Exercise 6

Deadline: 24.01.2024 16:00.
Ask questions in discord to #ask-your-tutor

In this exercise, we will explore symbolic regression with the SINDy algorithm.

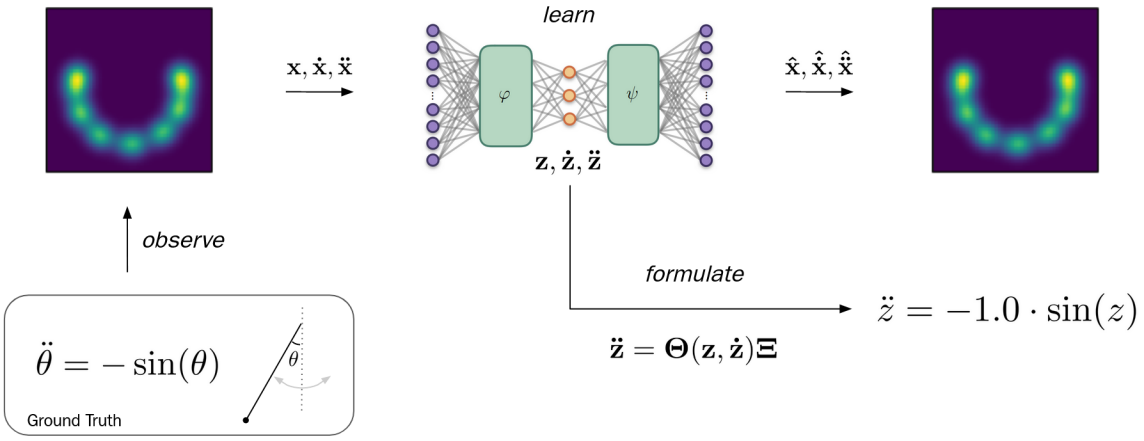

Abbildung 1: Overview of the SINDy-Autoencoder by Champion et al.

Regulations

Please implement your solutions in form of *Jupyter notebooks* (`*.ipynb` files), which can mix executable code, figures and text in a single file. They can be created, edited, and executed in the web browser, in the stand-alone app JupyterLab, or in the cloud via Google Colab and similar services.

Create a Jupyter notebook `sindy.ipynb` for your solution and export the notebook to HTML as `sindy.html`. Zip all files into a single archive `ex06.zip` and upload this file to MaMPF before the given deadline.
Moreover, please set your **Anzeigename/display name** and **Name in Uebungsgruppen/name in tutorials** in MaMPF to your real name, which should be identical to your name in `muesli` and make sure you **join the submission** of your team via the invitation code before the submission deadline. Check out https://mampf.blog/handing-in-homework-assignments for instructions.

1 SINDy in Ground Truth Coordinates z

1.1 Simulation

Let z_t the pendulum's deflection angle and $\dot{z}_t = \partial z_t / \partial t, \ddot{z}_t = \partial^2 z_t / \partial t^2$ its first and second derivatives with respect to time t. Physics tells us that the second derivative is inversely proportional to the sine of the angle:

$$\ddot{z}_t = -\sin(z_t) \tag{1}$$

Thus, larger deflections lead to larger forces pulling the pendulum back. In preparation for the SINDy algorithm, we express this differential equation as a scalar product between a library of functions terms $\Theta(z, \dot{z})$ and the vector of their coefficients Ξ, where Ξ^* denotes the ground-truth coefficients (a vector of all zeros except for a -1 for the sine term):

$$\ddot{z}_t = \Theta(z_t, \dot{z}_t) \cdot \Xi^* = \sin(z_t) \cdot (-1.0) \tag{2}$$

We start by simulating the pendulum to create training data with the ground truth ODE. Implement the following functions (they are a bit more general than needed here in order to be reusable for the subsequent tasks):

