Master’s Thesis

Dynamic Neural Networks for Multi-Body Simulation in Mechanical Systems

Student: Esteve Vives Pons

Supervisors: Prof. Dr.-Ing. Jochen Wiedemann
            Dipl.-Math. Jens Neubeck

Institut für Verbrennungsmotoren und Kraftfahrwesen
Universität Stuttgart
Stuttgart, October 2011
Abstract

This text deals with the simulation of the tyre/suspension dynamics by using recurrent dynamic neural networks. Recurrent neural networks are based on the multilayer feedforward neural networks, by adding feedback connections between output and input layers. The neural network can be trained with data obtained from the simulation of a physical model created using a multi-body simulation software (SIMPACK). The results obtained from the neural network demonstrate a good agreement that could be improved, depending on some factors, with the multi-body model simulation results. The neural network model can be applied as a part of vehicle system model to predict system dynamic behaviour. Although the neural network model does not provide a good insight of the physical behaviour of the system, it is a useful tool to help in vehicle ride dynamics performance due to its good efficiency and accuracy in computational terms.
Aknowledgments

The author would like to thank the following people:

Jens Neubeck, supervisor of this thesis, to offer the opportunity to develop an interesting topic for the work related with automobile and mechanical engineering, and of course for his help during the thesis development.

The family (parents and grandparents), for their unconditional support during the time of hard work and particularly in the difficult moments, and for the help they have given to me to stay in Stuttgart to develop this work.

The friends, for their help and the encouragement they gave me.

Universitat Politècnica de Catalunya, to provide me this great opportunity to finish my studies in Germany.
# Table of contents

Abstract..................................................................................................................................................2

Aknowledgments ..................................................................................................................................3

Introduction ..........................................................................................................................................6

1. Theoretical basis ...............................................................................................................................8

1.1. History ..........................................................................................................................................8

1.2. Biological inspiration ....................................................................................................................10

1.3. Neural networks introduction ......................................................................................................11

1.4. Neuron models ..............................................................................................................................12

1.4.1. Simple neuron ..........................................................................................................................12

1.4.2. Transfer functions ...................................................................................................................13

1.4.3. Neuron with vector input .........................................................................................................17

1.4.4. Abbreviated notation ...............................................................................................................17

1.5. Network architectures ...................................................................................................................18

1.5.1. A layer of neurons ..................................................................................................................18

1.5.2. Inputs and layers ......................................................................................................................20

1.5.3. Multiple layers of neurons ......................................................................................................21

1.6. Data structures .............................................................................................................................23

1.6.1. Concurrent inputs in a static network .......................................................................................24

1.6.2. Sequential inputs in a dynamic network ...................................................................................24

1.6.3. Concurrent inputs in a dynamic network ...................................................................................25

1.7. Backpropagation ............................................................................................................................26

1.7.1. Introduction .............................................................................................................................26

1.7.2. Architecture .............................................................................................................................27
Introduction

This document gives an introduction to the research in dynamic neural networks applied to the Multi-Body Simulation in mechanical systems. Emphasis is placed on the research to find what kind of network is the most appropriate to be used for this purpose, the implementation with the software used (MATLAB) and the analysis to determine if this is an adequate and successful method to be used in this kind of mechanical systems.

The usage of neural networks is extended to many different industry fields such as: aerospace, automotive, banking, electronics, financial, manufacturing, medical, robotics, telecommunications and many others. But they are more used in some industries than in others, because of investigation is more advanced and more money is invested for this purpose. And mechanical engineering, specially the part related to automotive mechanical systems is one of these areas in which neural networks usage is not largely extended, probably because of the Multi-Body Simulation software used in the automotive industry and its advanced development and efficiency level.

Efforts have been made to present the text in a clear and consistent manner so that it can be read and understood by any engineering student, even those without previous knowledge about neural networks, as the same situation the author found at the beginning. For that reason many specific details have been omitted, specially those mathematical based concepts that are not necessary to develop the research or to understand how networks work.

The main objective of the thesis is to study if is it possible to use dynamic neural networks as a complementary and efficient method to obtain good results in mechanical systems simulation. In this case, this work is the first step to set a base from which continue working and improving in the topic.

This text is not a catalogue or learning guide of all known neural network architectures and rules, but concentrates in a briefly and understandable manner in the description of the networks to be used in the development of the work.

Almost information required about the topic is exposed in the first chapter. It consists in a short introduction to the history of neural networks during the 20th century and the inspiration in the biological concept of neuron. And after this general information about
the topic, there is a wide theoretical focused explanation thought to be easy to understand by anybody with basic knowledge in mathematics or engineering. In this section, neural networks are described from the simple mathematical concept of neuron, going through the networks architectures and finishing with dynamic networks.

In the second chapter, the model for the mechanical multi-body system used to develop the study (the suspension system for a wheel of a real car model) is presented. And also the way to link the multi-body system implemented with SIMPACK to the analytical software used to work with neural networks (MATLAB).

In Chapter 3, the method to do the main purpose (create the correct network and work with it) is showed and explained. This includes the script and the commands to create the network and train it with MATLAB, in order to obtain good results.

The fourth chapter contains the results from the work of the previous chapter. Here are the different results from the research compared and discussed.

And finally, the conclusions from the research are presented, but also the recommendations and possible ways to continue the investigation started with this thesis, in order to improve the results obtained and get better performance in the topic.
1. Theoretical basis

1.1. History

The actual way in how neural networks are understood began in the 1940s with the work of Warren McCulloch and Walter Pitts, who showed that networks of artificial neurons could, in principle, compute any arithmetic or logical function. Their work is often considered as the origin of the neural network field.

After McCulloch and Pitts, Donald Hebb proposed that classical conditioning is present because of the properties of individual neurons. He proposed a mechanism for learning in biological neurons.

The first practical application of artificial neural networks took place in the late 1950s, with the invention of the perceptron network and the associated learning rule by Frank Rosenblatt. Rosenblatt and his colleagues created a perceptron network and proved its ability to perform pattern recognition. This early success generated a great deal of interest in neural network research. Unfortunately, it was later shown that the basic perceptron network could solve only a limited type of problems.

At about the same time, Bernard Widrow and Ted Hoff introduced a new learning algorithm and used it to train adaptive linear neural networks, which were similar in structure and capability to Rosenblatt’s perceptron. The Widrow-Hoff learning rule is still in use today.

Unfortunately, both Rosenblatt’s and Widrow’s networks suffered from the same implicit limitations, which were extensively publicized in a book by Marvin Minsky and Seymour Papert. Rosenblatt and Widrow were aware of these limitations and proposed new networks that would beat them. However, they were not able to successfully modify their learning algorithms to train the more complex networks.

Many people, influenced by Minsky and Papert, believed that further research on neural networks was a dead end. This fact, combined with the situation that there were no powerful computers on which to experiment, caused many researchers to leave the
investigations about the field. For a decade, neural network research was practically suspended.

Some important work, however, did continue during the 1970s. In 1972 Teuvo Kohonen and James Anderson, independently and separately, developed new neural networks that could work as memories. Stephen Grossberg was also very active during this period in the investigation of self-organizing networks.

Interest in neural networks had waned during the late 1960s due to the lack of new ideas and powerful computers with which to experiment. During the 1980s both of these problems were solved, and research in neural networks increased dramatically. New personal computers and workstations, which quickly grow in capability, became easily available. In addition, important new concepts were introduced in the field.

Two new concepts were especially responsible for the revival of neural networks. The first was the use of statistical mechanics to explain the way of work of a certain type of recurrent network, which could be used as an associative memory. This concept was described by physicist John Hopfield.

The second key development of the 1980s was the backpropagation algorithm for training multilayer perceptron networks, which was discovered independently by several different researchers. This algorithm was the answer to the criticisms Minsky and Papert had made in the 1960s.

