Project 2:

The Three-body Problem –
Solving a Chaotic Motion Problem Using Deep Neural Networks

Jonatan Haraldsson
jonhara@chalmers.se

Oscar Stommendal
oscarsto@chalmers.se

Abstract

We have explored a deep learning approach to solve the Three-body problem, using neural networks to approximate particle trajectories based on initial positions. Building on previous work by Breen *et al.*, the main goal was to use a similar approach by training neural networks on data provided by the N-body solver `Brutus` [1][2]. Using the `Keras` library in `Python` [3], various neural network architectures, including fully connected-, convolutional-, and recurrent layers, were tested and compared. During training, the mean absolute error with, and without an additional energy conservation constraint was minimized. By studying predicted trajectories visually, we concluded that a model architecture similar to Breen *et al.* with 10-layer fully connected model best replicated the trajectories from `Brutus`. Despite the similarities, our model was outperformed by the model used by Breen *et al.*, since they used a higher number of training epochs. To test how well the models handle the chaotic nature of the Three-body system, predictions with disturbed initial conditions were compared, revealing that small changes diverge over time. Lastly, adding an energy conservation constraint in the cost function improved energy conservation in predicted trajectories, however, these trajectories were visually less similar to the true trajectories from `Brutus`. In further investigations, the implementation can be adjusted, for instance, tuning the constraint parameter α could result in better performance.

October 2024
Course: *Learning from Data, 7,5 hp*
Course Code: *TIF285*

Physics, MSc.
Chalmers University of Technology

CHALMERS
UNIVERSITY OF TECHNOLOGY

1 Introduction

The last decades have introduced us to arguably the greatest threat – and opportunity – that mankind has ever experienced: Artificial Intelligence (AI). This has in many ways revolutionized our lives – from multibillion-dollar companies like Google or Microsoft, to leading universities, to parents that want assistance writing a speech for their graduating son or daughter – everyone uses AI. As a subcategory of AI, Machine Learning has emerged as a rising star in problem-solving and modelling. By feeding an algorithm (i.e., a computer) data from a problem or model, new data outcomes – essentially solutions – can be predicted without needing to fully understand the underlying model. Beyond its modelling advantages, this approach is also useful when the numerical approach to a problem becomes computationally costly.

A well-known problem that has been studied for more than 300 years, adhering to this arises when three point masses orbit each other in space. First formulated by Isaac Newton while studying the interactions between the Earth, the Moon, and the Sun, this is now known as the Three-body problem [4] [5]. The goal is to determine each mass' trajectory as a result of the gravitational influence from the other masses, i.e. solving Newton's equations of motion. However, due to the chaotic nature of the system – where small changes in initial conditions can lead to vastly different outcomes – numerically solving the equations of motion becomes computationally expensive.

However, as mentioned above, machine learning has shown potential in overcoming computational costs using previous known solutions to the problem. This was studied by Breen *et. al* [1], who used output from a N-body solver, `Brutus`, developed by Boekholt and Zwart [2], to train a *neural network* (a kind of machine learning model) in order to predict solutions to the Three-body problem. In this project, the aim is to study the usage of neural networks to solve the Three-body problem and to reproduce results from [1], where focus mainly will be on model architecture.

2 Theory

In this section, a brief introduction to neural networks and their function are presented. As mentioned earlier, a neural network is a type of machine learning model inspired by biological processes [6]. The principal function of a neural network can be explained as follows: we have a set of input data $\boldsymbol{X}$ that enters the *input layer* of the model, see Figure 1. The signal then travels through all N *hidden layers* until it reaches the *output layer*, where it exits the network as output $\boldsymbol{Y}$. Each layer contains a number of *neurons*, represented as circles in Figure 1 [6]. A neuron represents a non-linear function

$$f(x_0, x_1, ..., x_n; w_0, w_1, ..., w_n) = \Phi \circ g = \Phi(g(\vec{x}; \vec{w}))$$

that takes input data $\vec{x}$ and additional weight terms $\vec{w}$ and produces an output $\vec{y}$. The non-linear function Φ is known as the *activation function* and takes the output produced by the function $g(\vec{x}; \vec{w})$ as input. This approach of combining a linear and non-linear function in the neuron increases the modelling power, while also simplifying analytical results in the neural network. The output of g can differ, but it most often combines the input $\vec{x}$ and weights $\vec{w}$ in a linear manner such that

$$g(\vec{x};\vec{w}) = w_0 + \sum_{i=1}^{n} w_i x_i.$$

Note that x_0 is often set to 1, which introduces the so-called bias term w_0 [6]. This allows the output of the neuron to be independent of the input data, which makes the network more flexible. The main goal of the network is then to find parameters $\vec{w}^*$ for each neuron that minimizes the cost function $C(\vec{w})$. This function can be chosen arbitrary and often depends on the output distribution, but common choices are the mean squared error (MSE), mean absolute error (MAE) or cross-entropy. Minimizing $C(\vec{w})$ is an optimization problem and common optimizing algorithms such as AdaGrad, RMSProp, or Adam are often based on Gradient Descent.

The tuning part of a neural network, or any machine learning model in general, is referred to as the model *training* [6]. Here the model is fed input data $\vec{x}$ from a model or problem along with the known model predictions or solutions $\vec{y}$ for the given input, often referred to as *targets*. The optimizer then tunes the weights $\vec{w}$ in order to minimize the cost function, as explained above. This results in a model that, hopefully, predicts accurate solutions for unknown input data, known as *validation data*.

Moreover, one can imagine that the architecture of a neural network can be vastly varying. The most usual aspects to vary are the number of (hidden) layers, the number of neurons in each layer, the activation function for each layer's neurons, the connectivity between neurons and how the data travels through the network [6]. However, the choice of architecture is highly dependent on the problem and the balance between computational efficiency and predictive accuracy.

