SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
DEPARTMENT OF COMPUTER SCIENCE

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Solving the three-body problem with neural networks

Andreas Merrath

SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
DEPARTMENT OF COMPUTER SCIENCE

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Solving the three-body problem with neural networks

Lösung des Dreikörperproblems mit neuronalen Netzwerken

Author:	Andreas Merrath
Supervisor:	Prof. Dr. Felix Dietrich
Submission Date:	15.04.2024

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15.04.2024 Andreas Merrath

Acknowledgments

I want to express my gratitude to G. Gonçalves Ferrari, T. Boekholt, and S. F. Portegies Zwart for their work on the n-body simulator. Without their contributions to computational astrophysics, this thesis would not be possible.

I am particularly thankful to Simon Portegies Zwart, as well as Wil Schilders, Chinmay Datar, and Felix Dietrich, for the great discussions we had during our meeting.

My thanks also go to Jack Wisdom and David M. Hernandez for their implementation of the two-body problem, without which the n-body simulator would not be complete.

I appreciate the opportunity to build upon their work and am thankful for their contributions.

Abstract

This thesis presents a neural network that solves the three-body problem numerically. The three-body problem, a fundamental challenge in classical mechanics and astronomy, involves predicting the dynamic behavior of three celestial bodies under mutual gravitational influences. This thesis builds upon the n-body solver of G. Gonçalves Ferrari, T. Boekholt, and S. F. Portegies Zwart and introduces a novel methodology that conceptualizes solvers for the two-body problem as neurons within a neural network. Combining these solvers, the network predicts the complex gravitational interactions among celestial bodies. Unlike conventional neural networks, which often act as "black boxes", this approach maintains interpretability by integrating physical calculations directly into the network's architecture. This enables the learning of arbitrary calculational components of the underlying physical formulas, including the masses of celestial bodies. The thesis is structured to guide the reader from foundational concepts to the specific TensorFlow implementation of the n-body simulator, transforming the simulator written in Python into a neural network. To show the potential of this approach, learning algorithms for this unique model are developed, and the masses of celestial objects are accurately approximated using synthetic data, marking the beginning of further investigation, potentially leading to an improvement in the estimation of the planet's masses in our solar system.

Contents

1. Introduction

The three-body problem, a cornerstone of classical mechanics and astronomy, examines the motion of three celestial bodies under mutual gravitational attraction. It encompasses the challenge of predicting the trajectories of multiple bodies interacting through gravity. While solutions for specific cases, such as the two-body problem, are solved, the three-body problem remains analytically unsolvable. Figures 1.1 and 1.2 show the complexity of three-body interactions compared to two bodies.

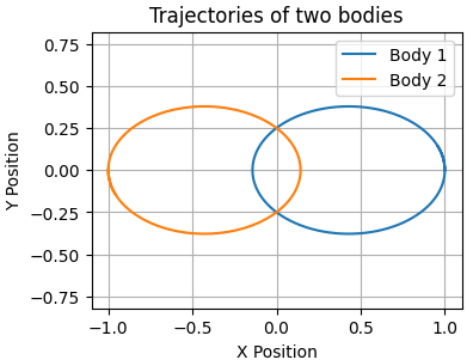

Figure 1.1.: Trajectories of two bodies.

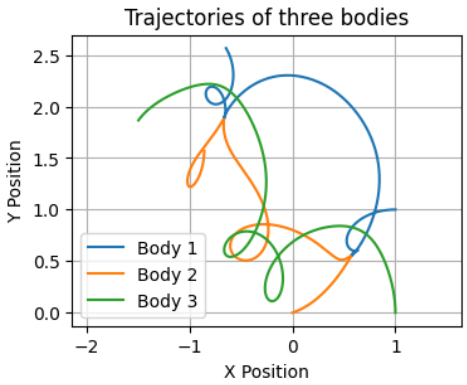

Figure 1.2.: Trajectories of three bodies.

This has spurred the development of numerous numerical methods and simulations to approximate such bodies' future positions and velocities. One of those numerical methods was developed by G. Gonçalves Ferrari, T. Boekholt, and S. F. Portegies Zwart [6]. They show how a composition of multiple independently evolved 2-body problems can approximate the solution of the general n-body problem.

This thesis demonstrates how those independent two-body problem solvers could be considered neurons within a neuronal network. This network distributes input to these neurons and combines their outputs to solve the n-body problem. It is created by implementing the code from the n-body simulator in TensorFlow and leveraging TenserFlow's computational graph and automatic differentiation capabilities.

This approach is different from other cases of machine learning in physics, like when

a deep artificial neural network was trained to solve the three-body problem [2]. In such models, the roles of individual neurons and the rationale behind the network's output remain obscured. Conversely, the network presented in this work integrates physical calculations directly into its structure, ensuring that each connection and computational unit fulfills a clear and discernible purpose. The advantage here is that every parameter and segment of the network is either fixed by the given physical approximation or can be left free to learn from data. This can potentially provide a new way of determining the masses of celestial objects like the planets of the Solar System or the mass of the supermassive black hole in the center of the Milky Way.

This thesis is structured to guide the reader from a foundational understanding of the n-body problem in section 2.1, neural networks in section 2.2, and TensorFlow in section 2.3 to the TensorFlow implementation of the n-body simulator in section 3.2. Following a review of the relevant information for this thesis, the technical details and capabilities of the model will be discussed. This includes the TensorFlow architecture in section 3.2 and the learning algorithms developed to learn the masses of celestial bodies in section 3.3. The last section—section 4—presents a conclusion and possible future work.

2. State of the art

2.1. The n-body problem

2.1.1. The two-body problem

To understand the solution to the three-body problem, it is essential first to grasp the two-body problem. The two-body problem is a critical and extensively studied problem in physics and astronomy. Despite its age (over 300 years), research on it continues. The goal is to predict the movements of two bodies influencing each other through gravity. Specifically, given the positions, velocities, and masses of two bodies at a time, it should be possible to calculate their positions and velocities at any time.

As stated in chapter 25 of Dourmashkin's work on Classical Mechanics [4], this issue is also known as the Kepler problem, named after Johannes Kepler to honor him for first describing the motion of planets in three laws. However, it was Isaac Newton who provided a solution to this problem. While the solution process is too complex to be detailed here, it's important to note that transformations can reduce the two-body problem to a one-body problem. This means that two planets moving around each other with positions (r_1, r_2), velocities (v_1, v_2), and masses (m_1, m_2) like shown in figure 2.1 can be considered as a single body with a reduced mass $m = \frac{m_1 \cdot m_2}{m_1 + m_2}$, position $r = r_1 - r_2$, and velocity $v = v_1 - v_2$, moving in a constant force field.

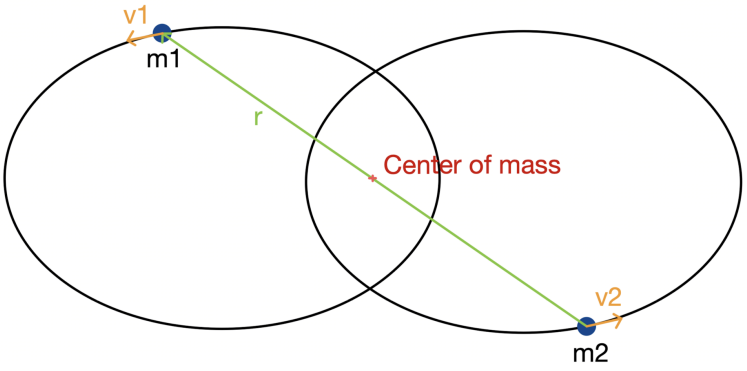

Figure 2.1.: Two bodies orbiting each other.

A particularly elegant solution is the Universal Variable Solution. Among its advantages is that it does not require distinguishing between different orbit types, such as elliptical, parabolic, and hyperbolic orbits [3, chap. 4, p. 55].

This approach is also recommended for the code in the paper [6]. This thesis utilizes the universal variable approach by Jack Wisdom and David M. Hernandez [16].

2.1.2. The three-body problem

For this section, the review of the three-body problem by Musielak and Quarles' [10] has been shortened and reformulated to fit this thesis. The three-body problem, initially formulated by Isaac Newton [12], examines the complex motion of three celestial bodies as they exert gravitational forces upon each other. Newton encountered this problem when he wanted to understand the Earth, Moon, and Sun dynamics. It has been a significant challenge in classical mechanics, known for its complexity and the inherent difficulties in predicting the precise movements of the bodies involved.

