Kepler problems with physics-informed neural networks

Semester thesis by

Justin Mazenauer

At

AMAS group PSI and
D-PHYS ETH Zurich

Supervised by

Dr. Andreas Adelmann

Switzerland
June 8, 2022

Abstract

In this semester thesis I introduce the Kepler problem and physics-informed neural networks (PINNs). The main result is a python class PINN that can solve the Kepler problem for given initial values over a given time interval by neural network approximation. For direct evaluation of the accuracy a symplectic Runge-Kutta solver delivers a high-fidelity solution. The network generally gives reliable results but the optimal hyper-parameters depend on the initial values. The accuracy of the prediction is influenced by higher order interactions of the hyper-parameters which makes it difficult to start with an educated guess. On the other hand, the PINN is flexible, fast, scaleable, and generalizeable to more than two bodies.

Contents

Chapter 1

Introduction

1.1 Overview

Newton's law of universal gravitation, together with Newton's law's of motion, describe classical celestial mechanics. The trajectories of massive bodies in a gravitational field, and by extension all potentials of the same form e.g. electrostatics, have been solved for isolated systems of two bodies. In fact, the system of two bodies is more easily calculated by considering the center of mass in the origin of the reference frame and introducing a single body orbiting the origin that mathematically stands for the relative motion. This problem is equivalent to the original problem but uses only one body, thus the two body problem has been reduced to the problem of one body in a central potential. For three and more bodies the corresponding system of equations is in general unsolvable by integrals. This result is elegantly described by the theory of integrable systems and in particular Liouville's theorem [Arn88, page 274]. A more quantitative and standard description of the problem in the Hamiltonian formalism is presented in section 1.2.

An important role in Hamiltonian mechanics is taken by conserved quantities. We will see that Newtonian gravity conserves energy and angular momentum and, in the two body case, the Laplace-Runge-Lenz (LRL) vector. The following chapters introduce a machine learning based approach to solve the dynamics of a body in a central potential. Conventional numerical solvers for ordinary differential equations (ODE) are agnostic to the physics of the underlying problem. Here, underlying physics means that just by inspecting the Hamiltonian of the system we can make an inference about the solution, e.g. we know the solution must conserve the energy and so on. Impressive progress has been made in the theory of symplectic integrators, for example Lobatto methods[Jay15], which do have nice properties. Loosely speaking they conserve energy and approximately solve the initial value problem (IVP).

what is a PINN

A physics-informed neural network (PINN) is a feed-forward neural network where we code the physical constraints into the network through an appropriate loss function. By the *universal approximation theorem* there exists such a network that solves the dynamics and by training the network we try to find one that approximates the solution as accurately as possible. More details and a brief introduction to the terminology used in this thesis are given in section 2.2. The PINN is based on a completely different principle than traditional ODE solvers. Instead of successively calculating the state of the system at consecutive discrete points in time and propagating the solution from the initial value, a PINN proposes a solution function to the problem and iteratively improves the prediction until it agrees with the requirements (dynamics, initial values, conservation laws) to a satisfactory degree. The difficulty lies in quantifying how good a solution is and how to improve it. This is also explained in section 2.2. The main difference to conventional methods is the following: instead of propagating the solution along discrete points we generate a sequence of functions and accept the solution once a certain condition on the convergence or the number of iterations is fulfilled.

The main goal of this thesis is to demonstrate that a PINN is able to solve dynamical systems and get an

idea how that solution depends on the several hyper-parameters that need to be chosen by the user. To achieve this objective I develop a python module using PyTorch and perform a Sobol sensitivity analysis. It is not my goal to build a PINN that beats conventional solvers in terms of accuracy or speed but to prove the concept. This method is not restricted to the two body problem, it can be applied just as well to many bodies and also different dynamical systems or partial differential equations (PDE) by tweaking the loss function. I chose to work within the framework of Newtonian gravity because its solutions are well understood and extensively researched from a mathematical point of view yet tangible to laypersons by imagining for example planetary motion. For the interested reader there is a repository on github that reproduces the plots from this thesis and includes the source files.

adapt this method to other problems

This first chapter includes an introduction to the mathematical description of N-body problems and an introduction to neural networks and PINNs. After the introduction I will discuss the implementation of the PINN and a conventional solver which serves as a reference since it delivers high-fidelity solutions. Further I explain the problem of finding suitable hyper-parameters and quantifying the sensitivity of the PINN solution to those hyper-parameters. In the results chapter I present some solutions by the PINN for qualitatively different solutions and the results of the hyper-parameter search and the sensitivity analysis. In the subsequent discussion chapter I discuss the both the results and the advantages and disadvantages of PINNs. Finally I give a brief outlook on further possible research and summarize the findings in the conclusion. Supplementary figures and tables are found in the appendix.

1.2 Background of N-body problems

1.2.1 Two bodies and body in central potential

The special case $N = 2$ can be reduced to the planar motion of a single body in a central potential. In a suitable choice of units the Hamiltonian of the system is given by

$$\mathcal{H} = \frac{p^2}{2} - \frac{1}{q} \tag{1.1}$$

where $(\mathbf{q}, \mathbf{p}) \in \mathbb{R}^{2\times 2} \equiv \mathbb{R}^4$ are the phase space coordinates that fully describe the state of the system. Throughout this chapter bold letters represent vector quantities and normal styled letters represent scalar quantities or the magnitude of the corresponding vector. The Hamiltonian equations of motion give Newton's law of gravity

$$\dot{\mathbf{q}} = \mathbf{p}$$
$$\dot{\mathbf{p}} = -\frac{\mathbf{q}}{q^3}$$

This system has four degrees of freedom and a total of seven conserved quantities, these are

1. energy E which is given by the Hamiltonian $E = \mathcal{H}$
2. angular momentum $\mathbf{L} = \mathbf{q} \times \mathbf{p}$
3. Laplace-Runge-Lenz vector $\mathbf{A} = \mathbf{L} \times \mathbf{p} + \frac{\mathbf{q}}{q}$

We call a system with n degrees of freedom and $2n - 1$ integrals of motion *maximally super-integrable*. This system can be solved by quadrature, and, for demonstration's sake, the solution in polar coordinates is

$$t = \frac{1}{2}\int_{r(0)}^{r(t)} \frac{dr'}{\sqrt{E - \frac{L^2}{2r'^2} - \frac{1}{r'}}}$$
$$\varphi(t) = \varphi(0) + \int_0^t \frac{L^2}{r^2(t')} dt'$$

One can show that the solution is given implicitly by

$$r(\varphi) = \frac{p}{1 + e\cos(\varphi - \alpha)}$$

where the parameters are given by $p = \mathbf{L}^2$ and $e = \sqrt{1 + 2EL^2}$ in suitable units and α depends on the orientation of the coordinate system. However, there is in general no elementary solution that allows to calculate $\mathbf{q}(t)$ without resorting to numerical integration.

1.2.2 General case

The general case of N bodies can be described by a $6N$ dimensional state vector $x = (\mathbf{q}_1, \mathbf{p}_1, \ldots \mathbf{q}_N, \mathbf{p}_N)$ and the Hamiltonian

$$\mathcal{H} = \sum_{i=1}^{N} \frac{p_i^2}{2m_i} - G \sum_{1 \leqslant i < j \leqslant N} \frac{m_i m_j}{\|\mathbf{q_i} - \mathbf{q_j}\|}$$

The equations of motion become

$$\begin{aligned} \dot{\mathbf{q}}_i &= \frac{\mathbf{p}_i}{2m_i} \\ \dot{\mathbf{p}}_i &= G \sum_{j \neq i} m_i m_j \frac{\mathbf{q_j} - \mathbf{q}_i}{\|\mathbf{q}_j - \mathbf{q}_i\|^3} \end{aligned}$$

There are a total of ten integrals of motion: the energy, the center of mass, the total momentum, and the total angular momentum. In contrast to the two body problem, states with $E < 0$ are not necessarily stable as some bodies individually can have positive energy and allow unbounded solutions. Due to Liouville's theorem on integrable systems the N-body problem is in general not solvable by quadrature for $N \geqslant 3$ and one has to resort to numerical simulations[Sch07, page 150]. This motivates the research for fast and reliable ODE solvers.

1.3 Neural networks

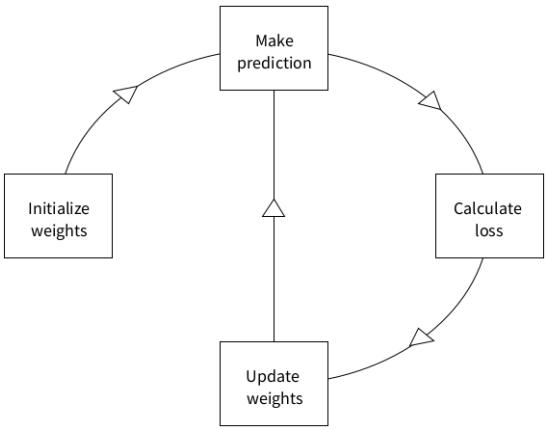


Figure 1.1: The process of initializing and training the neural network.