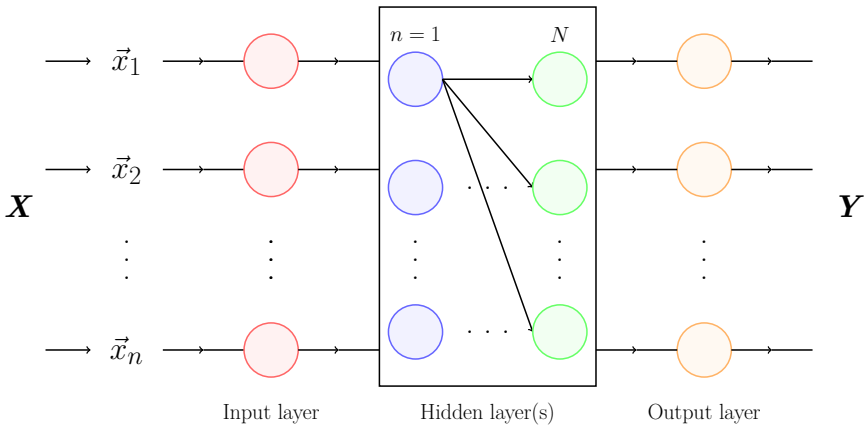


Figure 1: Schematic view of a neural network where each circle represents a single neuron. An input signal $\boldsymbol{X}$ enters the model in the input layer, proceeds through the N hidden layers, and exits from the output layer. This process produces an output $\boldsymbol{Y}$, which can be interpreted as a prediction based on the input data.

3 Method

In this section, methods used to solve the Three-body problem and reproduce the results found by Breen *et al.* [1] are presented. Focus will mainly be on the data processing, neural network design and optimizer choice. Note that the complete `Python` script used in the project is available on GitHub. Moreover, to spare our beloved personal laptops from an early retirement, most of the heavy training was performed in Google Colab [7].

3.1 Data Processing

Initially, data – provided by `Brutus` – was split into 90 % training and 10 % validation data, and then further "massaged" into inputs and outputs manageable for the neural network. The given data consisted of 9 000 trajectories for two particles with 1 000 time steps, each containing 9 data points. So, in total a $(9\,000 \times 1\,000 \times 9)$ tensor, where the last column consists of positions, velocities and a time step for two of the three particles. With the knowledge that some trajectories converged to $(\vec{x}_i, \vec{y}_i) = (0, 0)$ for all particles, these were removed, leaving in total 7 623 trajectories.

In this case, the inputs, $\boldsymbol{X}$, were set as initial positions for particle two with coordinates (x_0, y_0) (particle one always started in $(1, 0)$), while the outputs, $\boldsymbol{Y}$, were set to the trajectories for particle one and two $(\vec{x}_1, \vec{y}_1; \vec{x}_2, \vec{y}_2)$. With this set-up, the network was designed to predict trajectories for particle one and two given the initial position for particles one. The third trajectory was obtained from the condition

$$\sum_{i=1}^{3} (\vec{x}_i, \vec{y}_i) \equiv (0, 0).$$

3.2 Neural Network Design and Training

Next, the pre-trained model `Breen_NN_project2` used by Breen *et al.* was examined and used to make predictions. This model contained ten `Dense` layers with 128 neurons each and one `Dense` output layer with 4 neurons, which in total gave $\sim 150\,000$ trainable parameters. In addition to the pre-trained model, five different models, whose architecture are displayed in Table 1, were designed, trained and evaluated on the validation data. Similar to [1], models with added convolutional layers (`Conv1D`) was tested. The layers of models 1 and 2 contained 64 and 128 neurons each, respectively. The layers of models 3, 4 and 5 all contained 64 neurons each.

Table 1: Overview of different models used to solve the Three-body problem.

Model	**Type of layers**	***N* Layers**	**Trainable parameters**
Pre-trained	`Dense`	10	149 636
1	`Dense`	5	17 156
2	`Dense`	10	149 636
3	`Conv1D`, `Dense`	5	21 508
4	`LSTM`, `Dense`	5	65 092
5	`Conv1D`, `LSTM`, `Dense`	5	91 524

As outlined by [1], the ReLU activation function was applied to all neurons, and the mean absolute error was used as cost function. To obtain faster computational times, training data was divided into batches of 68 000. This meant a total of ~ 100 iterations to go through all data points (this number was also used for the validation data). According to [8], a batch size of one to a few hundreds usually yields the best compromise between performance and computational time. To arrive at the chosen batch size, model 1 was trained with different batch sizes. In the end, the batch size of 68 000 had the best combination of training time and error, and it was thus used throughout the project.

During training, the maximal epochs was set to 200. However, an early stopping technique was used to minimize training time, where the training was disrupted if the error was not improving within 10 epochs. After training, predictions made on an initial position for particle one was compared to trajectories obtained by `Brutus` and the pre-trained model. The comparison was done visually by plotting the trajectories.

Furthermore, to study the chaotic nature of the Three-body problem, predictions with an added disturbance to the initial position were plotted at $t \in \{1, 250, 500, 750, 1\,000\}$. Figure 2 presents an example of 100 disturbed initial positions, given by $r_{\text{dist.}}(\cos(\varphi), \sin(\varphi))$, with $r_{\text{dist.}} = 0.01$ and randomly sampled $\varphi \in (0, 2\pi)$. Lastly, the Lyapunov exponent λ, a quantitive measure of the how the disturbances propagate through time, was estimated using $\delta(t) = \delta(0)e^{\lambda t}$, where $\delta(t) = |T_{\text{non-dist}} - T_{\text{dist}}|(t)$ for predicted trajectory T.

