LMU: Fakultät für Physik

Lecture:
When Machine Learning meets Complex Systems

PD Dr. Christoph Räth, Sebastian Baur, Daniel Köglmayr, Joel Steinegger

Exercise Sheet 3

(Please prepare your answers until Wednesday, December 3rd)

Exercise 3.1: Networks Create the following networks:

- A regular ring lattice with $N = 250$ nodes, where each node is connected to its 4 nearest neighbors.
- An Erdös-Rényi random network with $N = 250$ nodes and an edge probability of $p = 0.01$.
- A small-world network with $N = 250$ nodes, 4 edges per node and a rewiring probability of $p = 0.1$.
- A scale-free network with $N = 250$ nodes where each new node is connected to 2 nodes of the existing network (with preferential attachement).

Visualize each network in a sensible, preferably interactable, way. Calculate and plot the distribution and the mean value for the node degree, clustering coefficient and path length.

Exercise 3.2: SINDy The SINDy algorithm (sparse identification of nonlinear dynamics) is a simple but powerful method for reconstructing the governing equations of a dynamical system. It was orginally introduced by Brunton et al. Here we will implement the most basic version of SINDy and apply it to reconstruct and predict the Lorenz System.

(a) Load the time series $\mathbf{X}$ of the Lorenz system from moodle and quickly plot it to convince yourself that you have loaded the data correctly. It has shape $t \times d = 10000 \times 3$, with t the number of time steps and d the system dimension. Alternatively you can also create your own dataset using the Runge Kutta implementations from previous sheets. Time step size is $\Delta t = 0.002$ and system parameters are $\sigma = 10, \rho = 28, \beta = 8/3$.

(b) For the purpose of this exercise we'll now pretend to not know what system was used to generate this time series but we'll assume that it was generated by some dynamical system following

a (stationary and deterministic) governing equations of the form

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t)).$$

The vector $\mathbf{x}(t)$ denotes the state of a system at time t, and the function $\mathbf{f}(\mathbf{x}(t))$ represents the dynamic constraints that define the equations of motion. Further, we'll guess that these equations can be approximated by combining a set of polynomials of up to second order. Doing so turns every data point $\mathbf{x}_i = (x_i, y_i, z_i)$ into a so called feature vector $\mathbf{r}_i$, here given ass:

$$\mathbf{r}_i = (x_i, y_i, z_i, x_i^2, z_i^2, y_i^2, x_i y_i, x_i z_i, y_i z_i),$$

which incorporates the original data point and all possible combinations of second order terms. Write function that turns any three dimensional data point into such a feature vector. By applying this function to every data point, turn the whole data set $\mathbf{X}$ of shape 10000×3 into our feature matrix $\mathbf{R}$ of shape 9999×9

(c) As a next step we need to define our target data. As we want to reconstruct an ODE, we want to have $\frac{d}{dt}\mathbf{x}(t) \approx \frac{\Delta \mathbf{x}}{\Delta t}$ as our target data. Produce for every $\mathbf{x}_i$ in $\mathbf{X}$ its $\Delta \mathbf{x}_{i+1} = \mathbf{x}_{i+1} - \mathbf{x}_i$ and store it in a target matrix $\mathbf{Y}$ (shape 10000×3). (Δt is not of interest for now.)

(d) We now have our input data $\mathbf{X}$, our best guess for what equations could be part of the system's governing equations $\mathbf{R}$ and our target data $\mathbf{Y}$, the thing that these equations in $\mathbf{R}$ are supposed to produce when given an input from $\mathbf{X}$. Now we just have to figure out the connections between them:

$$\mathbf{X} \xrightarrow[\text{function set}]{\text{polynomial}} \mathbf{R} \xrightarrow{\quad ? \quad} \mathbf{Y}$$

The basic idea for how to do this is to find to find a matrix $\mathbf{W}_{out}$ that is optimized to 'predict' the next value of the time series by multiplying the the feature vector of time i $\mathbf{r}_i$ and $\mathbf{W}_{out}$, such that

$$\mathbf{W}_{out}\mathbf{r}_i \simeq \Delta \mathbf{x}_{i+1}. \tag{1}$$

I.e. we are trying to solve

$$\mathbf{W}_{out} = \arg\min_{\mathbf{W}} \sum_i \|\mathbf{W}\mathbf{r}_i - \Delta \mathbf{x}_{i+1}\|. \tag{2}$$