In the context of this thesis a neural network is a feed-forward deep neural network (DNN). A neural network is a function $\Phi : \mathbb{R}^n \to \mathbb{R}^m$. If $z_0 \in \mathbb{R}^n$ is the input to the network then we define recursively

$$z_{i+1} = \sigma(A_i z_i + b_i),\ 0 \leqslant i < L \tag{1.2}$$

and the output $z_{L+1} = A_L z_L + b_L \in \mathbb{R}^m$. Importantly, A_i are called the weights and are matrices, b_i are called the biases and are vectors, and σ is an activation function acting on each component individually. We combine them to get the parameters that live in a high-dimensional parameter space $\theta := (A_i, b_i)_{i=0}^L \in \Theta$. Due to the *universal approximation theorems* for any continuous function F there exists a neural network Φ that approximates F to arbitrary degree in the L^∞ (or equivalently Ł^2) sense, given enough parameters. Proofs are e.g. in [KL19] for the bounded width case or [Cyb89] for the bounded depth case. The process of training the network is necessary to find suitable parameters in Θ. In figure 1.1 the training process corresponds to the closed cycle that gets repeated until a terminating condition is fulfilled, that is either a maximum number of cycles or a minimum tolerance in the loss. A cycle is also called an epoch.
As discussed in equation 1.2, the output is calculated as $z_{L+1} = A_L z_L + b_L$, where z_L is a vector of functions of the input t. Therefore the output is a linear superposition of the basis functions $z_L^i(t)$ where i ranges from 1 to the number of neurons in the last hidden layer. All output functions are built from this set of basis functions. The deeper the network the more flexible are the individual basis functions and the wider the network the larger is the set of functions at available to train.

The core part of the training process lies in the two boxes *calculate loss* and *update weights* in figure 1.1. The loss is a non-negative real functional and a measure of predictive accuracy of the network. For the correct solution, when Φ perfectly matches F, the loss should be zero and else positive. In supervised learning the mean squared error is used most commonly. To explain supervised learning consider the Banach space $V := L^2(X)$ for some set $X \subseteq \mathbf{R}^n$. Then $\theta \to \Phi_\theta \in V$ parameterizes a $\dim(\Theta)$-dimensional subset of V and the goal is to find the set of minimizers $\theta^* = \arg\min_\theta \|F - \Phi_\theta\|_2$ that give the point on the subset closest to F with regard to the L^2 metric. The metric is essentially approximated by sampling F and Φ on a finite set and Monte Carlo integration, the minimizing is done via gradient descent or quasi-Newton methods in Θ.

1.4 Physics-informed neural networks

PINN = unsupervised learning

A physics-informed neural network (PINN) can be used to solve an ODE without providing training data, it is thus an unsupervised model. In this case, F is the solution to an initial value problem and the true form of F is unknown. If F is a very complex function of the inputs, which is often the case in non-linear dynamical systems, then Φ can be used as a surrogate model that matches F to a limited extent but is considerably faster to find and to evaluate. Instead of using a loss function like $\|\Phi - F\|_2$, where some values of F must be known beforehand, we incorporate the initial values and the dynamics, hence the attribute "physics-informed". As in the supervised case the problem is solved by means of non-convex optimization, e.g. gradient descent methods. For an IVP of the form

$$\dot{F}(t) = f(F(t))$$
$$F(0) = F_0$$

where $\dot{\Phi}$ is the time derivative, one can implement a loss function of the form

$$\mathcal{L} = \left\| \dot{\Phi} - f(\Phi) \right\|^2 + \|\Phi(0) - F_0\|^2$$

which can now be used to train the network. Optionally one can include more terms to make the network fulfill all sorts of soft constraints. A popular term is $\lambda \|\theta\|_p^2$ with the p-norm, called L_p regularization. The most common types of weight regularization use $p = 1$ (Lasso) and $p = 2$ (Tikhonov regularization). Remember that Φ is parameterized by a point θ in Θ and a locally optimal θ' can be found via means of gradient descent. This construction is not limited to dynamical systems and is successfully used to solve partial differential equations by replacing the time derivative by any differential operator and including a term for boundary values. A good introduction with examples is Raissi et al [RPK19].

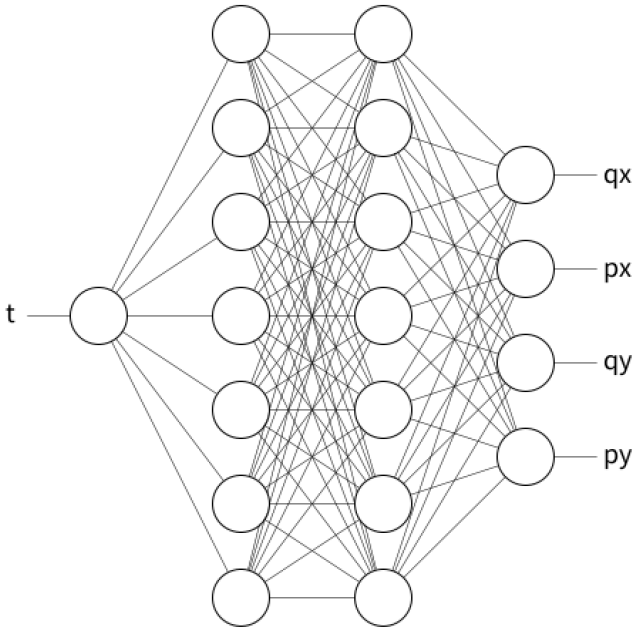


Figure 1.2: Architecture of the neural network. The input is time and the output are the four corresponding phase space coordinates. This illustration shows a network with two hidden layers that are seven neurons wide each.

Chapter 2

Implementation

2.1 Explicit solver

For the problem at hand a symplectic integrator of high order is a good choice for a conventional solver. Firstly, when integrating over multiple cycles of a periodic orbit we want to make sure the integrator respects the physical conservation laws and does not exhibit numerical dissipation. Secondly, we want accurate results for large time steps. I decided in favor of the symmetric Ruth algorithm (SRA), which is a symplectic and forth order partitioned Runge-Kutta method. It is semi-implicit, i.e. it has the form

method used →

$$p_{n+1} = p_n + hf(q_n)$$
$$q_{n+1} = q_n + hg(p_{n+1})$$

with some functions f, g. Therefore it does not require any systems of equations to be solved and is orders of magnitude faster in execution than implicit methods of comparable order. It is also straight-forward to implement. A single step of the algorithm is shown in figure 2.1 The performance of the SRA against some other fast ODE solvers is shown in figure 2.2. A derivation of the SRA with a detailed treatment of symplectic Runge-Kutta method is given in [HNW08, pages 326-330]. It is noteworthy that a purely explicit Runge-Kutta scheme like the widely used RK4 method cannot be symplectic.

```
def symmetricRuthStep(self,dt):
    b = [7/48,3/8,-1/48,-1/48,3/8,7/48]
    bhat = [1/3,-1/3,1,-1/3,1/3,0]
    n = len(b)
    for i in range(n):
        self.p += b[i]*dt*self.pdot(self.q)
        self.q += bhat[i]*dt*self.qdot(self.p)
```

Figure 2.1: Implementation of one step of the symmetric Ruth algorithm. One step takes a state with phase space coordinates q, p and advances them by a time step dt.

2.2 Neural network

how the NN works

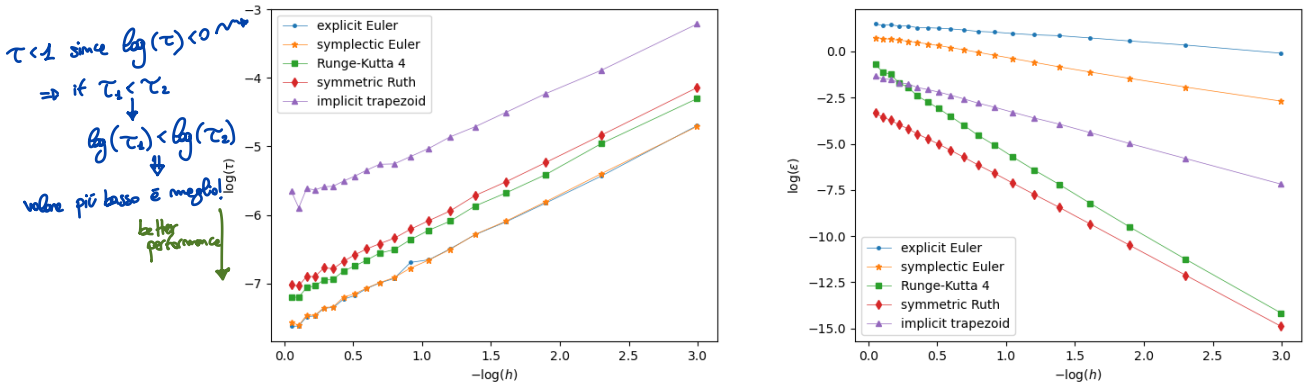
The state of a body in a central potential is uniquely characterized by the four phase space coordinates $(\mathbf{q}, \mathbf{p})$. We seek a neural network that takes one real input, the time t, and returns four real outputs (q_x, p_x, q_y, p_y). Besides the evaluation of the network at a given input the PINN needs two other important properties: a loss function to determine how good the network is with its prediction, and an optimizer to update the parameters of the PINN based on the loss. Recall the equations of motion are equivalent to

$$\dot{\mathbf{q}} = \frac{\mathbf{p}}{2m}, \quad q^3\dot{\mathbf{p}} = -\mathbf{q}Gmm$$