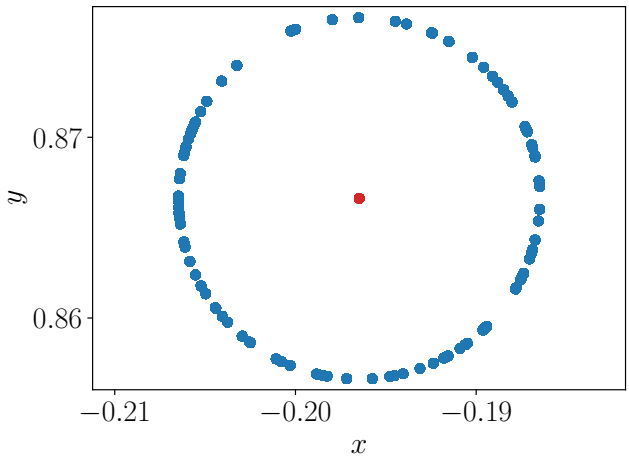


Figure 2: Disturbed initial position place on the circle $r_{\text{dist.}}(\cos(\varphi), \sin(\varphi))$, where with $r_{\text{dist.}} = 0.01$ and φ is randomly sampled in the interval $(0, 2\pi)$. The red dot indicate the true initial position.

3.3 Optimizer Choice and Hyperparameter Tuning

As outlined by [9], the Adam optimizer in general performs well in a variety of models. Therefore Adam was used throughout this project. However, to optimize the models further, an additional "autotune" of the optimizer's hyperparameters was performed using `keras_tuner`. Using `Bayesian Optimization`, the tuner was set to investigate the loss using different combinations of the learning rate $\eta \in \{0.0005, 0.0008, 0.001, 0.002, 0.005\}$ and moments $\beta_1 \in \{0.3, 0.5, 0.75, 0.99\}$ and $\beta_2 \in \{0.3, 0.5, 0.75, 0.99\}$. A maximum of 20 (in total 80 possible combinations) trials was set, with 10 epochs and two executions per trial in order to minimize the impact of rng.

3.4 Predicting with Conserved Energy

As mentioned by [1], an important aspect to consider when discussing the usage of ANNs instead of classical numerical methods is the ability to preserve a conserved quantity, in this case the energy. This was done by [1] by adding a so called projection layer, equivalent to finding new coordinates that minimized the energy loss while also being constrained by too high deviations in initial position and velocity. However, in this project, we instead trained our ANN with a new cost function according to

$$C(w) = \text{MAE} + \alpha Err(\vec{x}_j, \vec{y}_j)_i, \tag{1}$$

where MAE is the mean absolute error, α is a tunable parameter and $Err(\vec{x}_j, \vec{y}_j)_i$ is the energy loss for each trajectory i. Put differently, this cost function tries to minimize the energy difference for every trajectory in the data. The energy error function for trajectory i was given by

$$Err(\vec{x}_j, \vec{y}_j)_i = \sum_{j=0}^{999} \Big(\underbrace{E_k(\dot{\vec{x}}_j, \dot{\vec{y}}_j) + E_p(\vec{x}_j, \vec{y}_j)}_{\text{Timestep } j \text{ energy}} - \underbrace{E_p(\vec{x}_0, \vec{y}_0)}_{\text{Initial energy}} \Big)_i,$$

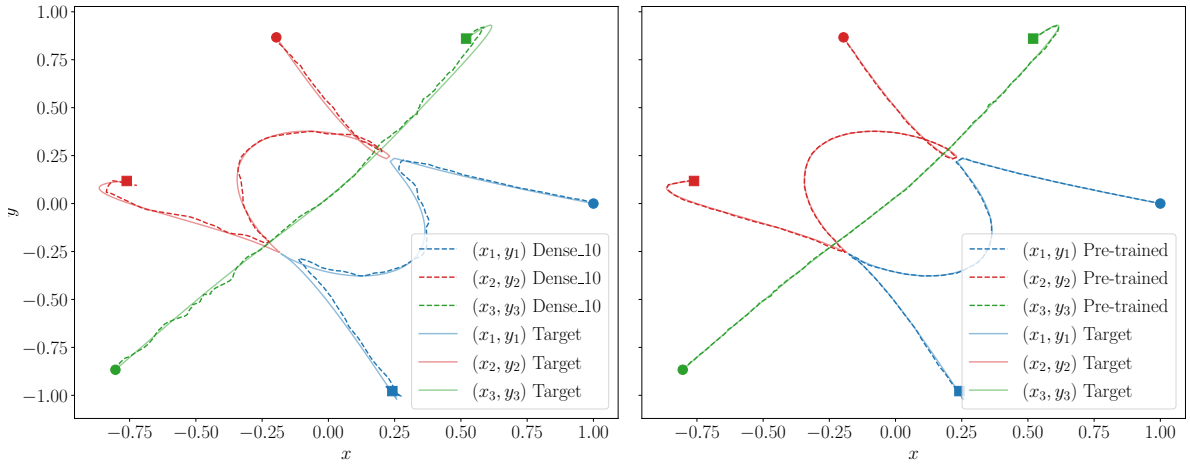
where E_k and E_p are the kinetic and potential energy and $(\vec{x}_j, \vec{y}_j) = ((x_{1j}, y_{1j}), (x_{2j}, y_{2j}), (x_{3j}, y_{3j}))$ are the particles' positions at time step j. Due to the new structure of the cost function, the ANN was trained using $\sim 68\,000$ batches of data, with one trajectory (1 000 data points) per batch. Other training conditions were kept the same. The predictions of this ANN was then used to produce a Figure similar to Figure 6 in [1], where the relative error at each time step for a certain trajectory was plotted for every model.

4 Results and Discussion

Below, we present our ANN's solutions to the Three-body problem while comparing to the pre-trained model used by [1]. We also visualize the chaotic nature of the problem by predicting with disturbed initial positions and evaluate the model when trained with a cost function that emphasizes energy conservation.

