Comparing and Studying Neural Network Models using a Double Pendulum System

FYS5429 at University of Oslo

Simen Løken

May 2023

Contents

1 Abstract

The double pendulum system is an iteresting system in physics given its chaotic behavior. In this project we will study such a system using various different Neural Network models and weighing them against each other. We will be using a Feed Forward Neural Network, a Recurrent Neural Network and a Convolutional Neural Network. To do this we will first study how to numerically generate sufficient double pendulum data using a Runge-Kutta 4 method, then compare the models. Ultimately, the models are very close, but we find that Convolutional Neural Networks are surprisingly effective in their modelling of such systems.

2 Introduction

When studying a system like a double pendulum, one of the most common ways to solve and study them is through the use of numerical methods like the Euler Method or the Runge-Kutta family of methods. Typically, one would take some initial variables and use them, with some suitable timestep, to predict how the system would unfold as time progresses. This is all well, but still somewhat limited. In this project, we're going to look at an alternative method of studying such numerical systems, namely by using Neural Networks

Neural Networks are deceptively good at handling and solving numerical systems. As opposed to traditional numerical methods, Neural Networks can solve systems without underlying explicit mathematical models or equations. That is, it can learn the system purely through data. This is called a data-driven approach to numerical modelling, and allows us to even model systems too complex or without sufficient domain knowledge.

Additionally to this, Neural Networks are often more stable than their numerical counterparts. With a problem like double pendulums, you'll often find that even small changes to the initial variables can have explosive results, and some times the simulation will break. This is highly unlikely with Neural Networks. As Neural Networks are trained on a large amount of data, it is rare, if not impossible for a well trained Neural Network to predict a "broken" simulation for a given set of initial variables.

It are these (and more) attributes that make Neural Networks a very interesting field within the computational sciences, and in this project we're going to study from the ground up how you can create a suitable Neural Network to study a rather chaotic system like that of double pendulums.

3 Theory and Method

3.1 Neural Networks - What are they?

Neural Networks are inspired by our brains in the way they are made. At their simplest, a Neural Network consist of an input layer, hidden layer(s) and an output layer. Input layers receive what we put into the Neural Network. It is then passed onto the neurons (hence the name) which is the hidden layer. Each neuron in the hidden layer has a set of weights and biases, which in turn determine the connections between the neurons and the sensitivity to input. The weights and biases are updated as the network trains. After being passed through all the hidden layers, we get an output.

3.1.1 Feed-Forward Neural Network

The simplest form of the Neural Network, the Feed-Forward Neural Network (FFNN) is named as such because the information flows forward. That is, information only ever flows from the input layer to the output layer. It never experiences any recurrent connections.
Although they are simple, that is not to say they are weak. Even just a shallow FFNN can be more than sufficient for many computational tasks.

3.1.2 Recurrent Neural Network

Perhaps the most relevant to what we're going to be exploring, Recurrent Neural Networks (RNN) is a Neural Network that is designed to handle sequential data. That is to say, the order of the inputs matter. This, in turn, makes it very suitable for solving time-sensitive numerical systems, such as the double pendulum system we're going to be examining.
RNNs, as the name suggests, has recurrent connections. Whereas in a FFNN, a given neuron only recieves information from a previous layer, a neuron in an RNN also recieves information from its previous state. In turn, this allows a given neuron to retain a sort of pseudo-memory or context.
There are many different types of RNN, but we're specifically going to be using a Long Short-Term Memory (LSTM) network. RNNs inherently have a problem where the gradients used to update the weights diminish exponentially as they propogate (either through layers or time steps), which in turn weakens the pseudo-memory of a given neuron and leads to poor learning. An LTSM model combats this by further expanding on the memory of an RNN, as LTSMs can selectively retain or forget information. This is done through introducing "gates" to each portion of the neural network. An input gate, a hidden(or forget) gate and a output gate, where the input gate determines how much of the input will be stored, the hidden gate determines which parts of the previous state to forget and the output gate which regulates the information recieved from the input and hidden gate. This in turn allows for a very powerful model.