Loss fct → 4 terms:

The loss function consists of four separate terms, each of which is a mean squared error. The first term MSE_1 regards the equations of motion and avoids division by q. The second term MSE_2 makes sure the initial conditions are satisfied to make the solution unique. The third term MSE_3 makes sure the conservation laws are respected and adds the physical phase-space constraints to the numerical solution, for example restricts the solution to an energy surface. Dropping this term will see the network switch

MSE_1: equation of motion
MSE_2: initial conditions
MSE_3: conservation law
MSE_4: L_2 regularisation



(a) Runtime τ vs stepsize h

(b) L^2 error ε vs stepsize h

Figure 2.2: (a) Comparison of the logarithmic runtime of different integration methods for different time steps h, all in appropriate units. The benchmarking problem is one revolution of a two-body system in a perfect circular orbit and the error is in the L^2 sense. (b) Comparison of the logarithmic error of different integration methods for different time step sizes h. The slope of a fitted line captures the order of convergence. According to those measurements explicit Euler has order 0.5, symplectic Euler has order 1.2, RK4 has order 4.6, SRA has order 3.9, and implicit trapezoid has 2.0 but SRA has the lowest error for large step sizes.

between solutions of the ODE at random times at a minimal increase of loss. Lastly, I include some regularization to the parameters. I choose L^2 regularization to prevent arbitrarily growing parameters at the price of introducing a non-physical bias to the solution. Finally, the loss is a weighted sum of the four individual losses. All those constraints are soft constraints, which is to say there will be a trade-off between the separate loss terms and none of them may be exactly fulfilled. Concretely,

$$
\begin{aligned}
MSE_1 &= \frac{1}{N}\sum_{i=1}^{N} \|\dot{\mathbf{q}}(t_i) - \mathbf{p}(t_i)\|_2^2 + \|q^3\dot{\mathbf{p}}(t_i) + \mathbf{q}(t_i)\|_2^2 \\
MSE_2 &= \|\mathbf{q}(0) - \mathbf{q}_0\|_2^2 + \|\mathbf{p}(0) - \mathbf{p}_0\|_2^2 \\
MSE_3 &= \frac{1}{N}\sum_{i=1}^{N} \left(\mathcal{H}(t_i) - E_0\right)^2 + \|\mathbf{L}(t_i) - \mathbf{L}_0\|_2^2 + \|\mathbf{A}(t_i) - \mathbf{A}_0\|_2^2 \\
MSE_4 &= \|\theta\|_2^2 \\
\mathcal{L} &= \frac{w_1 MSE_1 + w_2 MSE_2 + w_3 MSE_3 + w_4 MSE_4}{w_1 + w_2 + w_3 + w_4}
\end{aligned}
$$

loss fct (with weighted terms)

We see that the loss is invariant under the scaling $(w_1, w_2, w_3, w_4) \mapsto (\lambda w_1, \lambda w_2, \lambda w_3, \lambda w_4)$ so one of $w_1, \dots w_4$ can be set to 1 without loss of generality. This would reduce the number of the hyper-parameters by one but is not done as optimization is not the main goal. The way this loss function is implemented is as follows: evaluate the network on an array of time samples $\vec{t} = t_0, t_1, \dots t_N$. The result is a $(N+1) \times 4$ matrix. Let q_x, p_x, q_y, p_y be the four $N+1$ dimensional column vectors. For each of them also calculate the time derivative at $\vec{t}$ with PyTorch's built-in automatic differentiation method `torch.autograd.grad`. The mean squared error is calculated with the `.square()` and `.mean()` methods for PyTorch tensors on the residuals, e.g.

Implementation of the NN

$$
\begin{aligned}
MSE_1 =& (\dot{q}_x - p_x).square().mean() + (\dot{q}_y - p_y).square().mean() + \\
& ((q_x.square() + q_y.square()).pow(1.5) * \dot{p}_x + q_x).square().mean() + \\
& ((q_x.square() + q_y.square()).pow(1.5) * \dot{p}_y + q_y).square().mean()
\end{aligned}
$$

For the initial residual MSE_2 we take the values at t_0 and compare them to the initial values. The quantities are $\mathcal{H}, L, A_x, A_y$ and are also $N+1$ dimensional vectors. They are calculated according to their definition in 1.2.1. Note that $\mathbf{L}$ has only one component L and $\mathbf{A}$ is split into its two components A_x and A_y. Lastly, MSE_4 is calculated by squaring and summing all parameters of the network. We see that MSE_2 does not necessarily need to be based on the value of $t = 0$, one could also fix the final position at the final time t_N and simulate the motion backwards in time or somewhere in between. For the optimization procedure I use PyTorch's built-in *L-BFGS* optimizer. This optimizer is based on the idea of Newton's root-finding method. To find a minimum of $\mathcal{L}$ we seek a zero of $\nabla\mathcal{L}$. With the Hessian matrix $\mathbf{H}_\mathcal{L}$, or an approximation thereof, we define iteratively

$$\theta_{n+1} = \theta_n - (\mathbf{H}_\mathcal{L}(\theta_n))^{-1} \nabla\mathcal{L}(\theta_n)$$

This method is second order, as opposed to gradient descent methods which are first order[KW19, pages 87-95]. PyTorch calculates the gradient of the loss with its `loss.backward()` call. The Hessian is the gradient of the gradient.

2.3 Determining accuracy

To determine the accuracy of a prediction we define a validation loss. We use a high-fidelity solution from the SRA solver with small stepsize and compare the trajectories in phase space. In formula, the validation loss is

$$\mathcal{L}_{valid} = \frac{1}{2N} \sum_{i=1}^{N} \|\mathbf{q}_{exact}(t_i) - \mathbf{q}_{pred}(t_i)\|_2^2 + \|\mathbf{p}_{exact}(t_i) - \mathbf{p}_{pred}(t_i)\|_2^2$$

We necessarily need to compare different predictions and networks by their validation loss and not their training loss. To see why, imagine a network where $(w_1, w_2, w_3, w_4) = (a, 0, 0, b)$. Then the prediction $\mathbf{q}, \mathbf{p} \equiv \mathbf{0}$ where all weights and biases of the network are zero has zero training loss, but it is obviously not the correct solution. This validation loss will be used to establish a conclusion on the feasibility of PINNs as ODE solvers.

To reiterate, the training loss is used to track the training progress of the network but a low training loss does not guarantee an accurate prediction. The true accuracy of the prediction is assessed with the validation loss, which is not available for training.

2.4 Hyper-parameter search

By design the PINN has nine hyper-parameters for a given initial value problem. Those are the stepsize or resolution, the width and depth of the network, the learning rate of the optimizer, the weights $w_1, ...w_4$ in the loss function, and the seed that initializes the network. Additionally, a maximum tolerance for the error can be given to decide when the training diverges and a minimal tolerance to decide when the training has converged. Or one can abort training after a fixed number of training iterations.

For a configuration of parameters that converge we try random seed values and pick the one which leads to the best convergence. This seed will be used for all subsequent networks. As a uniform grid is infeasible in high dimensions the other eight hyper-parameters are scanned with an eight dimensional Sobol' sequence.

2.5 Sobol' sensitivity analysis

This section is based on Sobol's original paper[Sob01]. One method of finding the sensitivity of a complicated function with respect to the inputs is Sobol sensitivity analysis, also known as variance-based sensitivity analysis. As opposed to local sensitivity analysis based on the gradient of the loss at a point in hyper-parameter space the variance based analysis is global. Assume the input parameters are random variables each with their respective distribution and variance. Then the function is a random variable

and the variance is influenced by the variances of the parameters and possible interaction terms. Let $\mathcal{L} = f(p_1, \dots p_8)$ be the validation loss as a function of the hyper-parameters $p_1, \dots p_8$ and decompose

$$f = f_0 + \sum_{s=1}^{8} \sum_{i_1 < \dots < i_s} f_{i_1,\dots i_s}(p_{i_1}, \dots p_{i_s}) \tag{2.1}$$

with the condition

$$\int f_{i_1,\dots i_s}(p_{i_1}, \dots p_{i_s}) dp_k = 0 \qquad \forall k = i_1, \dots i_s \tag{2.2}$$

Now the function $f_{i_1,\dots i_s}$ captures the simultaneous influence from the parameters $p_{i_1,\dots i_s}$ to the function f. By integrating equation 2.1 and using equation 2.2 we can express $f_{i_1,\dots i_s}$ as integrals of f, for example

$$\begin{aligned}
f_0 &= \int f dp_1 \dots dp_8 = \mathbb{E}[f] \\
f_i(p_i) &= \int f dp_1 \dots dp_{\bar{i}} \dots dp_8 - f_0 = \mathbb{E}[f|p_i] - f_0 \\
f_{i,j}(p_i, p_j) &= \int f dp_{1,\dots \bar{i},\bar{j},\dots 8} - f_i - f_j - f_0 = \mathbb{E}[f|p_i, p_j] - f_i(p_i) - f_j(p_j) - f_0
\end{aligned}$$

where $\bar{i}$ means the integration along the ith dimension is omitted. The pattern continues for the higher order functions. We now let

$$\begin{aligned}
D &= \int f^2 dp_{1,\dots 8} - f_0^2 \\
D_{i_1,\dots i_s} &= \int f^2_{i_1,\dots i_s} dp_{i_1,\dots i_s}
\end{aligned}$$

be the conditional variances and call $S_{i_1,\dots i_s} = \frac{D_{i_1,\dots i_s}}{D}$ the Sobol indices of order s. All 2^8 indices are non-negative and sum to 1. They quantify the contribution of the interaction of the parameters $p_{i_1}, \dots p_{i_s}$ to the total variance of $\mathcal{L}$.