Since Newton, the problem has been explored extensively, and while it is often described as unsolvable, this characterization requires nuance. The term "unsolvable" refers to the absence of a general analytic solution —a precise mathematical expression that can predict the positions of the bodies at any given time. Henri Poincaré proved this absence in 1892 [15] during a competition created by King Oscar II of Sweden. Despite this, with advancements in numerical methods and computational power, generating highly accurate simulations of three-body systems is now possible. These simulations allow scientists to explore the movement of celestial bodies with remarkable precision. However, the chaotic nature of these systems limits long-term predictability, as even minuscule variations in the starting conditions can lead to exponential divergences in trajectories over time. Figures 2.2 and 2.3 show the chaotic nature by plotting the trajectories of three bodies over some time with only minuscule variations in starting positions.

During the studies of the three-body problem, certain special cases have been identified where the problem is solvable. These special cases are mostly solvable because they can be reduced to a two-body problem. One such instance, discovered by Euler, occurs when the three bodies are always aligned with the center of mass in a straight line [5].

Overall, the three-body problem remains an active field of study. Despite its inherent difficulty, the continued exploration of this problem continues to yield new insight.

Figure 2.2.: Bodies one, two, and three start at [1, 1, 0], [0, 0, 0], and [1, 0, 0], respectively, with equal masses of 1. Their initial velocities are [−1, 0, 0], [0, 0, 0], and [0, 1, 0]. Their trajectories for a given time period are simulated.

Figure 2.3.: The starting position of Body one is changed to $[1 + 1 \cdot 10^{-10}, 1, 0]$. This results in an exponential divergence in trajectories over time.

2.1.3. The general case

As seen earlier, the two-body problem involving just a pair of objects yields exact solutions, allowing for precise predictions of positions and velocities. However, the extension to the three-body problem significantly increased complexity, introducing scenarios where analytical solutions are not generally feasible. The n-body problem extends these challenges even further. Similar to the three-body problem, however, numerical methods can calculate the trajectories of n interacting bodies with high precision. This thesis mainly focuses on problems with $n < 10$, but the implementation

allows for relatively large n.

2.2. Neural Networks

The book on neural networks by Michael Nielson [13] gives a very good introduction. Everything in this section is based on his work.

Though this book does not explicitly define neural networks, an implicit definition of a neural network can be taken from it. A neural network can be defined as a parallel and distributed processor that can learn from data. This processor is made up of simple processing units called neurons. The Motivation for such a way of computing came from the human brain, a highly complex, nonlinear, and parallel computer. Though artificial neural networks have little in common with the brain, they are still potent tools for complex tasks because they can be automatically adjusted to learn from data and thus can discover patterns that a programmer would have a tough time finding. A common use case for them is image recognition.
The general structure of a Neural Network is that the neurons are structured in layers. Data is given to the input layer and fed through the network to produce an output. Figure 2.4 gives a general illustration of neurons. Figure 2.5 then illustrates a Network.

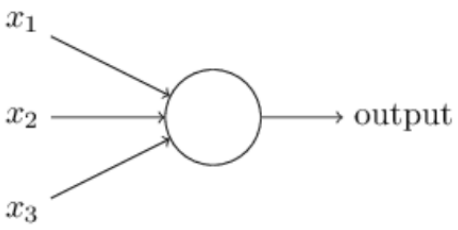

Figure 2.4.: A single neuron [13].

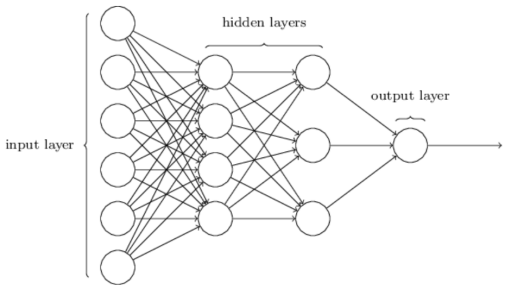

Figure 2.5.: A neural Network [13].

Many types of neurons can be used for networks. One common type is, for example, the sigmoid neuron. This neuron computes a weighted sum of its inputs, which is then passed through the sigmoid function, defined as

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

A neuron's output, y, is

$$y = \sigma\left(\sum_{i=1}^{n} w_i x_i + b\right),$$

where x_i are the inputs to the neuron, w_i are the weights of the inputs, b is the bias and σ is the activation function.

The problem with creating a network with sigmoid neurons is that the rationale behind the network's output often remains obscured. To avoid this problem, the neurons in this paper will be based on physics calculations. For example, one type of neuron used in the network will solve the two-body problem. The following will show what it means when a network learns and how it is done. For this, a measure of how well the network does is defined. It is then shown how the network could, with this information, be adjusted to perform better.

2.2.1. Cost functions

An untrained network often has some randomly initialized parameters. That means that the network is very likely to perform poorly at the calculations it should do. For example, a network with sigmoid neurons initially has randomly initialized weights and biases. That means that the network output will be a randomly created function of the inputs. The correct parameters for the problem the network has to deal with first have to be learned. To learn the parameters, a performance measure has to be defined. With this, different parameters can be compared, and better parameters can be found. One way to measure the performance of parameters is to assign a cost to each calculation the network does. The costs are higher or lower depending on how well the parameters are chosen. The notation in Table 2.1 will be used.

Table 2.1.: Notation used for cost functions.

Symbol	Definition
n	$\in \mathbb{N}$. The number of parameters that can be learned.
p	$\in \mathbb{R}^n$. The vector of parameters.
F_p	$F_p : \mathbb{R}^a \to \mathbb{R}^b$ with $a, b \in \mathbb{N}$. The function the network describes.
N	$\in \mathbb{N}$. The number of training data points.
x_i	The i-th training data point input.
y_i	The desired output of the i-th training data point input.
$\hat{y}_i$	$F_p(x_i)$
C	$C : \mathbb{R}^n \to \mathbb{R}$. The cost function gives the cost for every combination of parameters.

One often used cost function is the quadratic cost function

$$C(p) = \frac{1}{2N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2 = \frac{1}{2N}\sum_{i=1}^{N}(F_p(x_i) - y_i)^2.$$

This cost function ideally provides a good measure of the network's performance with the chosen parameters [13]. If the parameters are chosen well, the network performs well, which means the outputs will be close to the desired outputs, and the cost will be low. If the parameters are chosen poorly, the network performs poorly, which means the outputs will be very different from the desired outputs, and the cost will be high. Other cost functions also satisfy those requirements [13], but the quadratic cost function is used in this paper.

2.2.2. Gradient descent

In the last section, a measure of the performance of a network is introduced. This section will show how, with this measure, the parameters can be adjusted so that the network produces better results. This is done by finding a local minimum of the cost function. Because of the usually huge number of trainable parameters and the complexity of the derivative, an analytical solution is often unfeasible. That is why an algorithm called gradient descent is used. In the following, the mathematical idea of gradient descent is discussed [13].
Let $n \in \mathbb{N}$ be fixed but arbitrary, $\mathbf{v} \in \mathbb{R}^n$, $v_i \in \mathbb{R}$ is the i-th element in this vector and $f : \mathbb{R}^n \to \mathbb{R}$.
$\Delta f(\mathbf{v})$ is approximately

$$\sum_{i=1}^{n}\frac{\partial f}{\partial v_i}\Delta v_i = \begin{pmatrix} \frac{\partial f}{\partial v_1} \\ \vdots \\ \frac{\partial f}{\partial v_n} \end{pmatrix}^T \cdot \begin{pmatrix} \Delta v_1 \\ \vdots \\ \Delta v_n \end{pmatrix} = \nabla f(\mathbf{v}) \cdot \Delta\mathbf{v}.$$

Let $\eta \in \mathbb{R}$ be a small value. When $-\eta \cdot \nabla f(\mathbf{v})$ is chosen for $\Delta\mathbf{v}$,

$$\Delta f(\mathbf{v}) \approx \nabla f(\mathbf{v}) \cdot (-\eta \cdot \nabla f(\mathbf{v})) = -\eta \cdot \nabla f(\mathbf{v})^2$$

is smaller than zero. That means that if a very small vector pointing in the direction of the gradient is subtracted from $\mathbf{v}$, $f(\mathbf{v})$ decreases. The gradient descent algorithm then always computes the gradient $\nabla f(\mathbf{v})$ and updates $\mathbf{v}$ with $\mathbf{v} - \eta \cdot \nabla f(\mathbf{v})$. This is often referred to as taking a step. It can be easily shown that for any small fixed-length vector that can be subtracted from $\mathbf{v}$, the one that points in the direction of the gradient decreases $f(\mathbf{v})$ by the highest amount. Visually, that corresponds to the gradient always

pointing towards the steepest slope, making it a sort of compass with which the nearest minimum can be found by always going in the opposite direction. Researchers often use gradient descent for the parameters of neural networks to minimize the cost functions. That means that with each gradient step, the parameters are updated to reduce the error of the whole network.