1. `pendulum_rhs(zt, dzt, coefficients, terms)` to compute the scalar product $\Theta(z, \dot{z}) \cdot \Xi$ between the function terms and the coefficients at the given time points `(zt, dzt)`. Note that the function should be vectorized, so that `(zt, dzt)` can be vectors of time points.
2. `pendulum_ode_step(y, t, coefficients, terms)` to be used as the *func* parameter to scipy.integrate.odeint.
3. `simulate_pendulum(z0, dz0, coefficients, terms, T, dt)` that uses `odeint` to simulate the pendulum with the initial conditions $z_0, \dot{z}_0$ for T steps with step size Δt.
4. `create_pendulum_data(z0_min, z0_max, dz0_min, dz0_max, coefficients, terms, T, dt, N, embedding = None, rejection = True)` that creates a training set of N simulations from uniform random initial conditions within the range $z_{0,\min} \le z_0 \le z_{0,\max}$ and $\dot{z}_{0,\min} \le \dot{z}_0 \le \dot{z}_{0,\max}$. If the pendulum has too much angular momentum, i.e. if $\left|\frac{1}{2}\dot{z}_0^2 - \cos(z_0)\right| > 0.99$, reject the simulation and sample a new one. The *embedding* parameter will later be used to specify the type of artificial embedding into cartesian coordinates or a video.

Executing `create_pendulum_data()` will create N runs with T timesteps in each, so our training set is $\text{TS} = \{z_i, \dot{z}_i, \ddot{z}_i\}_{i=1}^{T \cdot N}$. Recommended parameters are $N = 100$ simulations with $-z_{0,\min} = z_{0,\max} = \pi$ and $-\dot{z}_{0,\min} = \dot{z}_{0,\max} = 2.1$ over $T = 50$ with $\Delta t = 0.02$.

Verify your implementation by picking 5 simulations and visualize $z_t, \dot{z}_t, \ddot{z}_t$ as a function of time. You should see oscillating curves corresponding to the swinging motion of the pendulum, and the curve of the three quantities should be shifted with respect to each other. Optionally, use `matplotlib.animation.FuncAnimation` to create an animation of the pendulum by calculating the position of the point mass at the tip via $x_1(t) = \sin(z_t)$ and $x_2(t) = -\cos(z_t)$.

1.2 Implementation & Training

Implement the basic SINDy algorithm, which is given the canonical coordinates $z_t, \dot{z}_t, \ddot{z}_t$ where the ODE works (i.e., no feature discovery is needed here):

$$\widehat{\Xi} = \text{argmin}_\Xi \left[\frac{1}{T \cdot N} \sum_{i=1}^{T \cdot N} \left\| \ddot{z}_i - \Theta(z_i, \dot{z}_i) \cdot \Xi \right\|_2^2 + \lambda \left\| \Xi \right\|_1 \right] \tag{3}$$

It is just an instance of LASSO regression with regularization parameter λ that encourages Ξ to be sparse, i.e. only a few function terms should be active in the solution. The function term library is built on the basis of prior knowledge and should consist of the following functions

$$\Theta(z, \dot{z}) = \left[1, z, \dot{z}, \sin(z), z^2, z \cdot \dot{z}, z \cdot \sin(z), \dot{z}^2, \dot{z} \cdot \sin(z), \sin(z)^2\right] \tag{4}$$

Implement two versions of the solver: One using the sklearn.linear_model.Lasso class and a pytorch version `SINDy(nn.Module)` which uses the ADAM optimizer. We will need the pytorch variant later in combination with the Autoencoder. It ensures sparsity by learning a boolean coefficient mask Υ, which is True for active function terms and False otherwise. Upon initialization, set $\Upsilon = \text{True}$ and $\Xi^{(0)} = 1$.

The function `SINDy.forward(self, z, dz)` returns the RHS of the ODE, i.e. $\Theta(z, \dot{z}) \cdot (\Xi \odot \Upsilon)$, where Ξ and Υ are the current guesses at the solution (stored as members of the `SINDy` class). To train the pytorch version, implement a `train_sindy()` function, and log the training and validation loss and the coefficient values over time. For now, the mask Υ remains unchanged all elements are True) throughout.

Train both versions on your training set and check that the solutions are equal within numerical accuracy.

1.3 Thresholding

Add an option `thresholding` to the `train_sindy()` function, which can take the values `None` (default), `"sequential"` or `"patient"` to apply the Sequential Thresholding (ST) or Patient Trend-Aware Thresholding (PTAT) methods explained in the lecture. Thresholding updates the coefficient mask Υ based on the coefficient history to determine which coefficients are active or inactive. As a reminder, the thresholding algorithms are:

Algorithm 1 Sequential Thresholding (ST)