These new developments reinforced the field of neural networks. Between the last 1980s and 1990s, thousands of papers were written, and neural networks found many applications. The field is going ahead with new theoretical and practical work.

The brief historical account given above is not intended to identify all of the major contributors, but is simply to give to the reader some feel for how knowledge in the neural network field has progressed. But this progress has not been slow but sure. There have been periods of dramatic progress and periods when relatively little has been accomplished.
Many of the advances in neural networks have had to do with new concepts, such as innovative architectures and training rules. Just as important has been the availability of powerful new computers on which to test these new concepts.

But the real question is if neural networks will take a permanent place as a mathematical/engineering tool, or will they fade away as have so many promising technologies? At present, the answer seems to be that neural networks will not only have their day but will have a permanent place, not as a solution to every problem, but as a tool to be used in appropriate situations. In addition, a little is still known about how the brain works. The most important advances in neural networks almost certainly lie in the future in parallel with computers technologies.

1.2. Biological inspiration

The artificial neural networks are only remotely related to their biological equivalents. In this section will be briefly described those characteristics of brain function that have inspired the development of artificial neural networks.

The brain consists of a large number (approximately $10^{11}$) of highly connected elements (approximately $10^4$ connections per element) called neurons. For the topic purposes these neurons have three principal components: the dendrites, the cell body and the axon. The dendrites are tree-like receptive networks of nerve fibers that carry electrical signals into the cell body. The cell body effectively sums and thresholds these incoming signals. The axon is a single long fiber that carries the signal from the cell body out to other neurons. The point of contact between an axon of one cell and a dendrite of another cell is called a synapse. It is the arrangement of neurons and the strengths of the individual synapses, determined by a complex chemical process, that establishes the function of the neural network.

Some of the neural structure is defined at birth. Other parts are developed through learning, as new connections are made and others waste away. This development is most noticeable in the early stages of life.
Neural structures continue to change throughout life. These later changes tend to consist mainly of strengthening or weakening of synaptic junctions. For instance, it is believed that new memories are formed by modification of these synaptic strengths. Thus, the process of learning a new friend’s face consists of altering various synapses.

Artificial neural networks do not approach the complexity of the brain. There are, however, two key similarities between biological and artificial neural networks. First, the building blocks of both networks are simple computational devices (although artificial neurons are much simpler than biological neurons) that are highly interconnected. Second, the connections between neurons determine the function of the network.

Biological neurons are very slow when compared to electrical circuits ($10^{-3}$ s compared to $10^{-9}$ s), but the brain is able to perform many tasks much faster than any conventional computer. This is in part because of the massively parallel structure of biological neural networks; all of the neurons are operating at the same time. Artificial neural networks share this parallel structure. Even though most artificial neural networks are currently implemented on conventional digital computers, their parallel structure makes them ideally suited to implementation using optical devices and parallel processors.

### 1.3. Neural networks introduction

Artificial neural networks consist of simple elements that work in parallel. These elements have the inspiration in biological nervous systems. As in the nature, the connections between the elements mainly define the network function. An artificial neural network can be trained to perform a special function changing the values of the connections (weights) between the network elements.

Neural networks are usually trained, so that a particular input leads to a specific target output. Figure 1.1 shows the training process. There, the network is adjusted, using a comparison between the output and the target values, until the network output matches the target. Normally, several input/target pairs of values are necessary to train a network.
Neural networks have been trained in order to perform many complex functions in different fields, like pattern recognition, identification, classification, speech, vision, and control systems.

Neural networks can also be trained to solve other problems that are difficult for usual computers or human beings, like engineering, financial or other practical applications in many other fields.

1.4. Neuron models

1.4.1. Simple neuron

A neuron with only one scalar input and without bias is shown on the left of figure 1.2.

Figure 1.1. Neural network training process schematic, [4].

Figure 1.2. Simple neurons schematics, [4].
On the right side of figure 1.2, a single input neuron with bias is shown. In that case, the scalar input $p$ is multiplied by the scalar weight $w$ to form $wp$, one of the terms that is sent to the summer. The other input, 1, is multiplied by a bias $b$ (also known as offset) and then sent to the summer. The summer output $n$ goes into a transfer function (or activation function) $f$, which produces the scalar neuron output $a$.

The output depends on the transfer function used. In reference to the bias, it is much like a weight, with the particularity that it has a constant input of 1. If it is necessary, the bias can be omitted.

Notice that $w$ and $b$ are both adjustable scalar parameters of the neuron. The main idea of neural networks is that such parameters can be adjusted so that the network has some desired or interesting behaviour. Thus, it is possible to train the network to do a particular job by adjusting the weight or bias parameters, or perhaps the network itself will adjust these parameters to achieve some desired end, [3].

### 1.4.2. Transfer functions

There are many transfer functions. A particular transfer function is chosen to satisfy some specification required by the problem that the neuron is trying to solve. Three of the most commonly used functions are shown below.

![Hard-limit transfer function](image)

$$a = \text{hardlim}(n)$$

Figure 1.3. Hard-limit transfer function, [4].

The hard-limit transfer function shown above (figure 1.3) limits the output of the neuron to either 0, if the net input argument $n$ is less than 0, or 1, if $n$ is greater than or equal to 0.
Figure 1.4 illustrates the linear transfer function.

![Linear transfer function](image)

Figure 1.4. Linear transfer function, [4].

Neurons of this type are used as linear approximators.

The sigmoid transfer function shown below (figure 1.5) takes the input, which can have any value between plus and minus infinity, and squashes the output into the range 0 to 1.

![Log-sigmoid transfer function](image)

Figure 1.5. Log-sigmoid transfer function, [4].

This transfer function is commonly used in backpropagation networks, in part because it is differentiable.

The symbol in the square to the right of each transfer function graph shown above represents the associated transfer function. These icons replace the general $f$ in the boxes of network diagrams to show the particular transfer function being used.
Transfer Functions. [4]

- compet: Competitive transfer function
- hardlim: Hard limit transfer function
- hardlims: Symmetric hard limit transfer function
- logsig: Log-sigmoid transfer function
- netinv: Inverse transfer function
- poslin: Positive linear transfer function
- purelin: Linear transfer function
- radbas: Radial basis transfer function
- satlin: Saturating linear transfer function
- satlins: Symmetric saturating linear transfer function
- softmax: Softmax transfer function
- tansig: Hyperbolic tangent sigmoid transfer function
- tribas: Triangular basis transfer function

In this list are all the representation symbols for the transfer functions available in MATLAB, including the code to use for each function.

Only three of the most used transfer functions have been explained above. But it is useful to know all this kind of functions that MATLAB can use. Showing only the transfer functions graphics is easy to understand how they work. For this reason, there is a figure (1.6) below that shows the graphics of all transfer functions.
Figure 1.6. Transfer functions graphics, [4].
1.4.3. Neuron with vector input

Usually, a neuron has more than one input. A neuron with R inputs is shown in figure 1.7. The individual inputs $p_1, p_2, \ldots, p_R$ are multiplied by weights $w_{1,1}, w_{1,2}, \ldots, w_{1,R}$ and the weighted values are fed to the summing junction. Their sum is simply $Wp$, the dot product of the (single row) weight matrix $W$ and the vector $p$.

The neuron has a bias $b$, which is summed with the weighted inputs to form the net input $n$. This sum, $n$, is the argument of the transfer function $f$.

$$n = w_{1,1}p_1 + w_{1,2}p_2 + \ldots + w_{1,R}p_R + b$$

1.4.4. Abbreviated notation

The figure of a single neuron shown before (figure 1.7) contains a lot of detail. When networks with many neurons are considered, and perhaps layers of many neurons, there is so much detail that the main thoughts tend to be lost. Normally is used an abbreviated notation for an individual neuron. This notation for multiple-input neuron is shown.
As shown in figure 1.8, the input vector \( p \) is represented by the solid dark vertical bar at the left. The dimensions of \( p \) are displayed below the variable as \( R \times 1 \), indicating that the input is a single vector of \( R \) elements. These inputs go to the weight matrix \( W \), which has \( R \) columns but only one row in this single neuron case. A constant 1 enters the neuron as an input and is multiplied by a scalar bias \( b \). The net input to the transfer function \( f \) is \( n \), which is the sum of the bias \( b \) and the product \( Wp \). The neuron’s output \( a \) is a scalar in this case. If there was more than one neuron, the network output would be a vector.