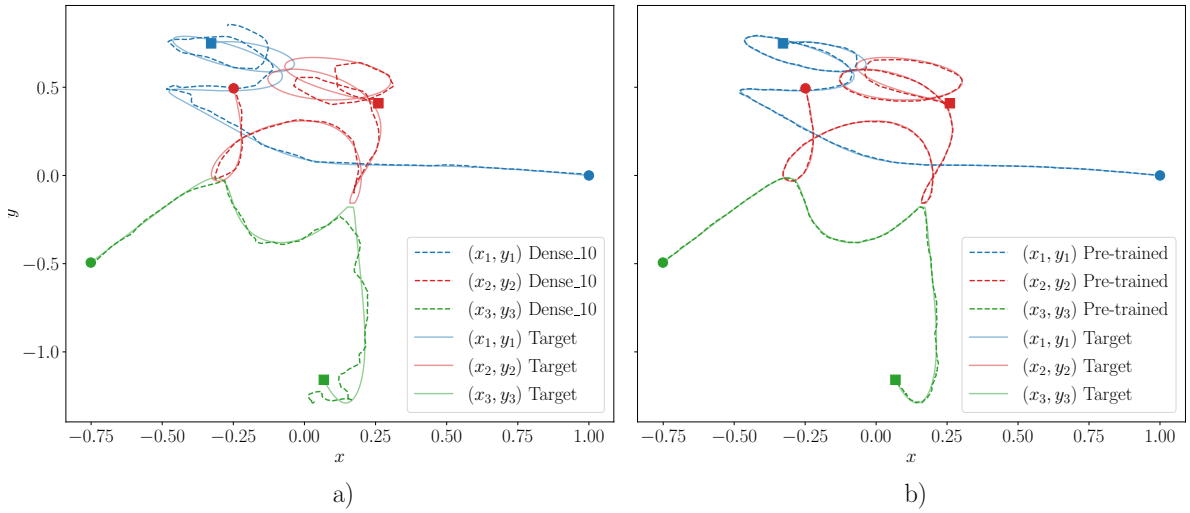
4.1 Predicted Trajectories

By studying predicted trajectories (available in Appendix A) for the models in Table 1, the trajectories predicted by model 2 were closest to trajectories from `Brutus`. In Figure 3a, trajectories from model 2, here labelled *Dense_10* are compared to the target trajectories from `Brutus`. In addition, Figure 3b contains predicted trajectories for the pre-trained model to the right. Furthermore, comparing predictions for a slightly more complex trajectory, as in Figure 4, model 2 is again performing slightly worse.



a) b)

Figure 3: (a) Predicted trajectories for model 2 in Table 1 (*Dense_10*) along with trajectories from `Brutus` (*Target*). (b) Predicted trajectories for pre-trained model along with trajectories from `Brutus` (*Target*). In both plots, the filled circles (●) indicate the initial position for each trajectory, while the filled squares (■) indicate the last position.



a) b)

Figure 4: (a) Predicted trajectories for model 2 in Table 1 (*Dense_10*) along with trajectories from `Brutus` (*Target*). (b) Predicted trajectories for pre-trained model along with trajectories from `Brutus` (*Target*). In both plots, the filled circles (●) indicate the initial position for each trajectory, while the filled squares (■) indicate the last position.

Despite having a similar architecture, 10 `Dense` layers with 128 neurons in each layer, the pre-trained model gave more accurate trajectories compared to model 2. This is most probably due to the number of epochs run during training. In [1], roughly 1 000 epochs were run, compared to 66 for model 2, which suggests that the early-stopping approach used during the training of model 2 might have been set too generously. However, if the loss at 50 epochs was compared to the loss at 150 epochs, the improvements was of order 0.001. Considering the computational time of $\sim$15 s/epoch, this improvement was judged sufficiently small to keep the early stopping during the training. In the training curve, see Figure 5, the training loss is reaching a plateau after $\sim$50 epochs. After 56 epochs, the validation loss reaches its minimum, which is not improved in the following 10 epochs and the early stopping therefore halted the training after 66 epochs.

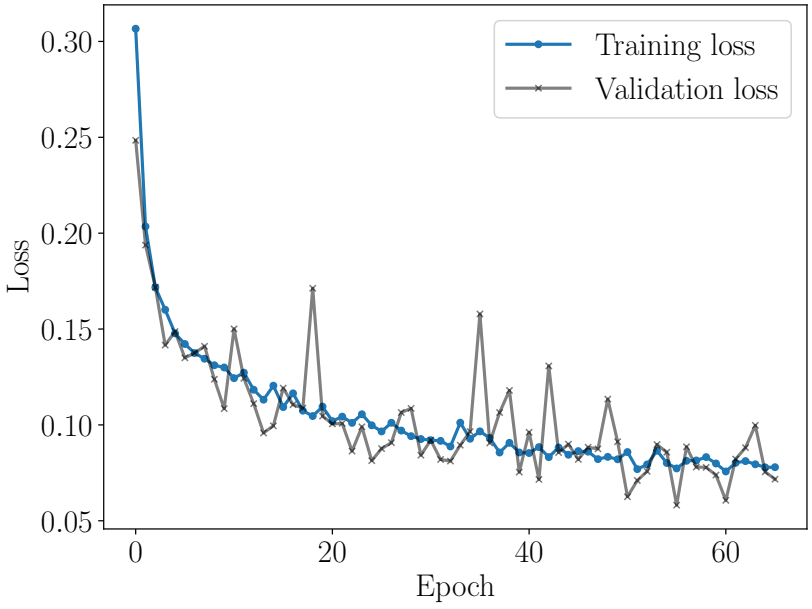


Figure 5: Validation- and training loss as a function of epochs, for the *Dense_10* model.

Lastly, recurrent neural networks such as `LSTM` or `Conv-LSTM` are often used to make predictions [10], however, in this case model 3, 4 and 5 in Table 1 performed worse than model 2. To be efficient, recurrent models in general need a sequence (a couple of data points) as input data [11]. Thus, the current problem set-up, where a complete trajectory are to be predicted from a single initial position, might not be ideal for a recurrent network. An alternative approach, more suitable for recurrent networks would be to, for instance, set the first 10 time steps as input.