2.2.3. Backpropagation

Up to this point, it is shown how a network can learn from training data by using the gradient descent algorithm on the cost function. What now needs to be done is to calculate the gradient of the cost function. For this, backpropagation is used. Backpropagation is a way of breaking down gradient computations with the help of the chain rule. The book by Michael Nielsen [13] shows backpropagation for neuronal networks with sigmoid neurons. This section generalizes the principles learned there so that they can be applied to any function.

The algorithm consists of two phases: the forward pass and the backward pass. In the forward pass, the function is calculated, and a computational graph is created. The computational graph for the function $f(x, y, z) = (x + y + z) \cdot z$ is shown in figure 2.6.

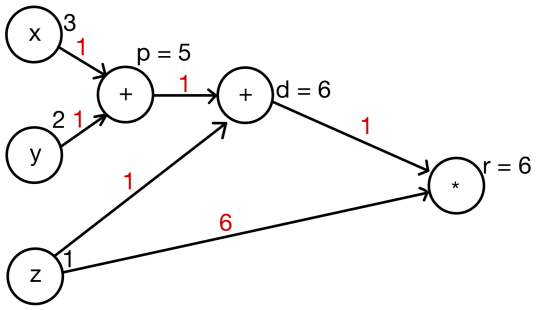

Figure 2.6.: An example for Backpropagation.

The nodes of this graph are arithmetic operations. After an arithmetic operation, the intermediate results are stored in a variable for that node. The outgoing edges from one node determine where the operation's result goes. In the backward pass, the nodes are traversed in reverse order. For each node, the partial derivative of the function with respect to the result of this node is calculated with the chain rule. To better understand the process and see how the chain rule is applied, figure 2.6 is given and $\frac{\partial f}{\partial z}$ for $x = 3$, $y = 2$ and $z = 1$ will be calculated. The variables in black are initialized in the forward pass. The information created in the backward pass is in red. First comes the forward pass. The result of an operation is stored in a variable to the corresponding node. The

result of $x + y = 3 + 2 = 5$ is for example stored in the variable $p = 5$. After the result of the computational graph is calculated, the backward pass starts. For every node, every edge is looked at, and it is asked how much a change in the node's value would result in a change in the value of the node the edge is pointing toward. This change is then multiplied by the amount, a change in the node being pointed to influences the result. Because the nodes are traversed in reverse order from the forward pass, only one derivative of one operation must be calculated per edge instead of the whole function derivative with respect to the node. Applied to the example, $\frac{\partial r}{\partial d}$ would be calculated first. r is equal to $d \cdot z = 6 \cdot 1 = 6$. That means, that $\frac{\partial r}{\partial d} = 1$. With the chain rule, $\frac{\partial f}{\partial z}$ is

$$\frac{\partial f}{\partial r} \cdot \frac{\partial r}{\partial d} \cdot \frac{\partial d}{\partial z} + \frac{\partial f}{\partial r} \cdot \frac{\partial r}{\partial z} = 1 \cdot 1 \cdot 1 + 1 \cdot 6 = 7.$$

2.3. TensorFlow

TensorFlow, an open-source library developed by the Google Brain team [9], has established itself in machine learning and deep learning. Launched in 2015, TensorFlow is designed for research and production, providing a versatile platform for developing and deploying machine learning models across various devices—from servers to mobile devices. Central to TensorFlow's design is its computational graph architecture, where data (represented as tensors) flows through a network of operations. That is how TensorFlow got its name. This graph-based approach allows for backpropagation, similar to the one described above. This is especially useful when creating a custom, very complex neural network like the one in this thesis.
The following provides a rough overview of the most important aspects of TensorFlow for this paper. The information is from the official TensorFlow documentation [9].

2.3.1. Tensors

Tensors are the primary data structure used in TensorFlow. They represent multi-dimensional arrays of data or operations outputs. Once a tensor is created, it becomes immutable, meaning its values cannot be changed. Tensors are used to pass data through the network and to hold the network's operational results. Important terminology for a tensor is:

- rank: This represents the dimension of the array. A rank-0 tensor, for example, is a scalar. A Vector would then be a rank-1 tensor.
- axis: A specific dimension of a tensor, with a scalar having none, a vector having one axis, and a matrix with two axes.

- shape: A list of values, where each value represents the number of elements of the corresponding axes.

Figure 2.7, for example, shows a visualization of a 3-axis tensor with shape [3, 2, 5]. The following code could create this tensor [8]:

```
tensor = tf.constant([[[0,  1,  2,  3,  4],
                       [5,  6,  7,  8,  9]],
                      [[10, 11, 12, 13, 14],
                       [15, 16, 17, 18, 19]],
                      [[20, 21, 22, 23, 24],
                       [25, 26, 27, 28, 29]]])
```

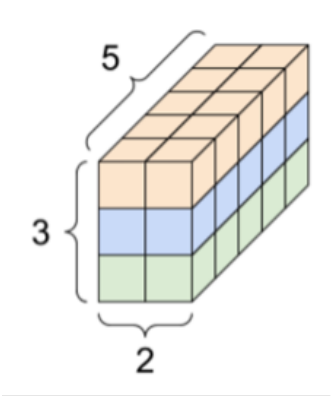

Figure 2.7.: Visualisation of a 3-axis tensor with shape [3, 2, 5], adapted from [8].

2.3.2. Variables

Variables are mutable tensor-like entities that are used to store the state of the program. Variables are designed to maintain and update the model's parameters during training. Unlike tensors, which are immutable once created, the value of a Variable can be updated and modified throughout the program's execution. This makes them ideal for representing parameters in a neural network that need to be learned and adjusted during training.

2.3.3. Graphs and functions

Tensorflow graphs are similar to the computational graph shown in figure 2.6. They have nodes representing operations in the graph and edges representing tensors carrying

data between those operations. Visualized with a tool called TensorBoard, a TensorFlow graph representing a two-layer neural network is shown in figure 2.8.

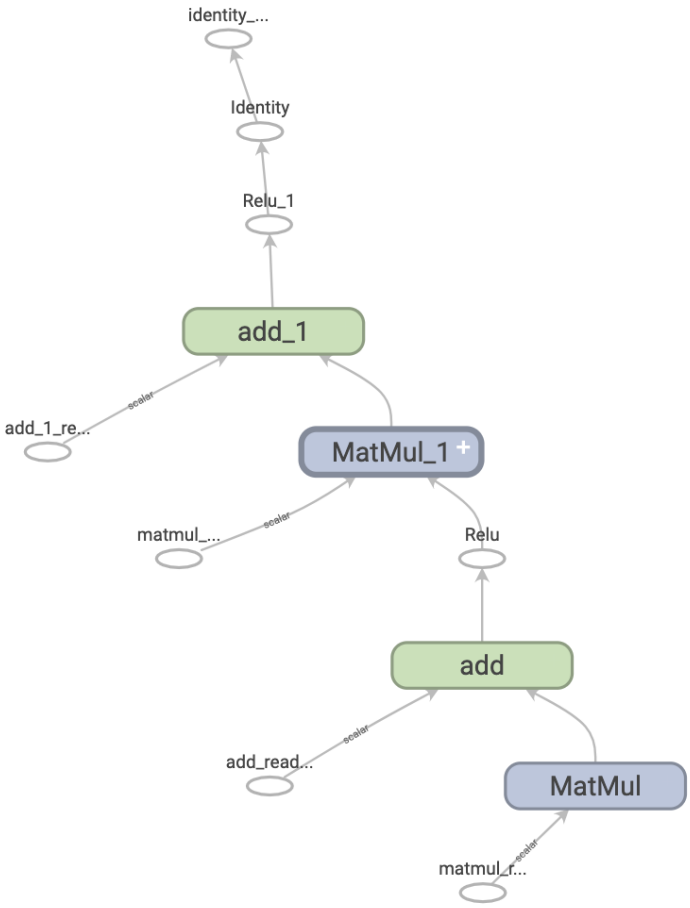

Figure 2.8.: TensorFlow graph representing a two-layer neural network [7].

