Solving differential equations: Double pendulum and deep learning

Bálint Hantos (DVTULF)

Introduction

The aim of my project is to investigate how neural networks perform at solving ordinary differential equations. Currently, machine learning solutions are trending in the IT industry. From image and sound processing methods through recommender systems to time series analysis it is getting widespread to use neural networks in fields where large amounts of data is available.

Motivation

The motivation behind this project stems from the strong intertwining between differential equation solving and the nature of neural networks. It is very important to mention, that there is a lot of related work in this area. Notably, recently solvers were used to suggest some new neural network architectures [1,2] as well as new training methods [3].

The usage of neural networks to tackle physical problems regarding ODE solving isn't very new. My personal motivation of choosing this project conceived upon reading an article about using the MLP-model to solve the chaotic three-body problem [4]. Firstly, I thought that the model presented in the article was poorly developed, due to the fact that the MLP-model can't take successivity into account.

On the other hand, recurrent neural networks, as a general tool of time series analysis, can be of a good use, to tackle the notion of events being sequential. I propose a model in this project, which might be capable of solving the equation of motion for the chaotic double pendulum.

Main ideas and basic equations

The double pendulum is one of the first encounters with analytically unsolvable physical systems during undergraduate physics. Although, its differential equations can be solved iteratively, usually on a computer. Also, it is a good example of a chaotic system.

During undergrad, we mostly used the Runge-Kutta-method with fixed and adaptive step sizes, and we compared it to the Euler-method and the Euler-Cromer-method (in the case systems of differential equations).

The double pendulum is a pendulum with another pendulum attached to its end. It exhibits rich dynamic behavior and it's strongly sensitive to its initial conditions. The two pendula can be described by the masses (m_1, m_2) hanging on them, and their l_1, l_2 lengths. The generalized coordinates can be conveniently chosen to be θ_1, θ_2 angles between each limb and the vertical.

The Lagrangian of such system is:

$$L = \frac{m_1 l_1^2}{2} \dot{\theta}_1^2 + m_1 g\, l_1\, cos\ \theta_1 + \frac{m_2}{2} \cdot \left(l_1^2 \dot{\theta}_1^2 + l_2^2 \dot{\theta}_2^2 + 2 l_1 l_2 \dot{\theta}_1 \dot{\theta}_2 \cos(\theta_1 - \theta_2) \right) +$$

$$+ m_2 g \cdot (l \cos\theta_1 + l_2 \cos\theta_2).$$

Thus the equations of motion:

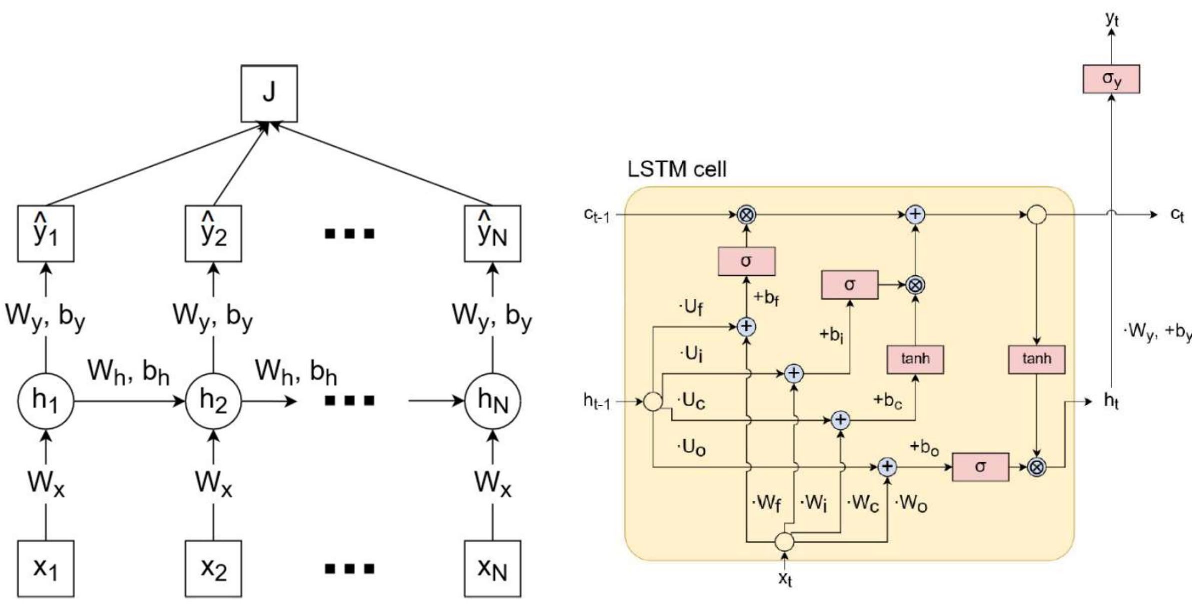
$$\ddot{\theta}_1 = \frac{-g(2m_1 + m_2)\sin\theta_1 - m_2 g \sin(\theta_1 - 2\theta_2)}{l_1(2m_1 + m_2 - m_2\cos(2\theta_1 - 2\theta_2))} - \frac{2m_2 \sin(\theta_1 - \theta_2)\left(\dot{\theta}_2^2 l_2 + \dot{\theta}_1^2 l_1 \cos(\theta_1 - \theta_2)\right)}{l_1(2m_1 + m_2 - m_2\cos(2\theta_1 - 2\theta_2))},$$