4.2 Predicting with Disturbed Initial Positions

The blue dots in Figure 6 displays values at time steps $t \in \{1, 250, 500, 750, 1000\}$ for 100 trajectories with disturbed initial positions. In addition, all predicted trajectories are given as the fainter blue areas, while the true trajectory is the dashed grey line. Comparing the predicted trajectories at the end position, the pre-trained model gives predictions closer to `Brutus`' trajectory. However, at other time steps the pre-trained model and *Dense_10* gives rather similar results. When further studying all predicted trajectories, the pre-trained models seem to "better" follow the true trajectory, while *Dense_10* seems to struggle at the sharp corners.

To make a quantitive measure of chaotic motion of the point masses, [1] determined the Lyapunov exponent, which gives insight into how the initial disturbances propagate through time. In this case, the mean difference between the trajectories with disturbed initial position $T(r_{\text{dist.}})$ and the trajectory with non-disturbed initial positions, $T(x_0,y_0)$ are given in Figure 7. Note that this is the same trajectory as in Figure 6, and it is, as concluded from Figure 6, clear that the pre-trained model is slightly less sensitive to disturbances at the end position. In this case, a rough approximation of the Lyapunov exponent between the first and last time steps $(0-1000)$ gives $\lambda_{10} = 0.0083$ for *Dense_10* and $\lambda_{\text{pre}} = 0.0076$ for the pre-trained model.

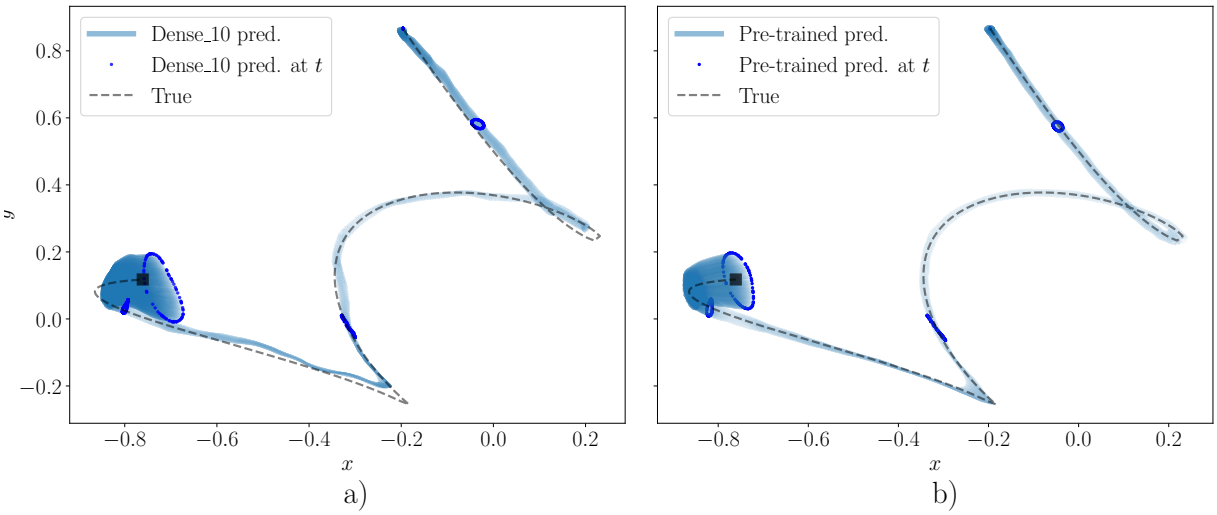


Figure 6: (a) All predicted trajectories from disturbed initial positions with marked values at $t \in \{1, 250, 500, 750, 1000\}$ for *Dense_10* along with the true trajectory given by `Brutus`. (b) All predicted trajectories from disturbed initial positions with marked values at $t \in \{1, 250, 500, 750, 1000\}$ for *Pre-trained* along with the true trajectory given by `Brutus`. The black square (■) marks the end position in the trajectory.

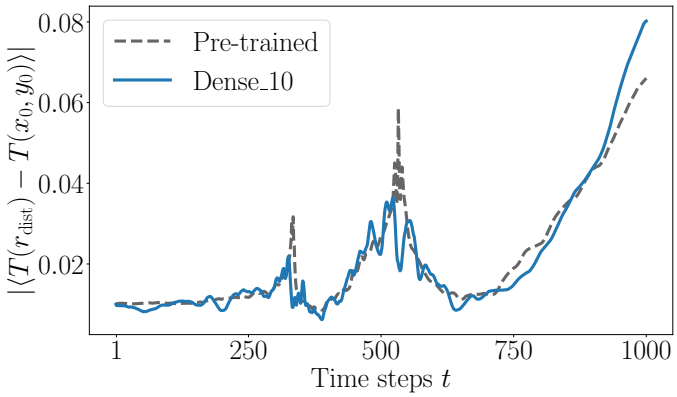


Figure 7: Absolute difference between disturbed trajectories, $T(r_{\text{dist}})$, and the non-disturbed trajectory, $T(x_0,y_0)$, for each time step t. The differences are given as a mean of 100 disturbed trajectories for both the pre-trained model and *Dense_10*.

4.3 Predicting with Conserved Energy

Figure 8 presents the energy error (a) and the relative energy error (b) for a trajectory predicted by every model as a function of the 1000 timesteps. In (b), our model *Dense_10* has been trained to conserve energy better, as described in section 3.4. For this trajectory, `Brutus` and our energy-trained ANN conserve energy significantly better than the pre-trained model and the standard *Dense_10* model. The peaks most likely follow from close encounters between the particles as the potential energy increase significantly. In this case, the energy error scaling parameter α was set to 0.01 (see Equation 1). After testing other options, this gave the best combination of minimizing the energy error term while not discarding the mean absolute error, i.e. solving for the correct trajectories, too much. In other words, the energy conservation could be emphasized more, with the downside that the predictions become worse.