A graph then serves as a blueprint of the whole execution. By encapsulating the entire computation as a graph, TensorFlow can optimize the computation as a whole. This optimization may include rearranging operations for execution efficiency, distributing computations across multiple CPUs, GPUs, or TPUs, and facilitating parallel processing of independent operations. A function written in Python can be transformed into a graph by annotating it with "@tf.function".

2.3.4. Automatic differentiation

TensorFlow's approach to automatic differentiation, crucial for optimizing model parameters during training, is encapsulated in the GradientTape API. TensorFlow "records" relevant operations executed on inputs—usually variables—inside the context of a GradientTape onto a tape and then uses that tape to compute the gradients using reverse mode differentiation, akin to Backpropagation. The inherent graph structure of TensorFlow computations is beneficial for this.

2.3.5. Comparison with similar libraries (PyTorch and JAX)

While TensorFlow remains a popular choice among developers and researchers, other libraries such as PyTorch [14] and JAX [1] offer their unique advantages and perspectives on deep learning and automatic differentiation. PyTorch, developed by Facebook's AI Research lab, emphasizes ease of use, flexibility, and dynamic computation graphs, making it well-suited for research and prototyping. PyTorch's dynamic graph (autograd) system allows for more intuitive modeling and debugging, as the computational graph is built on the fly during execution. Tensorflow, in contrast, is often very hard to debug as the computational graphs can be very complex and very different from the original code. JAX is designed for high-performance machine learning research. It extends Autograd (for automatic differentiation) and XLA (Accelerated Linear Algebra) to offer fast computations, especially on TPUs. JAX's functional programming approach promotes pure functions and immutable data structures, distinguishing it from TensorFlow's and PyTorch's object-oriented paradigms.

3. Neural network for solving the n-body problem

3.1. Motivation

This thesis aims to enable automatic differentiation of the Python code shown in figure 3.2 that describes the calculations for an n-body solver [6]. As the calculations involved in the n-body solver are long, complex, and impractical to do manually, TensorFlow is used for its automatic differentiation capabilities.

The n-body solver is chosen because of its approach to approximating the n-body problem. It analyzes isolated pairwise interactions for all combinations of bodies and combines the pairwise outcomes in such a manner that, for each body, the actual simultaneous influence of all other bodies is accurately approximated. That means the two-body problem is solved for every pair of bodies, and the results are combined to solve the n-body problem. This way of breaking down the whole problem—the n-body problem—into smaller problems—independently evolved two-body problems—is similar to how neurons in neural networks solve smaller problems that the network then combines to solve a bigger problem. In this thesis, a neural network that solves the n-body problem is created by having neurons that solve the two-body problem, combining their calculations to solve the n-body problem and making everything differentiable to facilitate learning. Figure 3.1 shows an illustration of this neural network. A color is chosen for each kind of neuron, and the corresponding code section in figure 3.2 is marked the same color.

Neural networks that try to solve the three-body problem have already been created [2], but their usage is mainly to speed up calculations, as they are not accurate enough. One factor limiting their accuracy is that they try to solve the whole three-body problem at once. Conversely, arbitrary parts of the neural network created in this thesis, which integrates physical calculations directly into its structure, can be approximated by machine learning models to make the calculations faster. This approach simplifies the function the model needs to learn as the model does not have to approximate the whole n-body solver, potentially leading to better results.

However, learning a part of the algorithm that solves the n-body problem is only some of what can be done with the neural network created in this thesis. As the inputs

are also part of a calculation, this network could be used to learn any input. This means, for example, that the masses of celestial bodies could be learned by just watching the trajectories or, given that celestial observations are often made in two dimensions, the position of an object in the third could be learned.

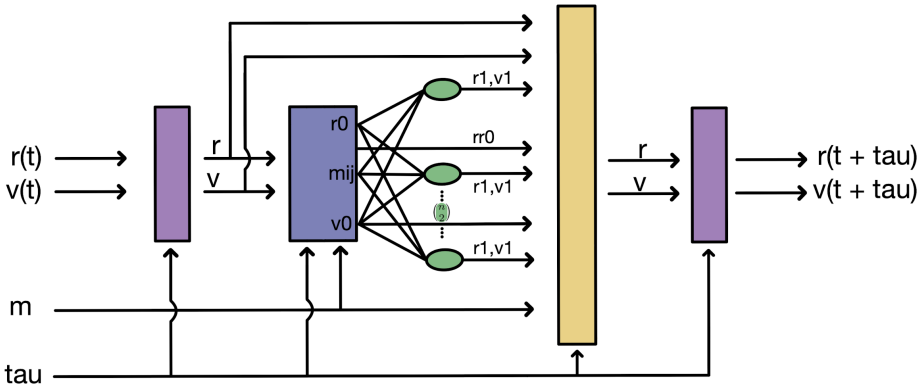

Figure 3.1.: The neural network for the solution of the n-body problem. The inputs are the positions `r` and velocities `v` at time `t` and masses `m` of all bodies. After a time step `tau`, the outputs are the positions `r` and velocities `m` of all bodies. The outer two boxes in purple are the neurons that calculate the `evolve_HT()` function. Everything in between is in the `evolve_HW()` function. The green neurons solve the two-body problem. For `n` bodies as input, there are $\binom{n}{2}$ neurons that solve the two-body problem.

```
def do_step(tau, n, m, r, v):
    r, v = evolve_HT(tau / 2, n, m, r, v)
    r, v = evolve_HW(tau, n, m, r, v)
    r, v = evolve_HT(tau / 2, n, m, r, v)
    return r, v

def evolve_HT(tau, n, m, r, v):
    for i in range(n):
        for k in range(3):
            r[i][k] += v[i][k] * tau
    return r, v

def evolve_HW(tau, n, m, r, v):
    for i in range(n):
        for j in range(n):
            if i != j:
                mij = m[i] + m[j]
                mu = (m[i] * m[j]) / mij
                for k in range(3):
                    rr0[k] = r[i][k] - r[j][k]
                    vv0[k] = v[i][k] - v[j][k]
                for k in range(3):
                    r0[k] = rr0[k] - vv0[k] * tau / 2
                    v0[k] = vv0[k]
                r1, v1 = kepler_solver(tau, mij, r0, v0)
                for k in range(3):
                    rr1[k] = r1[k] - v1[k] * tau / 2
                    vv1[k] = v1[k]
                for k in range(3):
                    dmr[i][k] += mu * (rr1[k] - rr0[k])
                    dmv[i][k] += mu * (vv1[k] - vv0[k])

    for i in range(n):
        for k in range(3):
            r[i][k] += dmr[i][k] / m[i]
            v[i][k] += dmv[i][k] / m[i]

    return r, v
```

Figure 3.2.: Abbreviated version of the Python code of the n-body problem solver [6].

In the following sections, the implementation of the Python code using TensorFlow is described. For this, the main loop shown in figure 3.2 and the missing `kepler_solver` function are implemented with TensorFlow. The application of the resulting neural network is then illustrated by introducing a local minimum-finding algorithm that enables the calculation of a planet's mass using only synthetic data. Subsequently, it is explained how this algorithm can be extended to estimate the masses of multiple bodies simultaneously using only synthetic data. Finally, the challenges of employing real data in these calculations are discussed.

3.2. TensorFlow implementation

3.2.1. The main loop

To see what has to be done to transform the Python code in figure 3.2 into a neural network, it is helpful to look again at the previously introduced definition of a neural network. Part of this definition is that a neural network is a parallel and distributed processor that can learn from data. The task at hand then involves making the code as parallel as possible. Implementing it with TensorFlow would enable automatic differentiation, facilitating the network's learning process.

The `do_step()` function accepts inputs listed in table 3.1.

Table 3.1.: Inputs of the `do_step()` function.

Symbol	Definition
`tau`	Specifies the duration of the time step.
`n`	Represents the total number of bodies.
`m`	A one-dimensional array containing the masses of each body, with each element corresponding to a distinct body.
`r`	A two-dimensional array that holds the positions of the bodies in three-dimensional space. For example, `r[0]` returns an array with three elements, denoting the position of the first body.
`v`	Similarly, a two-dimensional array that stores the velocities of the bodies, using the same indexing system as `r`.

The inputs can easily be transformed to tensors with `r`, for example, being a rank-2 tensor. With this, the `evolve_HT()` function can be written as `r = r + v * (tau / 2)`. The `evolve_HW()` function is not easily transformed. In this function, the i and j loops select two different bodies and then perform calculations on them. One way of doing all loop executions in parallel is to structure the calculation steps of the executions in

a matrix. The element in row i and column j of one matrix would then represent the calculations of that step done with the body selected by the loop variable i and the body selected by the loop variable j in the original implementation. It is important to note that in the case of the positions and velocities, this matrix would be a cube, as every element is a vector. However, to keep the formulations simpler, it is still referred to as a matrix, as in the context of making the code parallel, it is not important what exactly the elements are.