The total effect of a parameter is given by the sum of all indices which include the parameter, i.e.

$$S_i^T = 1 - \sum_{s=1}^{7} \sum_{\substack{j_1 < \dots < j_s \\ j \neq i}} S_{j_1,\dots j_s}$$

The extreme cases are when $S_i^T = 0$ and $\mathcal{L}$ does not depend on p_i or when $S_i^T = 1$ and $\mathcal{L}$ depends only on p_i. The problem is, we do not know the exact form of f and thus cannot calculate the integrals. Thus we resort to numerical integration by calculating f at a set of sample points in the eight-dimensional parameter space. A sensible choice of method are quasi-Monte Carlo methods, such as Sobol sequences.

More details and explanations are given in [Sob01]. To sum up this paragraph, each parameter p_i has a first order index S_i which describes the contribution it gives to the total variance. Each pair $p_i < p_j$ has a second order index S_{ij} that describes the contribution from the interaction, up until eighth-order $S_{1,\dots 8}$. Then there is a total contribution S_i^T which describes the total contribution by parameter p_i in all indices. The computation is done with the python package *SALib*[HU17], specifically its member `SALib.analyze.sobol.analyze` and the samples are generated with `SALib.samples.saltelli.sample`.

Chapter 3

Results

3.1 Examples

The main result of this thesis is a python script that builds a neural network to solves the Kepler problem for two bodies by solving the reduced one-body problem in a $\frac{1}{q}$ potential. To initialize the build and training the network one requires the initial state, the final time of the simulation, the nine hyper-parameters described in section 2.4 and finally a condition to stop the training, e.g. the number of epochs. The module contains a number of functions to show the result: one function to plot the phase space coordinates against time during the training process, one function to show the trajectory in position space in real time, a pure training function without plotting, and a function to show the details of a completed training process.

There are three qualitatively different classes of solutions to the dynamical system 1.1: bounded solutions (elliptical orbits), unbounded solutions (hyperbolic and parabolic orbits), and singular solutions that reach the $\mathbf{q} = \mathbf{0}$ state in finite time. I present some results for those possible situations. The hyper-parameters used are the ones with the best result from table 3.1 and the number of training epochs for all examples is $10,000$ without early stopping.

Some results are shown in the following figures. The top left image shows the training loss in blue and validation loss in yellow over the training epochs with two separate axes on a log scale. The top right image shows the final prediction of the phase space coordinates against time in different colors as a solid line and the high-fidelity solution as a dashed line of the same color. The bottom left image shows the trajectory of the body in the xy-plane, the blue line is the final prediction by the neural network and the yellow dashed line is the high-fidelity solution. Red marks the origin in the plane. Finally, the bottom right image shows the absolute deviation of the supposedly conserved quantities (solid) and the ODE residuals (dashed) against time in the final prediction.The closer the lines are to zero the better conserved are the quantities and the better satisfied is the ODE. The shape of the lines is not important but they give us an impression at which points along the trajectory the network has trouble approximating the solution and where it succeeds. In conclusion, all of the results for short-term integration are usable.

Elliptical orbit

The initial conditions are $\mathbf{q}_0 = (1, 0), \mathbf{p}_0 = (0, 1.2)$ and the final time is $T_{max} = 8\pi$. The optimizer converges after about $4,000$ epochs and the prediction behaves as expected. The main source of error comes from the region where the coordinate functions have higher curvature.

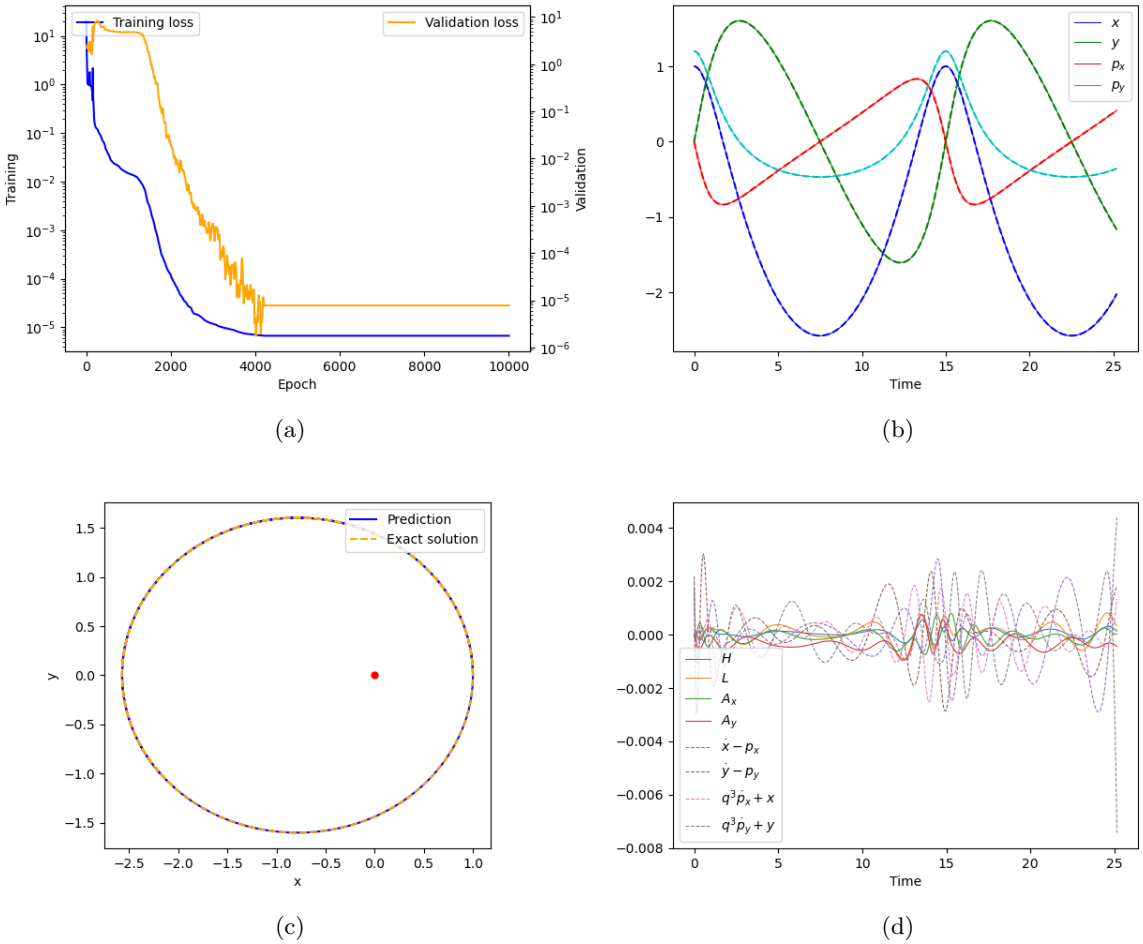


Figure 3.1: (a) shows the history of training (blue) and validation (yellow) loss on a log scale. (b) shows the final prediction (solid) and the exact solution (dashed) of the phase space coordinates as functions of time. (c) shows the predicted (blue, solid) and exact (yellow, dashed) trajectory in position space. (d) shows the absolute errors of the conserved quantities and the ODE residual over time along the trajectory.

Hyperbolic orbit

The initial conditions are $\mathbf{q}_0 = (-2, -1), \mathbf{p}_0 = (1, 1)$ and the final time is $T_{max} = \pi$. The optimizer converges after about only $3,000$ epochs with minimal validation loss. From figure (d) we see that the approximation had the most difficulties at the nearest point to the origin for the conservation laws. The ODE residual tends to be large at the beginning and the end of the simulation.