$$\ddot{\theta}_2 = \frac{2\sin(\theta_1 - \theta_2)\left(\dot{\theta}_1^2 l_1(m_1 + m_2)\right)}{l_2(2m_1 + m_2 - m_2\cos(2\theta_1 - 2\theta_2))} + \frac{2\sin(\theta_1 - \theta_2)\, g(m_1 + m_2)\cos\theta_1}{l_2(2m_1 + m_2 - m_2\cos(2\theta_1 - 2\theta_2))} +$$

$$+ \frac{2\sin(\theta_1 - \theta_2)\, \dot{\theta}_2^2 l_2 m_2 cos(\theta_1 - \theta_2)}{l_2(2m_1 + m_2 - m_2\cos(2\theta_1 - 2\theta_2))}.$$

RNN network architectures

The MLP-model in the [4] paper can only work with fixed size input and output. We need to use a different approach. Instead of processing the whole series of data in one step, we only look at two elements – the current and the previous element – of the series in each step. Therefore, it is necessary for the neural network to store the relevant information about the previous states. For this task a recurrent neural network is a good choice.



1. Figure: VRNN (left) and LSTM (right) architectures [5]

I intend to examine how the vanilla recurrent neural network (VRNN) and the long short-term memory (LSTM) architecture perform in this task. Firstly, I'm going to use the Runge-Kutta ODE solver to generate datapoints of a motion of a double pendulum with the same parameters (m_i, l_i) and different initial conditions. Secondly, train the above architectures on the generated data to predict the next step based on the previous step. Finally, measure the goodness of the fit of the trajectory with mean absolute error.

The tools to implement such networks can be found in the `Keras` open-source neural network library. It is written in `Python` and can run on top of `TensorFlow`.

Vanilla recurrent neural network (VRNN)

The vanilla recurrent neural network architecture is a simple architecture. On Figure 1. we can see, that it consist of shared W_h and individual W_x weights. Unlike standard dense neural networks, it has feedback connections, thus capable of processing sequences of data.

Long short-term memory neural network (LSTM)

The long short-term memory network architecture is similar to the VRNN architecture. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell. [6]

Loss functions

Mean absolute error (MAE)

The mean absolute error loss is calculated in the following way:

$$J(\boldsymbol{\theta}) = \frac{1}{NK}\sum_{k,n=1}^{K,N} abs\left(\tilde{y}_k^{(n)}(\boldsymbol{\theta}) - y_k^{(n)}\right),$$

where the $y_k^{(n)}$ is the n-th vector's k-th component.

Mean squared error (MSE)

It is similar to the absolute square error in the sense, that instead of taking the absolute value we square the difference between the true and predicted value of the regression and then compute the mean:

$$J(\boldsymbol{\theta}) = \frac{1}{NK}\sum_{k,n=1}^{K,N} \left(\tilde{y}_k^{(n)}(\boldsymbol{\theta}) - y_k^{(n)}\right)^2.$$

The main differences between these two loss functions is that a network is less sensitive to outliers with MAE but it converges slower.