A good example is the calculation step, where the combined mass `mij` is calculated. The mass vector can be seen on the left in figure 3.3. Body one has a mass of 1, while body two has a mass of 2. On the right, the tensor that holds all the combined masses for all loop iterations can be seen. For example, the combined mass of bodies one and two is in the first row and second column. The diagonal, where i is equal to j, is zero because no calculations need to be done for this case.

Figure 3.3.: The input vector, `m`, is shown on the left and the `mij` tensor is on the right.

It is easy to see why the matrix is symmetrical. The symmetry or antisymmetry resulting from each pair being looked at from each body's point of view can significantly accelerate calculations.

To see how this matrix is derived, looking at what an element represents is again good. The element in row i and column j of one matrix represents the calculations of that step done with the body selected by the loop variable i and the body chosen by j in the original implementation. Therefore, two other matrices have to be created. One has the bodies selected by i as elements, and the other has the bodies chosen by j. In the case of the combined mass, the two matrices and how they are combined are shown in figure 3.4.

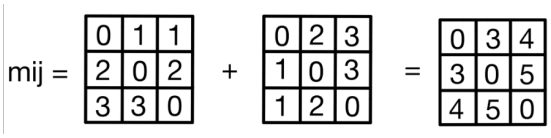

Figure 3.4.: The creation of the mij tensor.

The two matrices already have zeros on the diagonal because nothing has to be

calculated there. The matrix on the left is structured such that each row, aside from zeros on the diagonal, is filled with the same repeated value across all other positions, as the body, the loop variable i refers to only changes every three iterations of the j loop. Similar to this, the second matrix does this for each column. It is then easy to see that the second matrix is just the transpose of the first. Figure 3.5 illustrates the process for `rr0` and `vv0`, and demonstrates how `r0` is subsequently calculated.

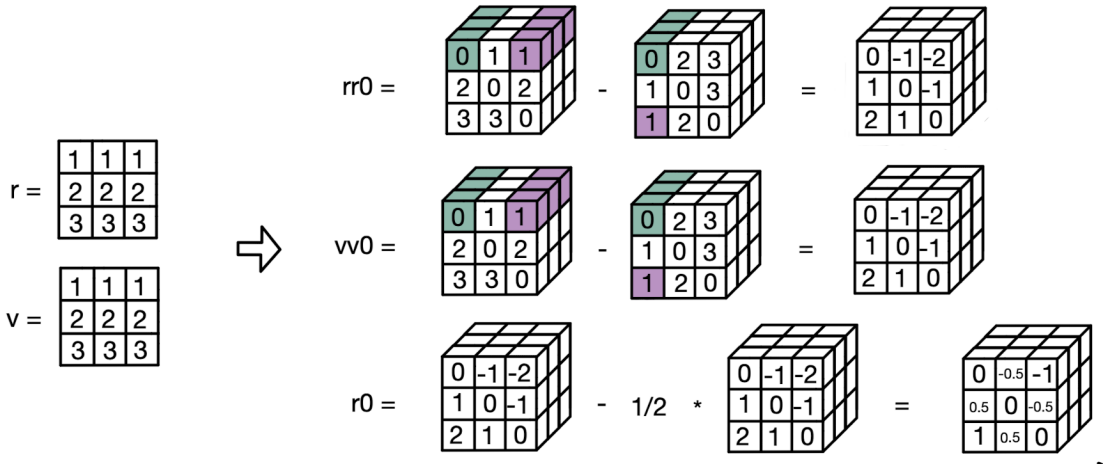

Figure 3.5.: How the relative positions and velocities are calculated.

`rr0`, `vv0` and `r0` are antisymmetric. It is logical for the relative positions and velocities of bodies to be antisymmetric, given that these values differ only in sign depending on the reference body under consideration. This means that when the perspective between two bodies is switched, the direction of the observed values for relative position and velocity reverses. This is very useful as that means, that the `kepler_solver` function only has to be called $\binom{n}{2} = \frac{n(n-1)}{2}$ times instead of $n^2 - n$ times. In figure 3.1, this is indicated by the $\binom{n}{2}$ highlighted in green.

By using the `map_fn` function from TensorFlow, the call of the `kepler_solver` for each pair of bodies can be done in parallel. For this, the input parameters must be concatenated so that the `kepler_solver` function is called for each eight-dimensional vector element. This is shown in figure 3.6.

Every element in the upper triangular prism of the concatenated cube is set to zero, as no calculations have to be done. If the elements are zero, the `kepler_solver` returns zero. After the call, the upper triangles of `r1` and `v1` are filled so that the matrices with vectors as elements are antisymmetric. Due to the impracticality of writing the values of the tensors `r1` and `v1` in figure 3.5, letters are used as placeholders instead.

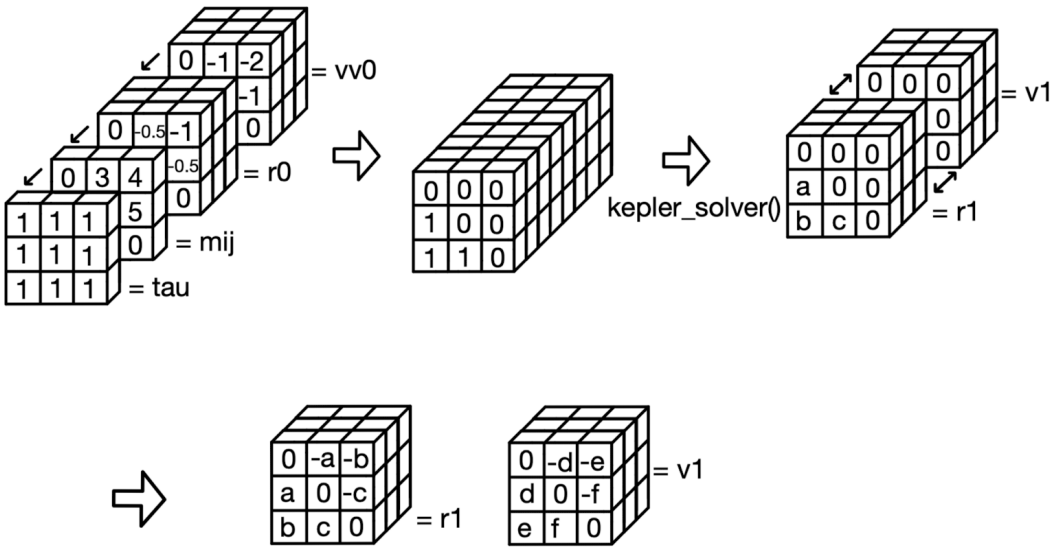

Figure 3.6.: How the `kepler_solver` function is called.

All the changes in positions and velocities then have to be summed up. For this, a few more tensor operations must be done, and the result is reduced along its first axis with the TensorFlow function `reduce_sum`. Figure 3.7 visualizes this last step.

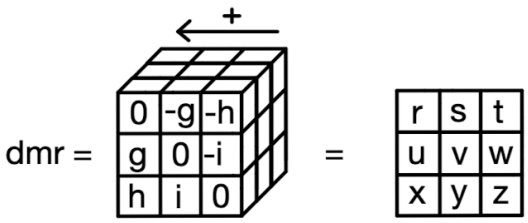

Figure 3.7.: The calculation of the change in positions.

The resulting rank-2 tensors `dmr` and `dmv` are added to the input tensors `r` and `v`, which are then returned.