Inputs: $\Xi \in \mathbb{R}^{L \times D}$, threshold a, interval S
Initialize: $\Upsilon \in \mathbb{R}^{L \times D}$ to True # Coefficient Mask
for $e = 1$ **to** E **do** # In each epoch of the training loop
  if $e \mod S = 0$ **then** # Check if thresholding should be applied
    $\Xi\big[|\Xi| < a\big] \leftarrow 0$ # Set small coefficients to 0 using boolean indexing
    $\Upsilon\big[|\Xi| < a\big] \leftarrow$ False # Update mask for coefficients smaller than a
  end if
end for

Algorithm 2 Patient Trend-Aware Thresholding (PTAT)

Inputs: Coefficients $\Xi \in \mathbb{R}^{L \times D}$, thresholds a, b, patience P
Initialize: Previous coefficients $\Xi^{prev} \in \mathbb{R}^{L \times D}$ to zero
Initialize: Last overshoot epochs $E_a \in \mathbb{N}^{L \times D}$, and $E_b \in \mathbb{N}^{L \times D}$
Initialize: $\Upsilon \in \mathbb{R}^{L \times D}$ to True # Coefficient Mask
for $e = 1$ **to** E **do** # In each epoch of the training loop
  $E_a\big[|\Xi| > a\big] \leftarrow e$ # Update exceeded threshold epoch
  $E_b\big[|\Xi - \Xi^{prev}| > b\big] \leftarrow e$ # Update exceeded trend threshold epoch
  $\tilde{\Upsilon} \leftarrow ((e - E_a) < P)$ **element-wise OR** $((e - E_b) < P)$ # Keep coefficients that exceeded either threshold in the last P epochs
  $\Upsilon \leftarrow \Upsilon$ **element-wise AND** $\tilde{\Upsilon}$ # Retain previously disabled coefficients
  $\Xi^{prev} \leftarrow \Xi$ # Update previous coefficients for next epoch
end for

1.4 Evaluation & Visualization

Compare the resulting coefficients and visualize the loss and coefficient history. Check that SINDy identifies the $-\sin(z)$ term. Verify that the coefficients of the unused terms are close to zero, and that the corresponding entries of the mask Υ are False when thresholding was active. Resimulate the system with initial conditions from a test set and the learned equation. Visualize the resimulated $\hat{z}(t)$ and the average error to the ground truth simulations over time.

1.5 Small Angle Approximation

Train SINDy again using initial conditions z_0 and $\dot{z}_0$ with much smaller magnitude than in the original training set and determine at which point $-\sin(z)$ is not the only non-zero coefficient anymore. Explain why. Since the rejection of extreme scenarios is not needed for small initial conditions, your may simulate the data with `rejection = False` and accept all sampled initial conditions.

2 SINDy-Autoencoder

2.1 Cartesian Embedding

Suppose that in real life we didn't know that the deflection angle z_t is the most appropriate variable to express the pendulum behavior. Instead, we measure the location of the pendulum's tip in 2-

dimensional cartesian coordinates x_t with derivatives $\dot{x}, \ddot{x}$. To simulate this situation, implement a `embed_cartesian(z, dz, ddz)` function that generates a new training set $x_i, \dot{x}_i, \ddot{x}_i \in \mathbb{R}^2$ from the canonical representation $z_i, \dot{z}_i, \ddot{z}_i$ via the chain rule:

$$x = [\sin(z), -\cos(z)] \tag{5}$$
$$\dot{x} = [\cos(z) \cdot \dot{z}, \sin(z) \cdot \dot{z}] \tag{6}$$
$$\ddot{x} = [-\sin(z) \cdot \dot{z}^2 + \cos(z) \cdot \ddot{z}, \cos(z) \cdot \dot{z}^2 + \sin(z) \cdot \ddot{z}] \tag{7}$$

2.2 Hyperparameter Optimization

We now need an additional autoencoder whose code space recovers the canonical variables z_t for the ODE, given the cartesian coordinates x_i. (Note that the derivatives are not learned, but will be analytically computed by a forward propagation algortihm in the next subtask.) To find a suitable autoencoder architecture, implement and train a simple Autoencoder with Linear and Sigmoid layers to encode x_t into a one-dimensional latent representation z_t and reconstruct $\hat{x}_t$. Try different hyperparameters such as the number and the size of the hidden layers. Use the Adam Optimizer and a learning rate of 10^{-3}. Report the best hyperparameters and use them for the following tasks.