Custom mean absolute error

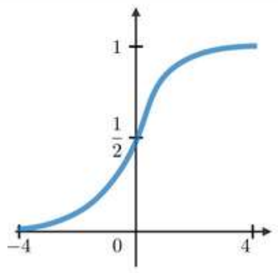
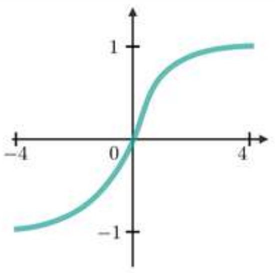
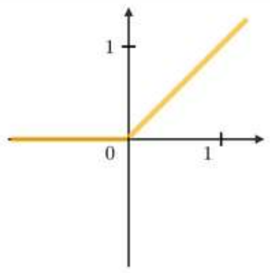
It is crucial for a double pendulum to satisfy the constraint, that the length of each pendulum must be constant. With adding a Lagrange multiplier term to the MAE loss, it takes this criterion into consideration:

$$J(\boldsymbol{\theta}) = \frac{1}{NK}\sum_{k,n=1}^{K,N} abs\left(\tilde{y}_k^{(n)}(\boldsymbol{\theta}) - y_k^{(n)}\right) + \frac{1}{N}\sum_{n} \alpha\ abs\left(L_1 - \widetilde{L_1^{(n)}}\right) + \beta\ abs\left(L_2 - \widetilde{L_2^{(n)}}\right),$$

where $\widetilde{L_1^{(n)}}$ is the first pendulum's predicted length at the n-th timestep.

Activation functions

The most common activation functions used in RNN modules are described below.

Sigmoid	Tanh	RELU
$g(z) = \frac{1}{1+e^{-z}}$	$g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	$g(z) = \max(0, z)$
		

2. Figure: commonly used activation functions [3]

Mostly the tanh(z) activation function is used in RNNs.

Methods

Before sharing the results of the work, I'd like to give some insights on the methods I used in my project.
Firstly, numerous data was generated, namely 300 instances of the motion of a double pendulum, with different initial conditions. The data consists of vectors representing the state of the pendulum in 10000 timesteps. Each vector was constructed in the following way:

$$\boldsymbol{y}(\mathrm{t}) = [\ x_1(\mathrm{t}), y_1(\mathrm{t}), \omega_1(\mathrm{t}), x_2(\mathrm{t}), y_2(\mathrm{t})\ ],$$

where x_i, y_i are the cartesian coordinates of the bobs of the pendula and ω_i are the angular velocities. The justification of the cartesian coordinates is that I wanted to preserve the length of the pendulum, although looking back, it might have been more effective to use the angles.

After generating the data it was split into a training, a test and a validation set in 80%-10%-10% ratios. Then multiple RNN models were trained with trying out different hyperparameters. From the pool of these trained models I've chosen the first five with the best MAE-scores.

Teacher forcing

In many cases RNNs are used to predict the next step of a time series based on the previous steps. The method is also critical in the development of natural language processing models, for example machine translations and image captioning. The first step is to give the model x[t] input from which we want to predict $\tilde{y}(\mathrm{t}) \approx x(\mathrm{t}+1)$. In the next step, the model is given the true $x(\mathrm{t}+1) = y(\mathrm{t})$ input and we anticipate $\tilde{y}(\mathrm{t}+1) \approx x(\mathrm{t}+2)$. This recursive method is similar to solving differential equations. It might be a good idea to add some noise to the data or try it with multiple different time series of, thus the model generalizes better. [5]

Results

The models were trained with the following hyperparameters:

- loss function
  - MAE
  - MSE
  - custom MAE
- activation functions
  - tanh
  - ReLu
  - sigmoid
- layer type
  - VRNN
  - LSTM
- number of layers
  - 1
  - 2

Thus, the number of potential models were 60, from which the best 5 models were chosen using hyperparameter tuning and rank ordering with a metric (namely MAE). In our case with the double pendulum MAE is the mean of the absolute differences of the coordinates of each pendulum.

Hyperparameter tuning

Firstly, with possible hyperparameters multiple trainings were run with only one epoch. This way I could filter for some unknown phenomena, observed during exploring the possible model variations: some hyperparameters didn't work out to an extent that the loss function became NaN at some point in the training. With the hyperparameters, which had loss values other than a NaN, I trained for multiple epochs. Out of the possible 60 hyperparameter configurations there was 45, which passed this criteria.

no.	MAE loss	number of layers	loss function	activation function	layer type
1	0.7725	1	custom MAE	tanh(z)	VRNN
2	0.8363	1	MAE	tanh(z)	VRNN
3	0.8593	1	custom MAE	tanh(z)	LSTM
4	0.8807	1	custom MAE	tanh(z)	LSTM
5	0.8867	1	custom MAE	tanh(z)	LSTM