3.2.2. The kepler_solver function

The code shown in figure 3.2 invokes the `kepler_solver` function. This function solves the two-body problem and can be considered a type of neuron in a neuronal network when implemented using TensorFlow. In the following, the implementation of Jack

Wisdom and David M. Hernandez [16] is translated into TensorFlow code. One thing to note is that the code written by Jack Wisdom and David M. Hernandez was C code. The translation to Python was not difficult, just time-consuming, and thus is not described here. The most challenging part was transforming the code into a TensorFlow Graph. For this, a recursion had to be made iterative, as TensorFlow demands that the whole computational graph be known when starting the execution. In the `kepler_solver` function, an attempt is made to solve the Kepler problem for a single time step. If this attempt fails, the Kepler problem is solved recursively for four time intervals, being one-quarter the size of the original. The results are combined to solve the problem for the original time interval. Figure 3.8 shows the relevant Python code for this recursion. The `kepler_step_internal` function tries to solve the Kepler problem for the given time step.

```
def kepler_step_depth(kc, dt, beta, b, s0, s, depth, r0, v2, eta, zeta):
    if depth > 30:
        print("kepler depth exceeded")
        exit(-1)

    flag = kepler_step_internal(kc, dt, beta, b, s0, s, r0, v2, eta, zeta)

    if flag == False:
        ss = State(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
        kepler_step_depth(kc, dt / 4.0, beta, b, s0, ss, depth + 1, r0, v2,
                          eta, zeta)

        r0 = math.sqrt(ss.x * ss.x + ss.y * ss.y + ss.z * ss.z)
        v2 = ss.xd * ss.xd + ss.yd * ss.yd + ss.zd * ss.zd
        eta = ss.x * ss.xd + ss.y * ss.yd + ss.z * ss.zd
        zeta = kc - beta * r0

        kepler_step_depth(kc, dt / 4.0, beta, b, ss, s, depth + 1, r0, v2,
                          eta, zeta)

        r0 = math.sqrt(s.x * s.x + s.y * s.y + s.z * s.z)
        v2 = s.xd * s.xd + s.yd * s.yd + s.zd * s.zd
        eta = s.x * s.xd + s.y * s.yd + s.z * s.zd
        zeta = kc - beta * r0

        kepler_step_depth(kc, dt / 4.0, beta, b, s, ss, depth + 1, r0, v2,
                          eta, zeta)

        r0 = math.sqrt(ss.x * ss.x + ss.y * ss.y + ss.z * ss.z)
        v2 = ss.xd * ss.xd + ss.yd * ss.yd + ss.zd * ss.zd
        eta = ss.x * ss.xd + ss.y * ss.yd + ss.z * ss.zd
        zeta = kc - beta * r0

        kepler_step_depth(kc, dt / 4.0, beta, b, ss, s, depth + 1, r0, v2,
                          eta, zeta)
```

Figure 3.8.: The recursive algorithm within the `kepler_solver`.

The iterative form was achieved by conceptualizing the problem similarly to a depth-first search in a tree. Each node of the quaternary tree corresponds to a calculation of the Kepler problem for a time interval. If the original time interval is of length dt, then the assigned time interval of a node at depth d is $dt/4^d$.

Two variables are required to navigate this tree and determine the correct length of the current time interval being calculated. The first variable indicates the position in the tree, and the second indicates the depth.

Determining the depth is relatively straightforward. Each time a calculation fails, and the algorithm moves one level deeper, the depth is incremented. Conversely, each time the fourth quarter of a time interval is calculated, the position moves up a level inside the tree, or the problem for the entire original time interval is solved, and the depth is decremented. Thus, starting at a depth of 0, the computation is finished when the depth reaches -1. This serves as an elegant termination condition and is a reason for depth being a separate variable.

How, then, is the current position in the tree determined? Initially, it is helpful to consider how a path from the origin node to the current node would be described. Starting at the root node, if a calculation fails, four possible time intervals are being calculated next. 0 would indicate that the first quarter is being computed. Hence, the position would be at the first node. One would then denote the second node, two the third, and three the fourth. This path description can be extended recursively. The current position is represented by a 64-bit integer, interpreted in base 4. Visually, the number $312_4 = 54_{10}$ would represent the correspond to the path highlighted in figure 3.9.

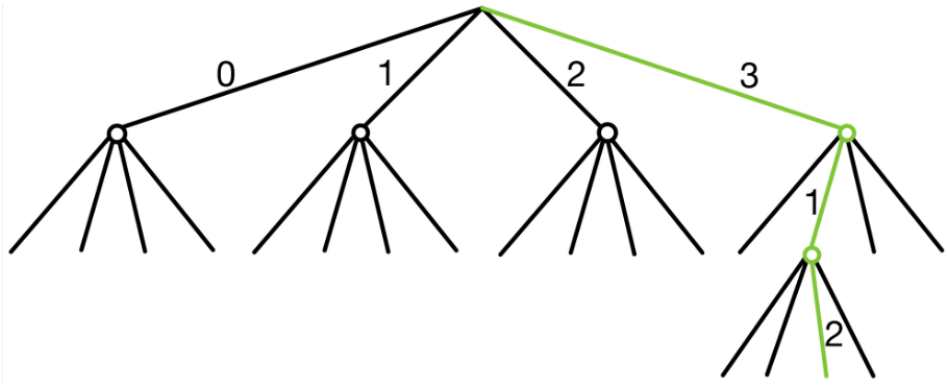

Figure 3.9.: Quatenary tree representing the `kepler_step_depth` calculations. Each tree node corresponds to a calculation of the Kepler problem for a time interval. The depth is the number of edges traversed to reach the root.

With the 64-bit integer, a path of length 30 can be represented, matching the maximum recursion depth in the original implementation. This is evident since $2^{63} - 1 > 2^{62} = 4^{31} > 4^{30}$. The algorithm thus performs a depth-first search of this constructed tree. However, the tree is not infinitely large; a node only has children if the Kepler problem cannot be computed for that node. Once a node is visited —meaning the next node at the same level or one level above is considered— the Kepler problem for that node's assigned time interval is deemed solved. The algorithm terminates when the depth is -1, indicating the Kepler problem for the origin node has been solved.

It is also important to note that in the actual implementation of this iterative algorithm, the integer does not directly store the path to the current node but indicates which child node needs to be visited next for each level of the tree. If no child node needs to be visited next, 0 is stored.

Algorithm 1 shows pseudo-code for the solution of the Kepler problem

while *depth* $\leq$ *30* ***and*** *depth* $\neq$ *-1* **do**
    `// If the depth is greater than 30, implying the problem cannot be solved even after dividing the time interval into 1/4-sized segments 30 times, the computation is terminated.`
    `// If the depth is -1, the computation is complete.`
    if *least significant digit of the position variable = 0* **then**
        Attempt to solve the Kepler problem.
        if *not successful* **then**
            The position variable is incremented and then multiplied by 4, and the depth is incremented.
            `// This describes moving one level deeper to solve the Kepler problem four times for time intervals 1/4 the size and also indicates that the first child node will now be visited and the next child node after that to be visited is the second.`
        end
        else
            The position variable is divided by four, and the depth is decremented until the position variable can no longer be divided without remainder or the depth is -1.
            `// This describes going up the tree to the next node that is waiting for this calculation's result. This is not necessarily the direct parent node.`
        end
    end
    else
        Variables like `eta` or `zeta` are updated for the next calculation.
        if *least significant digit of the position variable* $\neq$ *3* **then**
            The position variable is incremented and then multiplied by 4.
        end
        else
            The least significant digit of the position variable is set to 0. The position variable is then multiplied by 4.
            `// This indicates that the current node has no more work to do and can be skipped on the way up the tree.`
        end
        The depth is incremented.
    end
end

Algorithm 1: The iterative algorithm of the `kepler_step_depth` function.

3.3. Learning

The preceding sections show how the n-body solver is implemented with TensorFlow, creating a neuronal network that solves the n-body problem. The subsequent discussions focus on the training of specific components within this network. This is illustrated through the task of learning the masses of celestial bodies. Initially, using synthetic data for the cost function, the mass of a single body is learned. For this, a minimum finding algorithm is introduced. This method is subsequently extended to simultaneously learn the masses of multiple bodies. Although successful with synthetic data, the limitations of applying this approach to real-world data are demonstrated, and insights into the underlying reasons are provided.

3.3.1. Learning with synthetic data

Learning the mass of one body

Most neural network training employs gradient descent, utilizing a step size initially set and subsequently decreased over time. This approach is particularly effective when the objective is to locate any local minimum with multiple minima. However, in celestial body attributes, it is hypothesized that a unique minimum exists when only one body at a time is considered. For given positions and velocities, the body has only one correct mass, which would explain the resulting positions and velocities after a time step. Further deviation of this correct mass would result in further deviation from the correct positions and velocities. Given this premise, searching for the unique minimum can be significantly accelerated.

For the cost function, the non-TensorFlow code is run for a few time steps, and the results are seen as the correct data, with which the network results given the learned parameter are tested. After the synthetic data is calculated, part of the input—in this case, the mass of one planet—is forgotten and then learned again by first guessing the input and then, with the help of a custom gradient descent algorithm performed on the quadratic cost function, is learned again.