3.1.3 Convolutional Neural Network

Lastly we're going to be using Convolutional Neural Networks (CNN). Typically, CNNs are used to study or model images and video, which may raise some eyebrows as to why we're using this here. However, if we assume we have an input of some x timesteps worth of 2d data, length y, such that the shape is $(y, x, 2)$. A given input is then of shape $(x, 2)$, and we can reshape it to be $(x, 2, 1)$. This in turn, effectively allows us to model our data as if it was an image, which is what CNNs are made for.
As hinted at above, CNNs are uniquely good at images and video, or to be more accurate, grid-like data, which may be more of an explanation as to why we're rearranging our data to be a grid. As to why CNNs are so useful with grid-like data, a CNN has three types of layers. Firstly, there are convolutional layers (hence the name). Secondly there are pooling layers and lastly the fully connected layers. Convolutional layers are a set of learnable filters that perform convolutional operations on the input data. this is done by letting the filters scan the input and extract local patterns. The pooling layers reduce the spacial dimensions of the data while retaining important features. This is important, as CNNs are very computationally heavy compared to the other Neural Networks in this project. Lastly, the connected layers are similar to those in normal FFNN and have the final say before it reaches the output layer, that is to say, the connected layers produce the output through aggregated extracted features from the previous layers.

3.2 Data Generation

As we need data to train our Neural Network, we will be using a Runge-Kutta4 numerical method to generate the data. The Runge-Kutta4 is a well known and popular method, but for simplicity's sake, let us go through it:

$$y_{n+1} = y_n + \frac{h}{6}\sum_{i=1}^{4} b_i k_i = y_n + \frac{h}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \tag{1}$$

where:

$$k_1 = hf(x_n, y_n)$$
$$k_2 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right)$$
$$k_3 = hf(\left(x_n + h, y_n + \frac{k_2}{2}\right)$$
$$k_4 = hf(x_n + h, y_n + k_3)$$

where h is the timestep.

3.2.1 Double Pendulums - Numerically

We also need to find the appropriate solution to the Double Pendulum system, numerically. First, we must find the Lagrangian L:

$$L = T - V = T_1 + T_2 - (V_1 + V_2)$$

where

$$T_1 = \frac{1}{2}M_1\left(\dot{x}_1^2 + \dot{y}_1^2\right)$$
$$T_2 = \frac{1}{2}M_2\left(\dot{x}_2^2 + \dot{y}_2^2\right)$$

and

$$V_1 = gM_1y_1$$
$$V_2 = gM_2y_2$$

Using then that x and y for pendulum 1 and 2 is expressed as:

$$x_1 = L_1\sin(\theta_1)$$
$$y_1 = -L_1\cos(\theta_1)$$
$$x_2 = x_1 + L_2\sin(\theta_2)$$
$$y_2 = y_1 - L_2\cos(\theta_2)$$

The Lagrangian then becomes:

$$\begin{aligned} L = {} & \frac{1}{2}(M_1 + M_2)L_1^2\dot{\theta_1}^2 + \frac{1}{2}M_2L_2^2\dot{\theta_2}^2 \\ & + M_2L_1L_2\dot{\theta_1}\dot{\theta_2}\cos(\theta_1 - \theta_2) \\ & + (M_1 + M_2)gL_1\cos(\theta_1) + M_2gL_2\cos(\theta_2) \end{aligned}$$

Solving then:

$$\frac{d}{dt}\left(\frac{\delta L}{\delta \dot{\theta}_1}\right) - \frac{\delta L}{\delta \theta_1} = 0$$

$$\frac{d}{dt}\left(\frac{\delta L}{\delta \dot{\theta}_2}\right) - \frac{\delta L}{\delta \theta_2} = 0$$