A layer of a network is defined in figure 1.8. A layer includes the combination of the weights, the multiplication and summing operation (here realized as a vector product \( Wp \)), the bias \( b \), and the transfer function \( f \). The array of inputs, vector \( p \), is not included in the layer.

Each time this abbreviated network notation is used, the sizes of the matrices are shown just below their matrix variable names. This notation will allow to understand the architectures and follow the matrix mathematics associated with them.

When a specific transfer function is to be used in a figure, the symbol for that transfer function replaces the \( f \) shown above in section 1.4.2.

1.5. Network architectures

Usually, one neuron, even with many inputs, may not be enough. In that case, two or more neurons can be combined, working in parallel, in what is called a layer. And a network could contain one or more layers. The concept of a layer is explained in section 1.5.1.

1.5.1. A layer of neurons

A single-layer network of \( S \) neurons and with \( R \) input elements is shown in figure 1.9.
The layer includes the weight matrix, the summers, the bias vector \( b \), the transfer functions boxes and the output vector \( a \). Normally, the inputs are not considered as a layer.

In that case, each element of the input vector \( p \) is connected to each neuron through the weight matrix \( W \). Each neuron has a summer that adds its weighted inputs and bias to form its own scalar output \( n_i \). All \( n_i \) outputs taken together form an \( S \)-element net input vector \( n \). Finally, the neuron layer outputs \( a_i \) form a column vector \( a \), which is the output of the layer. The expression for \( a \) is shown in figure 1.9, at the bottom of the layer graphic.

It is common for the number of inputs to a layer to be different from the number of neurons (i.e., \( R \) is not necessarily equal to \( S \)). A layer is not constrained to have the number of its inputs equal to the number of its neurons.

It is not compulsory that all the neurons in a layer have the same transfer function. It is possible to create a single (composite) layer of neurons having different transfer functions simply by combining two of the networks shown in figure 1.9 in parallel. Both networks would have the same inputs, and each network would create some of the outputs, [3].

The input vector elements enter the network through the weight matrix \( W \):
The row indices of the elements of matrix $W$ indicate the destination neuron associated with that weight, while the column indices indicate the source of the input for that weight. In this way, the indices in $w_{1,2}$ mean that the strength of the signal from the second input element to the first (and only) neuron is $w_{1,2}$.

To simplify, the $S$-neuron $R$-input one-layer network also can be drawn in abbreviated notation (figure 1.10).

1.5.2. Inputs and layers

Considering a network with several layers, each layer has its own weight matrix $W$, its own bias vector $b$, a net input vector $n$ and an output vector $a$. It is necessary to introduce additional notation to distinguish between these layers. In that case, superscripts are used to identify the layers. Specifically, it is necessary to make a distinction between weight matrices that are connected to inputs and weight matrices that are connected between layers. It also needs to identify the origin and destination for the weight matrices.
Weight matrices connected to inputs are known as input weights; weight matrices coming from layer outputs are called layer weights. For this purpose, superscripts are used to identify the origin (second index) and the destination (first index) for the different weights and other elements of the network. To show that, the single-layer multiple input network shown earlier is redrawn in abbreviated form below.

As shown in figure 1.11, the weight matrix connected to the input vector $p$ is labeled as an input weight matrix ($IW^{1,1}$) having an origin 1 (second index) and a destination 1 (first index). Elements of layer 1, such as its bias, net input, and output have a superscript 1 to say that they are associated with the first layer.

Multiple layers of neurons uses layer weight (LW) matrices as well as input weight (IW) matrices.

**1.5.3. Multiple layers of neurons**

A network can have several layers. Each layer has its own weight matrix $W$, its own bias vector $b$, a net input vector $n$ and an output vector $a$. Some additional notation is necessary to be introduced to distinguish between the weight matrices, output vectors, etc., for each of these layers. In that case, the number of the layer is appended as a superscript to the names for each of these variables. The use of this layer notation can be appreciated in the three-layer network shown below, and in the equations at the bottom of the figure 1.12.
The three-layer network shown in figure 1.12 has \( R \) inputs, \( S^1 \) neurons in the first layer, \( S^2 \) neurons in the second layer, etc. Commonly, different layers can have different numbers of neurons. In that case, in each layer a constant input 1 is fed to the bias for each neuron.

The outputs of each intermediate layer (1 and 2) are the inputs for layers 2 and 3. In this way, layer 2 can be viewed as a one-layer network with \( R=S^1 \) inputs, \( S=S^2 \) neurons, and an \( S^2 \times S^1 \) weight matrix \( W^2 \). The input to layer 2 is \( a^1 \) and the output is \( a^2 \). Now that all the vectors and matrices of layer 2 have been identified, it can be treated as a single-layer network on its own. The same approach can be taken with any layer of the network.

The layers of a multilayer network play different roles. A layer whose output is the network output is called an output layer. All other layers are called hidden layers. The three-layer network shown before has an output layer (layer 3) and two hidden layers (layer 1 and layer 2).

The same three-layer network used to explain the layers can also be drawn using abbreviated notation.
Multilayer networks are more powerful than single-layer networks. For example, a two-layer network having a sigmoid first layer and a linear second layer can be trained to approximate most functions (with a finite number of discontinuities) arbitrarily well.

In this example it is assumed that the output of the third layer, $a^3$, is the network output of interest, and this output is labeled as $y$. This notation is used to specify the output of multilayer networks, [3].

### 1.6. Data structures

In this section is discussed in which way the format of input data structures affects the simulation of the network. In first place, the static networks, and then the dynamic networks.

There are two basic types of input vectors: those that occur concurrently, that is at the same time or in no particular time sequence, and those that are as a sequence in time. For concurrent vectors, the order is not important, and if there were a number of networks running in parallel, one input vector could be presented to each of the networks. But in case of sequential vectors, the order in which the vectors appear is important.
1.6.1. Concurrent inputs in a static network

The simplest situation to simulate a neural network occurs when the network to be simulated is a static one, it means that has no feedback or delays. In this case, is not necessary to be concerned about whether or not the input vectors occur sequentially, so the inputs can be treated as concurrent. In addition, the problem is simpler with the assumption that the network has only one input vector.

A single matrix of concurrent vectors is presented to the network, and consequently the network produces a single matrix of concurrent vectors as output. The result would be the same in case there were several networks operating in parallel and each network received one of the input vectors and produced one of the outputs. The order of the input vectors is not important, because they do not interact with each other, [3].

1.6.2. Sequential inputs in a dynamic network

In the situation that a network contains delays, the input to the network would usually be a sequence of input vectors that occur in a certain time order. To explain this case, the next figure (1.15) shows a simple network containing only one delay.
The input of the network is a cell array that contains a sequence of inputs, and then the network returns a cell array containing a sequence of outputs. The order of the inputs is important when they are presented as a sequence. In this case, the current output is obtained by multiplying the current input by \( w_{1,1} \) and the preceding input (due to the delay) by \( w_{1,2} \) and adding both values as a result. In case the order of the inputs was changed, the numbers obtained in the output would change.

1.6.3. Concurrent inputs in a dynamic network

If a definite input were applied as a set of concurrent inputs instead of a sequence of inputs, a completely different response would be obtained as a result. But this situation is not usual, because it has no sense. It would be as if each input were applied concurrently to a separate parallel network.