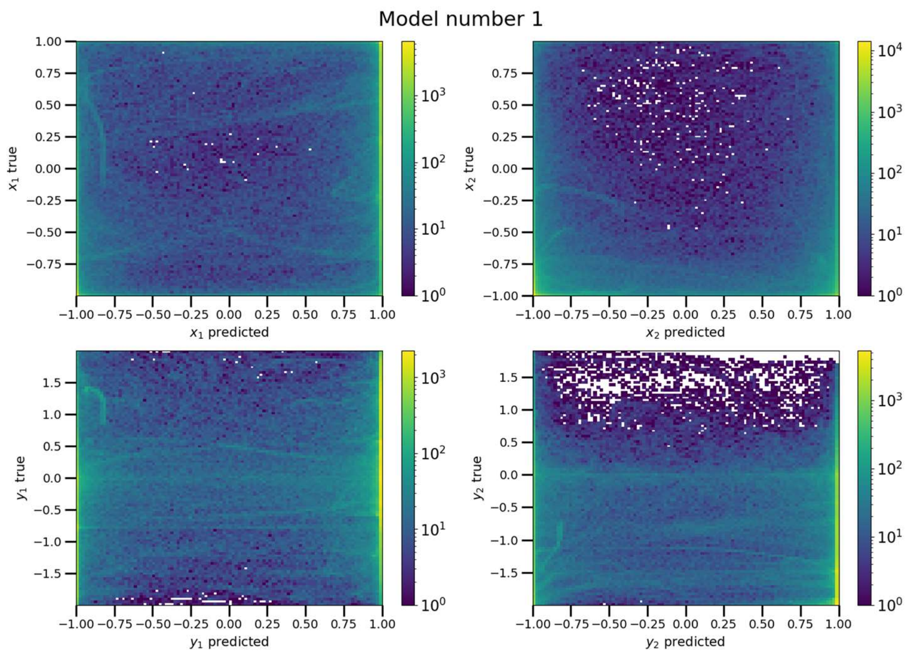
1. Table: top 5 models with lowest MAE-scores

Secondly, with these configurations I trained the model for 30 epochs in the single layer case and 15 epochs because it took a lot of time with 2 batches. (It took a lot of time.) After that, I rank ordered these models (1. Table) on the basis of their MAE-score on the test set.

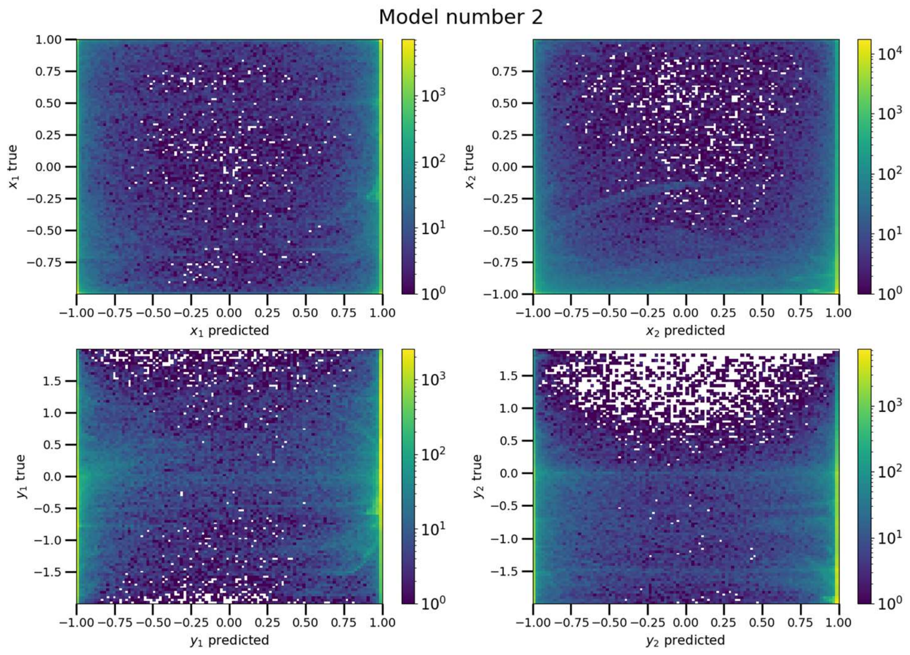
Unfortunately, the parameters of the custom loss couldn't be retrieved, since I've only returned the keywords or functions themselves (it was in the form of: <function loss_fixed_len.<locals>.loss at 0x000001B4F87A6438>).