We find the accelerations:

$$\ddot{\theta} = \alpha_1 = \frac{-g(2M_1 + M_2)\sin(\theta_1) - M_2 g \sin(\theta_1 - 2\theta_2) - 2\sin(\theta_1 - \theta_2)M_2(\omega_2^2 L_2 + \omega_1^2 L_1 \cos(\theta_1 - \theta_2))}{L_1(2M_1 + M_2 - M_2\cos(2\theta_1 - 2\theta_2))} \tag{2}$$

$$\ddot{\theta} = \alpha_2 = \frac{2\sin(\theta_1 - \theta_2)(\omega_1^2 L_1(M_1 + M_2) + g(M_1 + M_2)\cos(\theta_1) + \omega_2^2 L_2 M_2 \cos(\theta_1 - \theta_2))}{L_2(2M_1 + M_2 - M_2\cos(2\theta_1 - 2\theta_2))} \tag{3}$$

We can imagine such a system to look like this:

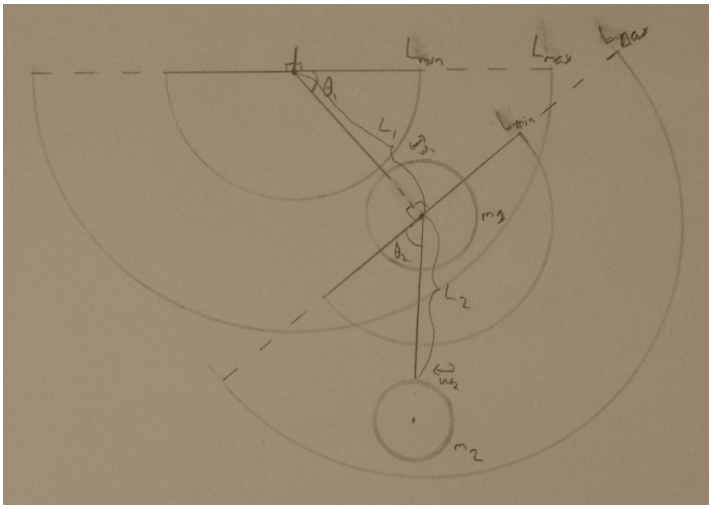

Figure 1: One possible configuration of our system
We have a first pendulum 1, whose length L_1 lies within L_{min} and L_{max} the possible values θ_1 lie within the θ_{min} and θ_{max} in relation to the vertical. Additionally, ω_1 and m_1 are also randomized within the boundaries of $\omega_{min}, \omega_{max}$ and $_{min}, m_{max}$
Similarly, pendulum 2 also uses the same limits, but its angle θ_2 is instead relative to the angle θ_1, essentially making its limits $\theta_1 + \theta_{min}$ and $\theta_1 + \theta_{max}$

where the limits are:

	L [m]	θ [rad]	ω [m/s]	m [kg]
min	0.5	$-\frac{\pi}{2}$	-0.05	0.25
max	1	$\frac{\pi}{2}$	0.05	0.5

Table 1: The limits we define for the randomized initial variables for our system.

4 Results

First, we implement our RK4 model to generate a sufficient data set for us to use. To ensure a sufficiently accurate model, we create a heat map for pendulum 2, with a total of 1000 runs.
We find:

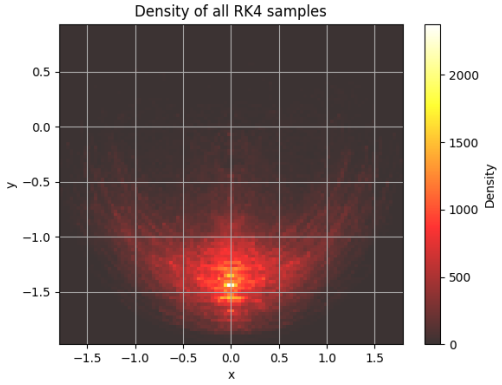

Figure 2: A heat map of all positions x_2, y_2 over a period of 5 seconds. As predicted, it takes the shape of an ω or a w. It may at first be odd that we do not see a crescent shape, but this is because occasionally pendulum 2 will pivot about pendulum 1 as it normally along the "bottom" of its path. This behavior can be seen easier in Fig. [3]

