UCLouvain

École polytechnique de Louvain

Deep learning methods for multibody modeling

Application to railway dynamics

Author: **Paul Michiels**
Supervisors: **Nicolas Docquier, Paul Fisette**
Readers: **Pierre-Antoine Absil, Sébastien Timmermans**
Academic year 2023–2024
Master [120] in Electro-mechanical Engineering

Abstract

In recent years deep neural networks have emerged as an effective approach to tackling various problems. In parallel, multibody simulation allows the generation of large amounts of data. This data can be used to train neural networks. For multibody modeling, a deep learning approach has two main advantages. It allows us to make accurate models which are time-efficient. They also do not require explicit mathematical equations between input and output data and are thus very suited for surrogate models. The main objective of this report is to demonstrate how deep-learning approaches can be used for multibody modeling. This master's thesis focuses on two deep neural networks, applied to railway applications. During the first experiment a deep neural network predicts the vertical acceleration of a railway vehicle's car body. The predictions are made based on the track profile. This network is composed of two independent sub-networks: one for handling low frequencies and another for high frequencies. Both sub-networks are mainly made of long short-term memory layers, treating the accelerations as time series. The second experiment performs parameter identification on the secondary suspension of a railway vehicle. The network is a multi-layer perceptron. This master's thesis presents both experiments; how the networks were constructed, as well as their obtained accuracies.

Acknowledgments

This work has only been possible thanks to the efforts of many people. I want to thank them all for their help, advice and encouragement.

First of all, I want to thank two persons who made this master's thesis becoming true: Nicolas Docquier, for being the supervisor of this master's thesis, and for his valuable expertise, advice and feedback.
Sébastien Timmermans, for his trust, encouragement and invaluable advice throughout this project. Thank you for your availability, and always motivating answers to my questions and problems.

I want to thank the entire IMMC team, for their advice during meetings and presentations, and especially: Paul Fisette for being supervisor of this master's thesis. Olivier Lantsoght for his help in Robotran and Linux.

Thanks to Estelle Massart for her advice on deep learning.

Thanks to Pierre-Antoine Absil for being part of the jury.

I'm particularly grateful to my family, for their daily support during this whole project.

Last but not least, my friends, for their support and motivation throughout this entire year.

Computational resources have been provided by the Consortium des Équipements de Calcul Intensif (CÉCI), funded by the Fonds de la Recherche Scientifique de Belgique (F.R.S.-FNRS) under Grant No. 2.5020.11 and by the Walloon Region.

Preliminary Note

For this report, two AI-based tools were used: ChatGPT from OpenAI and Grammarly from Grammarly Inc. Both tools assisted in correcting spelling and grammar mistakes, as well as providing synonyms. These tools were not employed for generating explanations and are not responsible for the content of this report.

Contents

Nomenclature

Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
DL	Deep Learning
DNN	Deep Neural Network
FFNN	Feed-Forward Neural Network
LSTM	Long Short-term memory
ML	Machine Learning
MLP	Multilayer Perceptron
MSE	Mean Squared Error
NN	Neural Network
NRMSE	Normalized Root Mean Squared Error
PINN	Physics Informed Neural Network
PSO	Particle Swarm Optimization
ReLU	Rectified Linear Unit
RMSE	Root Mean Squared Error
RNN	Recurrent Neural Network

Variables

$\mathcal{L}$	Loss function
σ	Sigmoid function
b	Bias
f	Activation function
h	Cell state

w	Weight
x	Input
y	Output
z	Pre-activation value

Chapter 1

Introduction and Objectives

1 Context

Multibody dynamics are nowadays a well-known field of research. In recent years, sensors technology has arisen in many industries, providing large amounts of real-time data. This is the case with multibody applications, like railway vehicles. In parallel, artificial intelligence (AI) has become a main topic of research. The coupling of AI with traditional multibody techniques can be divided into two distinct approaches.

First, the fusion of AI with the data obtained from sensors holds immense potential for augmenting traditional multibody approaches. This huge number of real-time data has to be processed fast, and at all times. AI can be used in railway applications for predictive maintenance. For example, for rail defects detection or to detect suspension anomalies and wear. Compared to classical multibody solutions, AI is a good option, as it is very fast. AI can also handle large amounts of data. Being slightly less accurate might not be problem, as the huge numbers of data can mitigate the errors.

The second perspective is AI that can be used inside multibody simulations. This has two main objectives. The first one is the reduced computation time. Solving the equations of motion in a traditional way costs a lot of time, whereas AI can predict the accelerations, velocities and positions of bodies much faster. However, there is always a trade-off between computation time and accuracy. The second advantage is making a very accurate model, only based on input and output data. The mechanical relationship between both can be unknown. This is a surrogate model. For example, computing the force exerted by a double spring-mass-damper, without knowing the spring constants, the friction coefficients, and the wear accumulated over time. Simple measurements of inputs and outputs can lead to a very accurate model.

The coupling of AI with multibody dynamics is a very large field of study. In this work, the main focus is set on railway vehicles. Note that the techniques demonstrated on railway vehicles through this work can be extrapolated to other multibody applications.

2 Objectives

The first aim of this master's thesis is to research how deep learning can be used as an improvement for multibody modeling. The fields of multibody dynamics and artificial intelligence have significant attention in academic research. Numerous papers are focusing on these fields. While articles that merge both approaches are relatively new, they do emerge in recent years. A comprehensive overview of the current state of research in this domain is needed. We want to know the main advantages and disadvantage of merging deep learning with multibody modeling.

This master's thesis has also as purpose to emphasize real implementations of deep learning techniques. We will showcase 2 applications, based on a railway vehicle. The goal is to demonstrate how the techniques are implemented, but also to understand how they can be used for different multibody applications. Simplicity, accuracy and robustness of the deep learning methods are important points we want to study.

The first application aims to showcase the predictive results of a recurrent neural network. The first step is to understand how it works. This experiment will also highlight the accuracy and efficiency of such an approach, as well as the trade-off between both. Specifically, the objective is to forecast the vertical accelerations of a railway vehicle. Vertical acceleration is an important indicator of both the safety and comfort of trains. Moreover, in various multibody applications, understanding acceleration is key, as it is rather easy to obtain velocity and position by taking the integrals of acceleration.

The second application has as main purpose to understand how deep learning can be used for obtaining model parameters. In the case of railway vehicles this can be important to model existing trains. Due to wear and tear, we do not know precisely the mechanical properties of an old train. The suspension acts as a kind of black box. In that case it is not easy to measure the suspension constants. Deep learning can be used for computing these constants. It is also important for fault detection. Both suspension and track defects can be found using deep learning. For this application, more traditional neural networks will be used. The objective will be to study the simplicity of the model and the possibility to apply one model to different cases. Finally, the obtained accuracy, as well as the amount of data needed to make such a model is emphasized.

3 Outline

This master's thesis is divided into 6 chapters. The general outline is as follows. The first chapter gives the general introduction, objectives and outline of this work.

Chapter 2 is a state of the art. First, it will give the basics of deep learning techniques. In particular for 3 types of neural networks: feed-forward, recurrent and long short-term memory neural networks. It will start with the very basics of these networks, and give all the knowledge needed to understand the rest of this work. The second part of this state of the art will be a literature review. It handles deep learning, multibody dynamics, and more important, the coupling between both deep

learning and multibody dynamics. It will also set the state of the actual research, with a focus on railway applications.

The model of the railway vehicle used in this work is explained in chapter 3. It will tackle its mechanical details, as well as its implementation in the Robotran software. At the end of this section, a short explanation of the programming languages used for the model, as well as for the deep learning algorithms, is given.

Chapter 4 covers the first application. This implementation handles the predictions of accelerations of the vehicle car body. This is done while the train is driving on a straight track with vertical, sinusoidal track defects. The model used for this application is based on a recurrent neural network.

The second case is explained in chapter 5. It gives a deeper look at parameter identification with a deep learning point of view. In particular spring and damper constants of the suspension of the train are predicted, while running on random, unknown track defects. First, using two independent networks. Next, merging both networks together could give an increase in time efficiency. At the end, the amount of data needed for such a model is highlighted.

Finally, chapter 6 concludes this work. It resumes the advantages of deep learning for multibody applications, as well as the encountered challenges. Leads for future research are given.

Chapter 2

State of the Art

1 Deep Learning (DL)

Artificial intelligence (AI) is a term used to describe the behavior of machines to think like humans. AI involves three different approaches: data-driven, rule-based or a combination of both. An example of AI that is not a machine learning technique is the 'if-else', rule-based expert system. [11]

Machine learning (ML) is a subset of AI. It only considers the data-driven approaches and makes use of statistics, mathematics and optimization techniques. ML is also known as the part of AI that learns on its own through a large amount of data, without being specifically programmed for it. Examples of ML techniques are: decision trees, linear regression...

Deep learning (DL) is a subset of ML and includes all deep neural networks (DNN). A DNN is an artificial neural network (ANN) with at least one hidden layer. This is further explained in section 2.2. DNNs work similarly to the biological neural networks in human brains. [1]

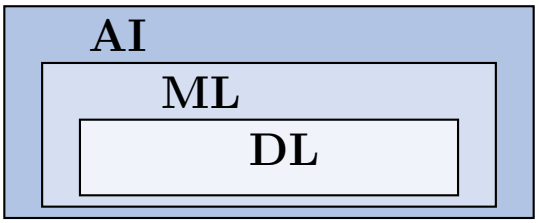


Figure 1.1: AI, ML and DL

Deep learning will always be used to predict an output, based on an input. It is important to note that there must be a mathematical relation between the input and output. This relation can be unknown, but has to exist. For example, in this work we try to predict the vertical acceleration of a train, based on the shape of the track. The equations are complex, but there is a clear relation between track shape and vertical acceleration. It is not possible to predict this acceleration, based on the color of the train.

Deep learning is used for both classification and regression tasks. In classification tasks, the output consists of predicting which class an input belongs to. In regression tasks, the output consists of predicting a continuous value rather than a categorical label. This work will focus on deep learning used for regression tasks in multibody systems.

2 Feed-Forward Neural Networks

Artificial neural networks were mentioned for the first time in the literature as early as 1943. The mathematician Walter Pitts and neurophysiologist Warren McCulloch wrote the article "A logical calculus of the ideas immanent in nervous activity"[20]. Fifteen years later, in 1958, the psychologist Frank Rosenblatt invented the perceptron[24]. It is the simplest form of a neural network. It is still a keystone of most modern neural networks, which are built by putting multiple perceptrons together.

2.1 Perceptron

A perceptron is made of exactly one single neuron. It is represented on figure 2.1.

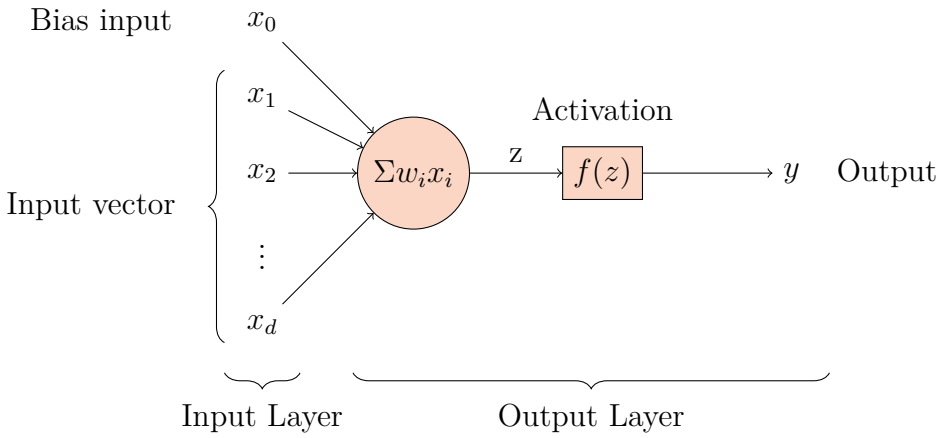


Figure 2.1: Perceptron

The perceptron maps the input vector $\vec{x}$ to the output value y. This input vector consists of d input features: $x_1, x_2, ..., x_d$. The perceptron will first multiply the input vector by a weight vector, $\vec{w}$. The resulting linear combination is called the pre-activation value, z. During this linear computation, a bias term b is added to the weighted sum. A common approach is adding a constant input feature equal to 1. The weight associated to this constant value will be the bias. We will refer to the bias term with index 0, ($x_0 = 1$) and w_0 is the bias. This gives $d + 1$ input features. The aim of the bias is to shift the weighted sum vertically. We will omit writing the bias term in neural networks in this work, but we will start the weighted sums at 0 instead of 1, as this gives the same result. The pre-activation value is calculated as follows:

$$z = \vec{w} \cdot \vec{x} \tag{2.1}$$

At this stage, the perceptron is only a linear regression. The weights are the coefficients and the bias the intercept. As the original perceptron is a binary classifier, it is needed to activate and deactivate the neuron. This is done by passing the pre-activiation value through an activation function, f.

In the perceptron described by Frank Rosenblatt, the pre-activation value is passed through a Heaviside step function in order to obtain the output value, y. This step function is the activation function and defined as follows:

$$f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{2.2}$$

The output is 1 or 0, so the original perceptron is indeed a binary classifier. In a multi-body mechanical system it could, for example, predict the derailment of a train. The input features could be the speed of the train, the track cant and the curve radius. The perceptron will output 1 if the train is going to derail and 0 if it is not. Of course the mathematical relation between input and output has to be quite simple in the case of one single neuron, as it is 'only' a step function applied to a linear regression. The weights of the perceptron have to be tuned in order to fit the data. This process is called training and is explained in section 2.4.

The perceptron can be continuous, by using a continuous activation function. This is explained further in section 2.2. In that case the output could be the speed of the train, which is a continuous value.

On figure 2.1, one can observe the first layer of the perceptron is called the input layer, containing the input vector. The second layer is the output layer. It computes and returns the output. Since only the output layer performs calculations, the perceptron is called a single-layer neural network. The perceptron is equivalent to a neuron. Even if both terms can be swapped, the term 'neuron' is mostly used when the perceptron is part of a bigger network. While 'perceptron' is used when the perceptron is a stand-alone, single-layer neural network.

2.2 Multi-Layer Perceptron

The multi-layer perceptron (MLP) is the neural network obtained by putting together multiple layers, each consisting of multiple neurons[1]. Its advantage with respect to the perceptron, is its capability to handle more difficult relationships between input and output data. This is because a larger network has more parameters. Consequently, it can fit more complex patterns.

A representation of an MLP is depicted in figure 2.2. In the notations, the superscript indicates the layer, the subscript refers to the neuron in this layer.

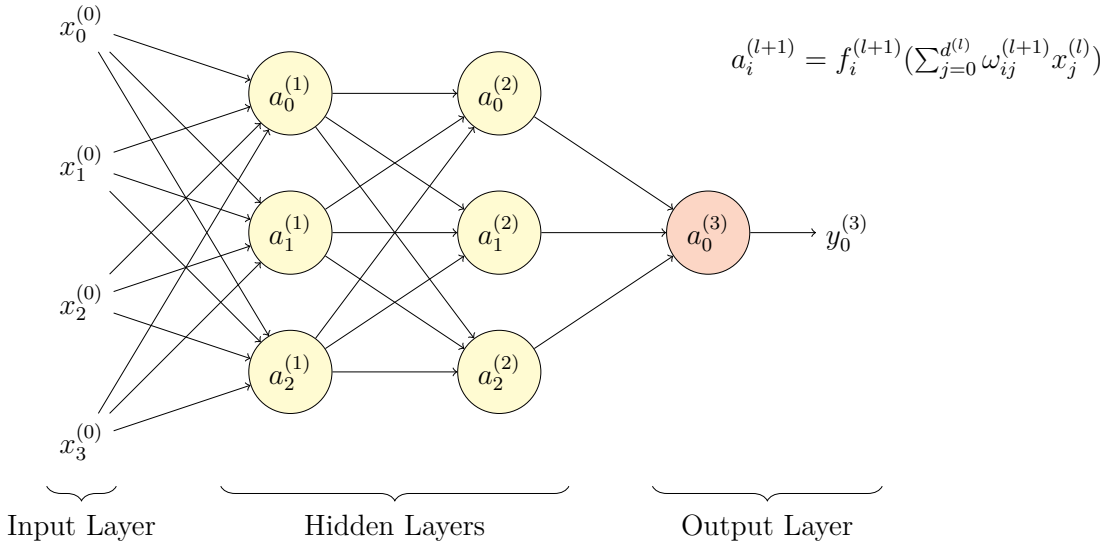


Figure 2.2: Multi-layer perceptron with 3 layers

The first layer is called the input layer. The output layer is the last one. All the layers in between are hidden layers. The MLP on figure 2.2 has 2 hidden layers. Both have 3 neurons. The output layer consists of 1 neuron. For every layer $(l+1)$ in the network, we can write the constitutive equations of the MLP as follows:

$$\begin{aligned} \vec{z}^{(l+1)} &= W^{(l+1)}\vec{x}^{(l)} \\ \vec{y}^{(l+1)} &= f^{(l+1)}(\vec{z}^{(l+1)}) \end{aligned} \tag{2.3}$$

- $\vec{z}^{(l+1)}$ is the pre-activation vector of layer $l+1$. It has size $d^{(l+1)}$.
- $W^{(l+1)}$ is the weight matrix, it has dimension $d^{(l+1)} \times d^{(l)}$.
- $\vec{x}^{(l)}$ is the input vector of layer $l+1$, which is the output vector of layer l. It has size $d^{(l)}$.
- $\vec{y}^{(l+1)}$ is the output vector of layer $l+1$, with size $d^{(l+1)}$.
- $f^{(l+1)}$ is the activation function of layer $l+1$, used in a relaxed way.

The choice of the activation function is a crucial step in the design of an MLP. It needs to be a non-linear function in case of non-linear patterns between input and output data. This activation function precisely gives the MLP its non-linear property. During tuning of the parameters of an MLP, gradient descent will be used. Consequently, the activation function must be differentiable. Common choices for activation functions are sigmoid, tanh or ReLU, as visible on figure 2.3. [9]

The sigmoid function is defined as follows:

$$sigmoid(z) = \frac{1}{1+e^{-z}} \tag{2.4}$$

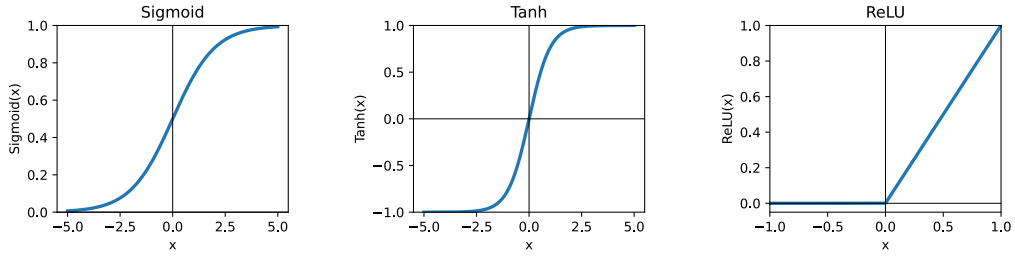


Figure 2.3: Non-linear activation functions

The sigmoid function was one of the first non-linear activation function used in ANN. It is still very popular in deep learning applications today. [8] However, there are two main drawbacks with the sigmoid function. First, the sigmoid saturates for very low or very high values. This saturation makes the derivative of the sigmoid function very small. When using a gradient descent algorithm (see section 2.7), the increase (or decrease) of the weights will be very small, as the increase is proportional to the derivative. This result in a very slow tuning of the weights. The second problem is the fact the sigmoid function is not zero-centered. The lack of zero-centering in the sigmoid function means that its output is biased towards either positive or negative values. This bias can result in gradients that are predominantly positive or negative. This leads to poor convergence during parameter tuning with gradient descent algorithms, as explained in section 2.7. [21]

To solve the problem of the not zero-centered sigmoid function, the tangent hyperbolic function is used. Its mathematical expression is:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.5}$$

This function is zero-centered. It has a better convergence rate. However, the tanh function still has the saturation problem.