This is solvable using linear algebra, namely by using least squares regression. In practice though it turns out that adding a penalty term to results of regression decreases overfitting substantially. Hence we'll actually solve

$$\mathbf{W}_{out} = \arg\min_{\mathbf{W}} (\sum_i \|\mathbf{W}\mathbf{r}_i - \Delta \mathbf{x}_{i+1}\| + \alpha \|\mathbf{W}\|. \tag{3}$$

where α is the coefficient for the L2 penalty. This type of regression is called ridge regression. $\alpha = 1e{-}8$ happens to work well here. Luckily for us, ridge regression has already been implemented in many different python packages. You can for example use the sklearn library to find $\mathbf{W}_{out}$ with

```
from sklearn.linear_model import Ridge
regressor = Ridge(alpha=1e-8)
regressor.fit(R,Y)
W_out = regressor.coef_
```

Note that the last entry of $\mathbf{R}$ has no target data, hence you'll want to shorten $\mathbf{Y}$ correspondingly.

(e) To figure out what equations SINDy has actually reconstructed, divide $\mathbf{W}_{out}$ by Δt (in our case Δt=0.002) and have a look at it. Can you identify matrix elements which look similar to the σ, ρ and β parameters of the initial Lorenz equations? Using this, write down your best guess of the Lorenz equation parameters. Congrats you just found a system's generating equations solely by looking at the data!

(f) As this is just an exercise sheet we already know the true governing equation, hence it's easy to see if the parameters we found actually agree with the real ones. In practice, you typically don't know the correct equations, you'd need to use different methods to see if the equations you have found are actually correct. One of the simplest ways to do this for SINDy is to use our results to directly predict the system.
You can do so easily by taking a data point $\mathbf{x}_i$, turning it into a feature vector $\mathbf{r}_i$ and then multiplying it with the optimized $\mathbf{W}_{out}$ to get $\Delta\mathbf{x}_{i+1}$ and from that $\mathbf{x}_{i+1}$

$$\begin{aligned}
\mathbf{x}_i &= (x_i, y_i, z_i), \\
\mathbf{r}_i &= (x_i, y_i, z_i, x_i^2, z_i^2, y_i^2, x_i y_i, x_i z_i, y_i z_i), \\
\Delta\mathbf{x}_{i+1} &= \mathbf{W}_{out}\mathbf{r}_i \\
\mathbf{x}_{i+1} &= \Delta\mathbf{x}_{i+1} + \mathbf{x}_i.
\end{aligned}$$

Implement the above in a for loop for 10000 predictions steps and visually compare it with the real Lorenz system.

Exercise 3.3 Reservoir Computing: reservoir computing, at least in its traditional form, is conceptually quite similar to the SINDY algorithm. In both cases one uses nonlinear functions to create a higher dimensional space from the input data, which is then fitted to the target data using linear regression. Atypical for modern machine learning, gradient descent is not used in either method.

The goal of this exercise is predict the Lorenz system from data using your own reservoir computing implementation. Should you get stuck though feel free to get inspired by the open source reservoir computing libraries out there.

(a) Here too we'll use the given Lorenz system time series as input $\mathbf{x}$. Plot it. You can also simulate our own using a time step of $\Delta t = 0.02$ and system parameters are $\sigma = 10, \rho = 28, \beta = 8/3$. (Note that our time step here is an order of magnitude larger as in exercise 3.2.)

(b) As a first step, we'll create the archetypal reservoir for our reservoir computer. In principle a reservoir can be any sufficiently high-dimensional, non-linear transformation of the input. Even real world physical systems, such as coupled non-linear springs or LASERS can be used.

Here we'll keep it maximally simple though. Our reservoir will just be a Recurrent Neural Network and a nonlinear activation function.