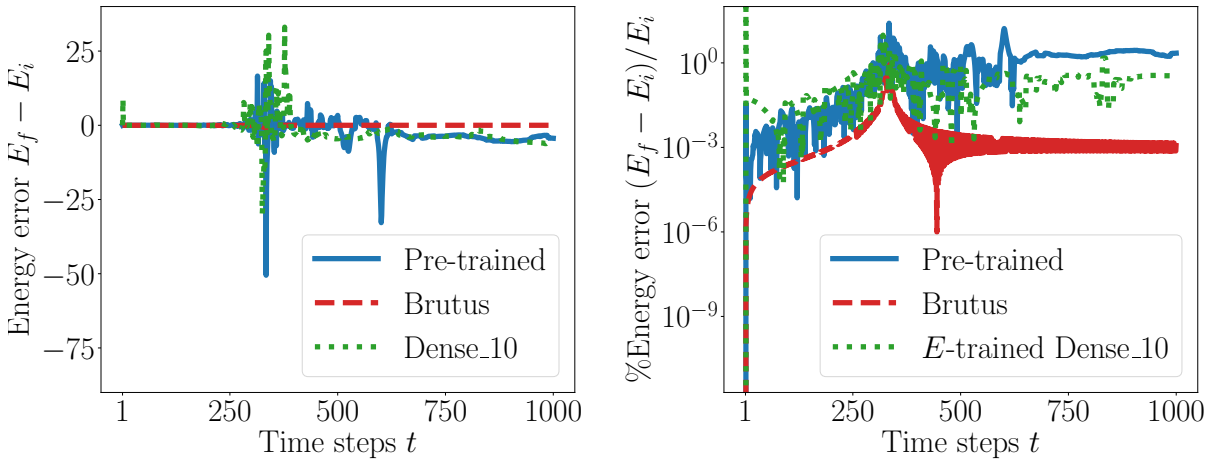


Figure 8: Energy error for all models (a) and relative energy error (b) with *Dense_10* trained to take the energy conservation in account as a function of time for a give trajectory. Note the peaks in (a) and that the energy-trained *Dense_10* has a slight advantage over the pre-trained model and the regular *Dense_10*.

Despite differences in the exact relative error, Figure 8b shows similar tendencies to Figure 6 in [1]. `Brutus` has the lowest energy error, followed by our energy-trained ANN and the pre-trained model (not trained with respect to energy).

The most likely reason for the differences in relative error is the features of the different trajectories, as we here use another example than the one showed in [1]. If a trajectory has more close encounters between particles for instance, the pre-trained model's lack of energy conservation is more prominent. However, for many other trajectories, the pre-trained model and the energy-trained ANN actually performs similarly considering the relative error, which points towards that other factors may influence this as well.

For instance, [1] uses a slightly different method with the projection layer as explained in section 3.4, which could be more effective when reducing the relative error. One other aspect that could influence the result is the scaling parameter α used in the cost function (see Equation 1). This should probably be set differently depending on the trajectory, as the size of the energy error highly depends on the particles paths, again, many close encounters significantly increases the error as explained above. By tuning α, the relative error might decrease, but this requires further simulations to confirm.

5 Conclusion

In conclusion, several neural network was designed, trained and solutions to the Three-body problem was compared to a pre-trained neural network and a high-end solver. Although the best performing model (*Dense_10*) and the pre-trained model had a similar architecture, the pre-trained model were able to predict trajectories more precisely. In addition, the pre-trained model also seems to be less sensitive to disturbances in initial positions, as it had lower Lyapunov exponent. In summary, the differences in performance between the models is most probably due to the number of epochs run during training, where we run fewer epochs due to computational time.

Furthermore, adding an energy conservation constraint to the cost function, yields more physical predictions in the sense that energy is better conserved. Although this, the trajectories without the energy conservation constraint still performed better when compared to trajectories obtained by `Brutus`. In the future, it is, thus, suggested to explore different ways of implementing the energy conservation constraint, which perhaps could lead to a better result. Lastly, the fact that such good trajectories can be obtained from a model trained on a laptop and with no prior knowledge in physics would not only have thrilled Newton, it also proves that neural networks truly are universal problem solvers.

References

[1] P. G. Breen, C. N. Foley, T. Boekholt and S. P. Zwart, "Newton versus the machine: Solving the chaotic three-body problem using deep neural networks", *Monthly Notices of the Royal Astronomical Society*, vol. 494, no. 2, pp. 2465–2470, Apr. 2020, ISSN: 1365-2966. DOI: 10.1093/mnras/staa713. [Online]. Available: http://dx.doi.org/10.1093/mnras/staa713.

[2] T. Boekholt and S. P. Zwart, *On the reliability of n-body simulations*, 2014. arXiv: 1411.6671 [astro-ph.IM]. [Online]. Available: https://arxiv.org/abs/1411.6671.

[3] Keras.io. (), [Online]. Available: https://keras.io (visited on 29/10/2024).

[4] I. Newton, *Philosophiæ Naturalis Principia Mathematica*. London: Royal Society, 1687, Revised editions in 1713 and 1726.

[5] Z. E. Musielak and B. Quarles, "The three-body problem", *Reports on Progress in Physics*, vol. 77, no. 6, p. 065 901, Jun. 2014, ISSN: 1361-6633. DOI: 10.1088/0034-4885/77/6/065901. [Online]. Available: http://dx.doi.org/10.1088/0034-4885/77/6/065901.

[6] C. C. Aggarwal, *Neural Networks and Deep Learning*, 2nd. Springer, 2023, ISBN: 978-3-030-33060-0. [Online]. Available: https://link.springer.com/book/10.1007/978-3-031-29642-0.

[7] Colab.google. (), [Online]. Available: https://colab.google (visited on 30/10/2024).

[8] Y. Bengio, *Practical recommendations for gradient-based training of deep architectures*, 2012. arXiv: 1206.5533 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1206.5533.

[9] R. M. Schmidt, F. Schneider and P. Hennig, *Descending through a crowded valley - benchmarking deep learning optimizers*, 2021. arXiv: 2007.01547 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2007.01547.