The custom gradient descent algorithm begins with an arbitrary step size and an arbitrary initial guess at the value. As the gradient, in this case, is directed towards the opposite of the global minimum, the update rule for the mass being learned is mass = mass - (gradient $\cdot$ step size). This process is iterated, increasing the step size by a factor of eight as long as the gradient's sign remains unchanged, indicating that the global minimum still needs to be bypassed. If a change in the gradient's sign is observed, suggesting an overshoot of the global minimum, the step size is halved at every subsequent iteration to converge to the minimum.

This adaptive step size algorithm allows for arbitrarily precisely determining the sought mass. When the global minimum has yet to be bypassed, the exponentially increasing step size ensures rapid convergence, even though the initial guess was very far off. To facilitate control over the precision of this method, the implementation accepts a parameter named accuracy. This parameter specifies the threshold below which the step size must fall before the algorithm terminates, thereby defining the upper limit of error.

In Figure 3.10, a simplified illustration of the algorithm is presented. This depiction highlights a worst-case scenario, where the penultimate value learned by the algorithm slightly deviates from the actual minimum. As a result, the final error margin is just below the specified accuracy threshold.

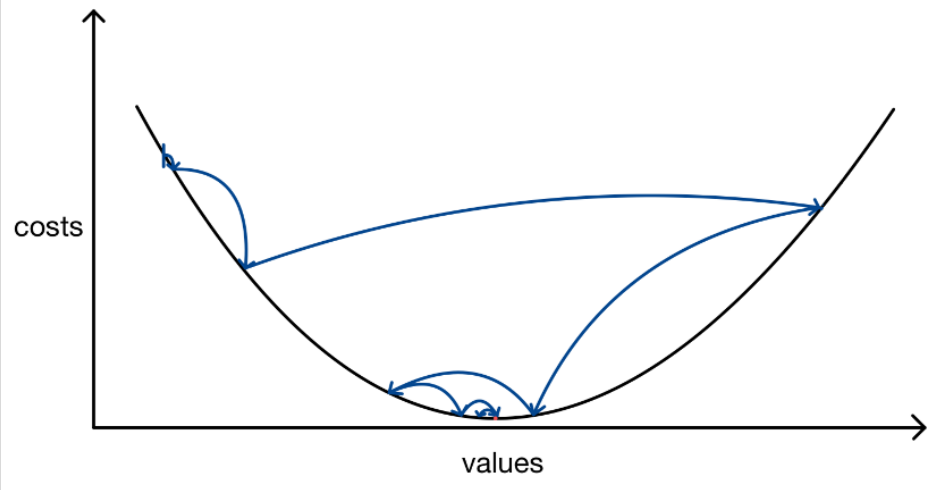

Figure 3.10.: Illustration of how the algorithm finds the global minimum.

As a test example, the mass of the Sun is learned. For this, the Sun, Mercury, and Venus values are given as input, the non-TensorFlow code is executed for ten time steps, and the result is used to evaluate the learned value of the Sun. If real data were used to evaluate the learned value, it would not be a good example to choose the first two planets and the Sun, as the influences of the other solar system planets are ignored. However, as synthetic data is used, this does not matter. The values of the celestial bodies are given in astronomical units, days, and mass relative to the Sun. NASA provides the starting values [11].

```
values = find_global_minimum(
    tau=0.1,
    n=3,
    m=[1, 3.302e23 / 1988500e24, 48.685e23 / 1988500e24],
    r=[[-7.673580434738827E-03, -3.302239590214931E-03,
        2.073556671274906E-04],
       [3.452597940573748E-01, -1.498122010844959E-01,
        -4.413780254122306E-02],
       [1.031676785783527E-01, -7.220322696819025E-01,
        -1.605894448118791E-02]],
    v=[[5.284080958425666E-06, -6.612725729575814E-06,
        -5.777615059407366E-08],
       [5.305044905894823E-03, 2.724866176413712E-02,
        1.741157849018483E-03],
       [1.986026764570113E-02, 3.004963198838360E-03,
        -1.104347674861390E-03]],
    learning_rate=0.1,
    starting_guess=100,
    num_of_steps_in_do_step=10,
    accuracy=1e-5,
    variable_to_learn="m"
)
```

The parameter `variable_to_learn` could also be set to `r` or `v` to learn the position or velocity of the Sun, respectively. It is always the first value in the mass array that is learned. The learned values after each training epoch are:

```
[99.9, 99.10000000000001, 92.7, 41.5, -368.1, -163.3,
-60.900000000000006, -9.700000000000003, 15.899999999999999,
3.099999999999998, -3.3000000000000025, -0.10000000000000231,
1.4999999999999978, 0.6999999999999977, 1.0999999999999979,
0.8999999999999979, 0.9999999999999979, 1.0499999999999978,
1.024999999999998, 1.012499999999998, 1.0062499999999979,
1.0031249999999978, 1.001562499999998, 1.000781249999998,
1.0003906249999979, 1.0001953124999978, 1.000097656249998,
1.000048828124998, 1.0000244140624979, 1.0000122070312478,
1.000006103515623]
```

It can be seen that, even though the initial guess was off by a factor of 100, the minimum algorithm converges fairly quickly and within the accuracy.

Learning the masses of multiple bodies simultaneously

In the last section, an algorithm was introduced that took advantage of the hypothesis that there can only be one correct mass and used it to learn the mass of one body within a predefined accuracy. This minimum finding algorithm is now expanded to also work when multiple masses are learned simultaneously. The difficulty with this is that the minimum of the cost function with respect to a specific mass when all other masses are held is fixed, depending on the other masses. So, if all masses are learned simultaneously, this would visually correspond to the curve in figure 3.10 moving around and changing.

Because of this, the algorithm described in the last section would not work. This is because it would detect a change in the gradient's sign and incorrectly assume that the minimum lies within the values range that the last gradient step skipped over. But because the other masses also changed, the minimum could have moved to a value the algorithm can not reach now, as it is too far away.

For this case, the approach includes three phases instead of two, like in the algorithm with only one mass. In phase one, the step size is now increased by a factor of t/r^k until a change in the gradient's sign is observed. t is the learning rate factor, r is the convergence rate, and k is the number of times phase 3 was executed. This mirrors the process used in single-variable minimum search. Upon a change in the gradient's sign, the algorithm always enters or stays in Phase 2, where the step size is halved, assuming proximity to the minimum. If no change in the gradient's sign after a step in Phase 2 is observed, the method transitions to Phase 3, maintaining the current step size without further halving, to explore the immediate vicinity of the estimated minimum. If, after a Phase 3 iteration, the gradient's sign remains unchanged, implying that the minimum has shifted, the algorithm reverts to Phase 1. The `convergency_rate` gradually reduces the multiplication factor of the step size in Phase 1. However, if the minimum remains within the explored interval —as indicated by a sign change during a Phase 3 step— the process moves back to Phase 2.

Figure 3.11 presents a conceptual visualization for simultaneously learning multiple masses. The illustration is not meant to depict precise functions but to convey the underlying concept. Within this figure, the vertical blue line represents the current mass values under consideration, while the arrow signifies the adjustments made to these values during a training epoch. The black curve illustrates the cost function's behavior for varying parameter values, assuming all other parameters remain constant.

The transition from the left to the right demonstrates a dynamic shift in the cost functions, marked by the gray curves that trace the function's position in the previous epoch. The illustration highlights the three phases:

1. At the top, phase one is depicted where the step size is increased by a factor of

eight because the learning rate factor is eight and the gradient sign never changed.

2. The middle part reveals a transition to phase two caused by a change in the gradient's sign. The moving minima between epochs show the necessity of adapting the step size more dynamically rather than merely halving it at each iteration.