2.3 Propagation of Time Derivatives

Extend your autoencoder with the necessary functionality to compute the derivatives $\dot{z}_t, \ddot{z}_t$ of the codes, given the cartesian coordinates and derivatives $x_t, \dot{x}_t, \ddot{x}_t$. Implement two layer classes `SigmoidDerivatives` and `LinearDerivatives` whose `forward(self, x, dx, ddx) -> z, dz, ddz` function computes the output (as the networks learned output) and its derivatives (via an explicit implementation of the chain rule through the network):

Algorithm 3 Propagation of Time Derivatives

for $l = 1$ **to** $L - 1$ **do**
  $z^0, \dot{z}^0, \ddot{z}^0 \leftarrow x, \dot{x}, \ddot{x}$
  Pre-Acticationes (Linear Layer)
  $\tilde{z}^l \leftarrow z^{l-1} W^l + b^l$
  $\dot{\tilde{z}}^l \leftarrow \dot{z}^{l-1} W^l$
  $\ddot{\tilde{z}}^l \leftarrow \ddot{z}^{l-1} W^l$
  Activations (Sigmoid Layer)
  $z^l \leftarrow g(\tilde{z}^l)$
  $\dot{z}^l \leftarrow g'(\tilde{z}^l) \odot \dot{\tilde{z}}^l$
  $\ddot{z}^l \leftarrow \left(g''(\tilde{z}^l) \odot \dot{\tilde{z}}^l\right) \odot \dot{\tilde{z}}^l + \left(g'(\tilde{z}^l) \odot \ddot{\tilde{z}}^l\right)$
end for

For the sigmoid activation function, the derivatives are

$$g(\tilde{z}) = \sigma(\tilde{z}) = \frac{1}{1 + e^{-\tilde{z}}} \tag{8}$$
$$g'(\tilde{z}) = \sigma(\tilde{z}) \cdot (1 - \sigma(\tilde{z})) \tag{9}$$
$$g''(\tilde{z}) = \sigma'(\tilde{z}) \cdot (1 - 2\sigma(\tilde{z})) \tag{10}$$

Verify your implementation on example inputs $x_t, \dot{x}_t, \ddot{x}_t$ by comparing the propagated time derivatives with the discrete first and second derivatives of the layer's output $z_t = \text{layer}(x_t)$. To compute discrete derivatives, you need the outputs of three consecutive time steps at distance Δt:

$$\dot{z}_t \approx \frac{z_{t+1} - z_{t-1}}{2 \cdot \Delta t} \qquad \ddot{z}_t \approx \frac{z_{t-1} - 2 \cdot z_z + z_{t-1}}{\Delta t^2} \tag{11}$$

2.4 Implementation

Implement the `SINDyAutoencoder` class storing an internal instance of the `SINDy` class from task 1 and an encoder/decoder pair constructed of the `LinearDerivatives` and `SigmoidDerivatives` layers. Use the Xavier uniform initialization and disable the biases. In the `forward(self, x, dx, ddx)` function, compute and return $\hat{x}, \ddot{z}, \hat{\ddot{z}}, \hat{\ddot{x}}$, where $\hat{\ddot{z}} = \Theta(z, \dot{z}) \cdot \Xi$ is the RHS in z-space. $\hat{\ddot{x}}$ is the RHS $\hat{\ddot{z}}$ decoded back into x-space (that is, you decode $\hat{x}$ from z and apply the decoder's derivative propagation algorithm to get $\hat{\ddot{x}}$ from $\hat{\ddot{z}}$).

Verify the correctness of the derivative propagation for multi-layer encoders and decoders again by comparing with descrete derivatives.

Implement a `train_sindy_autoencoder` function in which you use the output of the *forward* function to compute and combine the MSE loss components for the new training objective:

$$\widehat{\varphi}, \widehat{\psi}, \widehat{\Xi} = \underset{\varphi,\psi,\Xi}{\operatorname{argmin}} \left[\frac{1}{T \cdot N} \sum_{i=1}^{T \cdot N} \left(\left\| x_i - \hat{x}_i \right\|_2^2 + \lambda_{\ddot{z}} \left\| \ddot{z}_i - \hat{\ddot{z}}_i \right\|_2^2 + \lambda_{\ddot{x}} \left\| \ddot{x}_i - \hat{\ddot{x}}_i \right\|_2^2 \right) + \lambda_1 \left\| \Xi \right\|_1 \right]$$

The `SINDy` class acts now on the autoencoder's latent space, and the training of the coefficient vector Ξ and mask Υ must be adapted accordingly.