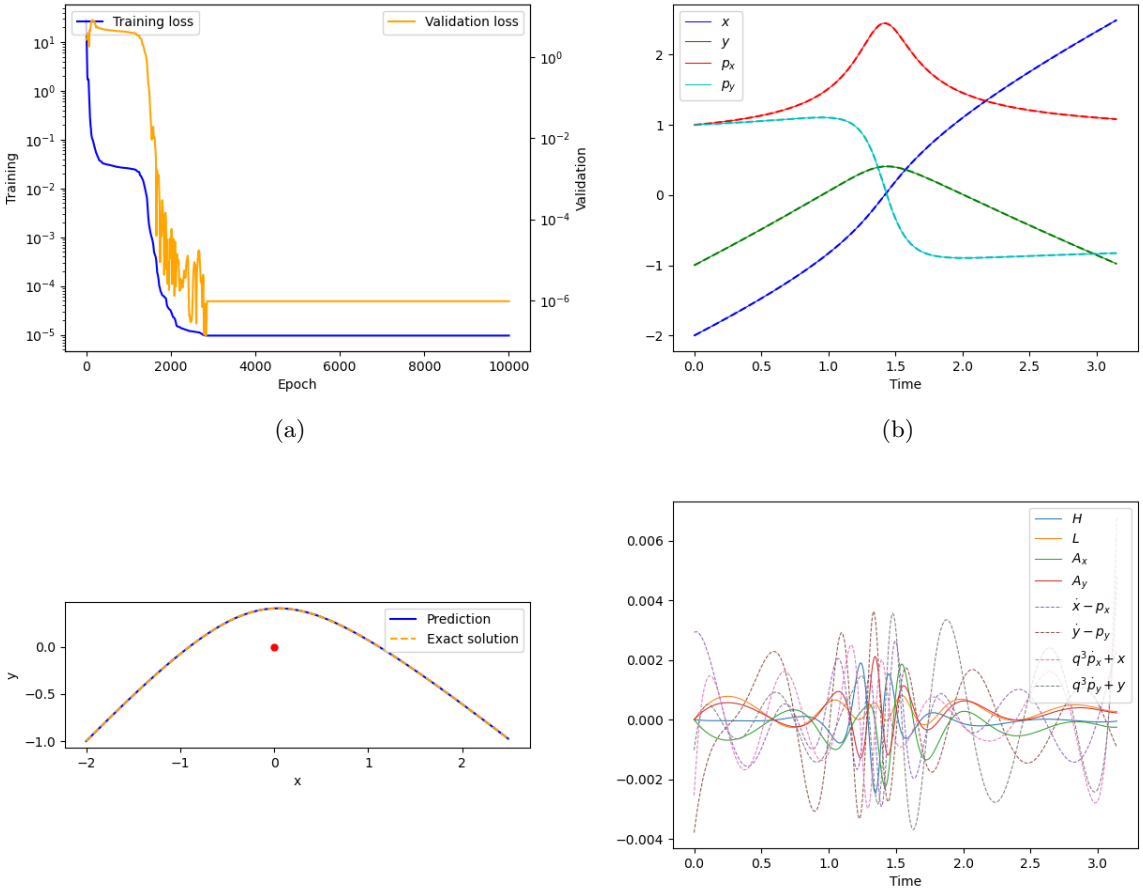
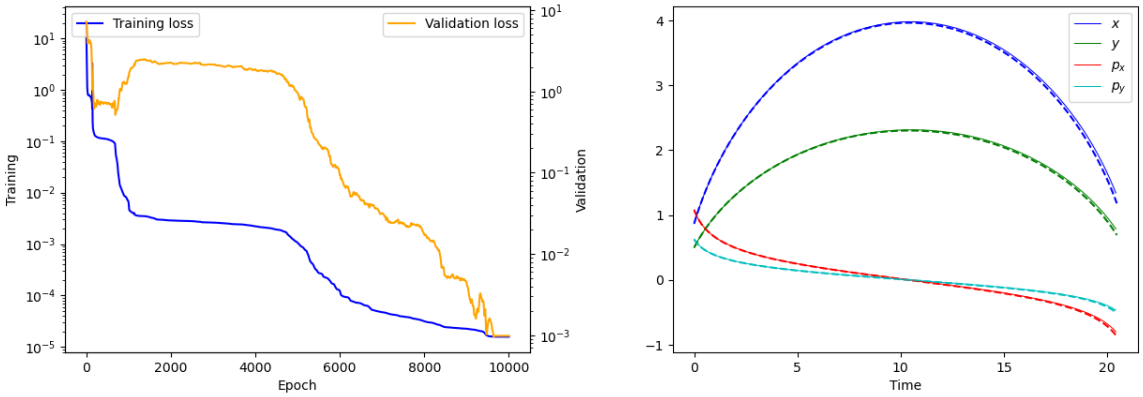


Figure 3.2: (a) shows the history of training (blue) and validation (yellow) loss on a log scale. (b) shows the final prediction (solid) and the exact solution (dashed) of the phase space coordinates as functions of time. (c) shows the predicted (blue, solid) and exact (yellow, dashed) trajectory in position space. (d) shows the absolute errors of the conserved quantities and the ODE residual over time along the trajectory.

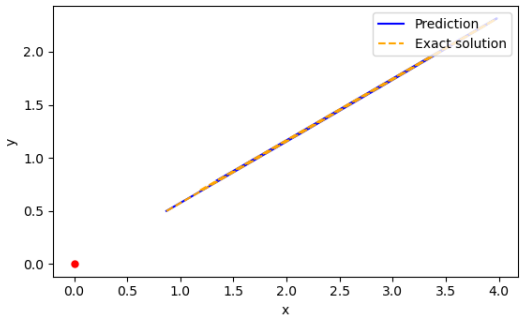
Singular orbit

The initial conditions $\mathbf{q}_0 = (0.87, 0.5), \mathbf{p}_0 = (1.08, 0.625)$ and the final time is $T_{max} = 6.5\pi$. These initial conditions lead to a body that would reach the origin in finite time and accelerate infinitely in the process. Note that we do not let the body reach the singularity. In this example the optimizer didn't converge to a satisfactory minimum in 10.000 epochs and it seems it stopped making progress just before the end of training. We also see that almost no progress was made between epoch 500 and 5,000, after that the network prediction continued improving. The ODE residual is considerably larger at the end of the simulation towards the singularity.

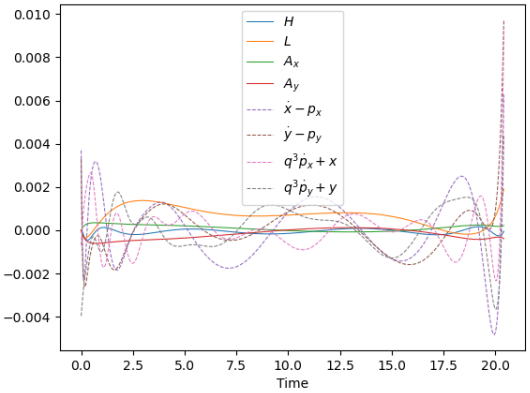


(a)

(b)



(c)



(d)

Figure 3.3: (a) shows the history of training (blue) and validation (yellow) loss on a log scale. (b) shows the final prediction (solid) and the exact solution (dashed) of the phase space coordinates as functions of time. (c) shows the predicted (blue, solid) and exact (yellow, dashed) trajectory in position space. (d) shows the absolute errors of the conserved quantities and the ODE residual over time along the trajectory.

Extrapolation

The initial conditions are $\mathbf{q}_0 = (1, 0), \mathbf{p}_0 = (0, 1)$ correspond to a perfectly circular orbit of period 2π. The network was trained for two full periods and then extrapolated another two periods. Figure 3.4 clearly shows that the PINN is not reliable for extrapolation of periodic functions. From equation 1.2 we see that asymptotically, the prediction will resemble the shape of the activation function for $|t| \to \infty$, in our case the prediction will be asymptotically constant, which is physically non-nonsensical as it does not comply with the governing laws of physics. This is notion is formalized in [ZHU20, theorem 2] and will be briefly replicated.

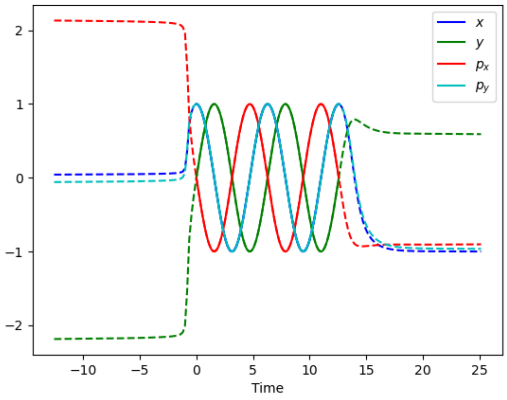


Figure 3.4: Extrapolation of circular orbit. Solid lines indicate the trained interval and dashed lines represent the extrapolated data. Evidently the neural network does not generalize well for periodic orbits.

Theorem 1. *Let $\Phi : \mathbb{R}^n \to \mathbb{R}^m$ be a feed forward neural network of fixed depth $L \geqslant 1$ and activation function $\sigma = \tanh$. Then for all non-zero $u \in \mathbb{R}^n$ there exists $v \in \mathbb{R}^m$ such that*

$$\lim_{z \to \infty} \|\Phi(zu) - v\|_2 = 0.$$

Proof. Consider the state of the first hidden layer $z_1 = \sigma(A_0 zu + b_0)$. For $\lim_{z\to\infty}$ the argument of the activation function approaches a vector where all entries approach $\pm\infty$, therefore z_1 is a vector with entries in $\{-1, 1\}$. All subsequent layers, including the output layer, receive a constant vector as input and will yield a constant vector as output. □

Ziyin, Hartweig, and Ueda propose the activation function $x \mapsto x + \sin^2 x$ on the output layer to introduce periodicity[ZHU20] but this is not further examined in this work.