In some particular cases, might be wanted to simulate the network response to several different sequences at the same time. In this case, it would be normal to present the network with a concurrent set of sequences. The input should be a cell array, where each element of the array contains the two elements of the two sequences that occur at the same time.

The first column of each matrix makes up the output sequence produced by the first input sequence. The second column of each matrix makes up the output sequence produced by the second input sequence. There is no interaction between the two
concurrent sequences. It is as if they were each applied to separate networks running in parallel.

The following diagram shows the general format for the input $P$ to the simulation function when there are $Q$ concurrent sequences of $TS$ different time steps. This sequence describes all cases where there is a single input vector. Each element of the cell array is a matrix of concurrent vectors that correspond to the same point in time for each sequence. If there are multiple input vectors, there will be multiple rows of matrices in the cell array, [3].

\[
\{ [p_1(1), p_2(1), \ldots, p_Q(1)], [p_1(2), p_2(2), \ldots, p_Q(2)], \ldots, [p_1(TS), p_2(TS), \ldots, p_Q(TS)] \}
\]

In this section, sequential and concurrent inputs are applied to dynamic networks. It is also possible to apply sequential inputs to static networks. It does not change the simulated response of the network, but it can affect the way in which the network is trained.

1.7. Backpropagation

1.7.1. Introduction

Backpropagation is a generalization of the Widrow-Hoff learning rule to be used with multiple-layer networks and nonlinear differentiable transfer functions. Input vectors and the corresponding target vectors are used to train a network until it can approximate a function, associate input vectors with specific output vectors, or classify input vectors in an appropriate way as defined by the user. Networks with biases, a sigmoid layer, and a linear output layer are capable of approximating any function with a finite number of discontinuities.
Standard backpropagation is a gradient descent algorithm, as is the Widrow-Hoff learning rule, in which the network weights are moved along the negative of the gradient of the performance function. The term backpropagation refers to the way in which the gradient is computed for nonlinear multilayer networks. There are a number of variations on the basic algorithm that are based on other standard optimization techniques, such as conjugate gradient and Newton methods.

Properly trained, backpropagation networks tend to give reasonable answers when presented with inputs that they have never seen. Usually, a new input drives to an output similar to the correct output for input vectors used in training that are similar to the new input being presented. This generalization property makes it possible to train a network on a representative set of input/target pairs and get good results without training the network on all possible input/output pairs. It is a time efficient method, [3].

There are generally four steps in the training process:
1. Assemble the training data.
2. Create the network object.
3. Train the network.
4. Simulate the network response to new inputs.

There are different training functions, but using each function generally follows these four steps. Only the one to be used will be explained.

In the next section about architectures, the basic feedforward network structure and demonstrates how to create a feedforward network object are described. Those points are the base for the dynamic networks used later in this work.

1.7.2. Architecture

This section presents the structure of the network most commonly used with the backpropagation algorithm, the multilayer feedforward network.
1.7.2.1. Neuron model

As follows, there is a brief explanation about the neuron elements. It is the same told in section 1.4.3, but used to remember simply how it works. So, an elementary neuron with \( R \) inputs is shown in figure 1.16. Each input is weighted with an appropriate \( w \). The addition of the weighted inputs and the bias forms the input \( n \) to the transfer function \( f \). Neurons can use any differentiable transfer function \( f \) to generate their output.

![Elementary neuron with \( R \) elements input vector](image1)

Figure 1.16. Elementary neuron with \( R \) elements input vector, [4].

Multilayer networks often use the log-sigmoid transfer function.

![Log-sigmoid transfer function](image2)

Figure 1.17. Log-sigmoid transfer function, [4].

The function log-sigmoid generates outputs between 0 and 1 as the neuron’s net input goes from negative to positive infinity.

Alternatively, multilayer networks can use the tan-sigmoid transfer function, which produces outputs between -1 and 0 for negative inputs and between 0 and +1 for positive inputs.

![Tan-sigmoid transfer function](image3)

Figure 1.18. Tan-sigmoid transfer function, [4].
Occasionally, the linear transfer function purelin (linear function) is used in backpropagation networks.

![Linear transfer function](image19)

**Figure 1.19. Linear transfer function, [4].**

If the last layer of a multilayer network has sigmoid neurons, then the outputs of the network are limited to a small range. If linear output neurons are used, then the network outputs can take on any value. It is recommended to use linear neurons in the output layer.

The three transfer functions described here are the most commonly used transfer functions for backpropagation, but other differentiable transfer functions can be created and used with backpropagation if desired. All other transfer functions are shown in figure 1.6, contained in section 1.4.2.

### 1.7.2.2. Feedforward network

A single-layer network of $S$ log-sigmoid neurons having $R$ inputs is shown in full detail on the left and with a layer diagram on the right in figure 1.20.

![Feedforward network diagram](image20)
Feedforward networks often have one or more hidden layers of sigmoid neurons followed by an output layer of linear neurons. Multiple layers of neurons with nonlinear transfer functions allow the network to learn nonlinear and linear relationships between input and output vectors. The linear output layer lets the network produce values outside the range –1 to +1.

On the other hand, if the purpose is to constrain the outputs of a network (such as between 0 and 1), then the output layer should use a sigmoid transfer function.

For multilayer networks, the number of layers determines the superscript on the weight matrices. The appropriate notation is used in the two-layer tan-sigmoid/linear network shown in figure 1.21.

![Figure 1.21. Two-layer network with tansig function in the hidden layer and purelin in the output layer, [4].](image)

This particular network can be used as a general function approximator. It can approximate any function with a finite number of discontinuities arbitrarily well, if the hidden layer contains sufficient neurons.

Referred specifically to the software used later for the research (MATLAB), the first step in training a feedforward network is to create the network object. The function with name `newff` creates a feedforward network. It requires three arguments and returns the network object. The first argument is a matrix of sample R-element input vectors. The second argument is a matrix of sample S-element target vectors. The sample inputs and
outputs are used to set up network input and output dimensions and parameters. The third argument is an array containing the sizes of each hidden layer (number of neurons in each layer). The output layer size is determined from the targets.

More optional arguments can be provided. For instance, the fourth argument is a cell array containing the names of the transfer functions to be used in each layer. The fifth argument contains the name of the training function to be used. If only three arguments are supplied, the default transfer function for hidden layers is the tan-sigmoid, and the default for the output layer is the linear purelin, [3].

This command creates the network object and also initializes the weights and biases of the network; so, the network is ready for training.

1.7.3. Backpropagation algorithm

There are many variations of the backpropagation algorithm, but only one (that used for this work) of them will be explained afterwards without going deeply inside the mathematics associated, because the main purpose is to explain briefly the working method. The simplest implementation of backpropagation learning rule updates the network weights and biases in the direction in which the performance function decreases most quickly, the negative of the gradient. An iteration of this algorithm can be written as follows:

\[ \mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k \]

where \( \mathbf{x}_k \) is a vector of current weights and biases, \( \mathbf{g}_k \) is the current gradient, and \( \alpha_k \) is the learning rate.

There are two different ways in which this gradient descent algorithm can be implemented: incremental mode and batch mode. In the incremental mode, the gradient is computed and the weights are updated after each input is applied to the network. In batch mode, all the inputs are applied to the network before the weights are updated. That means the batch mode is quite faster than the incremental mode.
1.7.4. Faster training

Usual backpropagation training methods like gradient descent, and gradient descent with momentum are usually too slow for practical problems. There are high-performance algorithms that can converge from ten to one hundred times faster than other algorithms. All these faster algorithms operate in batch mode and can be classified in two categories. The first category uses heuristic techniques, which were developed from an analysis of the performance of the standard steepest descent algorithm. One heuristic modification is the momentum technique, for example. But there are two more heuristic techniques: variable learning rate backpropagation, and resilient backpropagation, between many others, [3].

The second category of fast algorithms uses standard numerical optimization techniques. This category includes three types of numerical optimization techniques for neural network training:
- Conjugate gradient
- Quasi-Newton
- Levenberg-Marquardt

Only the Levenberg-Marquardt will be explained because this is the algorithm used in this work.

1.7.4.1. Levenberg-Marquardt algorithm

As the quasi-Newton methods, the Levenberg-Marquardt algorithm was designed to approach second-order training speed without having to compute the Hessian matrix (the matrix of the second-order partial derivatives). When the performance function has the form of a sum of squares (as is typical in training feedforward networks), then the Hessian matrix can be approximated as

$$ H = J^T J $$

and the gradient can be computed as

$$ g = J^T e $$
where $J$ is the Jacobian matrix that contains first derivatives of the network errors with respect to the weights and biases, and $e$ is a vector of network errors. The Jacobian matrix can be computed through a standard backpropagation technique that is much less complex than computing the Hessian matrix.

The Levenberg-Marquardt algorithm uses this approximation to the Hessian matrix in the following Newton-like update:

$$x_{k+1} = x_k - [J^T J + \mu I]^{-1} J^T e$$

When the scalar $\mu$ is zero, this is just Newton’s method, using the approximate Hessian matrix. When $\mu$ is large, this becomes gradient descent with a small step size. Newton’s method is faster and more accurate near an error minimum, so the objective is to go to Newton’s method as quickly as possible. Thus, $\mu$ is decreased after each successful step (that means reduction in performance function) and is increased only when a tentative step would increase the performance function. In this way, the performance function is always reduced at each iteration of the algorithm.

This algorithm seems to be generally the fastest method for training moderate-sized feedforward neural networks (up to several hundred weights). It also has an efficient implementation in MATLAB software, because the solution of the matrix equation is a built-in function, [3].

### 1.7.4.2. Reduced memory Levenberg-Marquardt

The main inconvenience of the Levenberg-Marquardt algorithm is that it requires the storage of some matrices that can be quite large for certain problems. The size of the Jacobian matrix is $Q \times n$, where $Q$ is the number of training sets and $n$ is the number of weights and biases in the network. But this matrix does not have to be computed and stored as a whole. For example, if the Jacobian were divided into two equal submatrices, then could be computed the approximate Hessian matrix as follows:

$$H = J^T J = \begin{bmatrix} J_1^T & J_2^T \end{bmatrix} \begin{bmatrix} J_1 \\ J_2 \end{bmatrix} = J_1^T J_1 + J_2^T J_2$$
Therefore, the full Jacobian does not have to exist at one time. The approximate Hessian can be computed by summing a series of subterms. Once one subterm has been computed, the corresponding submatrix of the Jacobian can be cleared.

When this algorithm is used, there is a parameter that determines how many rows of the Jacobian are to be computed in each submatrix. If this memory reducing parameter is set to 1, then the full Jacobian is computed, and no memory reduction is achieved. If memory reducing is set to 2, then only half of the Jacobian is computed at one time. This saves half the memory used by the calculation of the full Jacobian.

There is a problem to using memory reduction. A significant computational overhead is associated with computing the Jacobian in submatrices. If enough memory is available, then it is better to set memory reducing to 1 and to compute the full Jacobian. If a large training set is used, and the computer is running out of memory, then memory reduction should be set to 2 and try again. If it still runs out of memory, continue to increase memory reduction.

Even if memory reduction is used, the Levenberg-Marquardt algorithm will always compute the approximate Hessian matrix, which has dimensions $n \times n$. If the network is very large, then the computer could run out of memory. If this is the case, try other training functions, or one of the conjugate gradient algorithms, [3].

## 1.8. Dynamic neural networks

### 1.8.1. Applications

Dynamic networks are generally more powerful than static networks (but generally more difficult to train). Because dynamic networks have memory, they can be trained to learn sequential or time-varying patterns. This has applications in such different areas as prediction in financial markets, channel equalization in communication systems, phase detection in power systems, sorting, fault detection, speech recognition and a long list with other areas.
1.8.2. Dynamic network structures

The software used (MATLAB) is designed to train a class of network called the Layered Digital Dynamic Network (LDDN). Any network that can be arranged in the form of an LDDN can be trained with this software. Here is a basic description of the LDDN.

Each layer in the LDDN is made up of the following parts:

- Set of weight matrices that come into that layer (which can connect from other layers or from external inputs), associated weight function rule used to combine the weight matrix with its input (normally standard matrix multiplication, known as dotprod), and associated tapped delay line
- Bias vector
- Net input function rule that is used to combine the outputs of the various weight functions with the bias to produce the net input (normally a summing junction, called netprod)
- Transfer function

The network has inputs that are connected to special weights, called input weights, and designated by $\text{IW}^{i,j}$, where $j$ is the indicator of the number of the input vector that enters the weight, and $i$ denotes the number of the layer to which the weight is connected. The weights connecting one layer to another are called layer weights and are denoted by $\text{LW}^{i,j}$, where $j$ indicates the number of the layer coming into the weight and $i$ denotes the number of the layer at the output of the weight. Figure 1.22 is an example of a three-layer LDDN. The first layer has three weights associated with it: one input weight, a layer weight from layer 1, and another layer weight from layer 3. The two layer weights have tapped delay lines associated with them, [3].
MATLAB can be used to train any LDDN, so long as the weight functions, net input functions, and transfer functions have derivatives. Most well-known dynamic network architectures can be represented in LDDN form.

**1.8.2.1. NARX network**

The Nonlinear AutoRegressive network with eXogenous inputs (NARX) is a recurrent dynamic network, with feedback connections to several layers of the network. The NARX model is based on the linear ARX model, which is commonly used in time-series modelling.

The defining equation for the NARX model is:

\[
y(t) = f(y(t-1), y(t-2), \ldots, y(t-n_y), u(t-1), u(t-2), \ldots, u(t-n_u))
\]

where the next value of the dependent output signal \(y(t)\) is regressed on previous values of the output signal and previous values of an independent (this means exogenous) input signal. The NARX model can be implemented by using a feedforward neural network to approximate the function \(f\). A diagram of the resulting network is shown in figure 1.23, where a two-layer feedforward network is used for the approximation. This
implementation also allows for a vector ARX model, where the input and output can be multidimensional.

There are many applications for the NARX network. It can be used as a predictor, to predict the next value of the input signal. It can also be used for nonlinear filtering, in which the target output is a noise-free version of the input signal. The use of the NARX network is demonstrated in another important application, the modelling of nonlinear dynamic systems. This is one of the reasons to use this type of networks to develop the research of this work, because the suspension system of a car is a nonlinear system and a dynamic network is needed, [3].

Without enter in the training of the NARX network, an important configuration that is useful in training needs explanation. The output of the NARX network can be considered to be an estimate of the output of some nonlinear dynamic system that is trying to model. The output is fed back to the input of the feedforward neural network as part of the standard NARX architecture, as shown in the left of the figure 1.24. Because the true output is available during the training of the network, it is possible to create a series-parallel architecture, in which the true output is used instead of feeding back the estimated output, as shown in the right of the figure 1.24. This has two advantages. The first is that the input to the feedforward network is more accurate. The second is that the

![NARX network structure schematic, [4].](image)

- 37 -
resulting network has a purely feedforward architecture, and static backpropagation can be used for training.

At this point, all the theoretical basement necessary to develop this work has been explained. Starting for the concept of a neuron, going through the architectures of the networks and the training to process, to finish with dynamic neural networks, trying to explain in a logical order the concepts used in the process to create, train and simulate a dynamic neural network.

In the next chapters, the application of the theory shown before and others will be developed.
2. Models

2.1. Physical model (SIMPACK)

Although the work is for multi-body mechanical systems in general, in this case the system used to develop the study and research is the suspension system of a touring car. More detailed, only one front wheel from the suspension system is used.