2.5 Refinement

Refinement means that we should switch off the L1 regularization term (i.e. set $\lambda_1 = 0$) once we are sure which coefficients should remain active in the final solution. Continuing training without regularization on only the active coefficients ensures that the bias introduced by the L1 term is eliminated from the final solution (provided we picked the correct terms at the beginning of refinement). Add a hyper-parameter to `train_sindy_autoencoder` that controls at which epoch the refinement begins.

2.6 Training

Train the SINDy-Autoencoder on the training set described in Section 1.2, but embedded into cartesian coordinates according to subtask 2.1. Train for 5000 epochs, followed by 1000 epochs of refinement, with a threshold of $a = 0.1$ and an interval of $S = 500$ epochs for the Sequential Thresholding, and a trend-threshold of $b = 0.002$ and patience of $P = 1000$ epochs for the PTAT. Use the loss weights $\lambda_{\ddot{z}} = 5 \cdot 10^{-5}$, $\lambda_{\ddot{x}} = 5 \cdot 10^{-4}$, $\lambda_{L_1} = 10^{-5}$ as in the original implementation by Champion et al.. Training should take a couple of minutes per model on a GPU.

Log and visualize the training and validation loss components, and the coefficient history. For debugging, you can use tqdm or tensorboard (which also works for pytorch) to monitor the loss and coefficients during training. You should notice that many coefficients quickly drop to 0 under the influence of the L_1 regularization and eventually get turned off by the thresholding algorithms.

Start by using Sequential Thresholding (ST) for training. When training succeeds, train 10 models with ST and PTAT, store their learned coefficients and compute the relevant metrics for evaluation.

2.7 Evaluation & Visualization

Report the average Fraction of Variance Unexplained in x, $\ddot{z}$, and $\ddot{x}$,

$$\mathrm{FVU}_y = \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}. \tag{12}$$

Analyze the resulting equations. How well do they correspond to the ground truth equation? You should get many equations with the $\sin(z)$ term or a z term.

Resimulate the system with the learned equation and the `simulate_pendulum` function and decode the resimulation with the trained decoder. Compare the ground truth and resimulated $\hat{z}(t)$ and $\hat{x}(t)$ and compute the error over time. For up to $t = 1\,s$, the resimulation should be very close to the ground truth.

3 Bonus: SINDy-Autoencoder on Videos

Repeat the previous task with high-dimensional video data. Additionally, consider the following:

3.1 Articial Embedding

Implement an `embed_grid(z, t, resolution, sigma)` function that creates a video of a Gaussian peak located at the tip of the pendulum. Compute the cartesian coordinates of the tip and fill the pixels with values of a 2D Gaussian located at the tip and a small standard deviation. This time, use the finite difference method to compute the time derivatives $\dot{x}$ and $\ddot{x}$. Visualize the data. It should look similar to Figure 2.

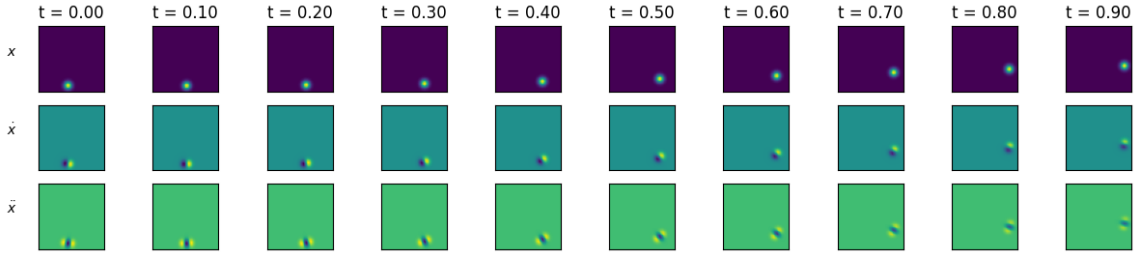

Abbildung 2: Expected result from the artificial embedding

3.2 Hyperparameters

Use three hidden layers of size $[128, 64, 32]$ for the encoder and $[32, 64, 128]$ for the decoder as in the work by Champion et al..

3.3 Implementation & Training

Flatten the video and train the SINDy-Autoencoder with a training set of the same size as with the cartesian coordinates.

3.4 Evaluation

Compare the identified equations using the Sequential Thresholding (ST) with the ones identified using the Patient Trend-Aware Thresholding (PTAT). Whereas ST can still identify the equation in the case of the cartesian coordinates, it should now become obvious that it does not work reliably in more complicated settings.