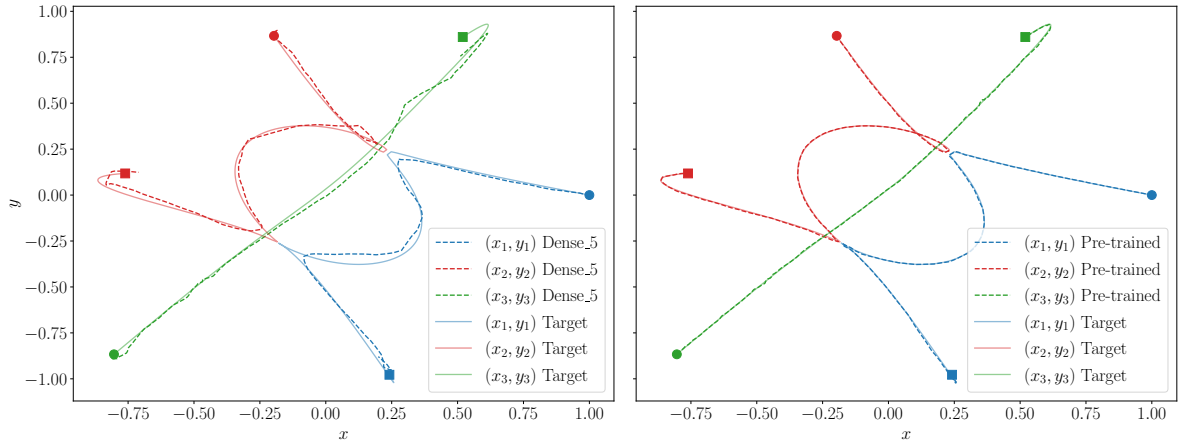
[10] X. SHI, Z. Chen, H. Wang, D.-Y. Yeung, W.-k. Wong and W.-c. WOO, "Convolutional lstm network: A machine learning approach for precipitation nowcasting", in *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama and R. Garnett, Eds., vol. 28, Curran Associates, Inc., 2015. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2015/file/07563a3fe3bbe7e3ba84431ad9d055af-Paper.pdf.

[11] A. Géron, "Hands-on machine learning with scikit-learn, keras & tensorflow", in R. Roumeliotis and N. Tache, Eds., 2nd ed. 1005 Gravenstein Highway North, Sebastopol, CA 95472.: O'Reilly Media, Inc, Sep. 2019, ch. 15, p. 497, ISBN: 9781492032649.

Appendix

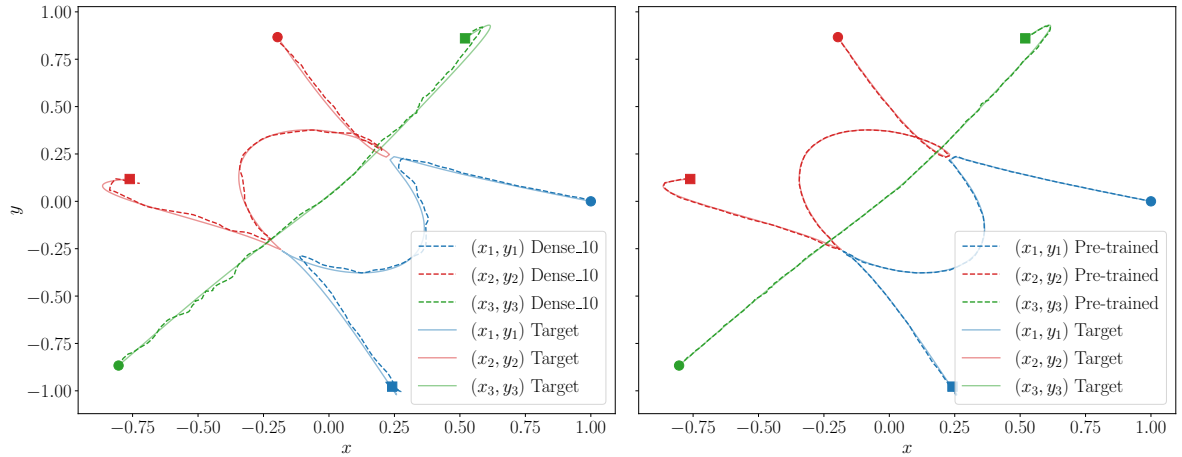
A Trajectories for alternative models

In this Appendix, trajectories for the alternative models in Table 1 are given. Here, Figure 1 corresponds to model 1 in Table 1, Figure 2 to model 2, Figure 3 to model 3, Figure 4 to model 4, and Figure 5 to model 5.



a) b)

Figure 1: (a) Predicted trajectories for model 1 in Table 1 along with trajectories from `Brutus` (*Target*). (b) Predicted trajectories for pre-trained model along with trajectories from `Brutus` (*Target*). In both plots, the filled circles (●) indicate the initial position for each trajectory, while the filled squares (■) indicate the last position.



a) b)

Figure 2: (a) Predicted trajectories for model 2 in Table 1 (*Dense_10*) along with trajectories from `Brutus` (*Target*). (b) Predicted trajectories for pre-trained model along with trajectories from `Brutus` (*Target*). In both plots, the filled circles (●) indicate the initial position for each trajectory, while the filled squares (■) indicate the last position.