3.2 Hyper-parameter search

The seed which gave the lowest training loss is $2,745$. The remaining eight parameters are scanned with an eight dimensional Sobol sequence and $2,048$ sample points. Figure B.1 shows the sample points and their pairwise distribution as well as a histogram of each value. The stepsize is exponentially distributed while all other parameters are uniformly distributed. Table 3.1 shows the ten parameter configurations with the lowest validation loss. From the limited samples it seems that large networks converge to a satisfactory prediction with fewer training epochs. Although smaller networks get good results occasionally, it seems to depend on the particular choice of problem.

timesteps	depth	width	lr	w_1	w_2	w_3	$10^3 \times w_4$	$10^7 \times$ loss
514	10	43	0.23	10.27	0.54	10.90	0.45	1.01
48	7	31	0.64	14.01	0.95	7.80	2.47	1.38
255	9	27	0.57	13.30	0.40	19.47	2.77	1.52
223	10	34	0.38	18.33	0.79	18.53	2.62	1.56
709	10	41	0.64	11.12	0.92	19.75	9.47	2.24
302	7	45	0.38	10.75	0.70	19.47	2.11	2.27
457	10	47	0.50	3.55	0.90	15.98	3.48	3.51
437	5	32	0.71	15.09	0.62	16.59	3.17	3.51
265	10	48	0.51	8.32	0.75	14.77	2.44	3.54
483	6	26	0.51	6.86	0.25	11.77	0.63	3.64

Table 3.1: The ten best results sorted by their validation loss.

3.3 Sensitivity analysis

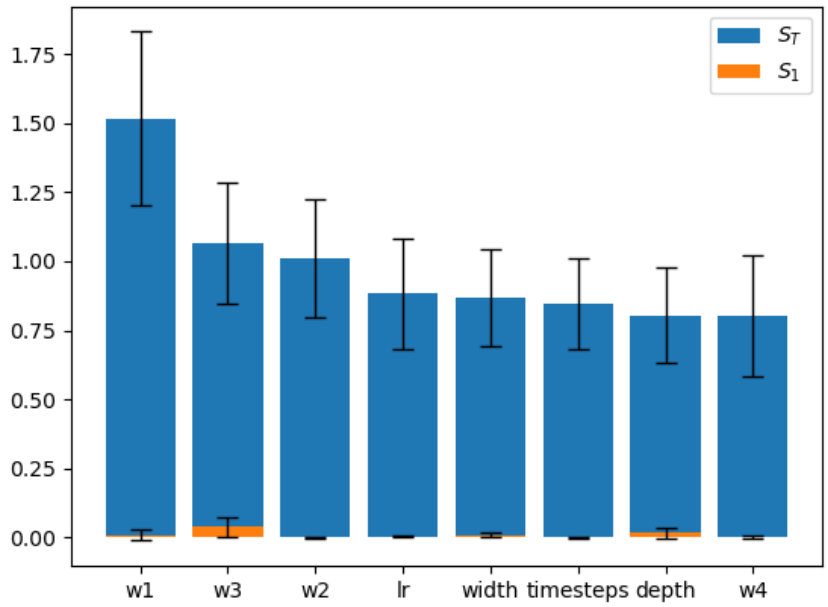
Around the optimal parameter point from table 3.1 we perform a Sobol sensitivity analysis. We assume for each of the parameters a uniform distribution with the lower end being 80% of the optimal value and the upper end 120% of the optimal value. We pick $73,728 = 2^{12} * (2 * 8 + 2)$ Sobol points according to their distribution and run the sensitivity analysis. The results of the first order analysis are shown in figure 3.5a with errorbars indicating the confidence interval and can also be found in table A.1. The second order results are shown as a heatmap in figure 3.5b and as a table in A.2.

Despite the relatively large number of samples the estimates from the sensitivity analysis are unreliable. Firstly, the length of the 95% confidence intervals for the total contributions are half the value of the estimates. The total contribution from the parameter w_1 is estimated larger than one, which is mathematically absurd. The other total contributions are plausible, their values all being close to one indicates that all of the parameters influence the accuracy of the prediction and that most variance stems from higher-order terms which include many parameters. In the same figure we see the first order contributions. All of them are estimated at least three orders of magnitude lower than the corresponding total contribution, when accounting for the huge relative uncertainties the first-order contributions are still negligible. The second order contributions are shown by their estimates in figure 3.5b but without uncertainties. Again the uncertainties are larger than the value of the estimate. However, it is clear that even the second-order contributions are negligible compared to the total contributions. Therefore we know that the higher-order interactions are the dominant source to the variance in the loss.

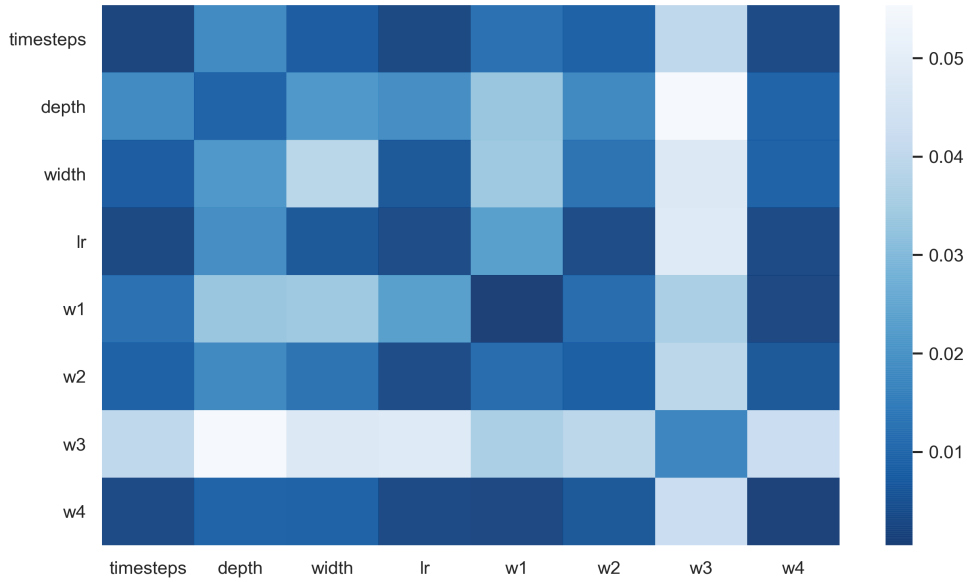
From the quasi-Monte Carlo samples used to perform the sensitivity analysis we can check some statistics on the 10% of samples with the lowest loss and the whole set of samples and see that there is no systematic trend or evident property that distinguishes the good points from the bad ones. Luckily, most samples give a low loss. In figure 3.6 we see a histogram of the best 75% and all of them have a loss between 14 millionths and 200 millionths with quartile limits at $36.8, 48.7, 70.6$ millionths, indicated by orange vertical lines. Some statistics are shown in table 3.2.

Data set	Mean	Variance	Skewness
Large data (75%)	$5.98 * 10^{-5}$	$1.17 * 10^{-9}$	1.73
Restricted data (10%)	$2.72 * 10^{-5}$	$1.08 * 10^{-11}$	-0.788

Table 3.2: Moments of the training loss from the large (75%) and the restricted (10%) data sets from the sensitivity analysis.



(a) Total and first order contributions to the variance with 95% confidence intervals.



(b) Second order contributions to the variance.

Figure 3.5: Results of the sensitivity analysis. Figure a shows the total contribution to the variance from all hyper-parameters in blue and the first order contributions in orange. Both are provided with errorbars corresponding to the 95% confidence interval. A table of the exact values is provided in table A.1. Figure b is a heatmap of the second order contributions. Dark shades of blue correspond to low contributions. The exact values and the confidence intervals are found in table A.2.