This suspension model has not been specially developed for this work, it was previously modelled in SIMPACK. The model is represented in the pictures below.

Figure 2.1. Perspective view of the SIMPACK model

Figure 2.2. Top view of the SIMPACK model
The most important elements for the suspension are included in the model. These parts are: the wheel, the joints to connect the wheel to the structure and the most important elements, the spring and the damper. All these parts can be easily appreciated in the images. In this case the shock absorber and the spring are coaxial, that means the damper is mounted in the interior of the spring.

In order to make the simulation easier, the wheel has no steering angle. The reason is because the main purpose is to determine if neural networks are appropriate to simulate and predict physical parameters in this type of mechanical systems, so that one of the premises is to have a simple model but really near to the real. If it works properly for this model, then some more detailed models could be used for future works.

SIMPACK allows, a part to create the model, to work and make simulations with it. In this case, the input parameter for the system to run the simulation is the road profile and the point to make all measurements is the centre of the wheel (respect to the simple body in the model that represents the car body). As MATLAB offers the possibility, the input signal is created with this software and then sent to the SIMPACK model to simulate the behaviour of the system. And the evolution along the time of the selected output variables (registered values) is also sent back and saved to be shown in MATLAB. This process to link the input data, the model and the output data between SIMPACK and MATLAB is known as co-simulation and is implemented in SIMULINK. This is a truly powerful and useful environment for multidomain simulation and model-based design for dynamic and embedded systems. It provides an interactive graphical environment and a customizable set of block libraries that let to design, simulate, implement, and test a variety of time-varying systems.
2.2. Simulink model

In the next picture (figure 2.3), the block schematic to implement the co-simulation is shown.

In the block diagram, and from left to right, first of all the input data block is found. As said before, the input signal is the road surface profile. A sine wave is used in that case, with four different sine waves changing the amplitude, the frequency and the phase during the time simulation (20 seconds, with 5 seconds for each type of wave). Because this input signal is made by different sine waves, the easiest and fastest way to create it is making each wave separately and then adding each signal with the others along the time simulation. The values of the compound signal (1 value each 0.001 seconds, it means 20000 values during the simulation time) are introduced in a matrix, saved with the name “f” and the file format for a matrix (.mat).

The next part is a Scope (Scope1), a block used to display the temporary evolution of the input signal.

Figure 2.3. Simulink co-simulation model block schematic
Following that, there is the SIMPACK co-simulation interface block, where some parameters can be defined and changed to configure co-simulation. This block is the link between MATLAB and SIMPACK.

The simulation output is sent to a demultiplexer to separate all data from the simulation output into the different variables, in this case the displacements and the rotations respect the three axis of the coordinate system. Then, these 6 signals are sent to a multiplexer, in order to show them at the same time in the graphic interface displayed by the Scope situated in the central position on the right side.

From Scope2 to Scope7, these are all to view the temporal evolution of the 6 output signals separately. And the blocks called “to Workspace” are to send the values of each variable to the workspace of MATLAB, but they are not necessary because all values of each variable are saved individually.

2.3. Co-simulation file

In the image below there is the code to write for the co-simulation file. Without this file it would not be possible to run the co-simulation.

```matlab
% Writte CoSimulation
% Über Aufruf 'runCooSim' ohne Argumente liefert eine kurze Einführung

simOut = runCooSim('SIMPACK Model file location', 'Simulink model file location', true, simulation time in seconds);

data_input = simOut(:,1);
data_output = simOut(:,3);

save data_input data_output
save x x
save y y
save z z
save alfa alfa
save beta beta
save gama gama
```

Figure 2.4. Co-simulation file MATLAB code
All the variables and values from the function “runCoSim” are saved together in the variable called “simOut”. For the function “runCoSim” is necessary to write the exact SIMPACK model file location, the Simulink model file location, the word “true” and the time for the simulation in seconds.

After running the co-simulation, the input and output signals are separated and assigned to a variable name to be able to distinguish them.

And finally, all data values are saved to use them later in MATLAB.

After running this file, the computer automatically sends the input data to the model in SIMPACK, runs the simulation, sends the output data to MATLAB and saves them in the correct variables name.
3. Method

The first fact to consider is which dynamic neural network type will be used. According to the characteristics described in the last section of the first chapter, the most suitable type to be used is the NARX architecture network.

This selection has many reasons. In this case, the network should be used to predict the output values (displacements and angles) for a selected point of the suspension system depending on the input signal values. In other words, the target is to model the behaviour of the suspension due to the road profile. Other reasons are that the variables are time-dependent, delays are needed and the suspension system is considered as nonlinear.

A dynamic network is used due to the mechanical system selected to work with and the input and output signals to use.

The NARX network is based on the feedforward network. The most reasonable option would be to use a NARX network from the beginning. This was attempted, but due to memory problems the training process was too slow and it was no possible to use this kind directly. Instead of that, the best way to solve this problem is using another configuration of the NARX architecture. The way to proceed is creating a series-parallel NARX to have no problems and make a fast training. Because the true output (necessary to know it to train the network) is available during the training of the network, a series-parallel architecture can be used, in which the true output is used instead of feeding back the estimated output. This has two advantages:

The input to the feedforward network is more accurate.

The resulting network has a purely feedforward architecture, and static backpropagation can be used for training.

After the series-parallel network has been trained, the next step is to transform the series-parallel network to a parallel network. After making this change, the new parallel network will be ready to work with new input data and simulate the output.

Then, the MATLAB code used to create, train and transform the network will be explained with detail in the Appendix A, line by line to let anybody understand why is
necessary every command. Following, there is a picture (figure 3.1), only to show the code used:

```matlab
% Create and train a new SF NARX for Z coordinate
1  load data_input;
2  load z;
3  u=(data_input)';
4  y=(z)';
5  u=conv2eq(u);
6  y=conv2eq(y);
7  p=u(3:end);
8  t=y(3:end);
9  d1=[1:2];
10  d2=[1:2];
11  net=newnarxsp(p,t,d1,d2,[10 8]);
12  net.trainParam.show=20;
13  net.trainParam.epochs=200;
14  Pi=[u(1:2); y(1:2)];
15  net=train(net,[p;t],t,Pi);
16  yp=sim(net,[p;t],P);  
17  o=cell2mat(yp);  
18  plot(e)
```

```matlab
% Transformation of the SF NARX to a Parallel NARX and simulation
19  net2 = sp2narx(net);
20  ys1=ys;
21  ul=u;
22  pl = ul(3:end);
23  tl = ys1(3:end);
24  Pi=[ul(1:2)];
25  ypl = sim(net2,P1,P);
26  plot([cell2mat(ypl)' cell2mat(tl)'])
```

Figure 3.1. MATLAB code necessary to create, train and simulate the network for this work
4. Results

In this chapter the results are presented graphically and discussed.

First of all, only the vertical displacement (in the direction of the z axis of the coordinate system) of the selected point from the suspension has been used for the study. The main reason is due to in the movement of a car, the most notorious in normal driving conditions is the vertical displacement of the car. And in the suspension system usually this is the largest value in terms of the wheel displacement, specially given by the road profile in a straight line (without steering angle).

These have been the previous considerations before applying the method described in the previous chapter.

Before show and discuss the results, it would be better to present and describe the input signal used during the training and the simulation of the network. It is presented in the figure below.

![Figure 4.1. Input signal (road profile) for the SIMPACK model and neural network simulation](image)

The road profile (input signal) is compound of 4 different sine waves:
- From 0 to 5 seconds the amplitude is 0.02 meters and the frequency is 4 rad/s.
- From 5 to 10 seconds the amplitude is 0.01 meters and the frequency is 6 rad/s.
- From 10 to 15 seconds the amplitude is 0.07 meters and the frequency is 0.62 rad/s.
- From 15 to 20 seconds the amplitude is 0.03 meters and the frequency is 3 rad/s.
The different samples used for the road profile are not realistic, although they could be implemented with a similar form in a test track for suspension systems. But this is a theoretical case to make a simulation, and a profile like this could be found in a test track or using a test bench for a suspension system.

The graphics presented below are a selection of the best results from more than one hundred attempts changing some parameters as the number of hidden layers and the number of neurons in each or the maximum number of epochs to reach before stop the training process.

Notice that in all graphics the x-axis represents the time in milliseconds and the y-axis represents the vertical displacement of the selected point from the car in meters. In blue is the output from the SIMPACK model simulation and in green colour is the result obtained using the neural network for the simulation.

Figure 4.2. Plot of the best neural network simulation results achieved
This graphic (figure 4.2) is the best obtained over all. The response of the system to the input signal using the neural network is really good. If it is compared with the output from the SIMPACK simulation, there are some differences between them, but the two outputs are really near in almost all points. So, this shows that is possible to simulate this system with a dynamic neural network, but the results are really dependent from the hidden layers, the number of neurons in each and the number of epochs. There are some graphics below that show the differences in the obtained results depending on these factors.

Figure 4.3. Graphics of the different simulation results (changing hidden layers and maximum epochs)
These are only a few of the results obtained among more than 100 simulations done. In all cases the input signal has been the same presented at the beginning of this chapter. This means that the input data used are the same for the network training process and for the simulation. Using more hidden layers or neurons in each one, or even a large number of epochs helps to obtain better results. Also, as more layers and neurons used, the training time increases in a fast way. It is necessary some experience and more investigation about this to find easily the values for these parameters without doing hundreds of simulations, because that means a lot of time would be necessary to do that. This is the first step to find the way to simulate with a neural network. After getting a good result in that situation, the next should be to try a simulation with the network but changing the input data.
That is the case for the graphic showed below (figure 4.5).

![Simulation graph](image)

**Figure 4.5.** Plot for the simulation results using only the first 5 seconds of the original input

The input used for this simulation is only a part of the input used to train and simulate the network for the first time, that is, the sine wave between 0 and 5 seconds of time (amplitude=0.02 meters and frequency=4 rad/s).

As seen in the graphic, the result from the simulation is quite different from the same with the SIMPACK model. In this case, 3 hidden layers with 3, 4, 4 neurons in each respectively and with 300 epochs have been used (the same configuration that gave the best result with the original input data). It would have been expected to obtain a good result, similar as the obtained before (shown in figure 4.2), because the input signal is exactly the first part of the signal used before in the best case to create and train the network.

The results shown in figure 4.5 are not good, but they are not as bad as they could seem. First of all, the output signal of the neural network shows a tendency to have a
similar behaviour as the SIMPACK model, but the frequency of the signal is higher than it should be. An explanation for this fact is that neural networks present sensitivity. This means that when creating a network with exactly the same parameters, training data, etc, several times, the network is never the same. This can cause different behaviour because the network changes, and is impossible to reproduce the network exactly even using all the same inputs, data sets, parameters, etc, as said before. A solution to reduce sensitivity in neural networks should be investigated for the application in further works.

As told some lines before, changing some parameters (hidden layers, neurons in each layer and number of epochs) would probably make to get better results. But this is not the main objective of the work. And this requires time to investigate how these parameters affect to the results given from the network, and if it would be a method or a rule to find which should be approximately the most appropriate values for these. And also, some experience in neural networks would be helpful to make it faster. Unfortunately, for the moment there is not any rule to deal with the problem, and only a large experience in the topic and neural networks is the best method to be more effective.
Conclusion

Simulation tools have been widely used to complement experimentation for mechanical design (for example in suspension systems) in the automotive industry not only for reducing development time, but also to allow the optimization of the vehicle performance. A mathematical representation that can accurately reflect the tyre dynamic behaviour while passing over different obstacles or road irregularities is fundamental for evaluating suspension quality and for developing new suspension control strategies, among others. Because the phenomenon is nonlinear, it is difficult to predict the actual performance by using a physical mathematical model. Two different types of models, which can be conveniently classified as physical models and purely numerical models (also known as black-box models), can be considered. And artificial neural networks, the object to make the research in the work, can be found among other numerical models.

The main advantage of using physically-based models is the ability to examine effects of physical changes of the system itself (variation of the system parameters). In addition, the behaviour of some system elements is not always well known (the tyre, for example) and, in this case, analytical models often require empirical adjustments of the model parameters due to the limited understanding of the physical phenomenon itself. Such parameters may be difficult to be estimated over a wide range of operating conditions, therefore they can limit sometimes the model applicability.

The artificial neural networks provide an alternative way for modelling a nonlinear dynamic system such as the tyre/suspension system. Neural networks offer a flexible modelling approach, which could model almost any tyre/suspension system once the correct architecture and the appropriate training data are defined. This has been the main objective of this work, to show if it was possible to model a suspension system using neural networks. And the results obtained conclude that it is possible with a medium-high accuracy, but is necessary a lot of work to do before achieving good results and they specially depend on the data used and the parameters selected when defining the network.

Commonly, networks are trained so that a particular input leads to a desired, or known, target output. Learning in neural networks is achieved by adapting the network
parameters so that the expected error between the network and the target output is minimized.

Recurrent networks allow a signal flow in the forward and backward directions, giving the network a dynamic memory useful to work with dynamic systems.

In order to capture the dynamics of nonlinear systems, there are two elements that can be used at the same time, to obtain a dynamic neural network model, memory lines, and feedback. In particular, a NARX is a neural network with an external memory. In fact, a NARX is a static Multi-Layer Perceptron Neural Network (MLPNN) whose inputs are filtered (by introducing delays only). The short-term memory used is the tapped delay line memory.

The second element used to give dynamic behaviour to the neural network is feedback. In NARX networks, the output is fed back to the input layer (global feedback) and so the MLPNN has information about the state of the system at previous temporal steps. The network needs two tapped delay line memories, generally not of the same order, applied to input and feedback signals.

It is necessary to define a network architecture in terms of input signals, number of layers, number of neurons of hidden layers, activation function of each layer, and orders of the tapped delay line memories to reproduce accurately the system dynamic behaviour. Because of all these reasons, the network architecture used for the work has been the NARX network. This is the best suitable neural network type to use with the suspension system due to its characteristics, that make it the best option. And the transfer functions associated to the layers has been used by default, the tan-sigmoid function for the hidden layers and the linear function for the output layer. These are the most recommended for this kind of systems.

About training, it is the procedure that allows the net to reach the desired response. Differently from other numerical methods, the parameters of the neural network (weights and offsets) are tuned by using the same input–output pairs more than one time. In this work, the weights and the offsets are found by batch procedure: at each iteration (an iteration is said epoch) all pairs are presented to the network. Usually, backpropagation
algorithm is used to train static neural networks. The algorithm applies input signals to the input layer to generate errors that are backpropagated from the output layer to the input layer to change weights and offsets. The algorithm minimizes a performance index that is normally represented by mean squared error (mse) or sum squared errors (sse) between the network response and the target values. It is possible to use backpropagation through time algorithm to train a dynamic network. Unfortunately, this algorithm is characterized by a low convergence rate. A different way to train a recurrent neural network is teacher forcing. With this method, the feedback loop is removed and the inputs coming from feedback are substituted by the responses of the actual system. This method is also called open-loop training or series-parallel training because the real system works in parallel with the network. This has been exactly the method used to train the network after trying to avoid the series-parallel training. That was the reason to use a series-parallel network for the training process and change to the parallel structure after that, in order to use this last structure for the network simulation.

So, the main conclusion is that it is possible to use dynamic neural networks in order to simulate a mechanical system, such as a vehicle suspension in that case. And this work has settled the base to continue investigating and performing in the topic. The basic method and the way to follow have been described. But there are some recommendations to do about future possibilities or research lines in order to develop and improve the method.