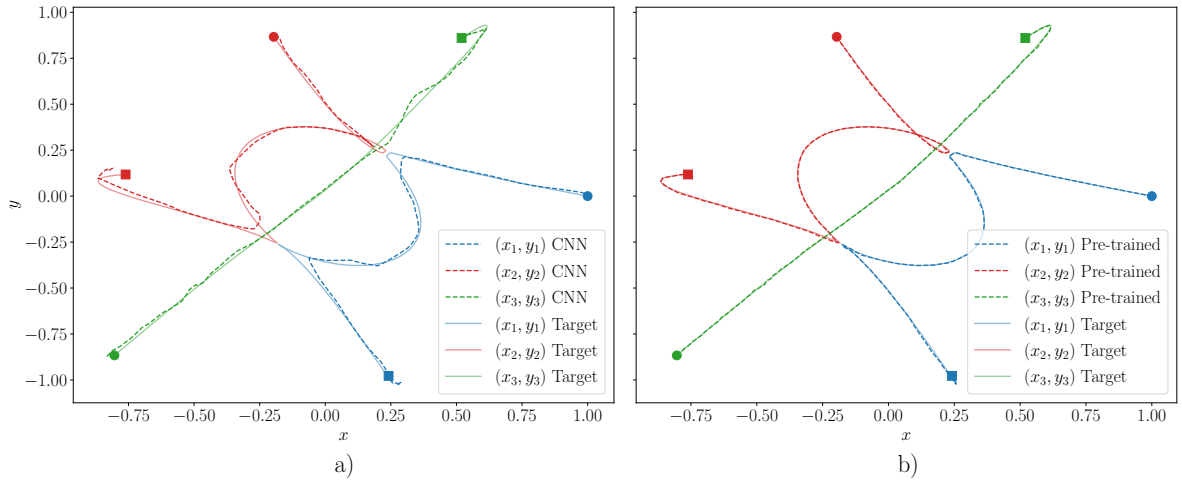


Figure 3: (a) Predicted trajectories for model 3 in Table 1 along with trajectories from `Brutus` (*Target*). (b) Predicted trajectories for pre-trained model along with trajectories from `Brutus` (*Target*). In both plots, the filled circles (●) indicate the initial position for each trajectory, while the filled squares (■) indicate the last position.

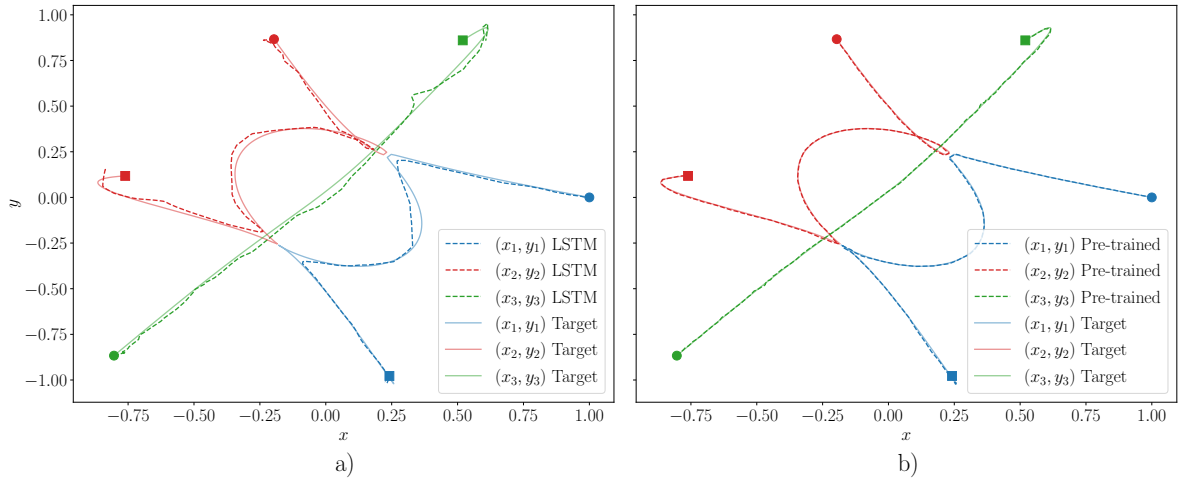


Figure 4: (a) Predicted trajectories for model 4 in Table 1 along with trajectories from `Brutus` (*Target*). (b) Predicted trajectories for pre-trained model along with trajectories from `Brutus` (*Target*). In both plots, the filled circles (●) indicate the initial position for each trajectory, while the filled squares (■) indicate the last position.

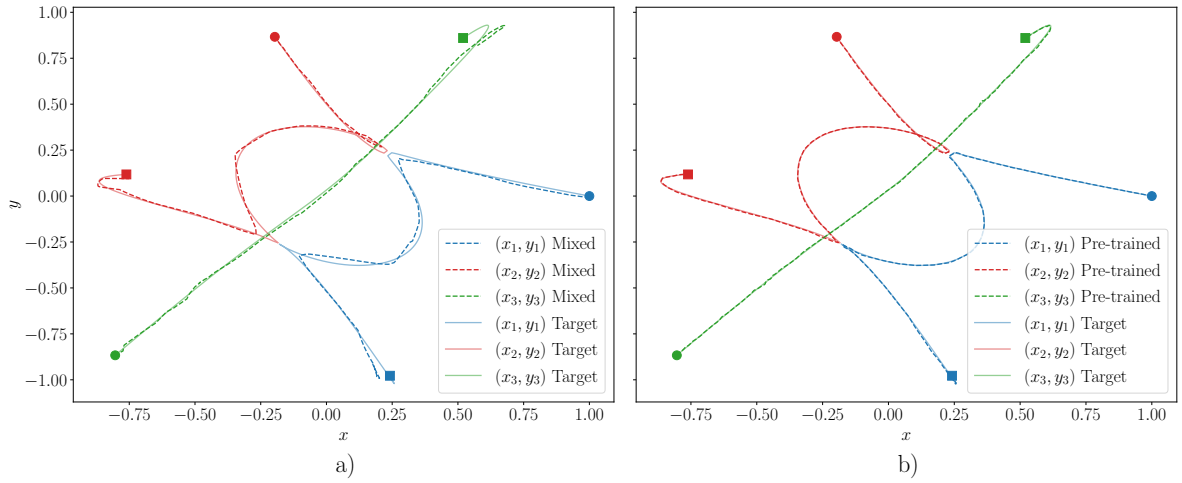


Figure 5: (a) Predicted trajectories for model 5 in Table 1 along with trajectories from `Brutus` (*Target*). (b) Predicted trajectories for pre-trained model along with trajectories from `Brutus` (*Target*). In both plots, the filled circles (●) indicate the initial position for each trajectory, while the filled squares (■) indicate the last position.