The Rectified Linear Unit (ReLU) function is given by:

$$ReLU(z) = \max(0, z) \tag{2.6}$$

Despite this function also being not zero-centered, it is nowadays the most popular activation function in deep learning. It is very simple and computationally efficient. It does not saturate for increasing values. The main drawback are the "dead neurons". These are neurons which have a negative pre-activation value. In that case the ReLU will always output 0. The neuron will not have any influence on the output of the neural network. Furthermore, it will not be possible to adapt these dead weights with a gradient descent algorithm, as the derivative is also 0 for negative inputs. [10] One possible solution is to adapt the ReLU function to the Leaky ReLU function. This function is defined as follows:

$$f(z) = \max(\alpha z, z) \tag{2.7}$$

Here α is a small positive value. This means for negative pre-activation values the output will be a small negative value, but not zero. The gradient will then also be non-zero, the gradient descent algorithm can be used.

There is no general 'best activation function'. It is important to test different ones in each specific application and observe which one gives the best results. However, depending on the specific case, some activation functions are more likely to work properly. If the expected output of the MLP is bigger than 1, the sigmoid and tanh functions are bad choices for the output layer. Indeed, the output will saturate at 1. The ReLU function is a better choice. If the outputs are normalized, the sigmoid or tanh activation functions can be used as well.

2.3 Types of MLP

The multi-layer perceptron shown in figure 2.2 is a feed-forward neural network (FFNN). This classification is due to the fact that neurons only transmit their output to neurons further forward in the network. They do not give information to neurons of the same or previous layers. This feed-forward neural network is fully connected, because each neuron takes as input all the neurons of the previous layer. It is possible to make networks without all the connections, as long as each neuron stays attached to the NN. Cutting some connections can be computationally efficient, because each connection represents some computations. If enough connections are kept, the output will not necessarily lose accuracy due to some cuts. Fully-connected NN are also referred to as dense NN. Finally, this MLP is also called a deep neural network (DNN), as there is more than one hidden layer. On the contrary, networks with 1 hidden layer or less are called shallow NN.

2.4 Training

The question is now: "How to find the optimal weights of all the neurons in the MLP, in order to achieve the best possible predictions?" This is done by a process called "training". As mentioned before, a widely used algorithm is gradient descent. [25] In order to train an MLP, we need input data. This data includes different samples. Each sample must contain the input vector $\vec{x}$, but also the output vector $\vec{y}$. This data is called 'labeled', as the output vector contains the true outputs. In order to train the NN on this data, all the data is first fed into the network. This is called the forward phase. Next, the error is computed using a loss function. Finally, the weights of the network are adjusted, during the backward phase. These iterations are done multiple times, until a threshold on the error is reached, or a maximum number of epochs is done. One 'epoch' is the entire dataset going once through all the data.

2.5 Forward phase

Before training, the weights of the MLP are set up randomly. The first phase of each training iteration (epoch) is the forward pass. During this phase the input vectors of the training data are fed into the network. The obtained outputs y_i can be compared to the expected training outputs $\hat{y}_i$, using a loss function.

2.6 Loss function

A loss function $\mathcal{L}$ measures the difference between y_i, the outputs of the MLP and $\hat{y}_i$, the true outputs. [26] For regression problems, a possible loss function is the mean squared error (MSE):

$$\mathcal{L}(y_i, \hat{y}_i) = \frac{1}{N_{outputs}} \sum_{i=1}^{N_{outputs}} (y_i - \hat{y}_i)^2 \tag{2.8}$$

MSE is the default loss function used through this work. However, other well-known loss functions exist [26], like cross-entropy. A good loss function needs to be differentiable, with respect to the network parameters. Being computationally efficient is also important, as the loss function is computed many times during training.

2.7 Backward phase: Gradient descent algorithm

The second training phase is called the backward phase [28]. During the backward phase, the weights of the model will be adjusted, starting from the output of the neural network. The idea is to change these weights in the direction of the gradient of the error. This is done with the gradient descent algorithm. We will increase the weights of the MLP as follows:

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \alpha \frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} \tag{2.9}$$

In this equation α is called the learning rate. The partial derivative (gradient) gives the direction of the change, and α sets the size of the increase. Using a larger learning rate accelerates weight adjustments, leading to quicker convergence with fewer iterations. However, it could overshoot the minimum in the error function. A smaller learning will be more safe to converge to the minimum, at the cost of more iterations.

The difficult part in this algorithm is computing the gradients. The chain rule is used, which states:

$$\frac{\partial a}{\partial c} = \frac{\partial a}{\partial b} \frac{\partial b}{\partial c} \tag{2.10}$$

This process is illustrated with the MLP shown on Figure 2.2. This network has 1 output layer, and 2 hidden layers. The first adjusted weights are the ones in the output layer, $w_{0j}^{(3)}$. Subscript $0j$ stands for the jth weight of neuron 0. Consequently, the derivatives of the loss function with respect to these weights are needed:

$$\frac{\partial \mathcal{L}}{\partial w_{0j}^{(3)}} = \frac{\partial \mathcal{L}}{\partial y_0^{(3)}} \frac{\partial y_0^{(3)}}{\partial z_0^{(3)}} \frac{\partial z_0^{(3)}}{\partial w_{0j}^{(3)}} \tag{2.11}$$

In this equation, the first term is the partial derivative of the loss function with respect to the output. Then, this term is multiplied by the derivative of the output to the pre-activation value. Finally, the last term is the derivative of the pre-activation

value with respect to the weight.

By using the chain rule we can apply the same process to update the weights of the penultimate layer. For the weights of the first neuron of the second hidden layer ($l = 2$) we obtain:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(2)}} = \frac{\partial \mathcal{L}}{\partial y_0^{(3)}} \frac{\partial y_0^{(3)}}{\partial z_0^{(3)}} \frac{\partial z_0^{(3)}}{\partial y_j^{(2)}} \frac{\partial y_j^{(2)}}{\partial z_j^{(2)}} \frac{\partial z_j^{(2)}}{\partial w_{ij}^{(2)}} \tag{2.12}$$

The same process can be applied to all the weights in all neurons of the MLP. Going backwards in the neural network when updating the weights is referred to as back-propagation. It shows the importance of the activation function to be continuously differentiable.

In this work, Adam optimizer [15] is used to adjust the weights in a way that ideally reduces the loss. It computes adaptive learning rates for each weight by maintaining running averages of both the gradients and their squares. These learning rates are individual for each weight. Adam optimizer benefits from an appropriate initial learning rate, which can be relatively large to accelerate the training process, but not too large to cause instability. Smaller learning rates, on the other hand, increase the chances of convergence

2.8 Overfitting

Overfitting is one of the main problems encountered while training a neural network. Fitting a model on a particular training dataset makes it less accurate on another test dataset. This problem and its solutions are well described in "An Overview of Overfitting and its Solutions "[29]. The most common solutions are shortly described here: increasing the training dataset, reducing the numbers of input features, using dropout and early stopping.

Increasing the training dataset will make it easier for the model to generalize some patterns. Indeed, with a larger dataset, the model is exposed to a wider range of variations. This makes it less likely to memorize specific examples and more likely to learn global patterns.

Decreasing the numbers of input features, reduces the number of false dependencies between the output and useless features. But, it is very important to keep features which are mathematically significant for the relationship between input and output features.

Early stopping is implemented during training. The entire dataset is split into a 'train' and 'validation' dataset. When evaluating the loss function during training, we will calculate $\mathcal{L}$(train) and $\mathcal{L}$(validation). Only $\mathcal{L}$(train) is used during back-propagation. The loss of the validation dataset is kept in memory. After the next iteration, both losses need to be equivalent and to decrease. However, at one point, $\mathcal{L}$(train) will decrease while $\mathcal{L}$(validation) is increasing. At this moment the network starts overfitting the data. It updates the weights to the specific train dataset, but

performs worse on the global validation dataset.

The idea behind 'dropout', is to randomly remove some neurons in the NN. This happens temporarily. In fact, at every forward phase some nodes are removed. This decrease the number of tunable parameters, and thus force the network to capture a more global pattern. Too many parameters would let the network adapt to each single sample of the training dataset. Dropout also assures all nodes in the network are equally important. When a node is removed, the network has to be able to give make good predictions. The ratio of nodes removed for every epoch, is given by the 'dropout rate'.

2.9 Hyperparameters optimization

During training we will optimize the parameters of the model, which are the weights associated to each neuron. These parameters are updated after each epoch. On the contrary, hyperparameters are set at the construction of the MLP. They decide the architecture of the MLP. Hyperparameters remain constant during training. The most important ones are: number of layers, number of neurons, activation functions and initial learning rate.

A perfect technique which computes the best choice of these hyperparameters before training the network does not exist. These hyperparameters are set during the construction of the network. Then, the network is trained. Finally, the performance of the NN is evaluated, by computing the loss function. This process can be done multiple times with different hyperparameters, it is then possible to compare the obtained performances. The process of designing a finite set of hyperparameters, then testing all possible combinations inside this set, is called grid search. [2] Grid search is computationally not efficient, as the entire network has to be trained multiple times. Still today, no perfect, efficient and accurate algorithm exists. An alternative way is random search, which is very similar to grid search. The difference is that only some sets of hyperparameters are tested, and this in a random way.

Another technique is called particle swarm optimization (PSO) [17]. It is a population-based stochastic optimization technique inspired by the social behavior of birds flocking or fish schooling. A swarm of particles moves iteratively through the search space, where each particle's position represents a potential solution. The movement of the particles is influenced by three factors: previous particle velocity, own best-known position and the best-known positions of the entire swarm. This process enables the swarm to converge towards the optimal solution over time. The primary advantage of PSO is its time efficiency. It does also not require any gradient information. Additionally, PSO is effective in exploring large search spaces and tends to converge faster than traditional methods like grid search, as not all possible solutions are tested. The main disadvantage of PSO is the risk of premature convergence, where particles may settle on local optima instead of the global optimum. This is why PSO is called pseudo-optimal. In this master's thesis PSO is used for hyperparameter tuning. PSO applied to hyperparameter tuning is given by algorithm 2.

Algorithm 2 Particle swarm optimization algorithm

```
Initialize 10 particles with random hyperparameters
Initialize personal best position of each particle as its current set of hyperparameters
Initialize global best position as the personal best of any particle
while iterations < 50 do
    for each particle do
        Update velocity based on current position, personal and global best positions
        Update position based on velocity
        Train network with parameters from current position
        Evaluate the trained network
        if new position improves personal best then
            Update personal best position
        if new position improves global best then
            Update global best position
return global best position
```

3 Recurrent Neural Networks

In multiple dynamical systems, time series are present. It is very hard to predict these time series with FFNN. Indeed, the MLP does not take into account any time related information. For this purpose we introduce recurrent neural networks (RNN). Unlike FFNN, RNN does not assume that outputs (and inputs) are all independent. Each output depends on both inputs and outputs from a given number of previous timesteps. RNN are widely used for time series forecasting, but also in other types of sequential data predictions, like: speech recognition, translation etc.

3.1 Global Formulation

A basic recurrent neural network is depicted on figure 3.1. This NN is recurrent because it has cyclic connections.

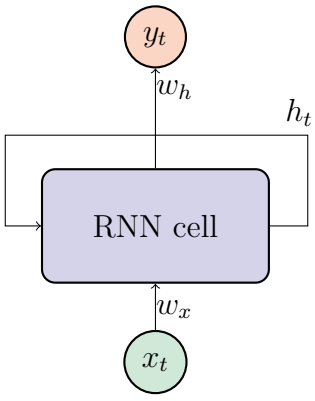


Figure 3.1: Recurrent neural network

The input and output of the RNN are respectively x_t and y_t. The index t gives information about the current timestep. The hidden state, denoted h_t, is computed

inside each cell. This is done by using information from the current input and the hidden state of the previous timestep $(t-1)$. The hidden state is the 'memory' of the cell, it stores information from past timesteps. This process is visible on figure 3.2. It shows the same RNN as figure 3.1, but in an unfolded/time-layered view.

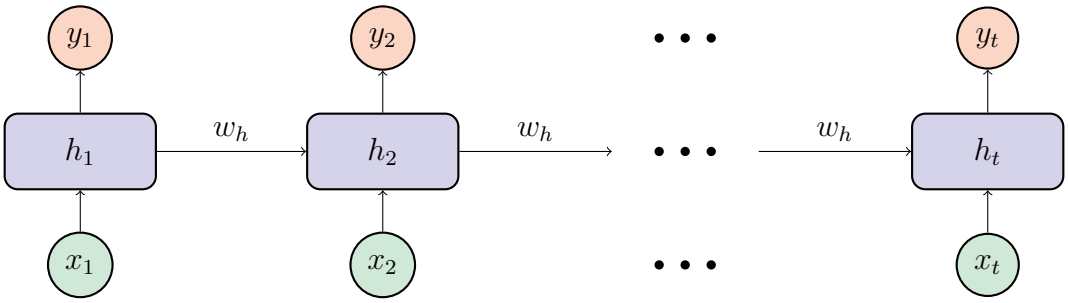


Figure 3.2: Unfolded view of a basic recurrent neural network

The inside of a recurrent neural network cell is visible on figure 3.3.

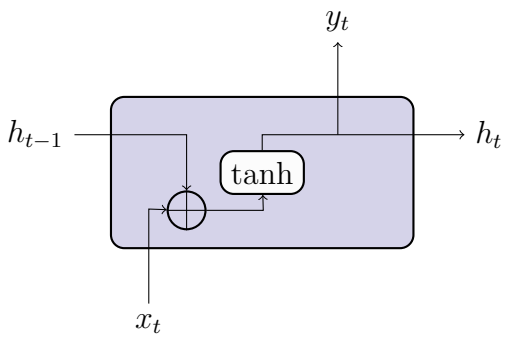


Figure 3.3: Recurrent neural network cell

The mathematical equations governing the RNN cell can be written as:

$$\begin{aligned} h_t &= \tanh(w_h h_{t-1} + w_x x_t) \\ y_t &= w_y h_t \end{aligned} \tag{2.13}$$

Three parameters (weights) are needed for these equations: w_h, w_x and w_y. It is important to note that these weights are the same in all cells of the RNN. This is visible in the unfolded view on figure 3.2. Only the input, output and hidden state are different at every timestep. Each output depends on the previous hidden state. Each hidden state depends on the current input and previous hidden states:

$$y_t = F_t(h_{t-1}, x_t) = F_t(x_1, x_2, ..., x_t) \tag{2.14}$$

In the case of the unfolded RNN visible in figure 3.2, there is an output at every timestep. It is possible to make RNNs with missing outputs. These are frequently used for word generation application. We give the 3 first letters of a word to the model. These are the inputs x_1,x_2 and x_3. The first two cells will not produce any output, however they do compute the hidden state and pass it to the next cell. Only

the last (3th) cell will produce an output, which could be the next letter in the word.

Other variations such as missing inputs are also possible. All these variations are depicted on figure 3.4. Adapting the equations of the RNNs to these different variants is quite simple, as it only requires to remove the terms with missing inputs or outputs. The hidden state is the only link between two cells, consequently it cannot be removed.

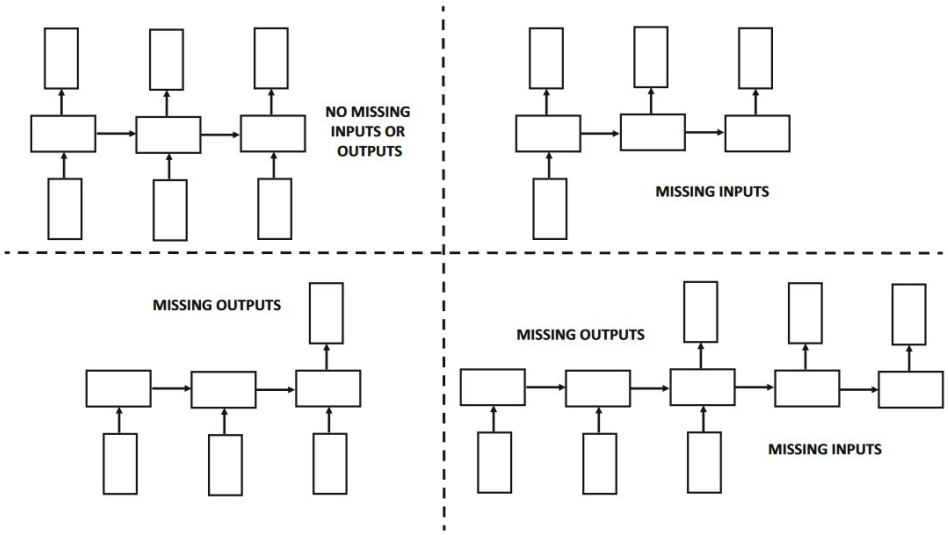


Figure 3.4: Recurrent neural network variants
Image source: Neural Networks and Deep Learning: A Textbook [1]

3.2 Backpropagation through time

As for an MLP, backpropagation is used to train RNNs. Though it is a different variant, called 'backpropagation through time' [27], the main concept is the same. A loss function, $\mathcal{L}$, is chosen to evaluate the output of the network. Exactly as for FFNN, this can for example be MSE. In order to find the changing rate of each of the three weights, we will first pretend the weights are different at every timestep. This is how we obtain $t \times 3$ weights: $w_x^{(t)}, w_h^{(t)}, w_y^{(t)}$. All the partial derivatives of the loss function $\mathcal{L}$ are computed, with respect to these weights. Finally, the contributions for all weights are summed, to obtain 3 mean updates for the weights.

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_x} &= \sum_{\tau=1}^{\tau=t} \frac{\partial \mathcal{L}}{\partial w_x^{(\tau)}} \\
\frac{\partial \mathcal{L}}{\partial w_h} &= \sum_{\tau=1}^{\tau=t} \frac{\partial \mathcal{L}}{\partial w_h^{(\tau)}} \\
\frac{\partial \mathcal{L}}{\partial w_y} &= \sum_{\tau=1}^{\tau=t} \frac{\partial \mathcal{L}}{\partial w_y^{(\tau)}}
\end{aligned} \tag{2.15}$$

The technique used is very similar to training of FFNN. Even if RNNs have only 3 weights, it is still needed to compute $3 \times t$ partial derivatives. The fact RNN has less weights is thus a advantage in time complexity during the forward phase, but not during the backward.

3.3 Exploding/Vanishing gradient

When the RNN has a lot of cells, it can encounter a big problem: exploding or vanishing gradients. We can illustrate the exploding and vanishing gradient problem with a simple example. [16] [13] Consider a deep RNN with over 100 cells. In order to apply backpropagation, we need to compute the partial derivatives. We try for every cell to compute:

$$\frac{\partial \mathcal{L}}{\partial h_{t-1}} = \frac{\partial \mathcal{L}}{\partial h_t} \frac{\partial h_t}{\partial h_{t-1}} \tag{2.16}$$

Accordingly to the equations governing an RNN cell, this equation can be rewritten as:

$$\begin{aligned} \frac{\partial h_t}{\partial h_{t-1}} &= \frac{tanh(w_h h_{t-1} + w_x x_t)}{h_{t-1}} \\ \implies \frac{\partial h_t}{\partial h_{t-1}} &= (1 - tanh^2(w_h h_{t-1} + w_x x_t)) * w_h \end{aligned} \tag{2.17}$$

As the derivative of $tanh(x)$ is $1 - tanh^2(x)$, it will always return a value between 0 and 1. When the weights are initialized with a normal distribution, the expected value of w_h is smaller than 1. The total value of the gradient will be smaller than 1. When applying backpropagation on all the previous layers, the gradients will be multiplied together. Because all the gradients are smaller than 1, the final multiplication of them will vanish. Hence, earlier layers will receive a very small update. The solution is not simply to take bigger weights, because in that case the gradient of the first layers will explode.