To doubly study this movement, we take a small subsection of all samples and plot the path traced by pendulum 2.
We get:

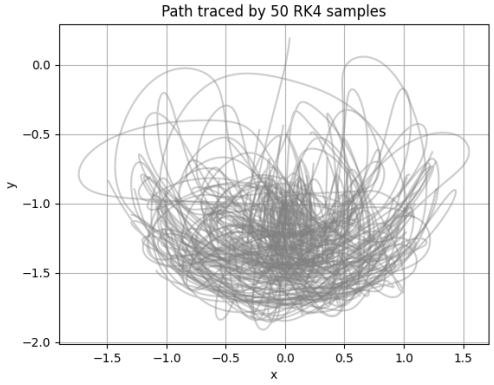

Figure 3: The path traced by 50 pendulum 2's over a period of 5 seconds. As mentioned above we see here the somewhat characteristic ω or w movement.

Let use now examine the three different Neural Networks we will be using head to head.

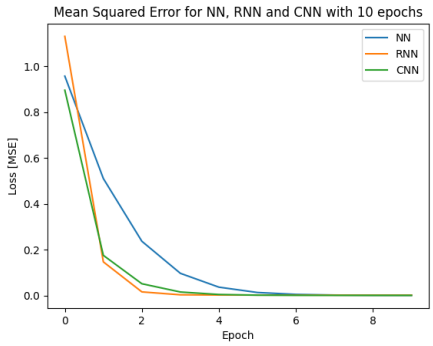

Figure 4: A macro image of the training error

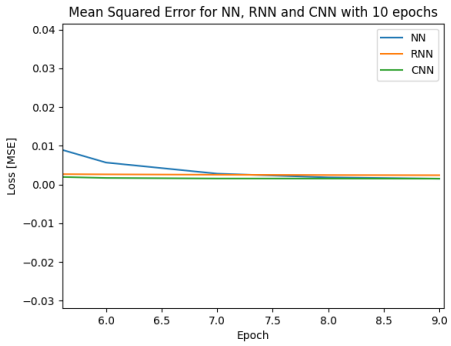

Figure 5: A micro image of the training error

Figure 6: A macro and micro image of the training errors for the different models over 10 epochs. We see here, perhaps somewhat surprisingly, that NN and CNN outperforms RNN, but only slightly. As for the test losses for this run, we get:
NN: 0.517 RNN: 0.529 CNN: 0.571

The fact that, relatively, CNN performs much worse (again, relatively) than RNN and NN in terms of test error, could be indicative of some kind of overfitting. We move now on to doing the simulations with the models. As it is hard to gauge the success rate of a model purely from the training and test error. It can give us a good benchmark, but to truly see if our model is good, we must compare it to the real deal.

To do this, we will create plots similar to those found in Fig. [2] and Fig. [3], but for each of the different models. Starting first with NN:

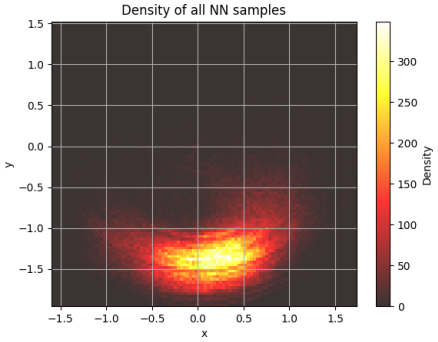

Figure 7: A heat map of pendulum 2 in the predicted NN model

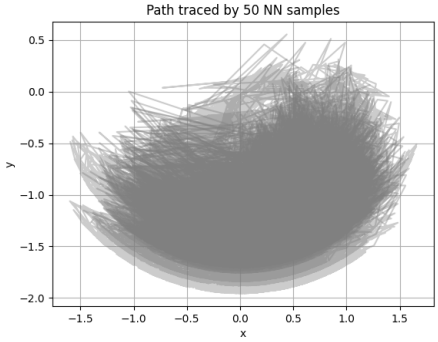

Figure 8: 50 samples of pendulum 2 traced with the NN model

Figure 9: We see here the a heat map and 50 traced samples respectively. At face value, it's easier to extract data from Fig. [7] than Fig. [8]. We see that the model somewhat understands what an oscillation is when averaged out, but looking at the density bar, we see that its peak value is much, much lower than that of Fig. [2]. This suggests that although the model somewhat understands, it is much more random in its prediction than it's "real" counterpart. As for the other image, we see here clearly that the movement of pendulum 2 is much more random than in the RK4 simulation. *It is worth noting that the RK4 heat map is 1000 simulations. This heat map (and subsequent ones) are 200. Thus, the density is expected to be lower, but not by this much (if equal, this density bar should've peaked at atleast 400)*

Next is RNN:

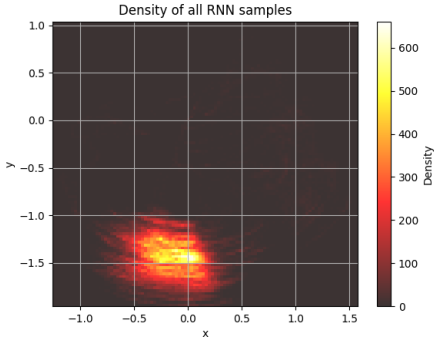

Figure 10: A heat map of pendulum 2 in the predicted RNN model

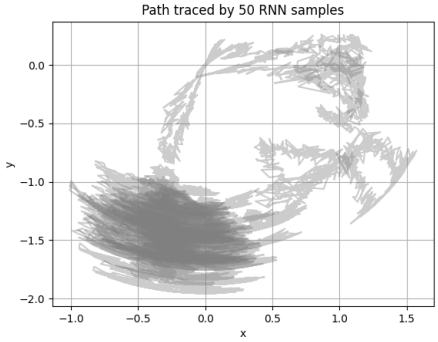

Figure 11: 50 samples of pendulum 2 traced with the RNN model

Figure 12: We see here better density than that of Fig [7], but it seems like the model has misunderstood the oscillation, and has a bias towards going to the left. A similar story is told in the traced image. This suggests that the model has understood oscillation, but has somehow not understood where it should take place. Additionally, while the motion seems more controlled than Fig. [8], there are still a few massive outliers.
Notice also that the oscillation is much more concentrated. This suggests that the model should've had less emphasis put on samples "mid oscillation", that is, in the process of going left to right or opposite.

Lastly, let us look at CNN:

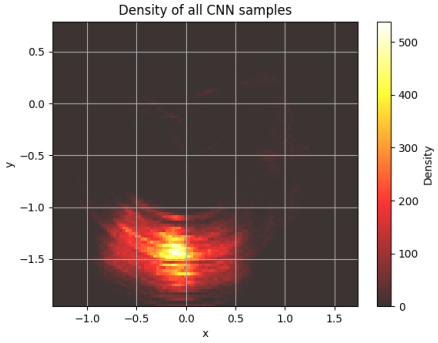

Figure 13: A heat map of pendulum 2 in the predicted CNN model

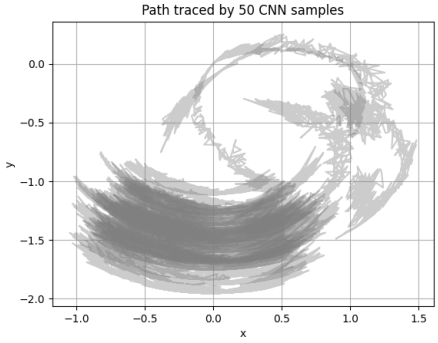

Figure 14: 50 samples of pendulum 2 traced with the CNN model

Figure 15: Here we see much the same behavior as in RNN, but the oscillating motion is better and less concentrated. It still has the outliers where it'll randomly shoot out.