Confusion histograms

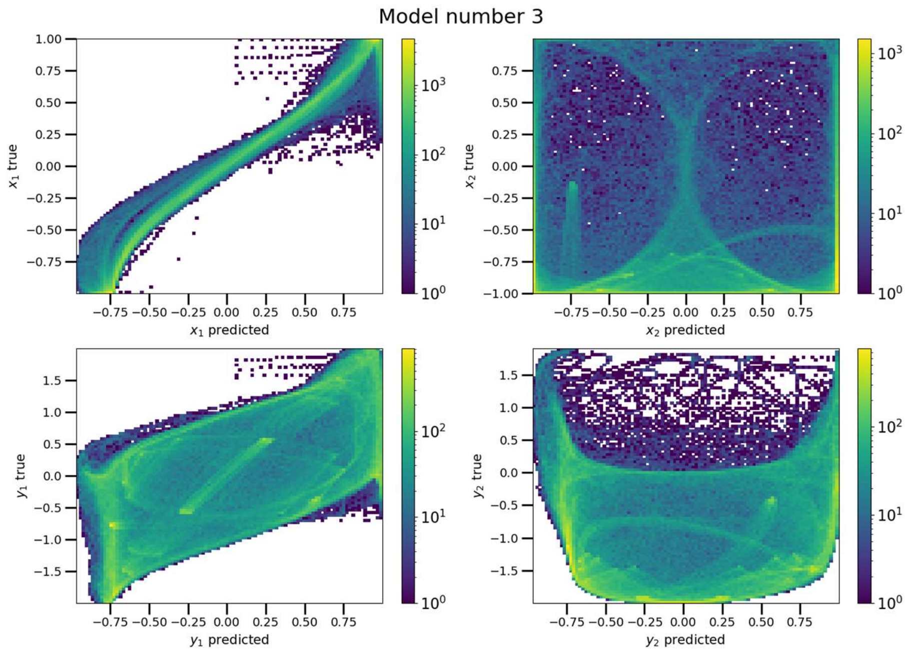
To visualize our models' predictive power, a confusion matrix-like method was used, but in this case for regressions. On the graphs below we can see, how each model performed: the predicted values of the coordinates are plotted against the true values. The ideal case would be a straight line between (-1,-1) and (1,1).



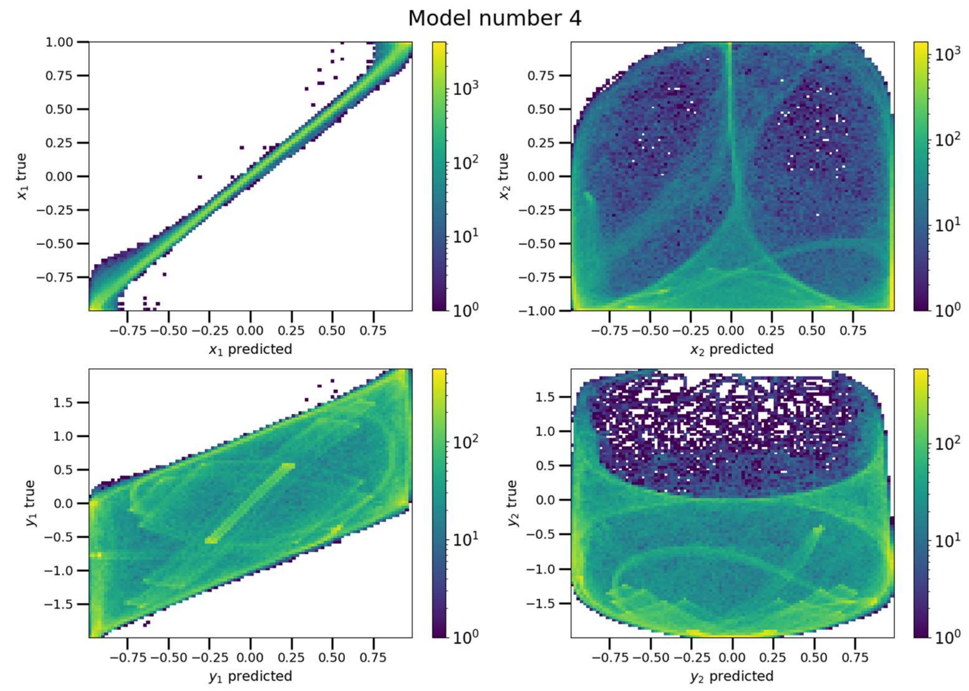
3. Figure: confusion histogram for model custom MAE loss, 1 VRNN layer