This problem can in theory also occur for FFNN, but only for deep networks with multiple hidden layers. The reason is the multiplication of the gradients of each layer. Consequently, the problem appears more often for RNN as they simply tend to have much more cells than FFNN. This is because RNN are used for long time series. To overcome this problem, the basic RNN cell has been updated to the long short-term memory cell.

3.4 Long Short-Term Memory (LSTM)

To counter the vanishing/exploding gradient problem, the long short-term memory (LSTM) model was defined in 1997 [12]. It is a recurrent neural network which has an intern memory to facilitate pattern recognition of long-term dependencies. LSTMs are widely used in forecasting problems. It has been proven that they obtain significantly better results compared to other RNN. Figure 3.5 shows an LSTM cell. The main difference in contrast to a normal RNN cell, is the added input and output cell states: c_t, c_{t-1}. These are sort of long-term memories that remember part of the past information, and forgets the other part.

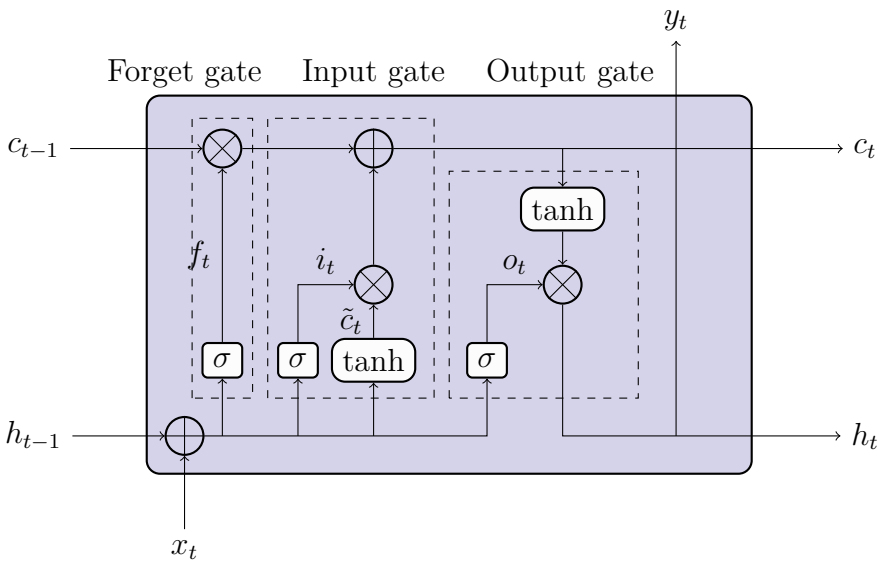


Figure 3.5: Long short-term memory cell

The equations governing this cell are written in equations 2.18 to 2.23.

$$f_t = \sigma(w_{xf}x_t + w_{hf}h_{t-1} + b_f) \tag{2.18}$$
$$i_t = \sigma(w_{xi}x_t + w_{hi}h_{t-1} + b_i) \tag{2.19}$$
$$o_t = \sigma(w_{xo}x_t + w_{ho}h_{t-1} + b_o) \tag{2.20}$$
$$\tilde{c}_t = \tanh(w_{xc}x_t + w_{hc}h_{t-1} + b_c) \tag{2.21}$$
$$c_t = f_t c_{t-1} + i_t \tilde{c}_t \tag{2.22}$$
$$h_t = o_t \tanh(c_t) \tag{2.23}$$

The LSTM cell is divided into 3 gates: forget, input and output gate. In these equations, σ represents the sigmoid activation function and tanh represents the hyperbolic tangent activation function. w are the weights, associated to each gate, and b are the corresponding biases.

3.5 Forget Gate

The forget gate's goal is to determine the amount of information coming from the long-term memory that will be kept. The rest is forgotten. This is given by the coefficient f_t. Coefficient f_t is computed with equation 2.18. The term $f_t c_{t-1}$ in equation 2.22 will apply this coefficient. This happens in the output gate.

3.6 Input Gate

Next, the input gate will determine how much new information is added to the cell state, c_t. This is done by equations 2.19 and 2.21. Later, equation 2.22 will increment the cell state. This is done after forgetting part of the information of the past cell state.

3.7 Output gate

Finally, the output gate will increment the hidden state, the output and the cell state. This is done respectively with equations: 2.23, 2.20 and 2.22.

3.8 Advantages for backpropagation

LSTM handles the vanishing/exploding gradient better than RNNs thanks to its gate mechanisms. These gates control the amount of information passing from cell to cell. During backpropagation, these gates also control the gradient flow.

In classical RNNs, multiplications are used between the weights. On the contrary, the equations modeling the LSTM involve summations of weighted inputs. This helps to maintain stability in the gradients during backpropagation. [13] When inputs are too large or too small, the summation operation in LSTMs tends to mitigate these effects.

4 Literature Review

The first artificial neural network in the literature was described in 1943. As mentioned before, the mathematician Walter Pitts and neurophysiologist Warren McCulloch wrote the article "A logical calculus of the ideas immanent in nervous activity"[20]. Fifteen years later, in 1958, the psychologist Frank Rosenblatt invented the perceptron[24]. His original perceptron is still used nowadays in a lot of neural networks. Deep neural networks are still in their infancy. Nowadays, a lot of researchers are studying this subject, applied in different fields. Unlike other domains, multibody applications will mainly handle with very complex, non-linear functions. This makes the multibody field different from other research domains like image and speech applications. For image and speech the patterns are often less complex, but they handle high-dimensionality instead. Multibody applications require robust and accurate solutions, which are optimized to be time-efficient.

4.1 Surrogate models in multibody applications

Surrogate models, also called black-box models, are mathematical models approximating the output of a system. Classical techniques like Bayesian learning exist, but neural networks are now emerging as an interesting, alternative solution for black-box modeling.

Ardeh et al.[3] constructed an FFNN to be used in a quarter-car vehicle model as a surrogate for the hysteretic, nonlinear tire model. Their research was motivated by two observations. First, tire models are inefficient in terms of computation time. Secondly, the tire model might not be available and only experimental data is accessible. Their surrogate model obtains high accuracy compared to the full, nonlinear simulation models. In this study, they focus on the efficiency obtained by the FFNN, while running on a CPU and GPU device. The FFNN is between 4 and 145 times faster than a classical mathematical model, depending on the integrator used. The

time efficiency in this article is impressive. This master's thesis will study if the same time efficiency can be obtained for a more complex railway vehicle model.

4.2 Inverse dynamics applications

Inverse dynamics problems are mainly researched in the field of robotic manipulators. Multiple authors emphasize that gathering data to train an NN is a very time-consuming task. It is also not always possible to obtain a large amount of data. For example, obtaining data of a train that is on the limit of derailment is very hard. It is not safe to obtain this with real-life information. However, we still need to be able to control the train in this part of the workspace. Multibody simulation can thus be used for this purpose.

Zhou and Schoellig[30] and later Ren and Ben-Tzvi[23] searched for solutions to these problems. They wrote about their adaptive approach for data gathering in the particular case of robot dynamics. In most DL applications, training data is assumed to cover the operational space sufficiently. The approach is then trial-and-error based. A training dataset is generated, the network is trained, and this process is started over again if the achieved performance is not satisfying enough. Zhou and Schoellig introduced an episode-based algorithm that can extrapolate data to parts of the workspace that are harder to reach. It also helps the NN to train, even if the data is not distributed homogenous in the operational space.

4.3 Forward dynamics applications

DL applications in forward dynamics mostly involve time-varying dynamic responses of complete or part of multibody systems. This can be done with NN instead of using the classical approach by solving the equations of motion. Once more, the main goal is increasing time efficiency and modeling unknown relations between input and output.

Choi et al.[4] showed the results they obtained by replacing multibody equations with a DNN. They use FFNN to predict the dynamic behavior of multibody applications. They obtained good results for 3 models: damped pendulum, double pendulum, and slider-crank mechanism. Despite the models being quite simple, the obtained smoothness of the solution is impressive for displacements, velocities, and accelerations. Their work shows the importance of choosing the range of input and output parameters. A wide range increases training dataset, and thus training time. However, the predictions made on a trained model but outside its range, are very bad.

In 2021, De Rosa et al.[5] published their study, where they trained an AI model to monitor lateral and cross-level track irregularities. They opt for binary classification. In the first class, the track geometry is acceptable. The track there does not need further maintenance. Where in the second class, the track geometry surpasses some thresholds and needs maintenance. They use two common ML techniques: decision tree and support vector machine. In order to obtain their dataset, they use real

data. In cases where there is no fault, but they use simulation results for situations where the defects are big. Indeed, in these cases a train could derail, it is thus nearly impossible to obtain thousands of real-life data. This article demonstrates very well the coupling between real-life measurements and multibody simulation measurements in order to augment the training dataset needed for AI.

4.4 Physics informed Neural Networks

Physics informed AI are models that use AI, but combined with real physics. For DL, physics-informed neural networks are trained, while respecting laws of physics. These laws are described with differential equations. Physics-informed NN (PINN) are very new in research. Even if PINN are a bit outside the scope of this master's thesis, they are worth mentioning. Future research in that particular field will be necessary to couple multibody modelling and AI fully.

Raissi et al.[22] are some pioneers of PINN. They published their work in 2019. The global structure of a PINN is the same as for a classical FFNN. The difference lies in the loss function. For classical NN, only the difference between predicted output and true output is considered. PINN it will also consider the loss with reference to the laws of physics. The errors are called data-driven loss and physics-informed loss. During the training of the neural networks, both are used. The conclusion of the article is that PINN are still in an early stage, but are very promising. Their goal is not to replace classical methods for solving differential equations, but to co-exist with them. The advantage is once more the computation time and modeling unknown relations between input and output data. The difference lays in the accuracy obtained compared to traditional NN.

In the same year, 2019, Lutter et al. published their article "Deep Lagrangian Networks: Using Physics as Model Prior for Deep Learning"[18]. Deep Lagrangian Networks apply lagrangian mechanics to FFNN. The network is trained with a classical loss function and a metric that represents the violation of the physical laws described by Lagrangian mechanics. Lutter et al. conclude that their Deep Lagrangian NN has shown the ability to learn the underlying physics of a system from sensor data and recover inertial, gravitational, and centripetal forces.

Chapter 3

Model

This section will tackle the railway model provided by Quandyga Engineering[1]. It is a railway vehicle based on the Manchester Benchmark [14]. The multibody model of this train was created on the Robotran software. This section will first describe how the Robotran software works, and will then introduce the actual train model.

1 Multibody formalism

The dynamic model of the train is based on the multibody formalism used by Robotran software [6][7]. This multibody formalism is used to derive the equations of motion for multibody systems. This section is based on the 'Robotran basics' document[2], where more detailed information is available.

The system is first described as a multibody system. This is an assembly of rigid bodies which are connected together with rotoid and prismatic joints. The structure is tree-like, without cinematic loops. If the model has some cinematic loops, these are cut. The equations of motion are written with equation 3.1, without any constraints. In order to add the constraints given by the user, equations 3.2, 3.3 and 3.4 are needed.

$$M(q)\ddot{q} + c(q,\dot{q},g) = Q(q,\dot{q}) + J^T\lambda \tag{3.1}$$

$$h(q) = 0 \tag{3.2}$$

$$\dot{h}(q,\dot{q}) = J(q)\dot{q} = 0 \tag{3.3}$$

$$\ddot{h}(q,\dot{q},\ddot{q}) = J(q)\ddot{q} + \dot{J}\dot{q}(q,\dot{q}) = 0 \tag{3.4}$$

Where:

- M is the symmetric generalized mass matrix of the system.
- c is the nonlinear dynamic vector that contains the gyroscopic, centrifugal, and gravity terms as well as the contribution of external resultant forces, frc and torques, trq.
- q denotes the relative generalized coordinates.
- $\dot{q}$ denotes the relative generalized velocities.

[1] http://www.quandyga.com/

[2] https://www.robotran.be/docs/documentation/

- $\ddot{q}$ denotes the relative generalized accelerations.
- $Q*$ represents the generalized joint forces (torques).
- $J = \frac{\partial h}{\partial q^t}$denotes the constraint Jacobian matrix.
- λ represents the Lagrange multipliers associated with the constraints.
- h denotes the algebraic constraints resulting from the cinematic loops.

Various methods can be used to solve the set of differential algebraic equations (DAE). Robotran uses a full index reduction of the system to a purely differential form, which can be obtained by means of the so-called coordinate partitioning method.

In this way, it becomes possible to reduce the original DAE system to a set of ordinary differential equations (ODE) with independent coordinates. This reduced set represents the equations of motion of the constrained multibody system. The coordinates are separated into two partitions: dependent coordinates (q_v) and independent ones (q_u).

The dependent coordinates can be expressed in terms of independent coordinates with means of nonlinear algebraic equations ($h(q) = 0$). The obtained system is then:

$$M_r(q_u)\ddot{q}_u + F_r(\dot{q}_u, q_u, Q_u) = 0 \tag{3.5}$$

Robotran uses a symbolic approach to automatically generate the DAE system, as well as matrix and vectors to obtain the reduced system 3.5. The temporal integrals are done by numerical libraries in C.

2 Railway vehicle

The model of the railway vehicle used in this work was provided by Quandyga Engineering. It is based on the Manchester Railway Benchmark [14].

The railway vehicle from the benchmark is a simplified model of a passenger coach. The hierarchy of the vehicle is depicted on figure 2.1.

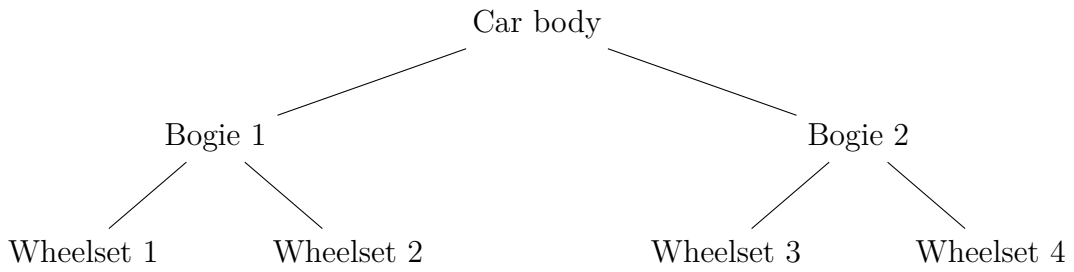


Figure 2.1: Hierarchy of the railway vehicle based on the Manchester Benchmark.

The suspensions between the wheelsets and bogies are called 'primary suspensions', while the ones between bogies and car body are 'secondary suspensions'.

All the model characteristics can be found in the Manchester Benchmark[14]. Only the ones that are very important for this master's thesis are listed in table 3.1.

Characteristic	**Value**
Mass car body	32 000 kg
Primary suspension vertical stiffness	1220 kN/m
Secondary suspension vertical stiffness	430 kN/m
Primary suspension vertical damping	1 kNs/m
Secondary suspension vertical damping	5 kNs/m

Table 3.1: Railway vehicle parameters.

In default configuration the train runs on a perfectly straight track, without track cant. It is possible to add track defects on top of this ideal track. When the train encounters irregularities or imperfections on the tracks, its natural frequencies are excited. The natural frequencies of the vehicle are described in table 3.2. This data is coming from [19]. Note that all car body natural frequencies are below 3 Hz, while all bogies' natural frequencies are above 7 Hz.

Mode	**Frequency [Hz]**
Car body	
Lower sway	0.56
Yaw	0.84
Bounce	1.07
Upper sway	1.27
Pitch	1.28
Longitudinal	2.85
Bogies	
Bounce in phase	7.47
Bounce out of phase	7.48
Lower sway out of phase	9.68
Lower sway in phase	9.70
Pitch out of phase	11.68
Pitch in phase	11.69
Upper sway out of phase	12.64
Upper sway in phase	12.64
Yaw out of phase	35.14
Yaw in phase	35.17

Table 3.2: Natural frequencies of the railway vehicle. Images illustrating these modes are visible in appendix 1.
Source: Solution Of Universal Mechanism Manchester Benchmark Results[19]

3 Programming Language

Two programming languages are used in this work. C for the modeling of the railway vehicle, and Python for coding neural networks.

3.1 Railway vehicle

The railway vehicle is fully coded in C/C++, with Robotran software. C/C++ is chosen as it is very computational efficient.

3.2 Neural Networks

All the neural networks in this master's thesis are coded in Python. The used library is called 'Keras', and runs on top of Tensorflow. Keras has the reputation of being fairly high-level, easy to use, and with fully automated training. It is important to note that while Keras is written in Python, it is integrated with C and C++ backends for low-level computations. This makes it more efficient, while keeping the simplicity of Python in front-end. A more detailed analysis of the various possible libraries for implementing neural networks can be found in Appendix 2.

In order to be able to compare the results of predictions made by the neural networks and results from Robotran, we need both to be evaluated in the same programming language. For this purpose, the neural network obtained in python will be saved in a file with extension .h5, which is specifically made for TensorFlow. The TensorFlow API in C will be able to open this network and make predictions. More detailed information about the TensorFlow .h5 files and the TensorFlow API in C can be found in Appendix 2.

Chapter 4

Predicting vertical car body accelerations

4.1 Objectives

Classical multibody software, such as Robotran, solves the equations of motion. In this chapter we test a different approach to predict vertical accelerations of a railway vehicle, using neural networks. While solving equations of motion is a relatively slow process, neural network's biggest advantage would be time efficiency. However, this is at the cost of a loss in accuracy. This will be covered in this chapter.

This study will focus on vertical accelerations of the car body, when the train is running on vertical track irregularities. When analyzing track defects on real trains, studies will mainly focus on vertical wheelset accelerations. The link between the wheelset and the track profile is more straightforward, as the wheels always stay in contact with the rails. However, for the periodic defects implemented, the wheelset accelerations are very periodic too. If one period is known, simply copying this period will already achieve good accuracy in predicting the acceleration over time. Our deep learning approach does not simply copy periodic signals; it also tries to predict how the vertical acceleration of the car body will behave over time. The amplitudes and frequencies of the track irregularities will play a significant role. The second reason of focussing on car body accelerations is that they are an important indicator for security (derailment) and comfort of railway vehicles.

The neural network implemented is an adapted combination of LSTMs and FFNNs. LSTMs are used since the vertical accelerations are time series. The FFNN is used to incorporate additional information about the track profile. This chapter describes first how the data was gathered and preprocessed. Then, the architecture of the NN is explained, as well as the hyperparameter tuning. Moreover, the achieved training phase is showcased and the obtained results are discussed.

4.2 Data Collecting and Preprocessing

In order to obtain data, the virtual railway vehicle model in Robotran is used. The vertical car body accelerations are saved. The track vehicle is used with its default configuration, as described in chapter 3. The speed of the train is set to $90km/h$

$(25m/s)$ during the whole experiment. This value is chosen according to 'track case 4', described in the Manchester Railway Benchmark [14]. At the start of the simulation, the train is at dynamic equilibrium, it is not accelerating and has maintained a constant speed for an adequate period to eliminate any vertical velocities or accelerations.

4.2.1 Track irregularities

The track is by default an ideal straight track. During each simulation, a rail defect is added to this ideal track. This irregularity will take the form of a sinusoidal vertical elevation, denoted h. The irregularity is identical on both the left and the right rail, which result in the same vertical acceleration on the right and left suspensions of the vehicle. The car body does not undergo roll. The track irregularity stays on the track until the end of the simulation. Defects are chosen randomly using a normal distribution. The range for amplitude and wavelength is described in table 4.1. Only these two parameters are set for each simulation. For clarity, frequency was also added in table 4.1, obtained by equation 4.1. Both amplitude and wavelength were rounded to 4 significant digits.

$$frequency[Hz] = \frac{velocity}{wavelength} = \frac{25m/s}{wavelength[m]} \tag{4.1}$$

	Amplitude [mm]	Wavelength [m]	Frequency [Hz]
Minimum Value	-3	0.75	33.33
Maximum Value	3	25	1

Table 4.1: Vertical track irregularities description

The simulation runs for 20 seconds. In this timespan, the train covers a distance of $500m$. Through this chapter, the graphs will focus on 3 cases, shown in figure 4.1. They are chosen among the 50000 simulations, to illustrate the behavior of the NN.