One similar focus to work with could be to try the implementation of a dynamic neural network to simulate other dynamics of the suspension system, such as forces transmitted to the suspension elements or to the vehicle structure. This is a different possibility to apply the method described in this text to the same or similar system. It would be interesting to know if it is more complicated for the neural networks to work and predict forces than displacements.

Another investigation line could be about how to improve the training process for the neural network, to make it faster and to get better results. This should be an investigation work about some of the most effective training algorithms to find the best training algorithm to be used depending on the system to simulate, the size of the network and other characteristics or parameters.
And finally, how to reduce the sensitivity for neural networks should be investigated, to avoid or reduce to the minimum this phenomenon.

Working and investigating about these main points should give a very important push to the development of the topic.
Bibliography


Appendix A: MATLAB code

There is a really important advice to give for the people who use MATLAB’s neural network toolbox, specially the functions “newnarx” and “newnarxsp”. It could be a mistake in these functions code that does not allow to use more than one hidden layer depending on the MATLAB version. To correct this error (found in the MATLAB Release 2007b) is necessary to go to the MATLAB script for every one of these functions. These files can be found through these directories in the disk that contains the program files:

ProgramFiles\MATLAB\R20...\toolbox\nnet\nnet\nnnetwork\newnarxsp.m.

Once the necessary MATLAB script opened, there is a code line that must be changed.

The mistake to correct is:

Change “net.inputConnect (1,N1) for this new code “net.inputConnect (1, 2)”.

After do that and save the changes to the file, this will allow to specify the hidden layers and how many neurons should contain each by the user.

The problem was that, with this mistake there was a problem in the connections of the neurons from the different hidden layers and the software had a bug that did not allow to work properly in case of use more than one hidden layer.

Then, the MATLAB code used to create, train and transform the network shown in figure A.1 is explained with detail, line by line, to let anybody understand why is necessary every command.
Line 3: load data_input
First of all is necessary to load all data to the MATLAB workspace, in order to have them available to work with. The input data, that is the road surface profile, is called "data_input".

Line 4: load z
This is to load to the workspace the output data, called "z" in the example because the displacement along z axis of the selected point from the mechanical system is the output to study.

Line 5: u=(data_input)'
This command assigns the transposed form of the input data (from a vector of 20000 rows by 1 column to a vector of 1 row by 20000 columns) to a new variable called “u”.

Line 6: ys=(z)'
In that case the transposed form of the output data is assigned to a new variable called “ys”.

Figure A.1. MATLAB code necessary to create, train and simulate the network for this work
Line 7: \texttt{u=con2seq (u)}

The purpose in that case is to convert the vector “u” from a concurrent to a sequential vector.

Line 8: \texttt{ys=con2seq (ys)}

Used to transform the concurrent vector “ys” to a sequential vector.

Line 9: \texttt{p=u (3:end)}

The new variable “p” contains the values from the third to the last of vector “u”.

Line 10: \texttt{t=ys(3:end)}

The new variable “t” contains the values from the third to the last of vector “ys”.

Line 11: \texttt{d1=[1:2]}

In that case “d1” are the number of delays for the input signal. To make the delays is necessary to write a sequence of numbers (as many of the number of delays the user wants to introduce). The condition is that the sequence must be of increasing positive integer numbers.

Line 12: \texttt{d2=[1:2]}

The variable “d2” contains the delays for the output data. In that case 2 delays.

Line 13: \texttt{net=newnarxsp (p, t, d1, d2, [10 8])}

This command is to create a new series-parallel NARX network called “net”. In the “newnarxsp” function there are: the input vectors (p), output or target vectors (t), input delay vector (d1), output delay vector (d2) and in the brackets the size of the hidden layers (in that case two layers with 10 neurons in the first and 8 in the second). The output layer size is determined from the dimensions of the target vector (t).

Line 14: \texttt{net.trainFcn='trainbr'}

This is to choose the training function for the network. The selected function is “trainbr”, that uses the reduced memory Levenberg-Marquardt algorithm.
**Line 15: net.trainParam.show=20**  
This command is not necessary, but is used to choose between how many epochs (iterations) MATLAB shows the progress of the training (here every 20 epochs). During the training a graphic user interface opens automatically and shows graphics of the training process progress that are really useful.

**Line 16: net.trainParam.epochs=200**  
Here, the user has the option to determine the maximum number of epochs used in the training. There are other parameters or conditions, like the mean squared error (mse), that can stop the training before reach the maximum number of epochs if the value is achieved.

**Line 17: Pi=[u (1:2); ys (1:2)]**  
This line is to define the initial input delay conditions. Here there are two delays for both inputs.

**Line 18: net=train (net, [p; t], t, Pi)**  
This is the function to train the network “net” according to “net.trainFcn” and “net.trainParam”. This function takes the network (net), the inputs (p; t) because although “t” is the output, at the same time it is one of the delayed inputs, the targets or output (t) and the initial input delay conditions (Pi).

**Line 19: yp=sim (net, [p; t], Pi)**  
The variable “yp” will contain the values from the network “net” simulation. To simulate, it is necessary to give to the function the network (net), the inputs (p; t) and the initial input delay conditions (Pi).

**Line 20: e=cell2mat(yp)-cell2mat(t)**  
The result of the subtraction between the network simulation “yp” (the simulated output using the network) and the output obtained from the simulation in SIMPACK is contained in the variable “e”. The subtraction is only possible converting before the variables from a cell array to a matrix. This is the reason to use the command “cell2mat”.

- 60 -
Line 21: `plot(e)`
As the command means, this is to create a plot with the values contained in “e” to show the difference between the simulated output using the network and the output from SIMPACK model simulation. This plot gives an idea of how good the simulation of the network fits the response of the SIMPACK model. If the difference is considerably in a lot of points, it would be better to go back and make some changes (for example in the number and dimensions of the hidden layers) before go away, in order to improve the results.

Line 28: `net2=sp2narx(net)`
To use the network created (net) with any input is necessary to convert it from the series-parallel to the parallel architecture. In the first part of the code it was necessary to use the series-parallel NARX to avoid the problems during the training.

Line 29: `ys1=ys`
The new variable “ys1” stores all values from “ys” in the given form of a sequential vector.

Line 30: `u1=u`
The new variable “u1” stores all values from “u” in the given form of a sequential vector.

Line 31: `p1=u1(3:end)`
At this point, “p1” will contain from the third to the last value of “u1”.

Line 32: `t1=ys1(3:end)`
From the third to the last value of “ys1” will be contained in “t1”.

Line 33: `Pi1=[u1(1:2)]`
Here the initial input delay condition is defined. In the parallel NARX network there is a delayed feed back for the output, but the delay is automatically applied.

Line 34: `yp1=sim(net2, p1, Pi1)`
This is the command to assign the results of the simulation to “yp1”. In that case the command for the simulation takes the NARX parallel network (net2), the input (p1) and
the initial delay conditions for the input (Pi1). Remember that in the NARX parallel network the output is fed back with a delay to the entrance.

Line 35: plot ([cell2mat(yp1)’  cell2mat(t1)’])

With this command MATLAB shows in the same plot the output after the simulation (yp1) using the NARX parallel network (net2) and the output from the SIMPACK model simulation. This plot will be used to compare easily if the output given from the network fits good or not with the same from the SIMPACK model. The comparison will be discussed in the next chapter.

This has been, step by step, the explanation of the code used to develop the work about dynamic neural networks. A personal comment to do is that, at first look, it seems an easy code that could be created in a short time. But nevertheless there is a lot of time spent behind to learn which commands should be used, learning how all the functions work, which are the variables and parameters to use with the functions and trying to put all together to work properly. Moreover, things are not as easy as they seem, and need a lot of work before apply them specially to understand how they work, the theoretical knowledge behind and finding and correcting all mistakes before apply them properly.