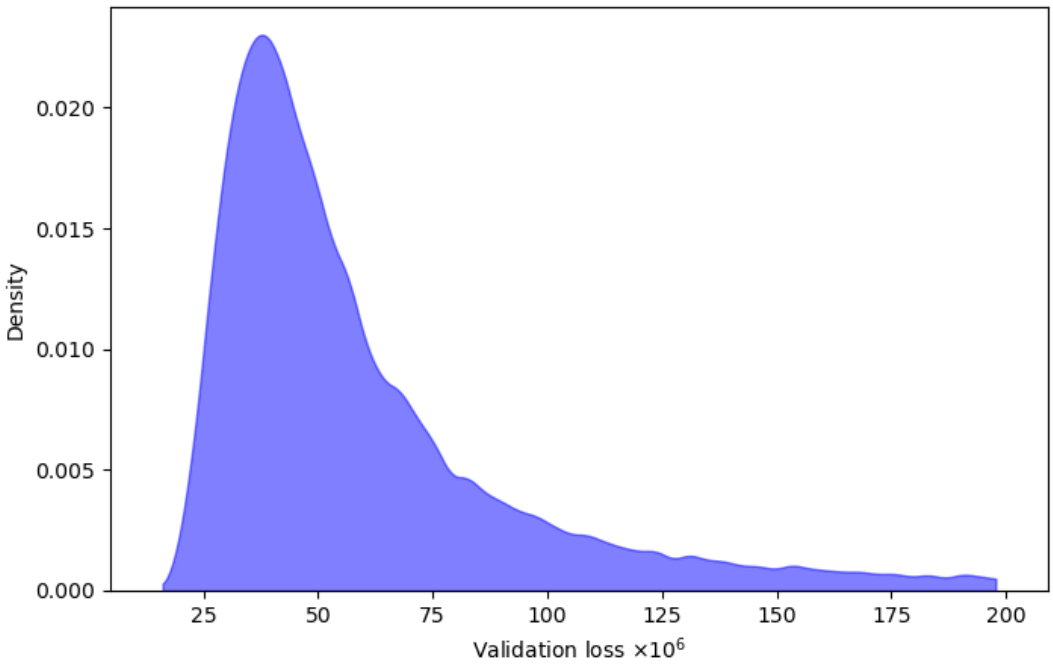


Figure 3.6: Density of the lowest 75% validation losses, which includes 54.244 samples. The density is based on a histogram with 50 bins and quadratic interpolation.

Chapter 4

Discussion

4.1 PINN

The PINN manages to produce sufficiently accurate predictions for a wide range of hyper parameters and initial conditions. For periodic motion the PINN is bad at extrapolating. This is expected since the network of the neural network is inherently non-periodic. Asymptotically, the prediction will resemble the activation function tanh and thus approach a constant value which is physically nonsensical. The PINN only works for bounded time periods. One possible solution is the choice of activation function with periodic components on the output layer.

A different approach of using periodic basis functions may be prove more useful for that case of periodic orbits. For example introduce a model where

$$\Phi_k(t) = \sum_{l=0}^{N} a_{kl} \cos(\omega_{kl} t) + b_{kl} \sin(\omega_{kl} t)$$

which is of course inspired by Fourier series. The different coordinates, indexed by k, may require different sets of angular frequencies and amplitudes. The parameters $(a_{kl}, b_{kl}, \omega_{kl})_{k=1\ldots4, l=0\ldots N} \in \mathbb{R}^{3\times4\times N}$ can also be optimized by the same gradient descent methods as feed-forward neural networks. I expect such a model to be slower in evaluation than a feed-forward neural network because of the sine and cosine computation but it may reach small errors with fewer parameters and training epochs and generalize to extend time periods. So-called Fourier neural networks are presented and compared with tanh neural networks in [NM21]. In conclusion for the general case of all possible Kepler orbits the feed-forward PINN is flexible enough to give satisfactory interpolating results and trains fast due to the quick computations and advanced backpropagation and automatic differentiation implementation of machine learning libraries such as PyTorch or TensorFlow.

4.2 Sensitivity

Right away it should be noted that the confidence intervals in the sensitivity analysis are relatively large and comparable in magnitude to the estimated value. Therefore the analysis is not very conclusive despite the large number of samples when it comes to details. Luckily the results are good enough to gain qualitative insights. We see that the largest part of the total variances S_i^T in the neural network model does not come from first and second order contributions. For example, the weight parameter w_1 has a total contribution of $S_{w1}^T = 1.517 \pm 0.628$ but the first order contribution is only a mere 0.008 ± 0.040, which attests a vanishingly small significance. All total contributions are around unity while the first and second order contributions are, with one exception and accounting for the estimation uncertainties, less than 0.1. Therefore the bulk of the error in the validation loss comes from complicated higher-order interactions. This makes it very difficult to find stable hyper-parameters. If a parameter contributed

almost only to first order then one could choose an optimal value independent of all other parameters and optimize in subspaces of Θ. The findings demonstrate that all parameters must be accounted for simultaneously. Figure 3.6 is reassuring considering the fact that most hyper-parameter combinations yield similarly small losses. It has been stated that 74% of the samples yield a validation loss under $2 * 10^{-4}$ with a median loss of $6 * 10^{-5}$.

Chapter 5

Conclusion

5.1 Summary

In this thesis I introduced physics-informed neural networks and present the results it gives for the particular case the Kepler problem. To be exact, it solves the reduced two body problem of one body in a central potential. I also explain the process of finding a good set of hyper-parameters and perform a sensitivity analysis to try to explain how the choice of hyper-parameters influence the accuracy of the network prediction. The network emulates the solution in terms of the phase space trajectory as a function of time.

While the network is able to make predictions for all $t \in \mathbb{R}$ the training is only done over a finite interval, usually $t \in [0, T]$, and the physical constraints are only enforced in this interval. We see that a network with this architecture is not good at extrapolating periodic motion beyond the training interval. In fact, asymptotically the network prediction approaches a constant, which is incompatible with physics. It needs to be said though that this network is flexible enough to approximate all qualitatively different solutions of the Kepler problem within the training range. We also saw that the accuracy of the prediction depends on the choice of hyper-parameters. Although the differences were Following are some thoughts on the pros and cons of PINNs for solving dynamical systems based on this thesis and compared to conventional ODE solvers as well as my suggestions for further investigations into this topic.

5.2 Reflection and outlook

1. Instead of using canonical coordinates $\mathbf{q}, \mathbf{p}$ it is possible to work with position variables only. The governing Newtonian equation of motion can be coded into the loss function by using automatic differentiation twice on the position. For the cost of working with a second order differential equation the output dimension of the neural network can be cut in half. This could lead to better results as fewer functions need to be trained while a second time derivative should not be problematic to calculate.

2. The coordinate functions can be simulated by either one large and flexible neural network or, alternatively, by a number of neural networks with a common loss function. I suspect by using more networks each one needs less flexibility and thus needs fewer parameters. This could reduce training time and improve accuracy. Other than that, a multi-body PINN can be constructed the same way with some corrections to the loss function. To be precise, for M bodies we can replace

$$MSE_1 = \frac{1}{NM} \sum_{i=1}^{N} \sum_{n=1}^{M} \left\| \dot{\mathbf{q}}^n(t_i) - \frac{\mathbf{p}^n(t_i)}{2m_i} \right\|^2 + \left\| \dot{\mathbf{p}}^n(t_i) - \sum_{k \neq n} G m_n m_k \frac{\mathbf{q}^k(t_i) - \mathbf{q}^n(t_i)}{\|\mathbf{q}^k(t_i) - \mathbf{q}^n(t_i)\|^3} \right\|^2$$

$$MSE_2 = \sum_{n=1}^{M} \|\mathbf{q}^n(0) - \mathbf{q}_0^n\|^2 + \|\mathbf{p}^n(0) - \mathbf{p}_0^n\|^2$$

which enforce the equations of motion:

$$\dot{\mathbf{q}}_i = \frac{\mathbf{p}_i}{2m_i}, \dot{\mathbf{p}}_i = \sum_{j \neq i} G m_i m_j \frac{\mathbf{q_j} - \mathbf{q}_i}{\|\mathbf{q}_j - \mathbf{q}_i\|^3}$$

For small M, e.g. the scale of the solar system, this should also be parallelizable.

3. Conventional solvers need the values of the previous state to calculate the next one. By design they work sequentially. In a PINN the loss function sums all time steps at the same time with the order being irrelevant. Here this part can be computed in parallel. For long-term integration or high temporal resolution this may be an advantage in favor of PINNs over conventional solvers. A PINN needs to go through training epochs sequentially but for example in the case of 10^8 time steps and 10^5 training epochs the PINN should be faster. Additionally, PyTorch has a GPU implementation which could be leveraged to speed the training process up. It did not give noticeable boost for me but my code was not written regards to optimization. But I think if the computational power is available a PINN should outperform a conventional solver in certain cases especially when multiple sets of hyper-parameter configurations are run simultaneously.

4. Another advantage of PINNs is that they can be pretrained. In general, if the network is already close to the solution then it will not diverge away and will converge in fewer training epochs than by starting from random initialization. If a model is pretrained and a parameter, e.g. the mass of a body or an initial condition, is perturbed slightly then the model does not need to be trained from scratch like in the case of a conventional solver.

5. I consider the main weakness of PINNs the sensitivity to the many hyper-parameters that have to be chosen beforehand. Because the loss landscape is highly non-convex the convergence to a local minimum is rare and the solution can at times be utterly useless and needs to be checked for physical plausibility by the user. When computational capacity is available one could train a number of networks with different hyper-parameter configurations and discard all but the one with the lowest training error. Also, it is complex loss landscape makes it difficult to find mathematical convergence results.

6. If the problem is suspected to have periodic solutions it may be worthwhile to choose activation functions with periodic components or use a Fourier neural network altogether. This would give better generalization results, provided all of the solutions are periodic.

Appendix A

Sensitivity analysis

The following tables show the Sobol' indices used in the sensitivity analysis. Table A.1 contains the first order Sobol' indices and the total contributions of the hyper-parameters to the validation loss for the samples as described in section 3.3. Table A.2 shows the pair interactions from the second order Sobol' indices. Table **??** shows the mean, standard deviation, min, and max for the total set of samples and a restricted set of samples of the 10% with the lowest validation loss.