5 Discussion

5.1 Overfitting and underfitting

It is hard to say specifically if overfitting or underfitting was the issue here. While the models managed to learn some of oscillating motion of the double pendulum system, it was still not "good enough" to be able to say that it could clearly model a double pendulum system. An other issue that can easy arise in cases like this that is quite hard to account for is that when splitting between train and test data, the majority of the test data were samples where the angles θ_1, θ_2 where initialized with negative values. This leads to a bias towards "left-side systems" and also increased the loss experienced when predicting the remaining "right-side systems". There is no easy way to fix this. Had it only been that the angles θ_1 and θ_2 that were randomized, we could've made it so both train and test had an even split, but this was not possible as all the other variables are also randomized (and could thus lead to similar biases). Perhaps evenly splitting θ_1, θ_2 in between train and test would've been the lesser of two evils.

5.2 Numerical Stability and volatility of initial variables

One of the inherent problems when generating data of a chaotic system like double pendulums is the numerical stability. Doubly so as we're dealing with randomized (within boundaries) initial conditions. This in turn means that we have to be careful when initializing the system. While this can be partially remedied by weeding out bad runs automatically, we still have a limited range of initial variables we can study. One of the key strengths of Neural Networks is that they can model for initial conditions they haven't seen before, and while that still remains true here, the limited number of initial variations does hurt this aspect of Neural Networks a bit.

5.3 Parallelization

One of the strongest cases and what could probably be done to further improve or expand upon this project is to implement some kind of parallel programming. Whereas traditional numerical methods like the RK4 is reliant on a previous data point to make predictions, thus making them cumbersome to parallelize, NNs are immune to this issue. As long as the model is trained, an infinite (theoretically) number of predictions can be run in parallel.

5.4 Further improvements - Unimplemented methods

There are two sections of the code that are left in despite not working and ultimately going unused. Firstly, a grid-search algorithm that intends to find the best possible hyper-parameters, activation functions, learning rates etc. This ultimately did not work, sadly, which does mean more time has to be spent manually checking possible combinations and their associated models/losses. This is doubly cumbersome for RNN and CNN which both take a fairly long time to train.
Secondly is a custom loss function. While the losses seem fine with a normal loss function implementation (in this case MSE), these are somewhat deceptive. When using the function `plot_random_sample()` we see that, despite very low losses, the predicted model doesn't look much like the real thing at all. Alternatively, these can be viewed here, here and here for NN, RNN and CNN respectively. The idea was that, assuming there is zero loss of energy in the system, we could take the initial energy of the system at $t = 0$, and use that as an upper limit to the energy. Thus, if at any point the energy of the system went beyond that, the model would be penalized. This in turn *should* have made for a better model, but ultimately ended up not working. This would probably also solved the most prevalent issue in the RNN and CNN model, with the random jittery movement where the second pendulum shoots out of the oscillating motion.

6 Conclusion

In closing, we've shown how to create a chaotic double pendulum system and use that system to generate data that we can then use to train different Neural Networks and weigh them against each other. We've also seen how the different models handle predicting the motion of double pendulums and how some models perform better than others. Ultimately, it is hard to draw a conclusion as to which model best predicts the chaotic motion of double pendulums, but a strong case can be made for CNNs, especially if we could "clamp" the energy of the system.

References

[1] Morten Hjorth-Jensen. *FYS4411 Lecture Notes*. 2023. URL: http://compphysics.github.io/ComputationalPhysics2/doc/web/course.

[2] Morten Hjorth-Jensen. Fys5429 lecture notes, 2023. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html.

[3] Wikipedia. Convolutional neural networks, 2023. URL: https://en.wikipedia.org/wiki/Convolutional_neural_network.

[4] Wikipedia. Neural networks, 2023. URL: https://en.wikipedia.org/wiki/Neural_network.

[5] Wikipedia. Recurrent neural networks, 2023. URL: https://en.wikipedia.org/wiki/Recurrent_neural_network.

Appendices

A Appendix Title

A.1 Code

Any code can be found at my github