Track defects	Amplitude [mm]	Wavelength [m]
Track case 1	1.04	6.141
Track case 2	0.551	11.01
Track case 3	2.19	4.391

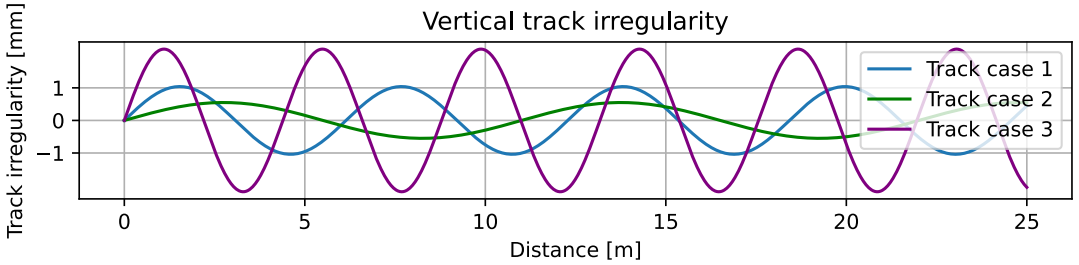


Figure 4.1: Track profiles with vertical irregularities.

In this study, data was only retained from simulations where the train did not derail. A derailment is indicated if any wheel loses contact with the rail, at which point the simulation is terminated. The pairs of amplitude and wavelength values for which the train remained on the track throughout the entire simulation are shown in figure 4.2. In total 100 000 Robotran simulations were performed, and 53 408 were retained. The left and right y-axis are respectively frequency and wavelength. They provide the same information, in a different scale, according to equation 4.1. The horizontal lines are the different natural frequencies of the system, as listed in section 3.

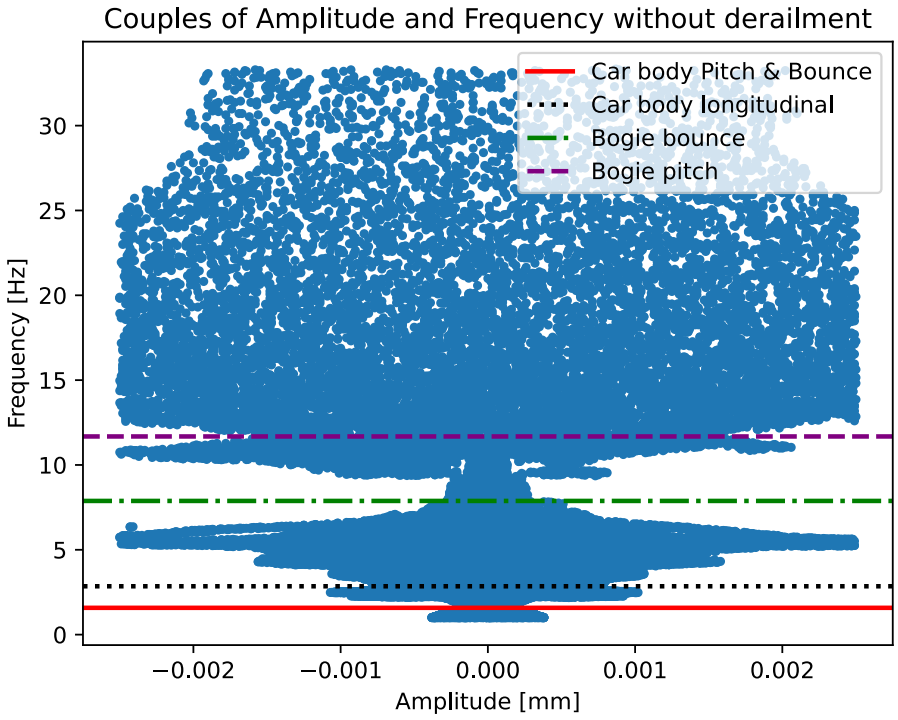


Figure 4.2: Track defects where the wheel-rail contact is not lost. The horizontal lines are the natural frequencies of the railway vehicle system.

Note that the upper part of the graph is less dense. This is because the track defects were chosen randomly with a normal distribution in the wavelength range, and not in the frequency range. This was done because this keeps the model more 'global'. The wavelength of the track defects does not change with the train speed. In contrast, the velocity of the train has an impact on the perceived frequency of the track defects.

As expected, the various natural frequencies of the model significantly impact derailment. This is particularly evident at the natural frequency of bogie bounce, which is $7.48Hz$. At this frequency, derailments occur more frequently, resulting in less available data.

4.2.2 Vertical Accelerations

Vertical accelerations of the car body are observed, over a timespan of 20s. They are saved with a timestep of 5 milliseconds. We limit our analysis of car body

accelerations to timespans of 20 seconds for two main reasons. Firstly, over longer periods, most car body accelerations tend to be damped and approach zero. Including numerous zero values in our dataset can degrade the performance of the neural network. Since neural networks compute weighted sums, entries of zero lead to weights that can take any value, making it impossible to train these weights effectively. Therefore, the best approach is to remove such inputs.

Secondly, not all car body accelerations reach zero. Some remain undamped and become very periodic. The goal of our network is to predict the evolution of these signals over time and determine how they will behave. Our aim is not simply to predict a periodic function, which would result in replicating the period.

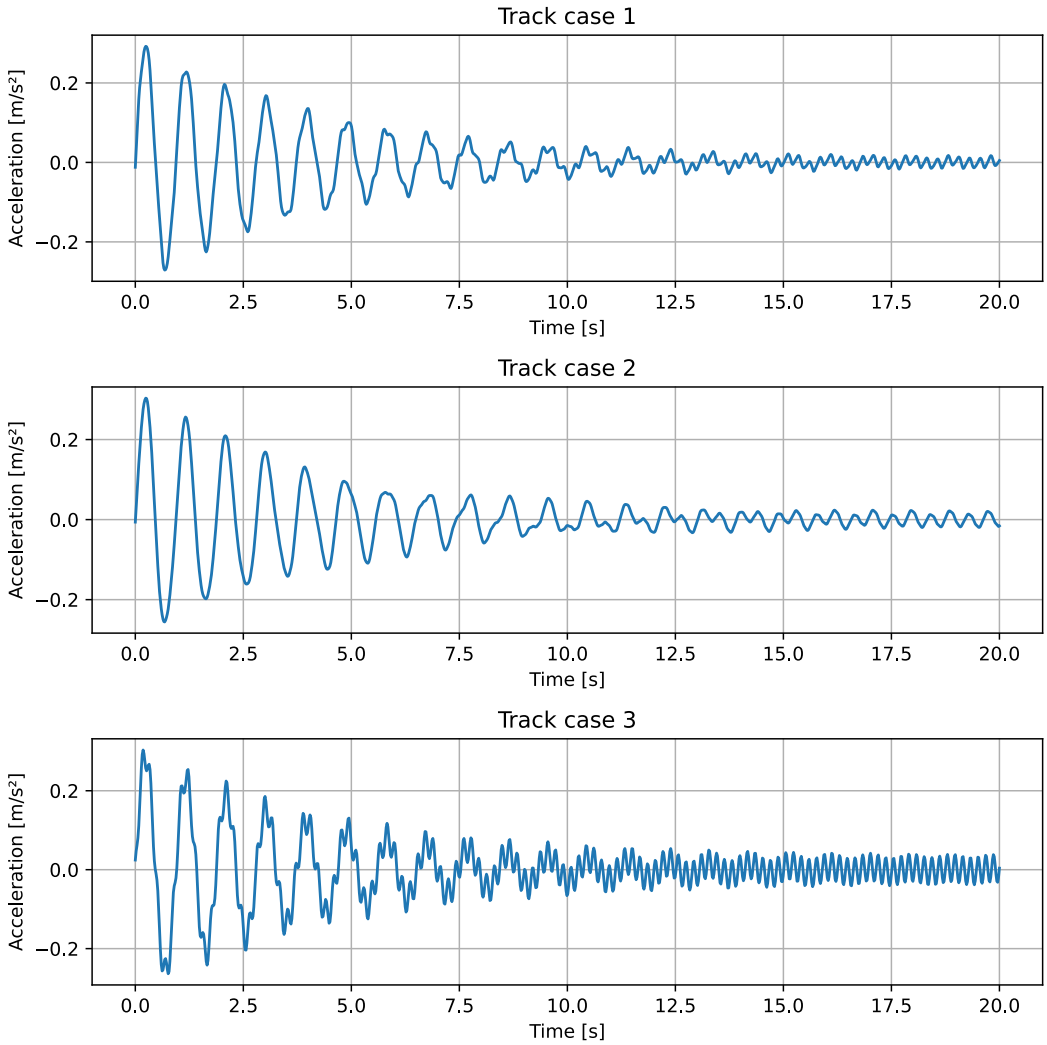


Figure 4.3: Example of vertical accelerations of the car body, for the track cases described in figure 4.1.

Figure 4.3 shows the vertical car body accelerations obtained when the train is running on the track cases described before, in figure 4.1. The first one is the most representative of the entire dataset, as it is quite similar to most samples. The

vertical acceleration is made of a low frequency and some higher frequencies part. The second and third ones are more different. Indeed, we observe almost only low frequencies for the second one, and lots of high frequencies for the third one.

In order to understand the data even more, we can look at the power spectral density (PSD) of the vertical accelerations. Figure 4.4 shows the PSD of the same three accelerations as figure 4.3, expressed in $\frac{m^2}{s^4 Hz}$.

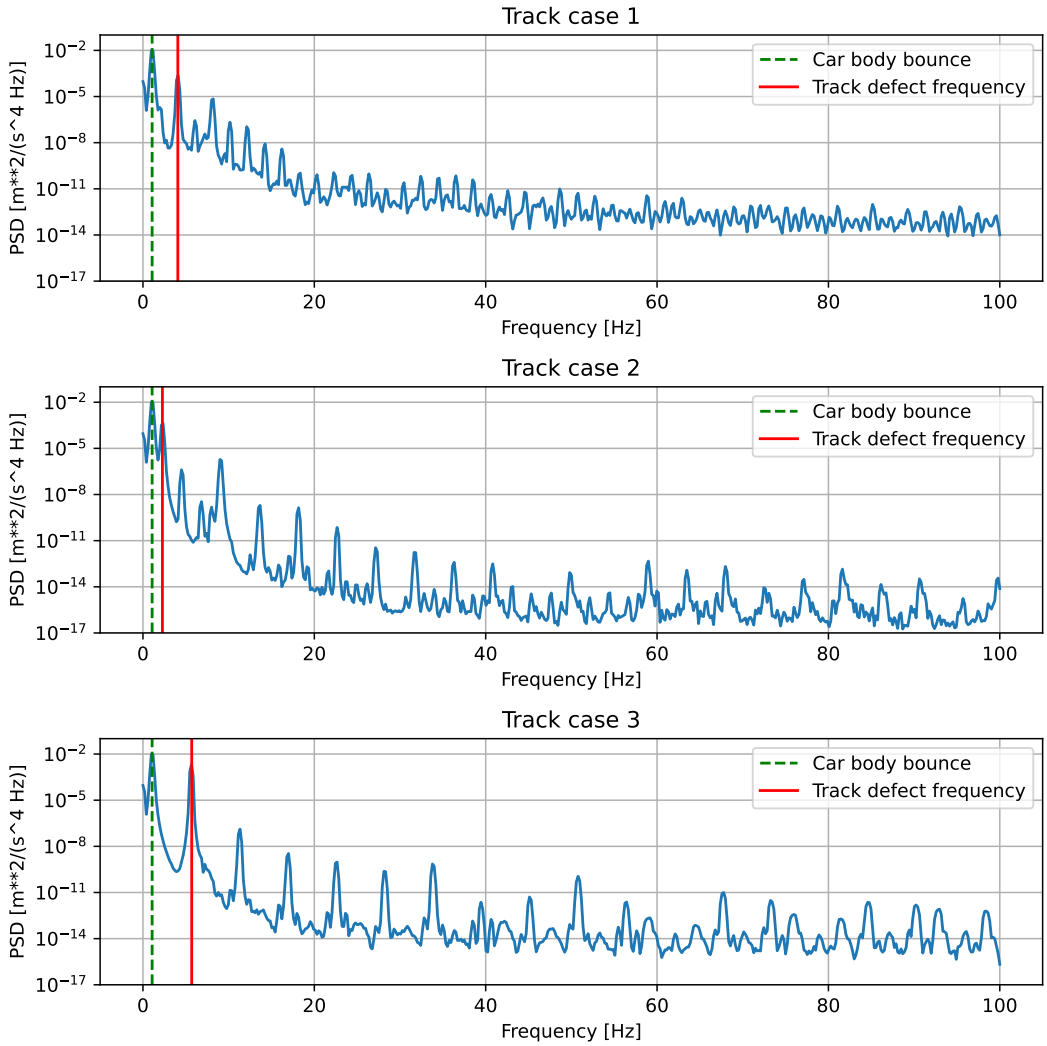


Figure 4.4: PSD of the vertical accelerations.

We observe all signals have a low frequency peak, at the natural frequency of the car body bounce. This frequency is also visible in figure 4.3, as the biggest amplitude has the lowest frequency. There is another peak in every PSD at exactly the frequencies of the track defects. This frequency will always be between 1 and 33Hz, as this is the frequency range of the defects.

4.2.3 Train, validation and test dataset

The total dataset consisting of 53 408 samples is split into a train, validation, and test dataset. They are respectively denoted $S_{train}, S_{validation}$, and S_{test}. S_{train} will be used during the training phase, to update the weights of the neurons. $S_{validation}$ will be used for early stopping. At every epoch, after the update of the weights, the predictions are made on $s_{validation}$. When the accuracy decreases on this dataset, the model starts overfitting. At that moment the training is stopped. Finally, S_{test} is used to evaluate the network performances once all the training is done. It is important to keep this data completely unseen for the network during training. Indeed, we want to see how it performs to capture global patterns, not to reproduce the data it saw during training. S_{train} needs to have enough samples to be representative of the complete dataset. 70% of the entire dataset is used for training, which represents 37 386 samples. Both $S_{validate}$ and S_{test} have 8011 samples, which is 15% of the entire dataset.

4.2.4 Input and output of the network

In order to train a network, we need to split the data into input, output, and feature(s). In this case, we want to predict the next acceleration at time $t + 1$. This is the output. The input will be the acceleration profile over the last lb seconds, thus starting at time $t + 1 - lb$ seconds. lb is called the look-back parameter, expressed in seconds. When the parameter lb is set to a higher value, it's like giving the network a longer memory. It can then consider events from further back in time when making decisions. The input feature will be the profile of the track defects, starting from $t + 1 - lb$ seconds, but going until $t + 1$ seconds. In other words, the track profile is known at the point we want to estimate the acceleration. The track profile will be referred to as h. The network can be globally represented as figure in 4.5.

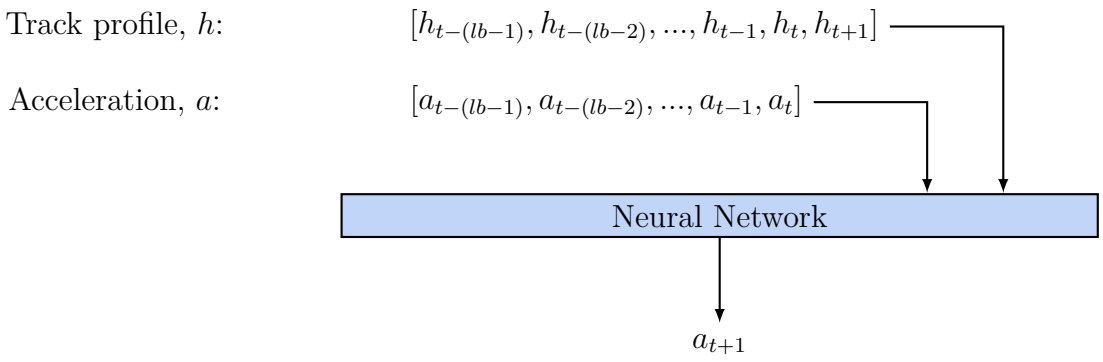


Figure 4.5: Global view of the model needed to predict the vertical acceleration at time $t + 1$.

In order to split the data into input and output accelerations, a technique called 'rolling window' is used. This principle is illustrated on figure 4.6. The process divides the input data into chunks. In every chunk, the first lb points are inputs, the last one is the output. The length, l_{chunks}, of the chunks, as well as the number of chunks, n_{chunks}, depends on the value of the look-back parameter. This is described

in equations 4.2 and 4.3.

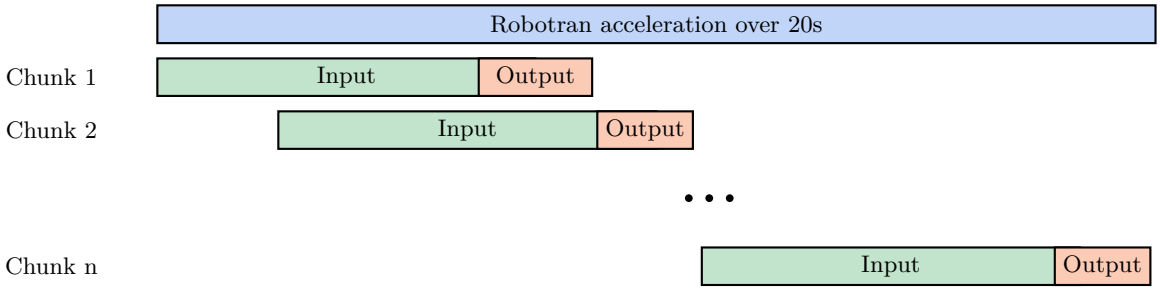


Figure 4.6: Rolling window principle.

$$l_{chunks} = lb + 1 \tag{4.2}$$

$$\begin{aligned} n_{chunks} &= n_{\text{samples}} \times n_{\text{chunks per samples}} \\ \Leftrightarrow n_{chunks} &= n_{\text{samples}} \times (n_{\text{timesteps}} - lb) \end{aligned} \tag{4.3}$$

To train the network more efficiently, min-max unity-based normalization is applied to the input and output data. This maps the input and the output linearly into the range $[0, 1]$. This is done by applying formula 4.4. X denotes the true input, as saved from the simulation. X' is the mapped input, which will be given as input to the network. X_{min} and X_{max} are respectively the minimum and maximum of the true input. Scaling helps to set up the weights, as the range is known for input and output values. It is also needed to give the same importance to all inputs. Otherwise, big amplitudes will overwhelm the entire network by forcing the weight to be very small in order to avoid exploding outputs.

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{4.4}$$

4.3 Neural Network Construction

In neural networks, it can be hard to model precisely sequences of different amplitudes. For this cause, the data is scaled in the range $[0, 1]$. However, using 1 network consisting of LSTM and dense layers did not give good results. The network always returned mean acceleration, oscillating around zero. The outputs had very small amplitudes. One solution is to split the neural network into 2 networks. One for very big amplitudes, thus low frequencies. The other one is for all higher frequencies.

The filters used for filtering the input data into a low and high frequency part are Butterworth filters of 1st order. The filtered data for the first track case is visible in figure 4.7. The cutoff frequency is set at 2.9Hz. This value was found optimal during hyperparameters optimization.