As basis for our network we'll construct the adjacency matrix $\mathbf{A}$ as a random Erdos Rényi Network with a network size $n = 500$ and an edge probability of 1%. We can use python's networkx package to do this for us, making sure to set a fixed random seed so that we get the same network every time we run the code:

```
import networkx as nx
import numpy as np
n = 500  # network dimension
p = 0.01  # edge connection probability
seed = 0  # seed for the random generation
network = nx.fast_gnp_random_graph(n=n, p=p, seed=seed)
network = nx.to_numpy_array(network)
```

This network is now a 500×500 binary numpy array. For reservoir computing to work you'll want to vary the strength of each connection randomly between [-1, 1]:

```
random_generator = np.random.default_rng(seed=0)
network[network != 0.0] = 2 * (
   random_generator.random(size=network[network != 0.0].shape) - 0.5
)
```

Then we want to scale the entire network's spectral radius ρ to be 0.4, a value that empirically works well for the Lorenz system. As our network is a real valued square matrix, the spectral radius is just the largest absolute value of its eigenvalues, which you can calculate using any of the various math libraries out there. Here is one way to do it, taking advantage of the sparsity of our network:

```
from scipy.sparse import csr_matrix
from scipy.sparse.linalg import eigs
s_rad = 0.4  # spectral radius we want
network = csr_matrix(network)
eigenvals = eigs(network, k=1, v0=np.ones(n), maxiter=int(1e3 * n))[0]
maximum = np.absolute(eigenvals).max()
network = (s_rad / maximum) * network
network = np.array(network.todense())
```

(c) As a next step we need to define an input matrix $\mathbf{W}_{in}$ to couple the input to our reservoir. Here, a purely random matrix with values between $[-0.1, 0.1]$ of shape 500×3 works well enough:

```
w_in = np.random.default_rng(seed)
w_in = w_in.uniform(low=-0.05, high=0.05, size=(n, 3))
```

(d) We can now implement the reservoir dynamics, for which we use the activation function tanh

$$\mathbf{r_{i+1}} = \tanh(\mathbf{A r_i} + \mathbf{W_{in} x_i}),$$

where $\mathbf{r}_i$ is the 500 dimensional reservoir state and $\mathbf{x}_i$ the 3 dimensional input data from the Lorenz System at time step i.

Create the 500 dimensional reservoir state as a numpy array and initialize it to zero, $\mathbf{r}_0 = \mathbf{0}$. Then iterate over 6000 steps, saving the reservoir state of each step $\mathbf{r}_i$ into a 500×6000 dimensional array $\mathbf{R}$. Throw the first 1000 steps of this array away, as these are transitory steps during which the reservoir is not yet synchronized with the input data. $\mathbf{R}$ should now be of shape 500×5000.

```
"""This part is for you to figure out. Probably something with a for loop though."""
```

(e) To use the reservoir for predictions, we need to fit an output matrix $\mathbf{W}_{out}$, which we do via a simple ridge regression. For this we define a target matrix $\mathbf{Y}$, where the target of each input $\mathbf{x_i}$ is defined as $\mathbf{y_i} = \mathbf{x_{i+1}}$. You can then run the ridge regression to create $\mathbf{W}_{out}$ via:

```
from sklearn.linear_model import Ridge
regressor = Ridge(alpha=10**-8)
regressor.fit(R,Y)
W_out = regressor.coef_
```

(f) To check if your regression worked, we can compare the target matrix $\mathbf{Y}$ with the regression output $\mathbf{W}_{out}\mathbf{R}$. They should be very close, though not quite equal.

$$\mathbf{Y} \approx \mathbf{W}_{out}\mathbf{R}.$$

Plot them on top of each other for e.g. 500 time steps and convince yourself that this is true.

(g) You can now use the above to generate successive internal reservoir states $\mathbf{r_{i+1}}$ and can use your trained $\mathbf{W}_{out}$ to transform these into the output time series via simple matrix multiplication

$$\begin{aligned}\mathbf{r_{i+1}} &= \tanh(\mathbf{A r_i} + \mathbf{W_{in} x_i}),\\ \mathbf{x}_{i+1} &= \mathbf{W}_{out}\mathbf{r_{i+1}}.\end{aligned}$$

Now use this this setup to predict the next 1000 time steps of your time series, beginning at time step 6000. Notably you can't trivially start the reservoir computing prediction at any arbitrary input x due to the need to synchronize the internal reservoir state r to this input. As such its easiest to start the prediction at the exact time step your training ended, i.e. at $\mathbf{x}_{6000}$. Plot your results.

```
"""This part is for you to figure out."""
```

Congratulations, you have implemented the basic form of reservoir computing! If your predictions don't work well, play around a bit with the rng seed of your setup and the hyperparameters. In any case, list all the parameters of your setup and we can discuss the results during the tutorial.