lapse (and also consistent with the previous observations of the delay).

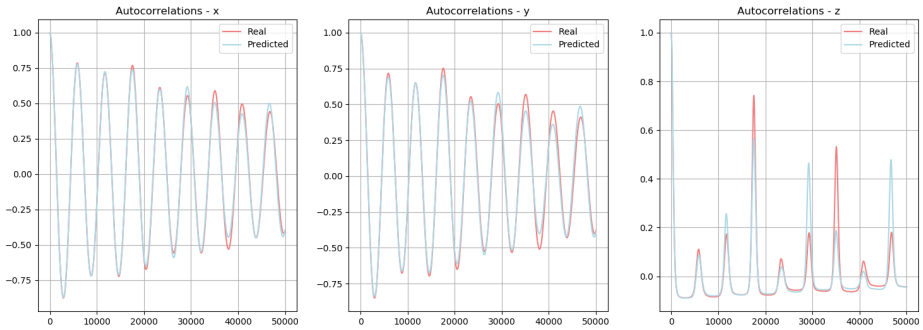


Figure 3: Auto-correlations of trajectories

3.3 Fourier frequencies

The trajectory of the Rössler attractor follows globally a loop, though not exactly periodic. We decided to analyse frequencies moving to Fourier domain. This analysis aims at exploring the similarities in terms of the trajectory itself considering it as a signal. The Fourier analysis should yield frequencies corresponding to the trajectory itself as well as very high frequencies corresponding to the distance between successive locations.

Results are satisfying : amplitudes are roughly the same, and frequencies stay in the same range (between 0 Hz and 1 Hz).

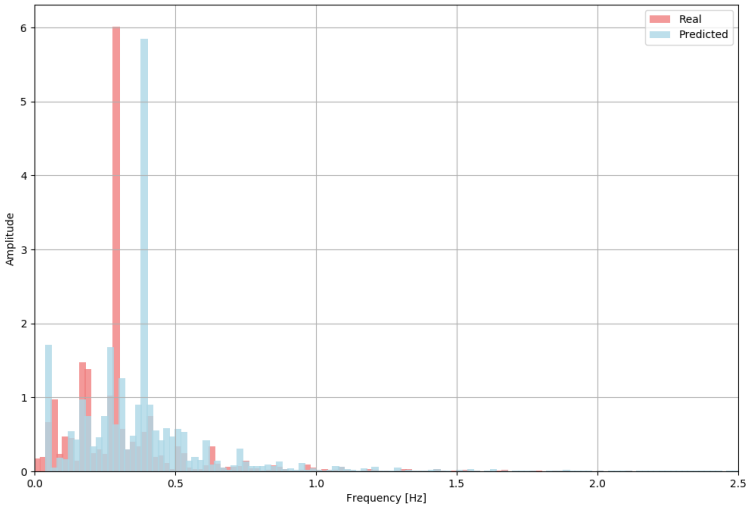


Figure 4: Fourier decomposition

To conclude, it seems our model mimics pretty well the behaviour of the Rössler attractor. Plus, it seems to be robust to the choice of the initial point.