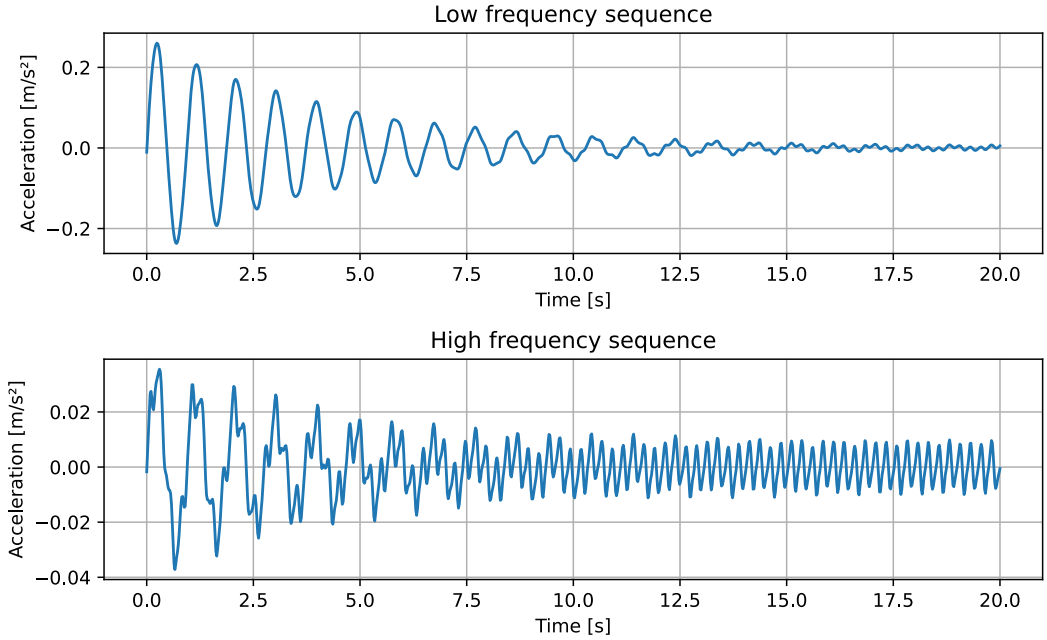


Figure 4.7: Filtered data for track case 1.

4.3.1 Architecture

The final architecture of the model for predicting vertical car body acceleration is visible on figure 4.8. The network is divided into two parts. The upper part is for preprocessing. It includes data scaling and filtering. The lower part is the true neural network part. It consists of 2 networks. The left one is for low frequency signals, the right one is for high frequency. The inputs for both networks are previous accelerations and the track profile. The first advantage of splitting the network into 2 parts, is that the look-back parameter lb can be different for low and high frequency. The one for low frequency will have a higher value. Indeed, we need to look further in the past to obtain the same information (periods) as for higher frequency. The second advantage of this separated structure is time efficiency while training. Indeed, it is possible to train and evaluate both sub-networks independently. Parallel training is thus possible.

The lower part of the networks are the true neural networks. As the inputs are time series (accelerations), LSTM layers are needed. Some dropout layers are put between the different LSTM layers. During training, they will cut a number of connections, given by their dropout ratio. Cutting these connections means that the network loses temporarily some parameters. This may look bad, but is an important technique to avoid overfitting. The network will have to retain only the 'global patterns', and cannot remember all details of the training data. At the end of each sub-network, there are two dense layers. These layers will add the input of the railway profile at time $t + 1$, because the input units of the LSTM layers need to have the same input shape. This shape is two, consisting of the acceleration and track profile at each timestep. However, at timestep $t + 1$, the acceleration is not known. The input shape is therefore one. Thus, those values needs to be added to the network after the LSTM layers.

The last layer of the network will simply add the outputs together of the low and high frequency networks.

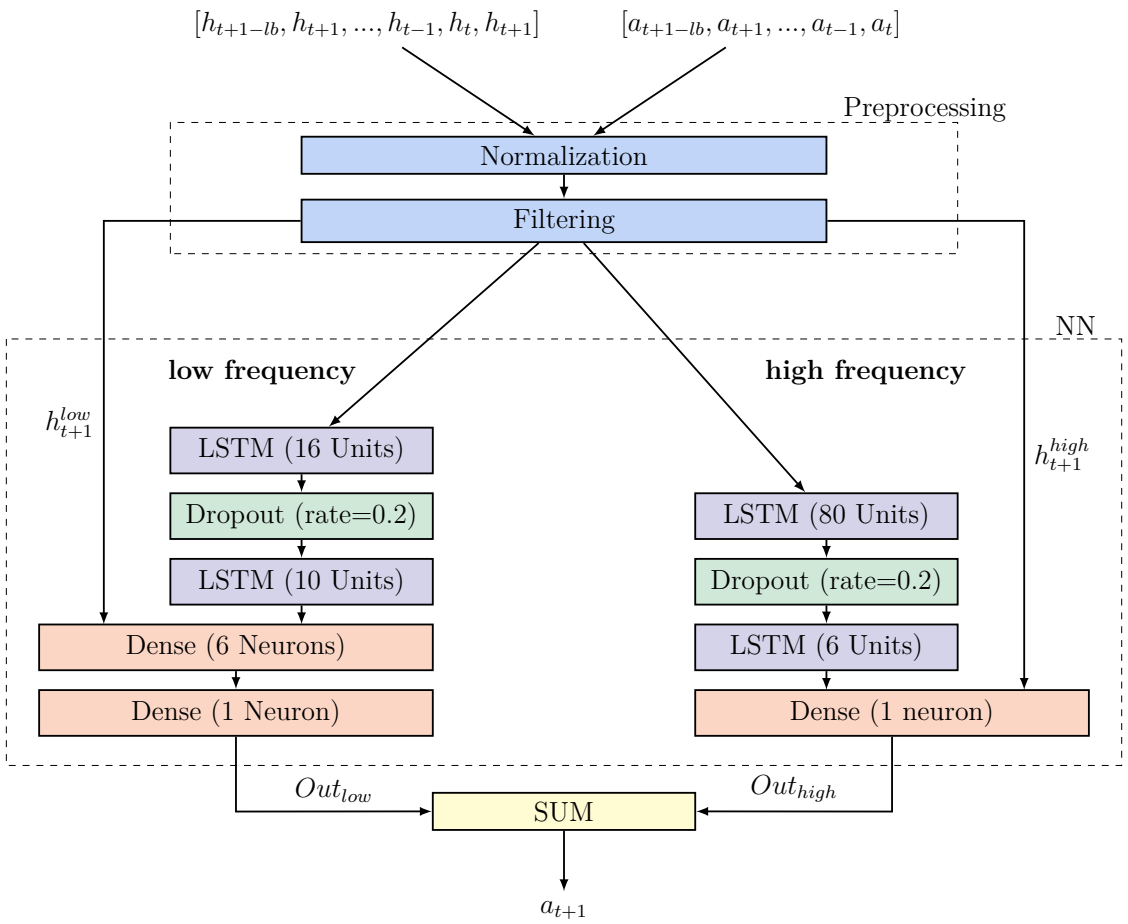


Figure 4.8: Neural network architecture

4.3.2 Hyperparameter tuning

In order to find a pseudo-optimal set of hyperparameters, particle swarm optimization (PSO) was used multiple times. The first times were to observe how many layers were needed, with a raw approximation of the number of neurons in each layer. Afterward, it was used to tune the number of neurons in each layer more precisely. Even if PSO only finds a pseudo-optimal maximum, the time gained compared to gridsearch is a huge advantage. The different hyperparameters tuned and their ranges are resumed in tables 4.2. The number of maximum iterations has been set at 25. There were 10 particles in the swarm. The total time needed for hyperparameter tuning is approximately 7 days. However, the accuracy of the model gained during the last 4 days is less than 0.6%. Thus, without having a significant impact on the final accuracy of the model, PSO could have been executed in 3 days.

Preprocessing	Min value	Max value	Optimal value
Cutoff frequency[Hz]	1	8	2.96

low frequency	Min value	Max value	Optimal value
Dropout rate	0	0.5	0.21
Number of LSTM layers	0	5	2
Number of Dense layers	0	5	2
Number of units LSTM layer 1	5	500	16
Number of units LSTM layer 2	5	500	10
Number of neurons Dense layer 1	1	20	6
Batch size	200	8000	1121
Initial learning rate	0.001	0.01	0.008
Lookback parameter (lb) [s]	0.005	2	1.4

high frequency	Min value	Max value	Optimal value
Dropout rate	0	0.5	0.17
Number of LSTM layers	0	5	2
Number of Dense layers	0	5	1
Number of units LSTM layer 1	5	500	80
Number of units LSTM layer 2	5	500	6
Batch size	200	8000	456
Initial learning rate	0.001	0.01	0.007
Lookback parameter (lb) [s]	0.005	2	1

Table 4.2: Hyperparameters tuning

The cutoff frequency coming from PSO was found at 2.96Hz. As expected, it cuts off the lowest frequencies with bigger amplitudes. We observe that all car body natural frequencies are lower than this cutoff frequency. On the contrary, all natural frequencies of the bogies are higher than the cutoff frequency. No relation between this cutoff frequency and the frequencies of track irregularities was observed. When comparing the PSD 4.4 and this cutoff frequency, we observe this will cut off the huge amplitude peak due to car body bounce. Sometimes, the peak due to the track defects will also be in the 'low frequency' part, but not always. We expect the low frequency signal to be similar for all data samples. This is because they will all have a power peak at the car body bounce natural frequency, even if the power of this peak is not exactly the same for all samples.

Both networks have two LSTM layers. However, the high frequency part requires 80 LSTM units for the first layer, in contrast to only 16 units for the low frequency network. This is because it needs to capture more complex patterns, as the data samples for high frequency are more varied.

As expected, the lookback parameter is lower for the high frequency network. However, the difference is only 0.5 seconds, even if the work frequencies are completely different.

4.3.3 Training

The metric used for training is a mean squared error (MSE). Early stopping was implemented with a patience of three. This means that if the loss of the validation

dataset increases during three consecutive epochs, training is stopped. The loss function for the training and validation datasets is visible in figure 4.9. This training is the one performed with the optimal hyperparameters. The y-axis gives the MSE, applied on respectively the input of the low and high frequency model. The input is not the raw data directly coming from the multibody simulation, but it is the normalized and filtered data. The x-axis gives the number of epochs, this is the number of times the entire training dataset was fed to the network. We observe the first epochs have a big impact, as the weights are set randomly at the beginning. The training loss and validation loss are very close. This is good, as it means the patterns captured through training can be applied on the validation dataset. We do not overfit the data. In the case of overfitting, the validation loss would be significantly higher than the training loss. Fewer epochs are needed to train the high frequency part. One of the reasons is the smaller batch size. The weights of the network are updated more frequently during one epoch, which results in fewer epochs needed. However, the number of epochs also depends on the number of parameters in the model, as well as the random initializing of these parameters and the complexity of the predictions.

Training on GPU for the low frequency part takes about 95 seconds per epoch, while the high frequency model takes on average 198 seconds per epoch. Training on CPU takes approximately 3 times more time.

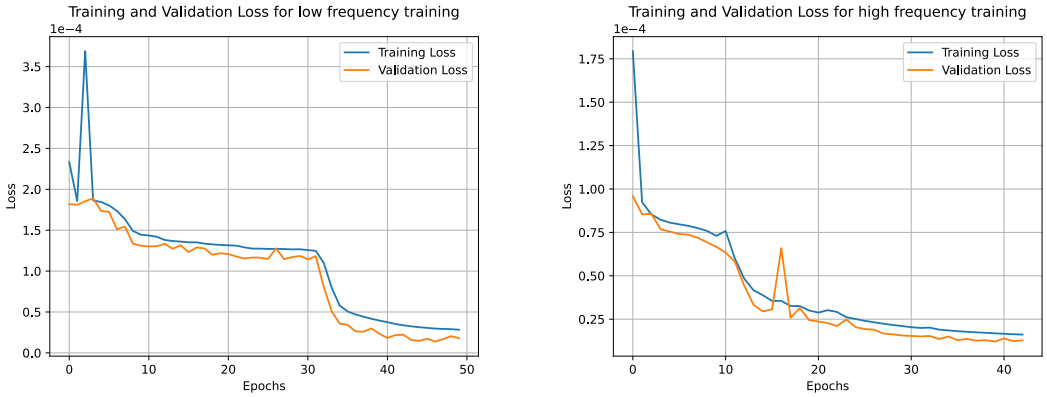


Figure 4.9: Training and validation loss for low frequency.

4.4 Results

4.4.1 Accuracy

The results for the low and high frequency parts of the three samples discussed earlier are visible in respectively figure 4.10 and figure 4.11. The blue lines are the values we try to approximate. They were obtained with the Robotran simulation, and are further referred to as the true output. Two predictions were made on each sample with each neural network. The first lb seconds have perfect predictions, as these are given as input to the network at the first iteration.

For the first predictions, the input at the first predicted timestep has only Robotran data as input. But as we go further, the inputs are the values predicted by the

neural network itself at previous iterations. In other words, the errors made at one timestep will propagate through time, as the predicted outputs are the inputs for future timesteps. These predictions are studied in this chapter and visible on figures 4.10 and 4.11.

For the second one, the input acceleration is only made of data coming from the Robotran simulation. This applies to the predictions at every timestep. An error made at one timestep will not affect the rest of the simulation, as the input data only rely on Robotran values. Indeed, the predictions are better. These predictions are shown in Appendix 5.

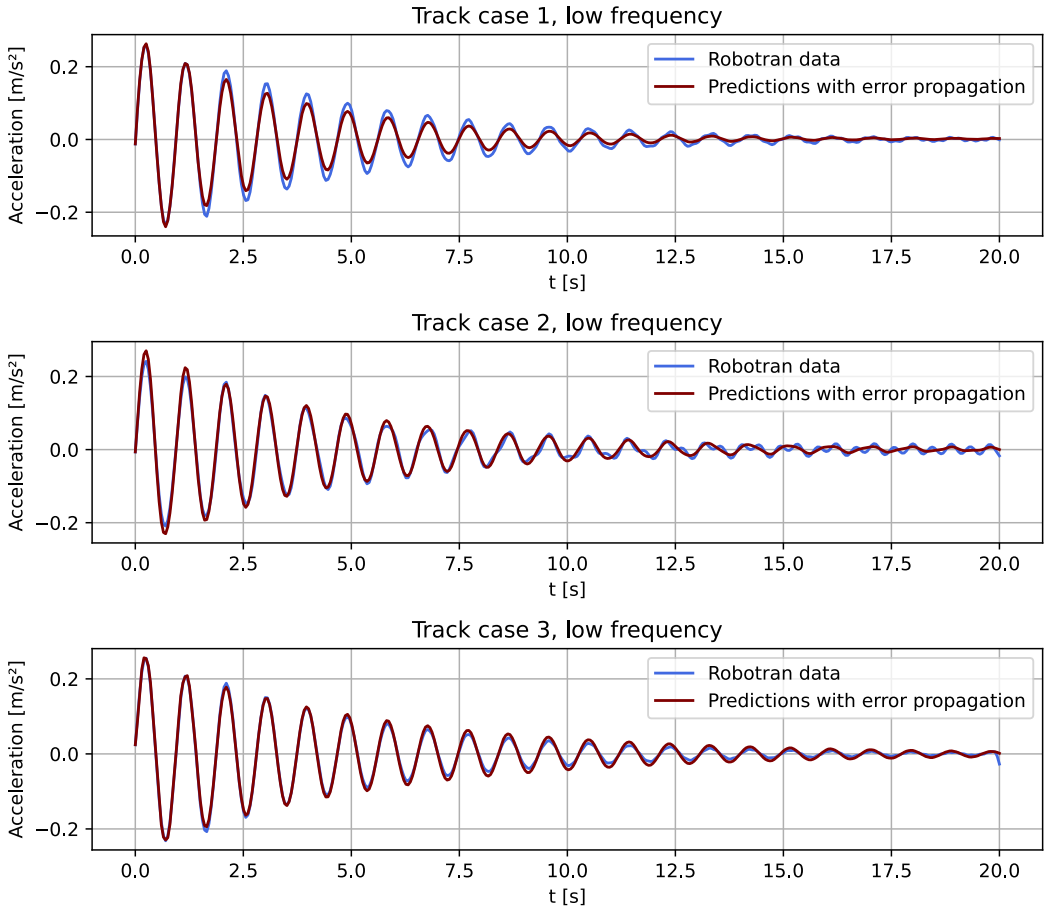


Figure 4.10: NN predictions for low frequency model.

As expected, the low frequency outputs are all quite similar one to each other. The predictions look good. However, the network tends to not be able to predict residual higher frequencies part. Using higher-order filters could have smoothed the signal more and avoided this problem.

The results for the high frequency part are more different one to each other. This also results in worse predictions. It is easy to observe that the very high frequency of track case 3 is hard to approximate with the network.

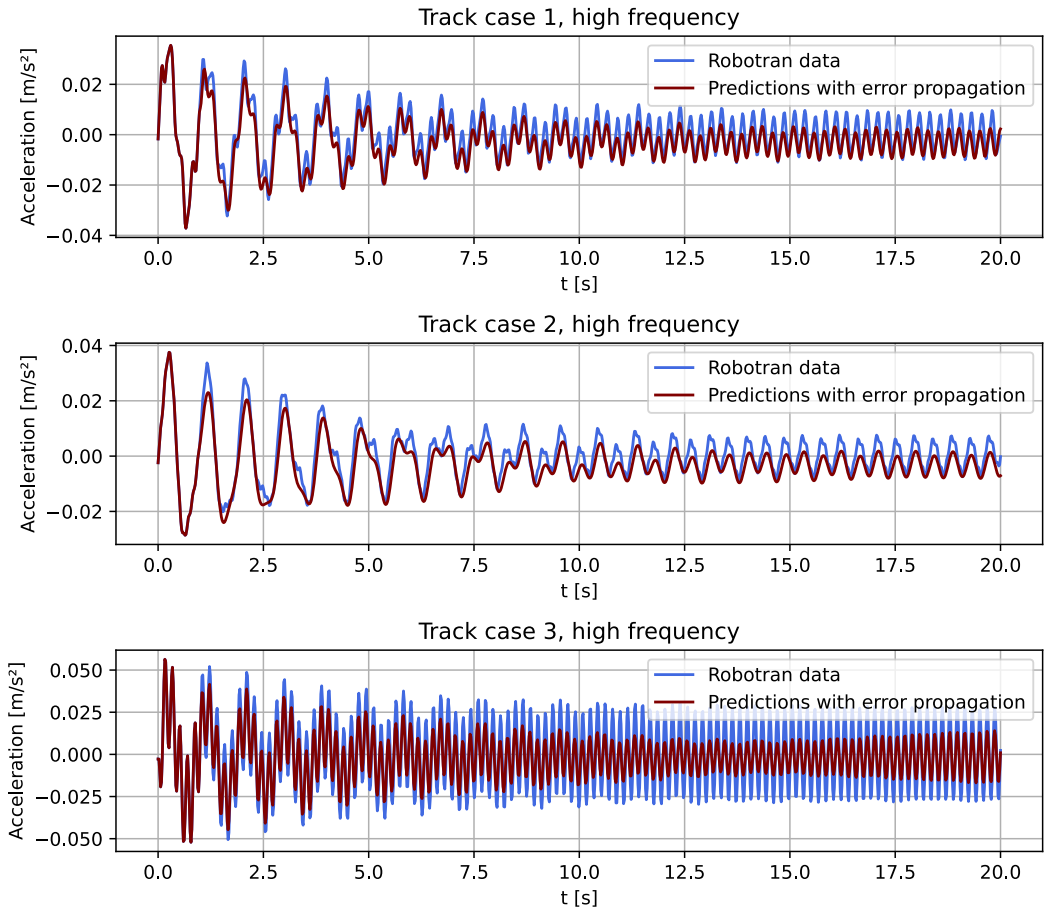


Figure 4.11: Neural network predictions for high frequency model.

In order to have a more objective metric, we look at the errors on the test dataset. The test data is completely unseen for the network. We compute the errors for both the low and high frequency models, as well as for the sum of both, which is the output of the complete model. Note that the term 'error' is further used in this chapter when comparing the results obtained from the Robotran simulation and the ones from the NN approach. This is under the assumption that the Robotran simulation outputs the 'true' outputs.

In order to be more easy to interpret, mean absolute error (MAE) is first computed. Next, we compute the root mean squared error (RMSE). In both metrics, 'mean' refers to taking the mean on all data points (at each timestep), inside a sample. We also want to know how the networks are performing, relative to the true outputs. Therefore, a relative metric is needed. We cannot simply divide the absolute metrics by the true output, as the true outputs contain a lot of (almost) zero values. This would result in errors or very high values, as we would divide by almost zero. Hence, we will compute normalized root mean squared error (NRMSE). The normalization happens by dividing the RMSE by the range of the 'true' data. All metrics are defined mathematically in equation 4.7. The obtained errors are written in table 4.3. 'mean' in the table refers to the mean of all the samples.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|\text{NN}_i - \text{Robotran}_i| \tag{4.5}$$

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\text{NN}_i - \text{Robotran}_i)^2} \tag{4.6}$$

$$NRMSE[\%] = 100 \times \frac{\text{RMSE}}{\max(\text{Robotran}) - \min(\text{Robotran})} \tag{4.7}$$

	mean MAE $[m/s^2]$	mean RMSE $[m/s^2]$	mean NRMSE [%]
low frequency model	0.009	0.011	1.38
high frequency model	0.005	0.006	9.33
Complete model	0.012	0.013	4.01

Table 4.3: Mean errors on the entire test datasets.

Due to the triangle inequality, the MAE of the complete model will always be smaller than or equal to the sum of the MAEs of both sub-models. We observe that the high frequency model has a smaller mean RMSE, but a higher mean NRMSE. This is because the predictions are not as good as for low frequencies, but the signals have a lower amplitude.

4.4.2 Time efficiency

The main advantage of deep learning approaches, is time efficiency. In this section, these results are showcased. The tests were performed on the CPU of a portable computer and a GPU. Detailed information about the devices used can be found in appendix 3. The computing time is compared for Robotran and the different neural networks. Each model is run 20 times, for 20 seconds. The average computing times are written in table 4.4.

	Mean CPU computing time	Mean GPU computing time
Robotran	89.2s	(not implemented on GPU)
Neural Network low frequency	17.4s	3.48s
Neural Network high frequency	53.3s	10.0s
Neural Network complete	70.7s	13.48s

Table 4.4: Computing times.

It is important to note that the time needed for Robotran can vary, depending on the track defects. When all bodies oscillate a lot, or the wheel-rail contact point is hard to find, the simulation will take more time. On the contrary, the neural

networks always have the same computing time. The number of computations made is independent of the track defects.

The time needed for the low frequency part is almost 3 times the time needed for the high frequency part. This is due to the number of LSTM units. Even if the high frequency part has 1 layer less, it has way more LSTM units. It gains 6 neurons in the dense layer but has 60 LSTM cells.

The average CPU computing time for the deep learning approach is 79% of the Robotran computing time. This is an improvement, but the biggest difference is visible when adapting the neural network to run on GPU. The neural network takes only 19% of the CPU time to do the same computations on GPU. The Robotran executable is not adapted to run on GPU, so no comparison could be made for this model. However, the NN on GPU is more than 6 times faster than Robotran on CPU.

Chapter 5

FFNN for parameter estimation

5.1 Objectives

In multibody applications, deep learning is essential for creating surrogate models. These models serve different purposes, such as predicting time series, covered in chapter 4, or parameter identification. The main focus of this chapter is parameter identification, applied to a railway vehicle.

Parameter identification is highly beneficial for the railway industry. First, predictive maintenance can be performed. In this example, railway suspensions are observed. To detect breakage or excessive friction due to wear and tear, it is important to know the suspension stiffness and damping parameters. It is hard to measure them directly on a working train, as accessing the suspensions is not easy. For this purpose, a surrogate model can be used. The input could be car body accelerations. The outputs are the suspension parameters. Acceleration is way more easy to measure than directly the suspension constants. Once the suspension constants are predicted by the surrogate model, they can be compared with threshold values. This gives an indicator if maintenance is needed.

However, parameter identification is not always feasible with traditional approaches through equations of motion. Indeed, the relationship between measured accelerations and suspension constants might be unknown. Even if this mathematical relation was known during the construction of a train, wear and tear could affect friction, making the original relation less valid. One solution to this problem is to construct surrogate models using deep learning methods.

This chapter will specifically focus on FFNN to create the surrogate models. The main objective is to understand what the FFNN can achieve in terms of accuracy, and the data required to construct the NN.

We aim for the network to predict both the stiffness and damping parameters of the secondary suspension of a railway vehicle, denoted as k_z and d_z respectively. The inputs of the model are the vertical car body accelerations. In real applications, the vertical track profile is not always known. For this reason, we introduce a random vertical deflection in the rail. The shape of this irregularity is unknown for the model. It varies for every simulation.

To predict k_z and d_z, two separate networks are constructed. Then, the results are compared by making one bigger network to predict both parameters. This is important to see if we can gain time efficiency without losing accuracy.
Another objective of this chapter is to detect how much input data we need. For this goal, we can act on 3 parameters:

1. Number of training samples
2. Length of each training sample
3. Timestep of each training sample

Decreasing the amount of training data has many advantages. When real-life data is used for training, it is often hard to obtain thousands of samples. When simulation data is used for training, this can take a long time to compute. Therefore, decreasing the number of training samples, and the length of each sample is mainly beneficial for data gathering. Additionally, it also allows to create a smaller network, as it has to handle fewer inputs. Finally, this is also the goal of increasing the timestep. Feeding fewer inputs to the network prevents the inclusion of excessive unnecessary data, and reduces the number of inputs.

5.2 Data Gathering and Preprocessing

5.2.1 Railway vehicle parameters

The railway vehicle from section 3 will be used. Its velocity is set at 20m/s (72km/h). During this experiment, different values for the secondary vertical stiffness and damping coefficient are selected randomly. These coefficients determine the vertical force exerted by the secondary suspension, as described in equation 5.1. q, $\dot{q}$ represent respectively the vertical displacement and velocity of the suspension. l is the neutral length of the suspension.

$$F_{vertical} = k_z(q - l) + d_z\dot{q} \tag{5.1}$$

The used ranges of the parameters are represented in table 5.1. The minimum and maximum values were selected to create a symmetrical range around the default value. The range was kept as large as possible, without exceeding 60% derailment when two random values were selected. In any other case, the number of simulations would have been excessive in order to create the dataset.

	Minimum value	Maximum value
Stiffness(k_z) [kN/m]	300	550
Damping (d_z) [kNs/m]	2	8

Table 5.1: Secondary suspension parameters.

5.2.2 Track parameters

A bump or sinking of the track is added on top of the default, straight track. The purpose is to introduce an unknown variation in the model. The anomaly will take the form of an amplified cosinus. The length of the irregularity is set at half of the wavelength of the cosinus. The values for the amplitudes and wavelengths of track irregularities are randomly selected from a normal distribution. This value is not given to the network, it remains completely unknown. The ranges are displayed in table 5.2.

	Minimum value	Maximum value
Amplitude (A) [mm]	-3	3
Wavelength (wl) [m]	1	25

Table 5.2: Track irregularity range.

One exception is made in the amplitude range. Zero amplitude is taken out. Otherwise, this would result in a perfectly straight track. As the train is starting at dynamic equilibrium, no vertical acceleration would be observed. Possible track profiles are shown in figure 5.1. The track defect is always starting at time $t = 0$.

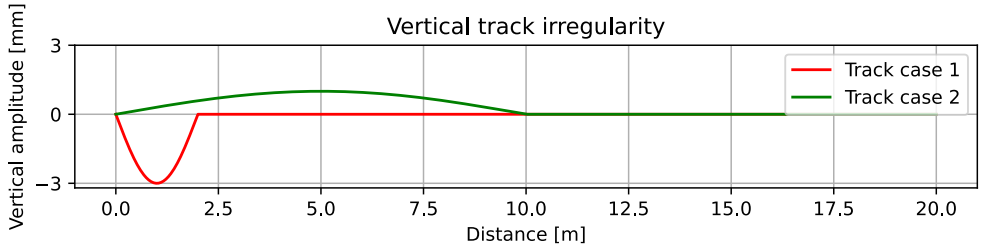


Figure 5.1: Possible track irregularities.

5.2.3 Train, validation and test dataset

In total, the simulation was run 20 000 times, with a random track irregularity and random parameters for the secondary suspension. Each time the car body accelerations are saved over 20 seconds. The timestep is 5 milliseconds. The total computing time for every full simulation was approximately 1 minute. This significant value is due to the necessity of setting the vehicle at dynamic equilibrium before starting the simulation. The equilibrium is different for each simulation, as the suspension parameters vary.

Only the 12 514 simulations in which the train did not derail were retained. The significant amount of derailment is due to the fact the railway vehicle suspension were adapted.

The train dataset consisted of 80% of the total amount of samples. Both the validation and test dataset were created out of 10%.

As for the LSTM example, chapter 5, the data was mapped linearly into the $[0, 1]$ range by applying min-max unity-based normalization.

5.2.4 Single output FFNN

As a preliminary step, two FFNN are constructed and evaluated. Both have a single-output, with the first predicting k_z and the second one d_z.

Initially, some simple NN architectures networks were evaluated, to obtain an approximate idea of the number of layers and neurons required. This allowed us to understand that all networks with 3 layers or fewer were not trainable. They resulted in constant outputs, independent of the inputs. The same is true for networks over 12 layers, which produced systematically very big or almost zero outputs. The large amount of parameters of such a network explains why it is impossible to train them with this dataset. We grouped each NN into four parts, with each part having have between zero and three layers composed of the same amount of neurons in each layer. The division into parts reduced the search space of PSO as only one parameter 'number of neurons' was required per part. Without this approach, PSO was too slow to converge towards a pseudo-optimal solution within 80 iterations. For the first parts, a wider range of neurons per layer was tested, as we had 4001 inputs (20 seconds of acceleration with a timestep of 0.005 seconds). Ultimately, we only require one output. Therefore, the size of the layers in the network must be reduced.

Once the approximate range of the hyperparameters was known, PSO was applied to find a pseudo-optimal solution. The swarm had 15 particles and the number of maximum iterations was 80. The parameters tested and their range, as well as their pseudo-optimal value, found by PSO are shown in table 5.3.

	Min value	Max value	Optimal value NN k_z	Optimal value NN d_z
Dropout rate	0	0.5	0.08	0.19
No. of layers part 1	1	3	3	3
No. of neurons per layer part 1	0	3000	841	1208
No. of layers part 2	1	3	3	3
No. of neurons per layer part 2	0	1500	496	641
No. of layers part 3	1	3	3	3
No. of neurons per layer part 3	0	500	64	89
No. of layers part 4	0	3	0	0
Batch size	10	1000	53	76
Initial learning rate	0.001	0.01	0.009	0.003
Activation function	tanh, Relu,	sigmoid	Relu	Relu

Table 5.3: Hyperparameters of the FFNN, obtained with PSO.

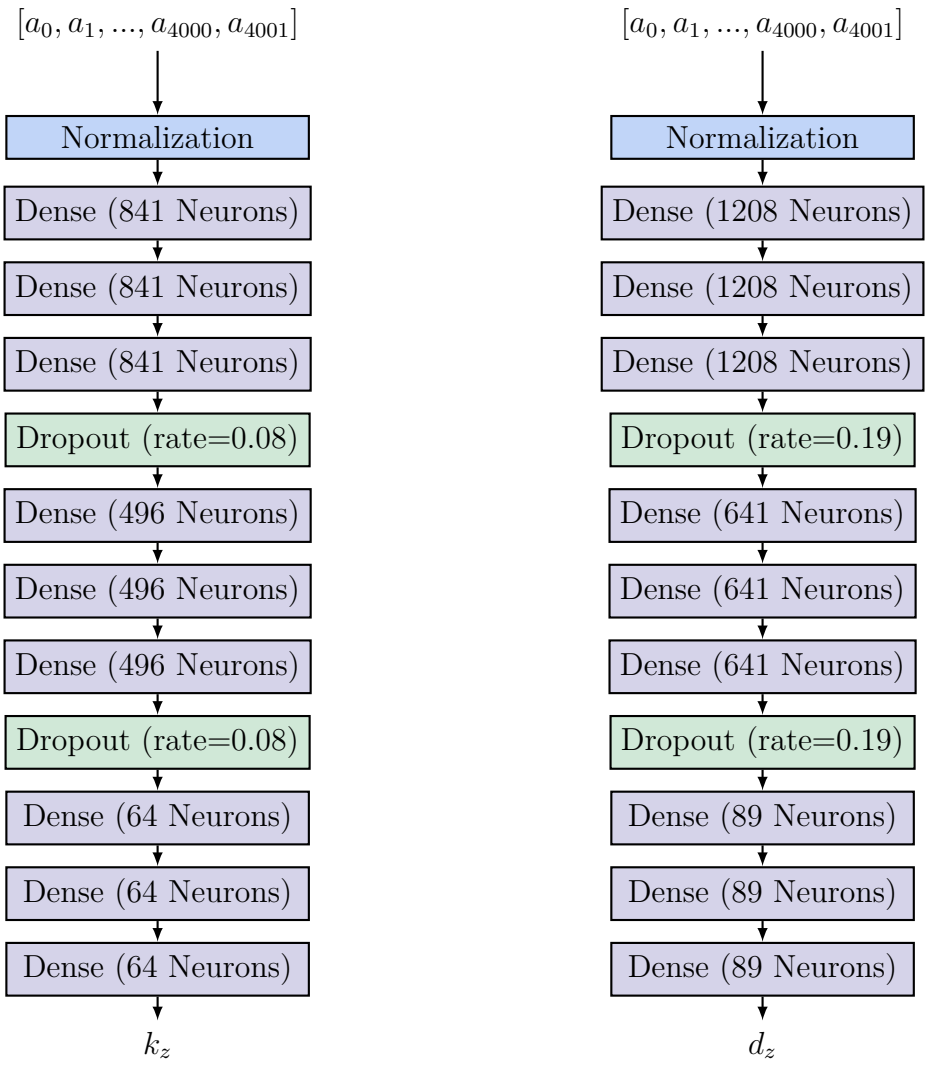


Figure 5.2: Architecture of the FFNN.

It is remarkable that both networks have the same number of layers. However, the network predicting d_z has more neurons in every layer. This implies that the relation between the input acceleration and damping coefficient is harder to capture. Figure 5.2 illustrates the neural network's architecture. The input of the networks are the 4001 accelerations of the car body. This equals 20 seconds of simulation with a frequency of one data point every 0.005 seconds. As most accelerations are damped after 20 seconds, taking samples over 20 seconds would imply to have many zeros in the damped accelerations. Having zeros in the input data decreases the accuracy of the network. Further analysis of sample length and timestep will be conducted in the next sections of this chapter.

The training of these networks takes respectively for k_z and d_z, seven and six minutes, on a CPU device (Appendix 3). The smaller network requires more time. This is explained by the number of epochs needed. The optimal weights were found using 24 instead of 11 epochs. For both networks, the validation loss is respectively 3% and 4% higher than the training loss. These very close losses indicate that the NN are not overfitting the training data.

The networks are each evaluated in terms of their relative errors and relative absolute errors. The errors are computed with equations 5.3. Note that both the values obtained by the NN and by Robotran are always positive. The relative absolute errors obtained for both networks are provided in table 5.4.

$$\text{Relative Error } [\%] = \frac{\text{Robotran}_i - \text{NN}_i}{Robotran_i} \times 100 \tag{5.2}$$

$$\text{Absolute Error } [\%] = \frac{|\text{Robotran}_i - \text{NN}_i|}{Robotran_i} \times 100 \tag{5.3}$$

	Maximum absolute error [%]	Mean absolute error [%]
Neural Network k_z	4.93	1.00
Neural Network d_z	20.36	3.21

Table 5.4: Neural networks relative errors.

Both neural networks perform quite well when looking at mean absolute errors. However, we observe that the mean absolute error is 3 times larger when predicting d_z, compared to k_z.

Figure 5.3 shows the relative errors compared to the value obtained by Robotran. The orange line constitute the median. The box indicates the interquartile range (IQR), which is the distance between the 25th and 75th percentiles. The horizontal whiskers are the minimum and maximum within the $1.5 \times IQR$ range. Points outside this range are considered outliers.

We first observe the network predicting k_z. The distribution of the error is not exceptionally homogeneous, with the tendency to overestimate the stiffness parameters in the $[400kN/m, 450kN/m]$ range. This region also includes some outliers. In the region of $[475kN/m, 550kN/m]$, it tends to underestimate the values and creates some outliers Globally, the obtained predictions are better for smaller stiffness values, even if they never exceed a 4 % relative error. For railway application, not having any outlier with an exceptionally high value is very good.

The NN predicting d_z has a more homogeneous distribution around the median. However, the average error is higher. It has a slightly lower number of outliers (17 compared to 25 for k_z), but they are more extreme in value, reaching over 20%.

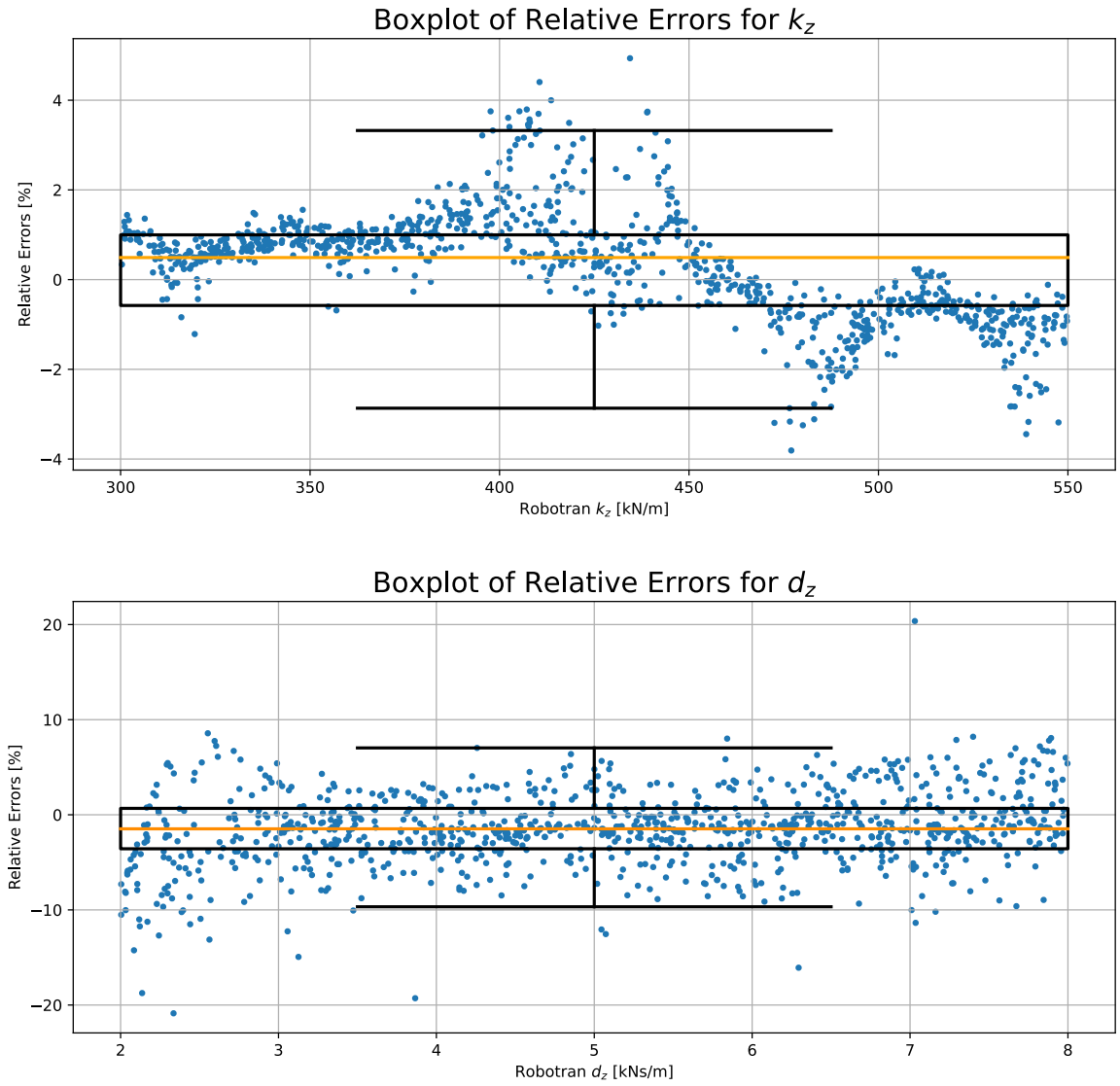


Figure 5.3: Relative errors for both neural networks.

5.2.5 Multiple output FFNN

A multiple-output neural network was build. Its goal is to predict both k_z and d_z, at the same time. This could be an advantage in time efficiency. If the same calculations are required in both single-output networks, they are only performed once in a multiple-output network. The accuracy should not decrease. This is because a multiple-output network can be constructed exactly the same way as two single-output networks, while adding connections between both. In the worst case, the weights and biases of these additional connections can be set to zero, effectively isolating the two tasks. As shown in figure 5.4, this means setting the weights and biases of the red connections to zero, resulting in accuracy that matches that of two separate single-output networks. The accuracy should also not increase. If the accuracy of the multiple-output NN was higher for one of the outputs, we could easily take this network and remove one output (by multiplying by 0), to obtain a single-output NN.

This is under the assumption all networks have optimal hyperparameters and weights. In practice, this is not completely possible. As no function exists to find the best set of hyperparameters, only pseudo-optimal solutions are used. Without guarantee, we might find a better pseudo-optimal solution with a multiple-output neural network.

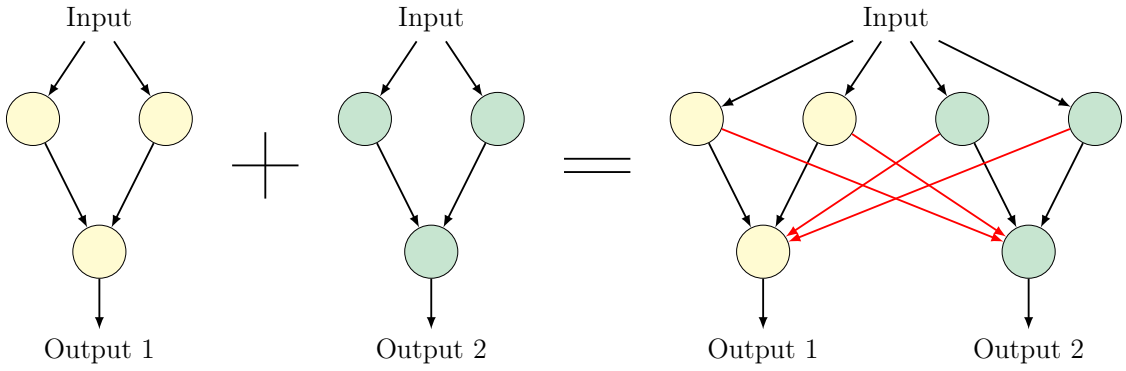


Figure 5.4: Merging 2 single-output FFNN into one multiple-output FFNN.

Making one multiple NN could potentially be a good solution in theory, as it could improve computation time. However, the training of such a bigger network is harder. The global minimum of the loss function is harder to find, as more parameters have to be trained. This also increases the risk of overfitting.

Once more, PSO was used to find a pseudo-optimal set of hyperparameters. The number of particles was 15, the number of maximum iterations was 80. The ranges of hyperparameters used were 2 times the ranges of the single-output networks. (Appendix: 4) The pseudo-optimal network found is shown on figure 5.5.

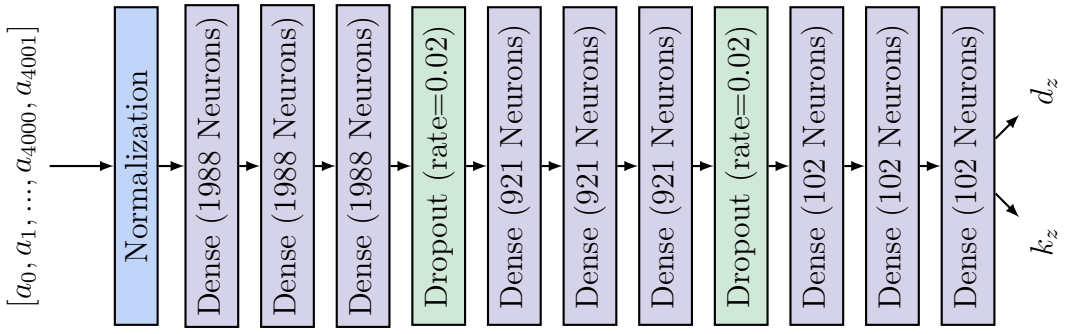


Figure 5.5: Global architecture of the multiple-output neural network.

The total amount of neurons for each layer is lower than the sum of neurons in each layer of the single-output networks. However, each neuron has more inputs, as the layers have more neurons than for each individual single-output NN. This implies each neuron has more weights, which have to be tuned.

	Maximum absolute error [%]	Mean absolute error [%]
k_z from multiple-output NN	6.01	0.96
d_z from multiple-output NN	22.36	3.51

Table 5.5: Neural network relative errors.

The obtained errors are displayed in table 5.5. The mean absolute error for k_z is slightly lower, but the maximum absolute error is a bit higher. For d_z, both values are worse. We did not reach the same accuracy as for the single-outputs models, due to the difficulty of training a network with more neurons.

In order to obtain these results, the loss function had to be adapted. Equation 5.4 is the metric used. It gives more importance to the loss relative to d_z, without this modification the network was not able to make accurate predictions for the damping coefficient.

$$\mathcal{L} = \sqrt{(\text{Robotran}_{k_z} - \text{NN}_{k_z})^2 + 2 \times (\text{Robotran}_{d_z} - \text{NN}_{d_z})^2} \tag{5.4}$$

Despite a decrease of accuracy, the time efficiency is analyzed. The results are in table 4.4. The values do not give any significant gain or loss in time efficiency. Even if the total amount of neurons of this network is smaller than for 2 separate single-output networks, there are more connections. This means every neuron has more weights. The multiple-output network does not gain any significant time efficiency, compared to using 2 single-output networks.

	Prediction time on CPU [ms]	Prediction time on GPU [ms]
k_z single-output NN	48	23
d_z single-output NN	44	21
k_z and d_z multiple-output NN	91	42

Table 5.6: Computation times.

In conclusion, the best multiple-output we could create is putting together both single-output NN. The weights and biases of the connections between both are set to zero. The networks are in fact independent. This is the network which is referred to when referring to a multiple-output NN in the rest of this chapter.

5.2.6 Decreasing training dataset

Improvement on the amount of training data has two main advantages. First, it allows making smaller networks when the input vector is smaller. Indeed, fewer entries have to be processed. Secondly, training time is decreased together with the number of samples in the dataset.

There are three possible ways to decrease the dataset. Reducing the number of samples, reducing the length of each sample, and increasing the timestep. These parameters have to be handled carefully, as less (accurate) data could result in worse training and thus bad predictions.

Bar chart 5.6 shows the mean relative absolute error obtained for both coefficients, with respect to the amount of training samples. In this experiment the hyperparameters never changed, only the values of the weights were adapted. The results could have been better if a specific set of hyperparameters was searched for each dataset size. This was not done as we want to observe the robustness of this neural network to changes of the training dataset.

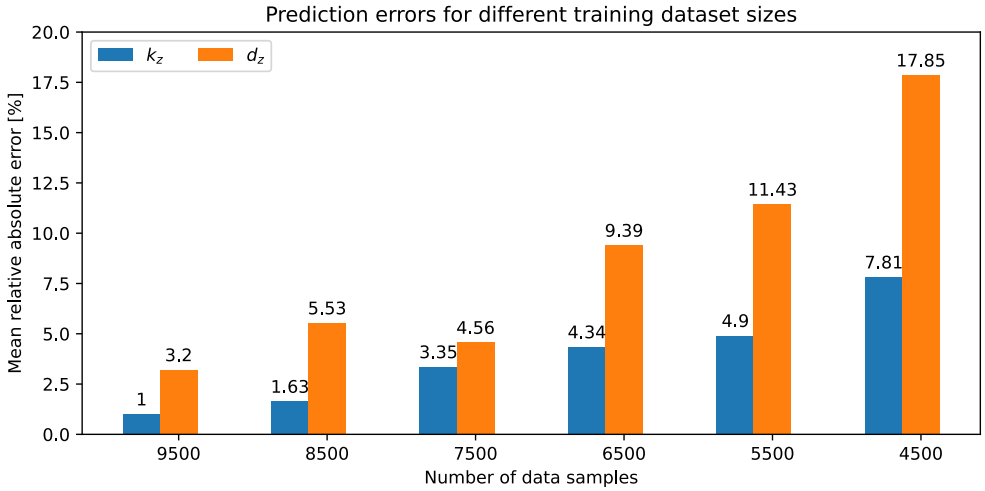


Figure 5.6: Possible track irregularities.

As expected, both errors increase, while the size of training dataset decreases. The graph shows that the error on d_z increases at a faster rate. For 7500 samples, the network performs acceptably on both coefficients, with relative absolute errors below 5%. For k_z, the dataset size can be decreased beyond 7500 samples, but for d_z this is not possible without a significant loss of accuracy.

The heatmap on figure 5.7 shows the arithmetic mean of both errors, in function of the length of each sample and the timestep. For every set of length and timestep, the model was trained with 9000 samples. That model is then evaluated on a test dataset. The global architecture was not changed for every couple of sample length and timestep. The numbers in each cell are the relative absolute errors of respectively k_z and d_z. The color is the arithmetic mean of both errors. A darker color means a more accurate network for that couple of sample length and timestep. The global trend is that a smaller timestep and longer simulation length obtains better results. We observe that by dividing the sample length by two, and by taking a three times bigger timestep, the accuracy is even better. However, this would decrease the total amount of data points from 4001 to 1001. It is highly possible a smaller network would obtain similar performances, as the number of inputs is decreased. By giving the network more information, the pseudo-optimal set of hyperparameters is not as

good as the one found with fewer inputs.

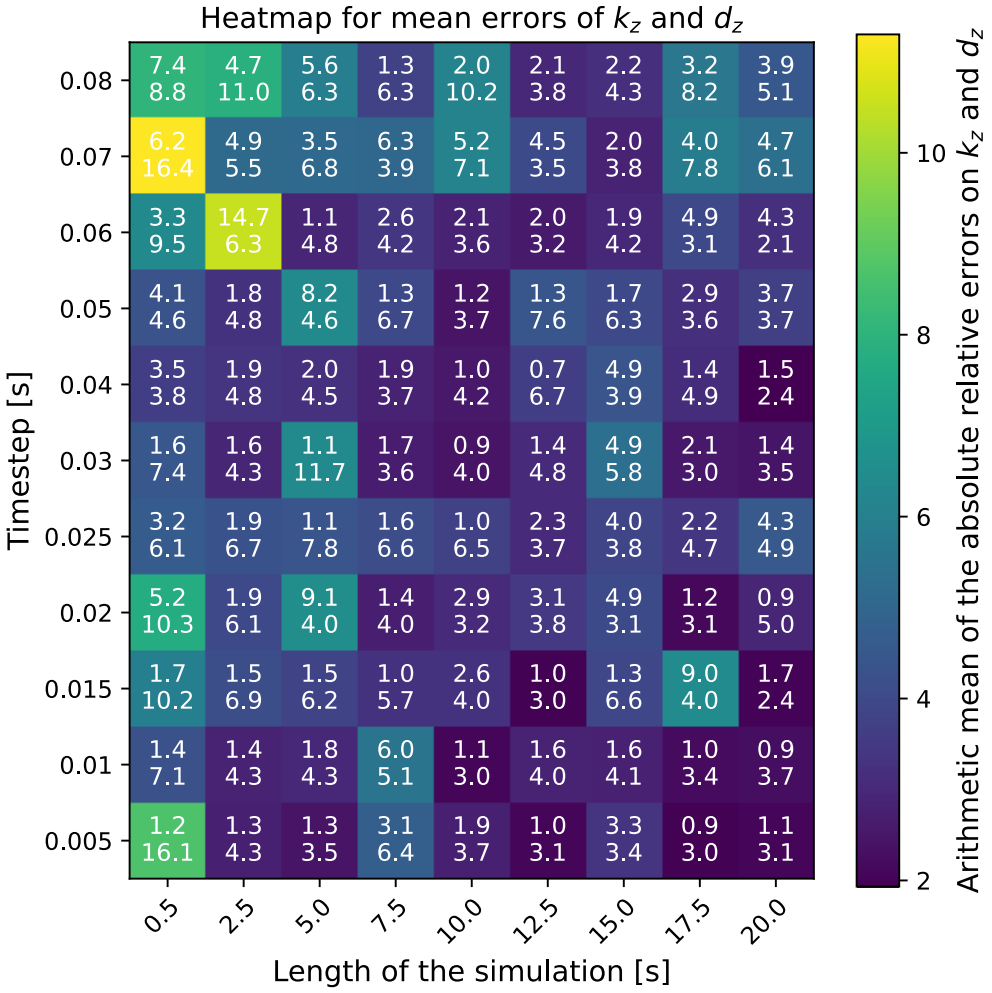


Figure 5.7: The color is the mean absolute relative error. The upper number in each cell is the mean absolute relative error for k_z, while the lower one is for d_z.

To make the best possible networks, these hyperparameters have to be tuned together with the other hyperparameters. The architecture of the network can be influenced by the size of the training samples.

5.2.7 Robustness

The robustness of the model is evaluated by trying to predict k_z and d_z with values outside the training range. Some results are given in table 5.7. They are not plotted, as some errors are almost 2000%. Only 19% of the points tested with parameters outside the ranges have both errors below 5 %. If one of the parameters is slightly outside its training range, the predictions are mostly very poor for both parameters.

The observed lack of robustness can be attributed to the network's high number of weights relative to the available training samples. This allows the network to fit the

k_z [kN/m]	d_z [kNs/m]	Absolute relative error k_z [%]	Absolute relative error d_z [%]
353	9	57	28
582	5	4	5
559	10	78	1981

Table 5.7: Relative absolute error for parameters outside the training range. As a reminder, the training range of k_z is [300, 550] kN/m, and for d_z it is [2, 8] kNs/m.

parameters within a limited range very precisely. However, expanding the range or reducing the network's size would reduce its accuracy, but force it to capture more global patterns. Consequently, this would enhance the robustness.

Chapter 6

Discussion and perspectives

The main objective of this master's thesis was to investigate the potential of deep learning as an improvement for multibody modeling. The key advantages of deep learning in this domain include reducing computation time and creating surrogate models.

For the first experiment, an LSTM model was developed. The input was the vertical track profile, and the outputs were the vertical car body accelerations. It was not possible to predict the accelerations with a single model consisting of only LSTM layers, as the low-frequency signal had larger amplitudes than the high-frequency signal. To address this, two sub-networks were created. The first one tackled low frequencies with larger amplitudes, while the second one handled high-frequency signals. The accelerations predicted by the global model were, in fact, the sum of the outputs of both sub-networks.

The model obtained good results for low frequencies, while it was harder to tackle high-frequency signals. The main reason for this difficulty is likely that the chosen timestep was too large. A smaller timestep would result in two consecutive data points being closer to each other. It would be easier for the network to capture the global patterns in the data. By taking advantage of the two sub-networks, this improvement can be applied only to the high-frequency network.

The LSTM requires 4 times less computation time compared to the classical approach used by Robotran. However, training the model requires a large amount of data and is time-consuming.

The second experiment involved constructing FFNNs for parameter identification. It highlighted the difficulty of obtaining accurate predictions for some parameters, especially when using larger networks, which are harder to train. Maintaining two independent, smaller networks yielded better results in terms of both accuracy and time efficiency, compared to a single global network.

In the case of a single multiple-output network, finding a good loss metric was challenging. The network tends to give more importance to one output over the others.

Another challenge faced during the creation of a FFNN is hyperparameter tuning. All hyperparameters need to be tuned simultaneously to find the best pseudo-optimal network. It was shown that the choice of input data has a significant impact. However, comparable results were obtained in terms of accuracy, even when the input length was reduced from 20 seconds to 10 seconds, and the timestep changed from 0.005 seconds to 0.01 seconds. Providing more information does not always result in a better network.

Additionally, the training range must be highly accurate. When the input data diverges from the training range, the accuracy of the results decreases.

A combination of both experiment could be interesting. The FFNN from the second experiment could be improved by changing the first dense layer by an LSTM layer, as the input acceleration is a time series. In that case all the timesteps given to the model would have the same importance, as the weights of every LSTM cell are the same. This is not true for dense layers, where the weights are different for every neuron.

In conclusion, deep learning has a significant potential in multibody applications. However, future research is needed to achieve accurate and useful results in real-life applications. Several directions for further research were identified during this master's thesis.

First, using more complex deep learning mechanisms is important. For example convolutional layers, which have demonstrated effectiveness in various applications. These could help to capture spatial dependencies in the data more effectively than the traditional methods used.

Next, incorporating real-life data into the training datasets will add complexity due to the presence of outliers and non-homogeneous distributions. However, this is an essential step towards developing more robust and global models.

Robotran and other multibody compute a significant number of physical parameters of the model. Taking advantage of those could be useful in the construction of physics-informed neural networks. These networks can integrate physical laws and constraints, which could result in more accurate and robust models.

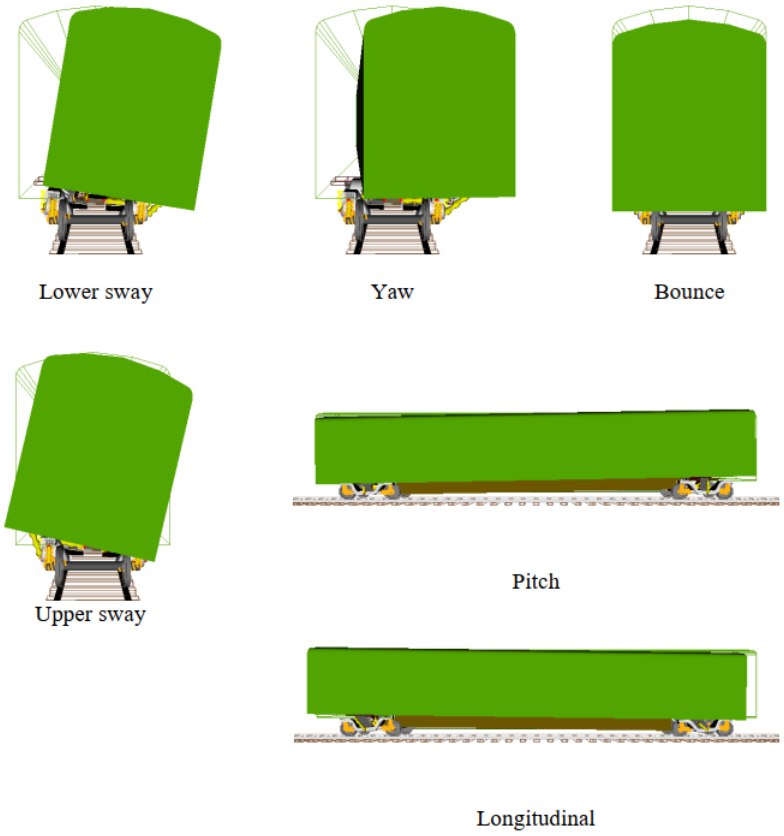
Finally, focusing on specific, time-consuming aspects of multibody simulations. For example, in our railway application, a significant amount of computational time is devoted to determining the wheel-rail contact points using dichotomic search methods. Depending on the simulation, looking for the contact points takes between 10 and 20% of the total simulation time. Optimizing this particular process could lead to improvements in overall simulation efficiency while keeping a better robustness and accuracy.

Chapter 7

Appendices

1 Natural frequencies of the railway vehicle

Image illustrating the modes of natural frequencies of the railway vehicle. The railway vehicle is based on the Manchester Benchmark [14]. The source of the image is "Manchester Benchmarks Results" [19].



2 Programming languages and packages

Python is the most common used programming language for deep learning. Various packages are easy to use and serve as building blocks for deep neural networks. The most common are PyTorch and TensorFlow. Both have a Python front-end, but are integrated with C and C++ backends.

Keras was used for this master's thesis, as the author had some more experience in using it. Keras is a library running on top of the TensorFlow library. It is very user-friendly, thanks to its great modularity. It is integrated with TensorFlow, which is useful to run the neural networks in a C environment, using the TensorFlow C API. For this purpose the weights of the networks are saved in files with the extension '.h5'. These can be opened in a C environment with the help of the open-source C API of TensorFlow. The drawbacks are the regular updates of the TensorFlow library, without official support for the C API.

On the other hand, PyTorch offers more flexibility for more complex network architectures. It is lower-level than Keras, offering an even better time efficiency.

3 Devices and packages used

The GPU used for hyperparameter tuning and training of the neural networks is the one from the IMMC team: Quadra P1000, with 3481MB of memory.

Following version of the drivers, Python and packages were used:

- NVIDIA-SMI 535.171.04
- Python 3.8.10
- TensorFlow 2.12.0
- CUDA 12.2
- cuDNN 8.6.0.163

The CPUs used for running the railway simulations with Robotran were the ones of the CECI clusters. They are Intel Skylake 16-cores Xeon 6142 processors at 2.6 GHz with a maximum of 192GB of Ram.

Finally, the CPU used for time efficiency analysis is an 11th Gen Intel(R) Core(TM) i7-1165G7, 2.80GHz with 16GB of Ram. This is the CPU from a portable computer.

4 PSO Multiple Output Neural Network

The hyperparameters and their ranges obtained with particle swarm optimization for the multiple output neural network are shown in table 7.1. The maximum values are the sums of the optimal values of both independent networks.

	Min value	Max value	Optimal value
Dropout rate	0	0.5	0.02
No. of layers part 1	1	3	3
No. of neurons per layer part 1	0	2049	1988
No. of layers part 2	1	3	3
No. of neurons per layer part 2	0	1137	921
No. of layers part 3	1	3	3
No. of neurons per layer part 3	0	153	102
Initial learning rate	0.001	0.01	0.003
Activation function	tanh, Relu,	sigmoid	Relu

Table 7.1: Hyperparameters of the multiple output neural network, predicting both k_z and d_z.

5 Predicted accelerations for low and high frequency

The following images show the prediction made by the neural networks, for both low and high frequency part. The blue lines are the values obtained by Robotran. The red lines the values obtained by the neural network. The output at a certain timestep is the input of the next timestep. In green, the values obtained by the neural network when the inputs are only values obtained by Robotran. In other words, there is no error propagation here. Each timestep is evaluated independently.

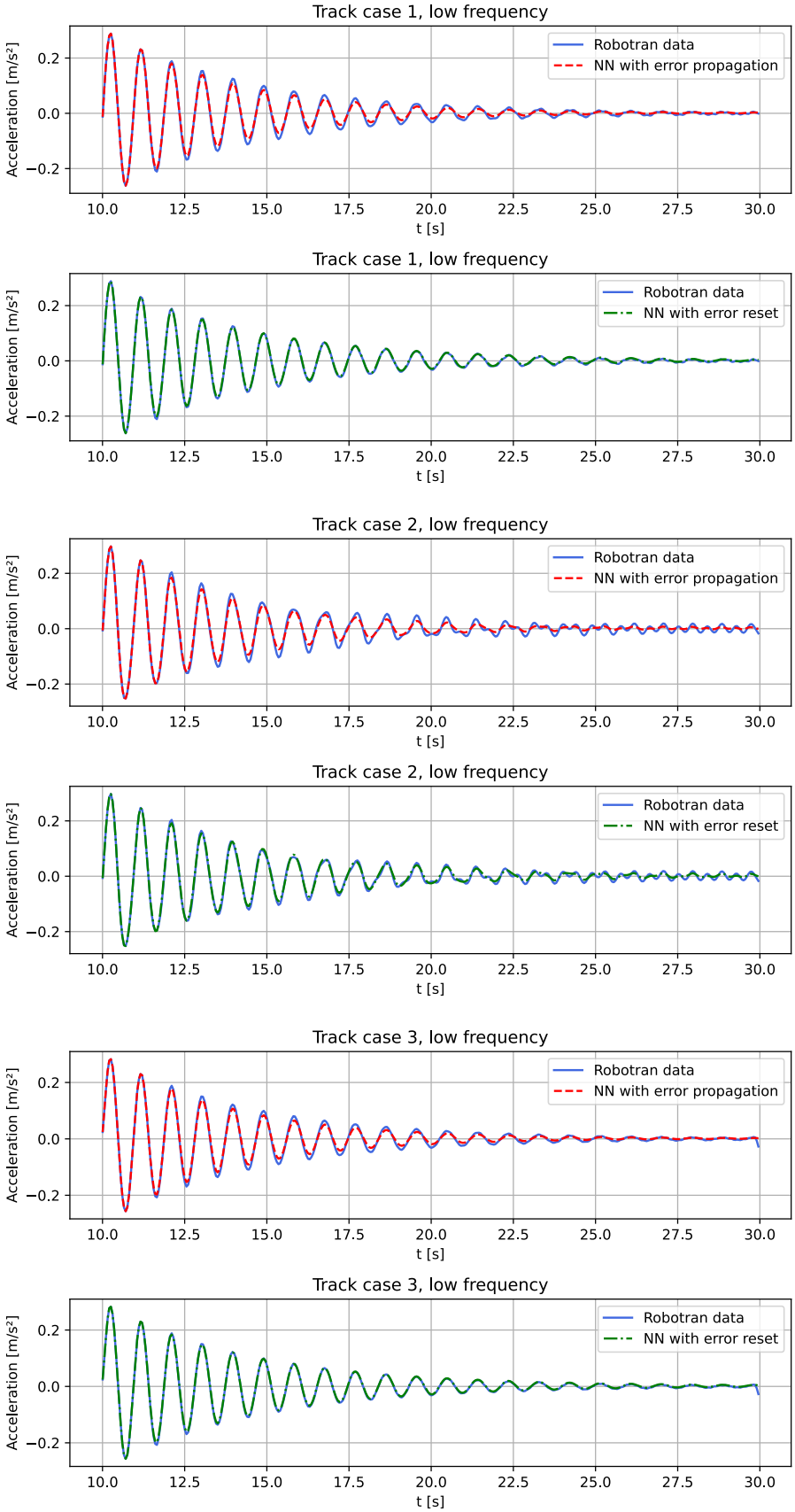


Figure 5.1: NN predictions for low frequency part

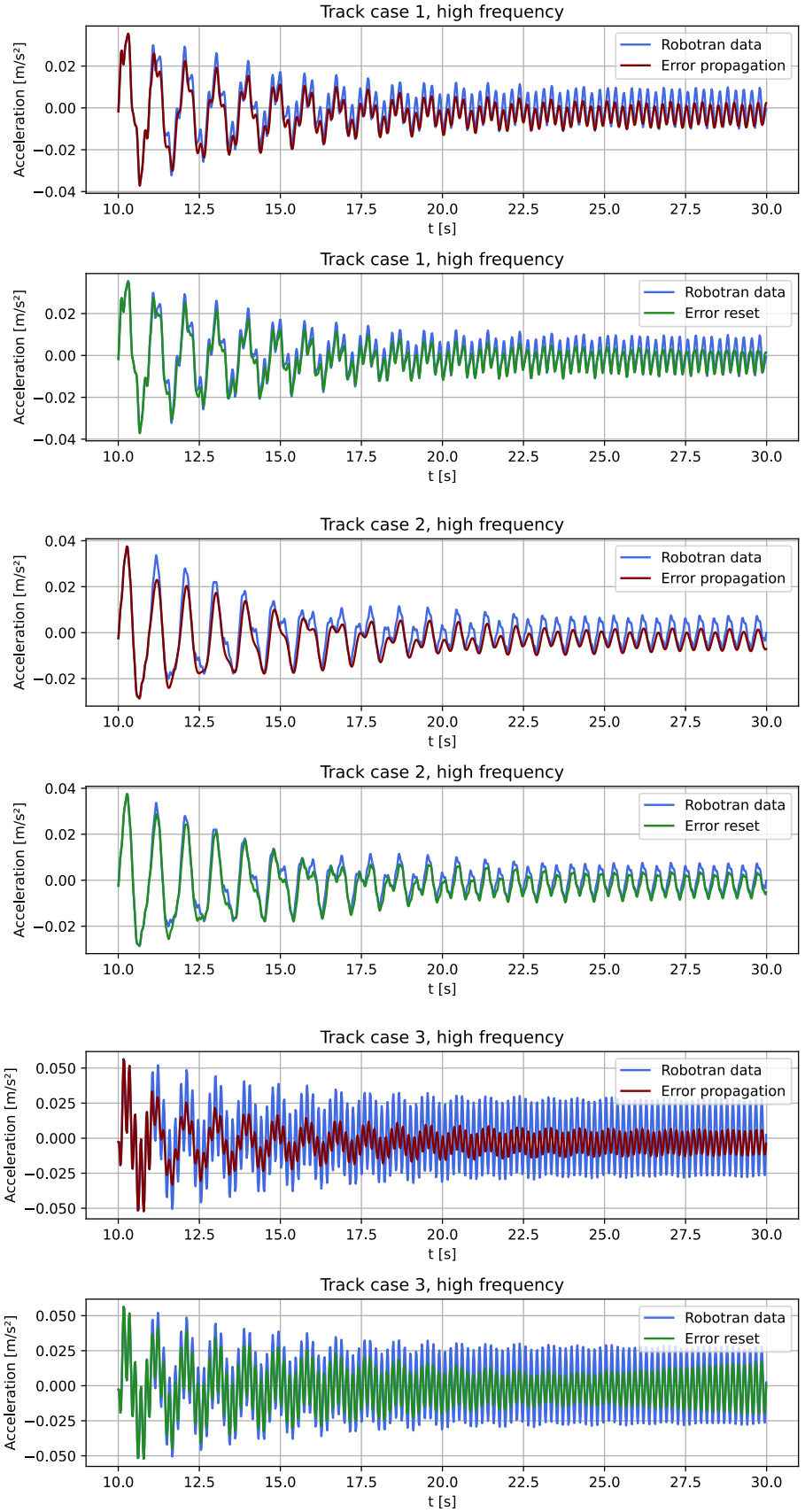


Figure 5.2: NN predictions for high frequency part

Bibliography

[1] Charu C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. Cham: Springer International Publishing, 2018. ISBN: 978-3-319-94462-3 978-3-319-94463-0. DOI: 10.1007/978-3-319-94463-0. URL: http://link.springer.com/10.1007/978-3-319-94463-0 (visited on 11/05/2023).

[2] Ahmed Akl et al. "Optimizing deep neural networks hyperparameter positions and values". In: *Journal of Intelligent & Fuzzy Systems* 37.5 (Nov. 22, 2019), pp. 6665–6681. ISSN: 10641246, 18758967. DOI: 10.3233/JIFS-190033. URL: https://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/JIFS-190033 (visited on 01/14/2024).

[3] Hamid Ardeh, Martin Tupy, and Dan Negrut. *On the Construction and Use of Surrogate Models for the Dynamic Analysis of Multibody Systems*. Vol. 3. Journal Abbreviation: SAE International Journal of Passenger Cars - Mechanical Systems Publication Title: SAE International Journal of Passenger Cars - Mechanical Systems. Jan. 1, 2009. DOI: 10.1115/IMECE2009-10277.

[4] Hee-Sun Choi et al. *Data-driven simulation for general purpose multibody dynamics using deep neural networks*. Sept. 2, 2019. arXiv: 1909.02391[cs,eess,stat]. URL: http://arxiv.org/abs/1909.02391 (visited on 09/21/2023).

[5] A De Rosa et al. "Monitoring of lateral and cross level track geometry irregularities through onboard vehicle dynamics measurements using machine learning classification algorithms". In: *Proceedings of the Institution of Mechanical Engineers, Part F: Journal of Rail and Rapid Transit* 235.1 (Jan. 1, 2021). Publisher: IMECHE, pp. 107–120. ISSN: 0954-4097. DOI: 10.1177/0954409720906649. URL: https://doi.org/10.1177/0954409720906649 (visited on 09/22/2023).

[6] N. Docquier, A. Poncelet, and P. Fisette. "ROBOTRAN: a powerful symbolic gnerator of multibody models". In: *Mechanical Sciences* 4.1 (May 2, 2013). Publisher: Copernicus GmbH, pp. 199–219. ISSN: 2191-9151. DOI: 10.5194/ms-4-199-2013. URL: https://ms.copernicus.org/articles/4/199/2013/ms-4-199-2013.html (visited on 04/29/2024).

[7] Nicolas Docquier et al. "Symbolic multibody models for digital-twin applications". In: *Multibody System Dynamics* (Dec. 18, 2023). ISSN: 1573-272X. DOI: 10.1007/s11044-023-09957-x. URL: https://doi.org/10.1007/s11044-023-09957-x (visited on 04/29/2024).

[8] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. *Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark*. June 28, 2022. arXiv: 2109.14545[cs]. URL: http://arxiv.org/abs/2109.14545 (visited on 04/09/2024).

[9] Luke B. Godfrey. "An Evaluation of Parametric Activation Functions for Deep Learning". In: *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)* (Oct. 2019). Conference Name: 2019 IEEE International Conference on Systems, Man and Cybernetics (SMC) ISBN: 9781728145693 Place: Bari, Italy Publisher: IEEE, pp. 3006–3011. DOI: 10.1109/SMC.2019.8913972. URL: https://ieeexplore.ieee.org/document/8913972/ (visited on 04/09/2024).

[10] Magnus Hansson and Christoffer Olsson. "Feedforward neural networks with ReLU activation functions are linear splines". In: ().

[11] Arash Hashemi et al. "Multibody dynamics and control using machine learning". In: *Multibody System Dynamics* 58.3 (Aug. 1, 2023), pp. 397–431. ISSN: 1573-272X. DOI: 10.1007/s11044-023-09884-x. URL: https://doi.org/10.1007/s11044-023-09884-x (visited on 09/22/2023).

[12] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 15, 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735 (visited on 04/12/2024).

[13] Yuhuang Hu et al. *Overcoming the vanishing gradient problem in plain recurrent networks*. July 5, 2019. arXiv: 1801.06105[cs]. URL: http://arxiv.org/abs/1801.06105 (visited on 04/12/2024).

[14] Simon Iwnick. "Manchester Benchmarks for Rail Vehicle Simulation". In: *Vehicle System Dynamics* 30.3 (Sept. 1998), pp. 295–313. ISSN: 0042-3114, 1744-5159. DOI: 10.1080/00423119808969454. URL: http://www.tandfonline.com/doi/abs/10.1080/00423119808969454 (visited on 01/09/2024).

[15] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. Jan. 29, 2017. DOI: 10.48550/arXiv.1412.6980. arXiv: 1412.6980[cs]. URL: http://arxiv.org/abs/1412.6980 (visited on 05/30/2024).

[16] Zachary Lipton. "A Critical Review of Recurrent Neural Networks for Sequence Learning". In: (May 29, 2015).

[17] Pablo Ribalta Lorenzo et al. "Particle swarm optimization for hyper-parameter selection in deep neural networks". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '17: Genetic and Evolutionary Computation Conference. Berlin Germany: ACM, July 2017, pp. 481–488. ISBN: 978-1-4503-4920-8. DOI: 10.1145/3071178.3071208. URL: https://dl.acm.org/doi/10.1145/3071178.3071208 (visited on 04/29/2024).

[18] Michael Lutter, Christian Ritter, and Jan Peters. *Deep Lagrangian Networks: Using Physics as Model Prior for Deep Learning*. July 9, 2019. arXiv: 1907.04490[cs,eess,stat]. URL: http://arxiv.org/abs/1907.04490 (visited on 09/26/2023).

[19] "Manchester Benchmarks for Rail Vehicle Simulation". In: ().

[20] Warren S Mcculloch and Walter Pitts. "A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY". In: ().

[21] Sridhar Narayan. "The generalized sigmoid activation function: Competitive supervised learning". In: *Information Sciences* 99.1 (June 1, 1997), pp. 69–82. ISSN: 0020-0255. DOI: 10.1016/S0020-0255(96)00200-9. URL: https://www.sciencedirect.com/science/article/pii/S0020025596002009 (visited on 04/09/2024).

[22] M. Raissi, P. Perdikaris, and G.E. Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378 (Feb. 2019), pp. 686–707. ISSN: 00219991. DOI: 10.1016/j.jcp.2018.10.045. URL: https://linkinghub.elsevier.com/retrieve/pii/S0021999118307125 (visited on 09/26/2023).

[23] Hailin Ren and Pinhas Ben-Tzvi. "Learning inverse kinematics and dynamics of a robotic manipulator using generative adversarial networks". In: *Robotics and Autonomous Systems* 124 (Feb. 1, 2020), p. 103386. ISSN: 0921-8890. DOI: 10.1016/j.robot.2019.103386. URL: https://www.sciencedirect.com/science/article/pii/S0921889019303501 (visited on 04/16/2024).

[24] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological Review* 65.6 (1958), pp. 386–408. ISSN: 1939-1471, 0033-295X. DOI: 10.1037/h0042519. URL: https://doi.apa.org/doi/10.1037/h0042519 (visited on 04/03/2024).

[25] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. June 15, 2017. arXiv: 1609.04747[cs]. URL: http://arxiv.org/abs/1609.04747 (visited on 04/12/2024).

[26] Juan Terven et al. *Loss Functions and Metrics in Deep Learning. A Review*. July 5, 2023.

[27] P.J. Werbos. "Backpropagation through time: what it does and how to do it". In: *Proceedings of the IEEE* 78.10 (Oct. 1990). Conference Name: Proceedings of the IEEE, pp. 1550–1560. ISSN: 1558-2256. DOI: 10.1109/5.58337. URL: https://ieeexplore.ieee.org/document/58337 (visited on 04/12/2024).

[28] Logan G. Wright et al. "Deep physical neural networks trained with backpropagation". In: *Nature* 601.7894 (Jan. 2022). Publisher: Nature Publishing Group, pp. 549–555. ISSN: 1476-4687. DOI: 10.1038/s41586-021-04223-6. URL: https://www.nature.com/articles/s41586-021-04223-6 (visited on 04/12/2024).

[29] Xue Ying. "An Overview of Overfitting and its Solutions". In: *Journal of Physics: Conference Series* 1168 (Feb. 2019), p. 022022. ISSN: 1742-6588, 1742-6596. DOI: 10.1088/1742-6596/1168/2/022022. URL: https://iopscience.iop.org/article/10.1088/1742-6596/1168/2/022022 (visited on 04/12/2024).

[30] Siqi Zhou and Angela P. Schoellig. "Active Training Trajectory Generation for Inverse Dynamics Model Learning with Deep Neural Networks". In: *2019 IEEE 58th Conference on Decision and Control (CDC)*. 2019 IEEE 58th Conference on Decision and Control (CDC). Nice, France: IEEE, Dec. 2019, pp. 1784–1790. ISBN: 978-1-72811-398-2. DOI: 10.1109/CDC40024.2019.9029973. URL: https://ieeexplore.ieee.org/document/9029973/ (visited on 04/16/2024).

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**