3. The bottom part illustrates phase three, where the step size remains constant over two epochs.

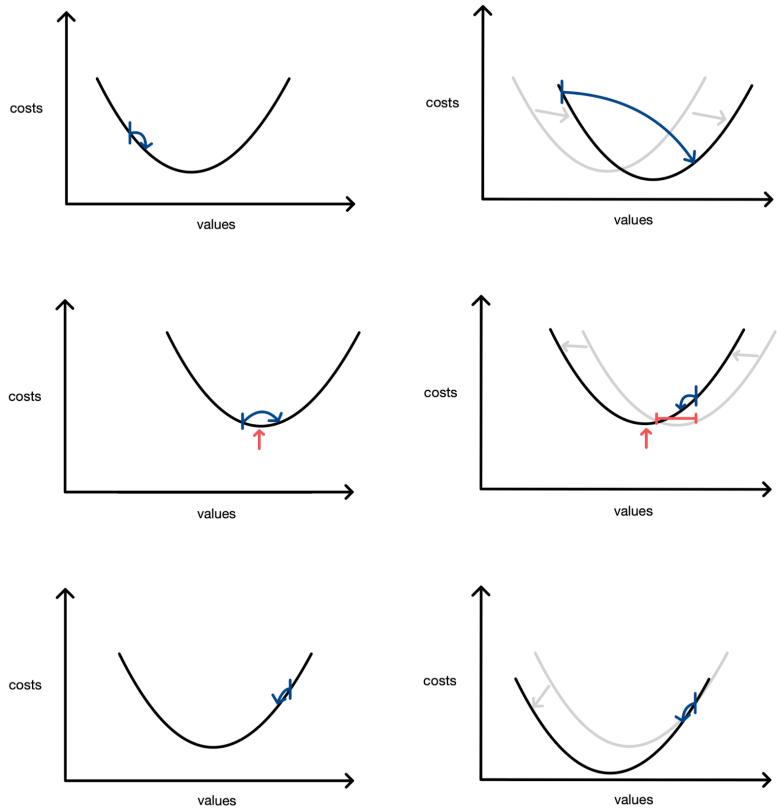

Figure 3.11.: Illustration of learning multiple masses simultaneously.

This dynamic adjustment strategy works very well, with initial guesses being even unreasonably far off. The algorithm, for example, learns the correct values for the Sun, Mercury, and Venus with the starting guess of `[1692, -342, 0.00000012301]`. However, if it is too far off, it seems to converge to completely wrong results. This, however, should not be a problem, as a realistic starting guess of the mass of the Sun is

not more than 100000 times the mass of the Sun like in this example, where the learned values are entirely wrong: `starting_guess = [169842, -342, 0.00000012301]`. Another problem arises in a different example: the masses of the Sun, Venus, Earth + Moon, Mars, Jupiter, Saturn, Uranus, and Neptune are learned. Figure 3.12 shows the cost per training epoch.

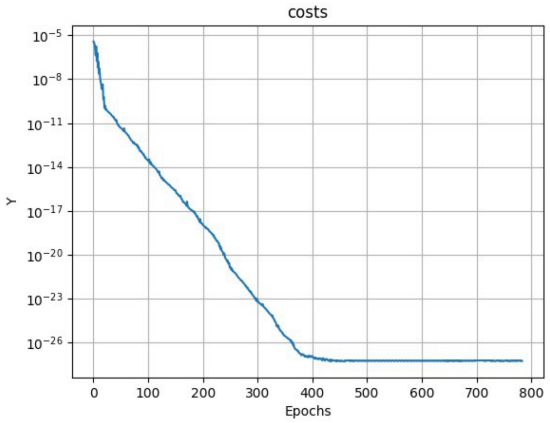

Figure 3.12.: Plot of the costs per training epoch.

It can be seen that, in the beginning, the cost decreases but stays constant at somewhere around 10^{-28}. This is because the calculated positions and velocities with the learned masses are very close to the target positions and velocities, leading to numerical errors in the gradient calculations. Because of this, gradient descent does not work anymore. This leads to limited accuracy of the learned masses. The differences between the learned and actual masses are still less than $1.5 \cdot 10^{-9}$.

3.3.2. Learning with real data

In the last section, synthetic data showed how the masses of celestial bodies could theoretically be calculated with high accuracy. When, however, real data is used, problems arise. In testing, the real starting values of the Sun, Venus, Earth and Moon, Mars, Jupiter, Saturn, Uranus, and Neptune for 28.02.2024 are taken from NASA's website, and a day is simulated. The difference between the data provided by NASA for 29.02.2024 and the calculated values for that day is less than $9 \cdot 10^{-7}$, which is too high to accurately calculate the masses.

4. Conclusion

This thesis solved the three-body problem, a fundamental challenge in classical mechanics and astronomy, using a neural network approach with TensorFlow. For this, the Python code written by G. Gonçalves Ferrari, T. Boekholt, and S. F. Portegies Zwart [6] was translated to TensorFlow. The resulting model interprets independent two-body problem solvers as neurons within a neural network, which combines the results of those neurons to predict complex gravitational interactions among celestial bodies. This new method contrasts with traditional 'black box' neuronal networks, providing a clear, interpretable network directly integrating physical laws into its architecture. As a result, arbitrary parts of the network can be learned, including but not limited to the masses of celestial bodies.

The experiments with synthetic data have demonstrated that the masses learned by the network are close to their actual values. However, challenges were encountered when it was tried to learn the masses from real data. To resolve these issues, future work could explore using longer trajectories or alternative n-body simulators, which could then be implemented with TensorFlow, analogous to how it is done in this thesis. The idea's potential to have an interpretable machine learning model with integrated physical laws and then learn unknown parameters, which would be very hard to estimate otherwise, is huge. Given that this approach calculates masses based on n-body interactions ($n > 2$) rather than limiting itself to binary relations, it has the potential to:

- Approximate the masses of all planets in the solar system,
- Enhance the precision of the gravitational constant measurement,
- Estimate the mass of Sagittarius A* (the black hole at the center of our galaxy) along with the masses of reference stars and
- Determine the planetary masses of extraterrestrial solar systems where orbital trajectories are known.

Furthermore, by learning arbitrary input parameters concurrently, the network could potentially deduce the masses of all celestial bodies using only two-dimensional inputs

for positions and velocities. This is particularly relevant given that celestial observations are often made in two dimensions.

This thesis could lay the groundwork for a better understanding of the universe by utilizing machine learning techniques and interpretable models.

List of Figures

List of Tables

Bibliography

[1] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018.

[2] 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." In: *Monthly Notices of the Royal Astronomical Society* 494 (2020), pp. 2465–2470. DOI: 10.1093/mnras/staa713.

[3] V. A. Chobotov, ed. *Orbital Mechanics*. 3rd ed. Reston, Va.: American Institute of Aeronautics and Astronautics, 2002, pp. XV, 460. ISBN: 1-56347-537-5.

[4] P. Dourmashkin. *Classical Mechanics*. LibreTexts Classical Mechanics Course Material. Available online at LibreTexts.org. Accessed on 02 April 2024. 2024.

[5] L. Euler. "De moto rectilineo trium corporum se mutuo attrahentium." In: *Novo Comm. Acad. Sci. Imp. Petrop.* 11 (1767), pp. 144–151.

[6] T. B. G. Gonçalves Ferrari and S. F. P. Zwart. "A Keplerian-based Hamiltonian splitting for gravitational N-body simulations." In: *Monthly Notices of the Royal Astronomical Society* 440 (2014), pp. 719–730. DOI: 10.1093/mnras/stu282.

[7] *Introduction to graphs and tf.function*. https://www.tensorflow.org/guide/intro_to_graphs. Accessed on 02 April 2024. 2023.

[8] *Introduction to Tensors*. https://www.tensorflow.org/guide/tensor. Accessed on 02 April 2024. 2023.

[9] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015.

[10] Z. E. Musielak and B. Quarles. "The three-body problem." In: *Reports on Progress in Physics* 77 (2014). Article number: 065901. DOI: 10.1088/0034-4885/77/6/065901.

[11] NASA Jet Propulsion Laboratory. *Horizons System*. https://ssd.jpl.nasa.gov/horizons/. California Institute of Technology. 2024.

[12] I. Newton. *Philosophiae Naturalis Principia Mathematica*. Royal Society Press, 1687.

[13] M. A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[14] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019.

[15] H. Poincaré. *Les Méthodes Nouvelles de la Méchanique Céleste*. vols 1–3. Gauthier-Villars, 1892.

[16] J. Wisdom and D. M. Hernandez. "A fast and accurate universal Kepler solver without Stumpff series." In: *Monthly Notices of the Royal Astronomical Society* 453 (2015), pp. 3015–3023. DOI: 10.1093/mnras/stv1862.

A. Appendix

The complete source code and resources for this project are available on GitHub. The repository includes a Dockerfile specifying all the tools and versions required to ensure the project's code runs seamlessly. *The project's code can be found at:* `https://github.com/AndreasMerrath/Solving-the-three-body-problem-with-neural-networks.git`. The project structure is:

- **NormalCode:** This folder contains the Python implementation of the n-body simulator.
- **TensorCode:** This folder contains the TensorFlow implementation
- **Learning:** This folder contains the learning algorithm and two Jupyter notebooks demonstrating the implementations and the testing examples discussed in Section 3.2.
- **Simulation:** This folder contains the visualizations of the trajectories shown in figures 1.1, 1.2, 2.2, and 2.3.