A.1 Total contributions and first order contributions

	S_i^T	95% CI	S_i	95% CI
w1	1.517	0.628	0.008	0.040
w3	1.065	0.435	0.039	0.071
w2	1.010	0.429	0.001	0.007
lr	0.882	0.403	0.003	0.007
width	0.869	0.347	0.007	0.017
timesteps	0.846	0.333	-0.001	0.005
depth	0.804	0.345	0.016	0.035
w4	0.800	0.437	0.001	0.013

Table A.1: Total and first order contribution to the variance.

A.2 Second order contributions

	S_{ij}	95% CI
('timesteps', 'depth')	-0.017	0.036
('timesteps', 'width')	0.007	0.025
('timesteps', 'lr')	0.002	0.008
('timesteps', 'w1')	-0.011	0.046
('timesteps', 'w2')	0.008	0.015
('timesteps', 'w3')	-0.040	0.071
('timesteps', 'w4')	-0.003	0.015
('depth', 'width')	-0.020	0.038
('depth', 'lr')	-0.018	0.037
('depth', 'w1')	-0.033	0.051
('depth', 'w2')	0.017	0.085
('depth', 'w3')	-0.055	0.087
('depth', 'w4')	-0.009	0.036
('width', 'lr')	-0.006	0.019
('width', 'w1')	0.034	0.123
('width', 'w2')	-0.012	0.018
('width', 'w3')	-0.048	0.075
('width', 'w4')	-0.008	0.021
('lr', 'w1')	-0.022	0.041
('lr', 'w2')	-0.003	0.013
('lr', 'w3')	-0.048	0.072
('lr', 'w4')	0.003	0.026
('w1', 'w2')	0.011	0.049
('w1', 'w3')	-0.036	0.089
('w1', 'w4')	0.002	0.051
('w2', 'w3')	-0.039	0.066
('w2', 'w4')	-0.006	0.014
('w3', 'w4')	-0.042	0.071

Table A.2: Second order contributions.

A.3 Statistics on Sobol sample points

	timesteps	depth	width	lr	w_1	w_2	w_3	$10^3 * w_4$
mean	514.006	10.000	43.000	0.230	10.270	0.540	10.900	0.450
std	59.352	1.155	4.965	0.027	1.186	0.062	1.259	0.052
min	411.219	8.000	34.401	0.184	8.216	0.432	8.720	0.360
max	616.794	12.000	51.599	0.276	12.323	0.648	13.079	0.540

	timesteps	depth	width	lr	w_1	w_2	w_3	$10^3 * w_4$
mean	513.709	9.905	43.132	0.229	10.221	0.540	10.942	0.450
std	59.362	1.138	4.844	0.027	1.188	0.062	1.256	0.052
min	411.219	8.000	34.401	0.184	8.216	0.432	8.720	0.360
max	616.794	12.000	51.599	0.276	12.323	0.648	13.079	0.540

Table A.3: Statistics on the whole data set of $73,728$ Sobol samples (top) and the best 10% (bottom). Both tables show more or less the same statistics, indicating there is no clear trend that distinguishes the better performing samples from the rest.

Appendix B

Distributions

Figure B.1 shows the pairwise distributions and histograms of the 2048 points that were used to find a good set of hyper-parameters. The best results are shown in table 3.1. These are not the samples that were used for the sensitivity analysis.

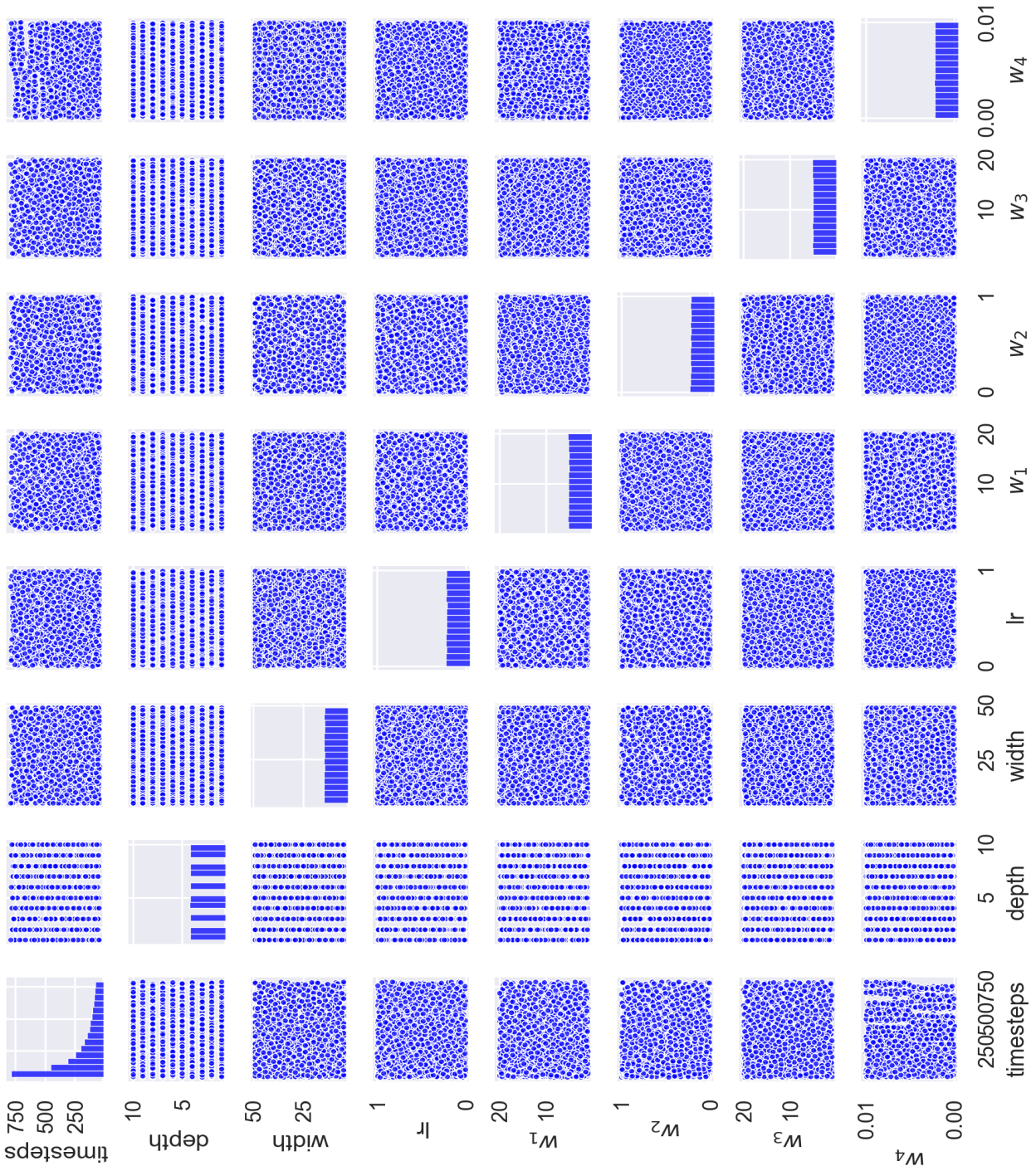


Figure B.1: A pairplot to show the distribution of the 2,048 Sobol samples in eight dimensional hyper-parameter space.

Bibliography

[Arn88] Vladimir I. Arnol'd. *Mathematische Methoden der klassischen Mechanik*. Birkhäuser, Basel, 1988.

[Cyb89] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[HNW08] Ernst Hairer, Syvert P. Norsett, and Gerhard Wanner. *Solving Ordinary Differential Equations 1*. Springer-Verlag, Berlin Heidelberg, 2008.

[HU17] Jon Herman and Will Usher. SALib: An open-source python library for sensitivity analysis. *The Journal of Open Source Software*, 2(9), 2017.

[Jay15] Laurent O. Jay. Lobatto methods. *Encyclopedia of Applied and Computational Mathematics*, pages 817–826, 2015.

[KL19] Patrick Kidger and Terry Lyons. Universal approximation with deep narrow networks. 2019.

[KW19] Mykel J. Kochenderfer and Tim A. Wheeler. *Algorithms for Optimization*. MIT Press, Cambridge, Massachusetts, 2019.

[NM21] Marieme Ngom and Oana Marin. Fourier neural networks as function approximators and differential equation solvers. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 14(6):647–661, 2021.

[RPK19] Maziar Raissi, Paris Perdikaris, and George E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[Sch07] Florian Scheck. *Theoretische Physik 1*. Springer-Verlag, Berlin Heidelberg New York, 2007.

[Sob01] Il'ya M. Sobol'. Global sensitivity indices for nonlinear mathematical models and their monte carlo estimates. *Mathematics and Computers in Simulation*, 55:271–280, 2001.

[ZHU20] Liu Ziyin, Tilman Hartwig, and Masahito Ueda. Neural networks fail to learn periodic functions and how to fix it. *Advances in Neural Information Processing Systems*, 33, 2020.