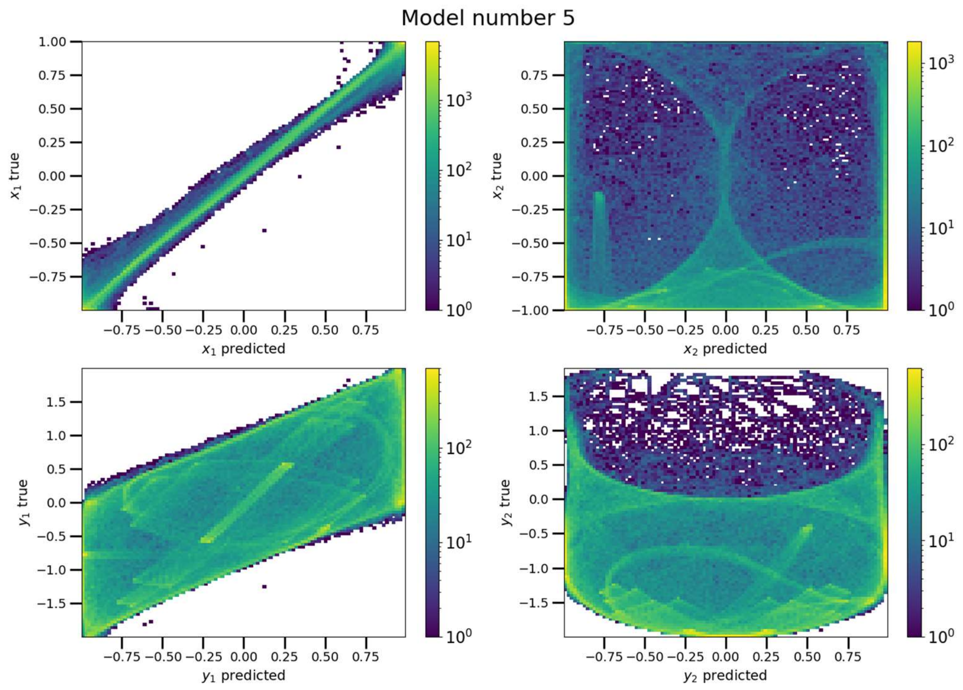
4. Figure: confusion histogram for model 2
MAE loss, 1 VRNN layer



5. Figure: confusion histogram for model 3
custom MAE loss, 1 LSTM layer



6. Figure: confusion histogram for model 4
custom MAE loss, 1 LSTM layer



7. Figure: confusion histogram for model 5
custom MAE loss, 1 LSTM layer

On these plots, we can see, that the LSTM layer performed significantly better than the VRNN layer in predicting the first pendulum's position. Especially the x_1 coordinate. The second pendulum's coordinates were predicted much more badly.

Even though the VRNN's MAE-score was better than the LSTM's, their predictions are useless. It's mostly noise in the central region, however it hits them perfectly in the boundary regions.

On Figure 5 and 7 the x_2 coordinates density resembles a tanh(z) and -tanh(z) function. This might be because there's only one layer, so I suspect, adding more layers would resolve these errors.

For demonstration, I've attached a video of every models prediction of motion (blue pendula) with true motion (red pendula). They can be found in the docs/visualization folder.

Conclusion

First and foremost, this method is based on the similar approach of the iterative differential equation solver integrators: we calculate the next steps of a time series with the help of the previous steps. This method needs much more computational power than the integrator, besides it needs the integrator, to generate the data in the first place.

Another observation is that VRNN architecture performed much worse in this task, than the LSTM layer. Moreover, it's quite likely, that using more layers would lead to more precise predictions. However, it'd use significantly more computational power, memory and time to make these calculations.

References

[1] Haber, Eldad, and Lars Ruthotto. "Stable architectures for deep neural networks." Inverse Problems 34.1 (2017): 014004.

[2] Chen, Ricky TQ, et al. "Neural ordinary differential equations." *Advances in neural information processing systems.* 2018.

[3] Bo Chang, Lili Meng, Eldad Haber, Lars Ruthotto, David Begert, and Elliot Holtham. Reversible architectures for arbitrarily deep residual neural networks. arXiv preprint arXiv:1709.03698, 2017.

[4] Breen at al.: Newton vs the machine: solving the chaotic three-body problem using deep neural networks

[5] Viktor Varga: Deep neural networks: algorithms and architectures course at ELTE Department of Informatics

[6] Wikipedia - Long short-term memory :https://en.wikipedia.org/wiki/Long_short-term_memory

[7] Standford CS230 course: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks