Bachelor of Science Thesis

Learning Control Strategies for Hybrid Vehicles Using Neural Networks

Rebecca Haehn

Examiners:
Prof. Dr. Erika Ábrahám
Dr. Walter Unger

Additional Advisor:
Johanna Nellen

Aachen, 31.03.2016

Abstract

This bachelor thesis has the topic to examine, whether it is possible to learn a control strategy for hybrid electric vehicles using artificial neural networks. In the course of this thesis this is examined for a basic control strategy. To achieve this goal, different training algorithms for artificial neural networks are tested. We make use of the vehicle model developed for the OASys project to generate training and test data and the open source libraries FANN and Open NN for the specification, training and test of neural networks. We demonstrate that it is possible to learn this basic control strategy with a satisfactory precision, but not exactly. It is assumed that even better results can be achieved with a higher computational effort.

Eidesstattliche Versicherung

______________________________	______________________________
Name, Vorname	Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

____

____

____

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

______________________________	______________________________
Ort, Datum	Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

______________________________	______________________________
Ort, Datum	Unterschrift

Acknowledgements

Thanks to my family for always supporting me.

I would also like to express my gratitude towards Johanna Nellen and Erika Abráhám for their patience and help with this thesis.

Contents

Chapter 1

Introduction

The environmental awareness of humanity grew in the last years. It became more important to reduce the emission of polluting and harmful gases and the consumption of limited resources. Cars have a significant share of both, by using fuel for combustion engines and producing exhaust gases.

In order to protect the environment cars should consume as little fuel as possible because exhaust gases are harmful to both the environment and human health. Therefore the automotive industry and research seeks for possibilities to reduce the emissions and fuel consumption. A first approach was to develop electric propulsion systems for cars. The result were electric vehicles. But those have another disadvantage, as they do not use fuel, but drive purely electric, they need a battery. As batteries have a certain capacity, the vehicles have only a limited driving range before the battery has to be recharged. Compared to conventional cars they have a short range, the charging times are long, also not everywhere are charging stations for electric vehicles. A larger battery is not a suitable solution, because that makes the vehicles much more expensive and heavy.

By combining both propulsion systems it is possible to achieve a wide range for driving due to the combustion engine and additionally support the combustion engine with the electric motor, which causes no polluting emissions, for a better efficiency, for more details see [9].

Vehicles, which do not drive solely with a combustion engine, but also with an electric motor, are called *hybrid electric vehicles (HEVs)*. In such vehicles fuel can not only be used for driving the car but also to recharge the battery which powers the electric motor. This way the combustion engine can always operate in its optimal rotational speed ranges with a high efficiency, because if the rotational speed is higher than requested, the battery becomes recharged, which is relatively favourable in such a moment. The electric motor can be used to support the combustion engine, which is called boosting, to improve the driving performance. Especially when starting the car, which often occurs in city traffic, the electric motor is important, as the combustion engine is not particularly efficient at low rotational speeds. So the total fuel consumption of HEVs is lower than in conventional cars, through increased efficiency and recuperation. Additionally, those cars are less heavy than pure electric vehicles since smaller batteries are sufficient. So HEVs have advantages compared to fuel-driven as well as electric vehicles.

There are mainly three types of HEVs: parallel, series and combined hybrids.

Series HEVs are electric vehicles with a combustion engine to recharge the battery, parallel HEVs have an engine and an electric motor, which can be used both at the same time, and combined HEVs are a combination of the two others, further described in [9].

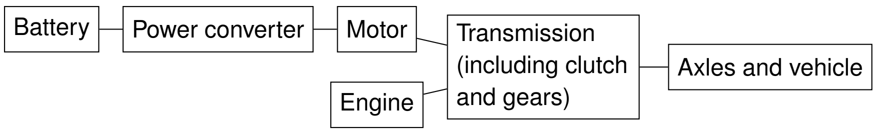

Figure 1.1: PHEV configuration as in [9]

In this thesis only *parallel HEVs (PHEVs)* are considered, as illustrated in Figure 1.1. These can be driven by both engines at the same time, so the required driving force to get the car at a requested speed can be split on both engines. A split value specifies which share the combustion engine provides, for example a split value of one means that just the combustion engine is used, while zero means that just the electric motor is used. When deciding in which situation which split value should be used our goal is to minimize the total fuel consumption. To examine for which split values the fuel consumption is minimal, a vehicle model computing the vehicle state, including the absolute fuel consumption, after driving with certain split values, is used, which was developed for the DFG project OASys. This vehicle model will be described in the next chapter.

So for PHEVs a control strategy which computes this split values is necessary. For every time step a split value has to be determined, to compute preferably continuous concrete values for the torques of the motors. The split value can be constant for a while, then several successive torque values become computed with the same split. Currently, different control strategies are implemented in the DFG project. They all have in common that they solve an optimization problem for each split they compute, the so called *energy management problem*, which will be described in Chapter 2.2.1. This causes a high computational effort, is time-consuming and needs also memory and processing units, which might be restricted due to hardware constraints in HEVs. However it is important that a control strategy computes the split values in real-time, because they are necessary for the car to drive. It is also desirable to reduce the computational effort, so a faster control strategy would be preferable, if it would cause a comparable reduced fuel consumption as the current strategies.

In this thesis an approach to develop a faster but equally good control strategy by using artificial neural networks is examined. *Artificial neural networks* are increasingly used to solve concrete application problems, for example optimization problems and will be explained in more detail in Section 2.3. So it might also be possible to learn a control strategy for a PHEV with an artificial neural network. A network which has learned a control strategy, could be used as a control strategy as well, after finishing the learning process. Instead of computing the split values with the original strategy, they could be computed by a trained neural network. A trained neural network consists of individual neurons arranged in layers and weighted connections from the neurons in each layer to the neurons in the next layer. The output of each neuron is the function value of its so called activation function for the sum of its weighted inputs. Activation functions are for example a step function, a linear function or the hyperbolic tangent. The output of the trained neural network is the output of its neurons in the last layer and is therefore computed by a fixed number of additions, multiplications and possibly potencies. So the computation time depends on the size of the network and on the number of its connections, but is approximately the same for each input and memory is not much more required than necessary for the network itself.

The topic of this thesis is to examine how well a control strategy for a hybrid vehicle can be learned by an artificial neural network, to decide if the network would be a suitable alternative to the original strategy. There are different types of strategies, for example basic strategies as the following:

- Rule-based strategies: e.g. ICE, where always the internal combustion engine is used or EM, which uses the electric motor whenever this is feasible
- Non-predictive strategies based on optimal control: e.g. equivalent consumption minimization strategies (ECMS), which minimize the sum of the current fuel consumption and the input power of the electric motor times a (time dependent) equivalence factor
- Strategies based on optimal control with prediction horizon: e.g. RDP and ADP, which are based on dynamic programming

There are also more complex strategies, for example learning-based strategies, which use those basic control strategies for learning. To test if it is in principle possible to learn a control strategy with a neural network, it is appropriate to start with a preferably continuous, basic strategy and to continue with a better, but more complex one, if it works to learn the basic strategy. As for learning strategies with a prediction horizon a neural network would need the predictions as inputs as well, which would increase the number of inputs, in this thesis just non-predictive strategies based on optimal control are tried to be learned.

The actual intention of trying to learn a control strategy using neural networks is to learn the genetic-algorithm-based control strategy GeneiAL, which is explained in [13]. But as this strategy currently uses a prediction horizon and would be quite difficult to learn, first a basic strategy is tried to be learned to test whether it is at all possible to learn a strategy like this. Also the generation of training data for GeneiAL would be way more time-consuming than for an ECMS, as more dimensions of inputs are necessary. If it is possible to learn a basic strategy, later we will make experiments with learning GeneiAL too.

Using a neural network instead of the original strategy could speed up the computation by the reduced computational complexity, optimization problems no longer need to be solved every time a new split is computed, as the neural network computes its output with basic arithmetic operations and the time-consuming training process of the neural network is done before using it as a control strategy. Solving the optimization problem to minimize the fuel consumption is more time and memory consuming compared to computing the output of a neural network, because for each considered split value a vehicle is simulated to be able to evaluate the total consumption, which will be described in more detail in Section 2.2.2.

This thesis begins in Chapter 2 with a description of the used vehicle model and the corresponding control strategies, especially those, which are supposed to be learned. Also the general energy management problem, which should be solved by a control strategy, is defined. This is necessary to be able to evaluate, how good a control strategy is. In Chapter 2 furthermore the basics of artificial neural networks are described, including the training and testing process and different libraries. The main part of this thesis is Chapter 3, where the decisions concerning the concrete training process of the artificial neural network, which is supposed to learn a control strategy, are described, including the generation of training data. The following testing process, especially the testing criteria and how a trained network becomes embedded into a control strategy, is explained in Chapter 4. In Chapter 5 the different networks are evaluated and the developed strategy is compared to the learned strategy. The thesis ends with a short summary, an evaluation of the achieved results and a brief discussion of possible future work in Chapter 6.

Notation

Especially in the first part, where the vehicle model is described, lots of different notations occur. For a better readability of the following chapters, the variables and constants used in Section 2.1 are listed here:

$v_{act} \in \mathbb{R}_{\geq 0}$	actual velocity in m/sec
$v_{req} \in \mathbb{R}_{\geq 0}$	requested velocity in m/sec
$acc_{req} \in \mathbb{R}$	requested acceleration in m/sec
$T \in \mathbb{N}_{\geq 0}$	duration of a driving cycle in sec
$t \in [0, T]$	current time in sec
$i \in \{1,2,3,4,5\}$	current gear
$r_i \in \mathbb{R}_{\geq 0}$	gear ratio for gear i
$\omega_{wh} \in \mathbb{R}_{\geq 0}$	wheel angular velocity in rad/sec
$\omega_{em} \in \mathbb{R}_{\geq 0}$	rotational speed of the electric motor in rad/sec
$\omega_{ice} \in \mathbb{R}_{\geq 0}$	rotational speed of the combustion engine in rad/sec
$T_{wh} \in \mathbb{R}$	actual torque at the wheels in Nm
$T_{cs} \in \mathbb{R}$	actual torque at the crankshaft in Nm
$T_{req} \in \mathbb{R}$	requested torque at the crankshaft in Nm
$T_{br} \in \mathbb{R}$	torque at the brake in Nm
$T_{em} \in \mathbb{R}$	torque at the electric motor in Nm
$T_{ice} \in \mathbb{R}$	torque at the combustion engine in Nm
$split_{ice} \in [0, 1.5]$	torque split for the combustion engine, values above 1 mean that the combustion engine is used to move the vehicle and recharge the battery at the same time
$\dot{m}_f \in \mathbb{R}_{\geq 0}$	fuel mass rate (instantaneous consumption) in g
$cons_f \in \mathbb{R}_{\geq 0}$	absolute fuel consumption in g
$SoC \in [0,1]$	battery state of charge
$r_{wh} \in \mathbb{R}_{\geq 0}$	wheel radius in m
$m_r \in \mathbb{R}_{\geq 0}$	equivalent mass of the rotating parts of the vehicle in kg
$P_{em} \in \mathbb{R}_{\geq 0}$	electric motor input power in W
$H_l \in \mathbb{R}_{\geq 0}$	lower heating value of the fuel m^2/s^2
$\eta_{gb} \in \mathbb{R}_{\geq 0}$	mechanical transmission efficiency
$m \in \mathbb{R}_{\geq 0}$	vehicle's mass in kg
$g \in \mathbb{R}_{\geq 0}$	acceleration of gravity in m/sec^2
$A \in \mathbb{R}_{\geq 0}$	vehicle frontal area in m^2
$\rho \in \mathbb{R}_{\geq 0}$	density of air in kg/m^3
$C_d \in \mathbb{R}_{\geq 0}$	air drag resistance
$\theta \in [0, 90]$	road slope
$f_r \in \mathbb{R}_{\geq 0}$	rolling resistance

Chapter 2

Preliminaries

In this chapter first the vehicle model, developed for the DFG project OASys, is described. Then the corresponding control strategies, as well as the energy management problem they are supposed to solve, are explained. Especially the strategies, which are supposed to be learned, are described in more detail. The reason for choosing this strategies is explained too.

It is continued with a description of artificial neural networks in general, for a better understanding of the main part of this thesis. This includes the training and testing process and the descriptions of two different libraries for artificial neural networks that have been used in this thesis.

2.1 Vehicle model

In the following, the vehicle model, as described in [13], is used in this thesis, to simulate a PHEV driving with a given control strategy. This is necessary to determine for example the absolute fuel consumption and the CO_2 emission for certain driving cycles, which are used as a metric to compare different strategies. Also some of the strategies use this vehicle model to simulate the behaviour of a vehicle for certain split values, to decide which one they return.

The car's propulsion consists of two engines, an internal combustion engine and an electric motor, which is illustrated in Figure 2.1. Both can accelerate the car simultaneously, as it is a PHEV. In order for this to be possible, both have to be coupled to the same axis. This axis is attached to the gearbox, which is connected to the wheels. The wheel angular velocity ω_{wh} depends on the velocity v of the vehicle: $\omega_{wh} = \frac{v}{r_{wh}}$, where r_{wh} is the wheel radius. The engines move with the same angular velocities, because they are connected to the same axis, so $\omega_{ice} = \omega_{em}$ holds. To convert the angular velocities at the engines to the one at the wheels, the gear i and the corresponding gear ratio r_i are relevant: $\omega_{wh} = \frac{\omega_{ice}}{r_i}$.

For a requested acceleration a a certain torque at the wheels T_{wh} is necessary, which can be computed with the following formula, taken from [4]:

$$T_{wh} = r_{wh} \cdot (\frac{1}{2}\rho C_d \cdot A \cdot v^2 + (m + m_r)a + m \cdot g \cdot f_r \cdot cos(\theta) + m \cdot g \cdot sin(\theta)) \quad (2.1)$$

where ρ is the density of air, C_d the air drag resistance, A the vehicle frontal area, m the vehicle's mass, m_r the equivalent mass of the rotating parts of the vehicle, g the

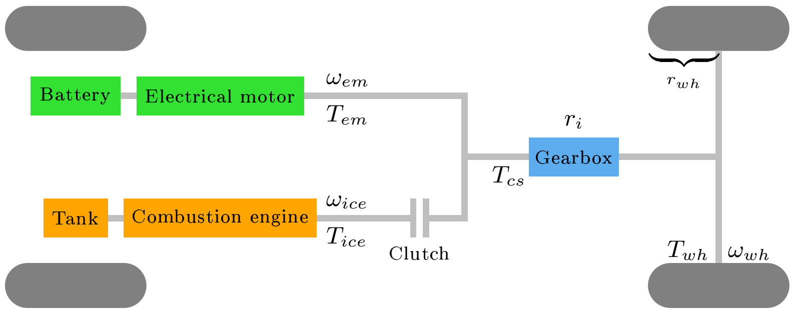

Figure 2.1: PHEV model as in [4]

acceleration of gravity, f_r the rolling resistance, θ the road slope. This formula takes the resistances that must be overcome by the vehicle, while driving, into account.

The torque at the crankshaft T_{cs} can be calculated from T_{wh}, depending on the current gear, using the formula from [13]:

$$T_{cs} = \frac{T_{wh} + T_{br}}{\eta_{gb} \cdot r_i} \tag{2.2}$$

where η_{gb}, which is the mechanical transmission efficiency, is assumed to be constant and T_{br} is the torque at the brake. In these formulas immediate torque responses are assumed.

The requested torque at the crankshaft T_{req} is generated by the electric motor T_{em} and the combustion engine T_{ice}:

$$T_{req} = T_{ice} + T_{em} \tag{2.3}$$

A control strategy u distributes T_{req} on the two engines:

$$T_{ice} = u \cdot T_{req} \tag{2.4}$$

$$T_{em} = (1 - u) \cdot T_{req} \tag{2.5}$$

How such a control strategy can be computed is described in the next section.

To calculate how much fuel would be used for a certain control strategy, a driving car is simulated. Therefore several components are necessary, as illustrated in Figure 2.2:

- Test route, so called driving cycle: contains for each time step (in seconds) of the simulation, the time step t itself, acceleration acc_{req}, gear i and requested velocity v_{req}; the values in-between are interpolated, if more data is needed
- Driver: gets requested velocity v_{req}, actual velocity v_{act}, and gear ratio r_i according to the driving cycle and calculates T_{req}
- Control strategy: computes a split, to distribute T_{req} to T_{ice}, T_{em} and T_{br}, further described in the next section
- Control converter: checks if the split is suitable and returns T_{ice}, T_{em} and T_{br}
- Vehicle: gets T_{ice}, T_{em}, T_{br} and gear i and calculates amongst other values the absolute fuel consumption $cons_f$ and the battery state of charge SoC

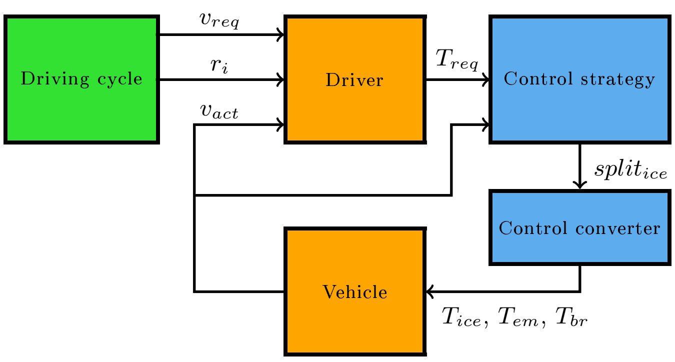

Figure 2.2: Simplified OASys model

With the above described model a control strategy can be evaluated by simulating a driving car. The evaluation criterion is mainly how much fuel is consumed, as every control strategy is an approach to solve the energy management problem, which has the goal to minimize the fuel consumption and is described in more detail in Section 2.2.1. For an exact comparison the SoC has to be considered too, as recharging the battery may consume fuel as well and a higher charged battery will enable to save more fuel while the next ride. So a control strategy with a slightly higher fuel consumption might eventually be better, when the SoC is higher. Another evaluation criteria could be the CO_2 emission, but this is neglected in this thesis. In the next section control strategies in general as well as some concrete strategies are explained.

2.2 Control strategies

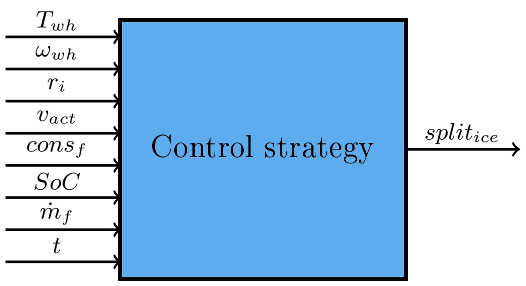

Figure 2.3: Control strategy as implemented for the OASys project

As mentioned before, the purpose of a *control strategy* is to compute a control u, here called $split_{ice}$, which distributes the T_{req} to T_{ice}, T_{em} and T_{br}. Therefore it gets several input values as illustrated in Figure 2.3:

- target values (requested torque $T_{req} \in \mathbb{R}$ in Nm, wheel angular velocity $\omega_{wh} \in \mathbb{R}_{\geq 0}$ in rad/sec, gear ratio $r_i \in \{r_1, r_2, r_3, r_4, r_5\}$ for the current gear i)
- vehicle state (actual velocity $v_{act} \in \mathbb{R}_{\geq 0}$ in m/sec, absolute fuel consumption

$cons_f \in \mathbb{R}_{\geq 0}$ in g, battery state of charge $SoC \in \mathbb{R}_{\geq 0}$, fuel mass rate (instantaneous consumption) $\dot{m}_f \in \mathbb{R}_{\geq 0}$ in g)

- current time $t \in [0, T]$ in *sec*, where $T \in \mathbb{N}_{\geq 0}$ is the duration of the driving cycle in *sec*

The only output the control strategy has to compute is the torque split $split_{ice} \in [0, 1.5]$ for the combustion engine, which is defined as the share of the requested torque the combustion engine has to provide. Split values above one mean that the combustion engine is not just used to drive the vehicle, but also to recharge the battery. While the battery becomes recharged the electric motor cannot be used to drive the vehicle, so in this case the combustion engine has to provide the requested torque and the additional necessary energy to recharge the battery. The strategies, which will be used in this thesis, compute a new split once every second, so the computation time is limited. A control strategy has to be an online algorithm, as the future states of the vehicle are unknown.

In this thesis the focus is on equivalent consumption minimization strategies (ECMS), which are explained in more detail in Section 2.2.2. There are also more complex strategies, for example online learning algorithms, which learn different basic control algorithms, but these will not be part of this thesis.

As the fuel consumption should be minimized, every approach for a control strategy tries to solve the energy management problem, which is described in the next section. So for computing the $split_{ice}$ an optimization problem has to be solved. Nevertheless are the control strategies, which are part of this thesis, just an approximation to the actual solution, because they are online algorithms and therefore have the no-regret property, which means, they can not change a split, if it was in retrospect not the optimal decision.

2.2.1 Energy management problem

The quality of a control strategy u can be determined by a cost function L, which calculates the costs for every time step: the lower the total costs, the better the control strategy. The input of L are the evaluation criteria for the control strategies, for example the fuel consumption and the SoC. The control $u(\tau)$, which is the $split_{ice}$ at the time step τ, is sufficient to compute the fuel consumption for time step τ when a driving cycle is given, as the fuel consumption depends just on the rotational speed ω_{ice} and the torque T_{ice}, which can be computed from the $split_{ice}$ and the information in the driving cycle. So here L depends on $u(\tau)$, $SoC(\tau)$ and τ: $L(u(\tau), SoC(\tau), \tau)$.

The *energy management problem*, as defined in [4], is to find a control strategy u, which is cost-minimal, optimally for arbitrary driving cycles. Such a control strategy solves the following minimization problem:

$$J^* = \min_{u(\cdot)} \sum_{\tau=0}^{T} L(u(\tau), SoC(\tau), \tau) \tag{2.6}$$

for a given driving cycle over a equidistantly discretized time interval $[0, T]$, where the $SoC(0)$ has to be the initial SoC and Formula 2.1 and the restrictions for the SoC in Formula 2.15 must hold, with acceleration a and velocity v given by the driving cycle. This basically means, the car must drive according to the driving cycle. The final SoC and other aspects are not considered here for simplicity.

Now this is the optimization problem, which should be solved as good as possible by a control strategy. It is possible to find the optimal solution, for example with dynamic programming, but not in real-time, and not without knowledge about the whole driving cycle. So the basic strategies, which are described in the next section and do not have knowledge of the prospective route, can just try to approximate the optimal solution.

2.2.2 Equivalent consumption minimization strategy

In this section different equivalent consumption minimization strategies, as well as the concrete energy management problem they are supposed to solve, are described. First the simplest variant is described, later different approaches are introduced. Those strategies should later be learned by artificial neural networks. In the following equivalent consumption minimization strategy is abbreviated as ECMS. In this strategies the cost of using the battery is weighted against the cost of using fuel with an equivalence factor s. The ECMS solves the following energy management problem from [4], under the conditions defined in Section 2.2.1:

$$J^* = \min_{u(\cdot)} \sum_{\tau=0}^{t} \dot{m}^u_{f,equ}(\tau) \tag{2.7}$$

Because only the already driven route is known, only the sum up to the current time step t can be minimized. As the name of the strategy already says, the cost function, which is tried to be minimized here, is the equivalent consumption $\dot{m}^u_{f,equ}(\tau)$:

$$\dot{m}^u_{f,equ}(\tau) = \dot{m}_f(\omega_{ice}(\tau), T^u_{ice}(\tau)) + \frac{s}{H_l} P_{em}(\omega_{em}, T^u_{em}(\tau)) \tag{2.8}$$

where $\dot{m}_f(\tau)$ is the fuel consumption, H_l is the lower heating value of the fuel and P_{em} is the electric motor input power.

There are different approaches to determine the equivalence factor s. The easiest possibility is to use a constant value. In this ECMS, a constant factor of 3.2 is used.

For given equivalence factor and input values, the strategy calculates every time step t a new split value $split(t)$. If $T_{req} \leq 0$ then $split = 0$, and if $T_{req} > 0$ a search to find the best split is performed. Depending on the implementation different search algorithms are used. In the C++ implementation a golden section search is performed, which is similar to a binary search. In the Matlab implementation, which can be used to generate C++ code, a search over discrete values is performed.

The search is performed in the interval $[0, 1.5]$ to find the split with the least equivalent consumption in the next simulation step. To determine this split, in every step of the search the fuel consumption of the car in the next simulation step, driving with the currently examined split, is computed, by simulating one time step with the vehicle model. The corresponding consumption of electric energy becomes computed as well to calculate the equivalent consumption with Equation 2.8.

This is computationally intensive and therefore time consuming, as in every step for different split values the equivalent consumption is calculated by computing a simulation step with the vehicle model and the electric motor input power. This computations are for a single split more time consuming when using the C++ implementation, as the Matlab computation is more efficient, but in the Matlab implementation more different split values are examined. The C++ implementation has the disadvantage that the search is based on the assumption that the equivalent consumption depending on the split is a strictly unimodal function, which can not be guaranteed.

Despite the computational effort, the computed splits are just an approximate solution for the minimization problem in Equation 2.7, which is to calculate the minimum of the given sum. A sum is minimal, when every single summand is minimal. The summand in time step τ as specified in Equation 2.8, depends on $\omega_{ice}(\tau)$, $\omega_{em}(\tau)$, $T^u_{ice}(\tau)$ and $T^u_{em}(\tau)$, because s and H_l are constants here. So the split just depends on the gear and the actual and requested velocities, which are already enough to compute the other values. It is noticeable that the split does not depend on the SoC, when using this strategy with a constant equivalence factor. Therefore a neural network, which should learn this ECMS, would get at most the three input values gear, actual velocity and requested velocity. This explains, why exactly such a strategy is supposed to be learned by a neural network, although it is not the best one. It is a basic strategy, so the split value does not depend on too many factors, especially as this strategy does not use a prediction horizon. And to test, if it is possible to learn a control strategy with a neural network, it should be started with a basic strategy. A problem might occur, if the output of this strategy is not continuous, as continuous functions are easier to learn for artificial neural networks.

In the following the abbreviation ECMS is only used for the equivalence consumption minimization strategy, which uses a constant equivalence factor. There are other ways of computing an equivalence factor, two of those are defined below.

PECMS

A possibility to improve the ECMS, described in the last section, is to choose a time-dependent equivalence factor $s(\tau)$. There are different approaches to determine the equivalence factor $s(\tau)$. One possibility is an equivalence factor depending just on the current SoC:

$$s(t) = s_0 + k_p(SoC_{ref} - SoC(t)) \tag{2.9}$$

with an initial value s_0 for $s(t)$ and a proportional feedback gain k_p. In Matlab, the implemented strategy uses $s_0 = 3.2$ and $k_p = 2.5$. The resulting strategy is the so called *proportional equivalent consumption minimization strategy (PECMS)*. This strategy also solves the energy management problem in Equation 2.7, only the equivalence factor s in the equivalent consumption in Equation 2.8 has to be replaced by $s(\tau)$.

PIECMS

In other approaches, the equivalence factor depends not just on the current SoC, but also on the sum of the previous $SoC(\tau)$, from [4]:

$$s(t) = s_0 + k_p(SoC_{ref} - SoC(t)) + k_i \sum_{\tau=t_0}^{t} (SoC_{ref} - SoC(\tau)) \tag{2.10}$$

with an initial value s_0 for $s(t)$, a proportional feedback gain k_p, as for the PECMS, and additionally an integral feedback gain k_i. In the Matlab implementation, s_0 and k_p are the same as in the PECMS and $k_i = 0.001$. This strategy is the so called *proportional integral equivalent consumption minimization strategy (PIECMS).* It also solves the energy management problem in Equation 2.7, but the equivalence factor s in the equivalent consumption in Equation 2.8 has to be replaced by $s(\tau)$.

One of these strategies should be learned by an artificial neural network later. To decide which one the input/output behaviours of the strategies have to be considered, which is done in Chapter 3. In the next section first a control instance is introduced, the so called control converter, which is supposed to ensure that the vehicle model stays within certain mechanical bounds. It also has the task to convert the split, which is computed by the control strategy, into the torque values, which are expected by the vehicle model.

2.2.3 Control converter

In the last section control strategies were explained. In those strategies it has usually not yet been considered, that for a safe and smooth operation the torque values and rotation speeds, as well as the SoC, resulting from the calculated split, must stay within certain bounds, as defined in [4]:

$$\omega_{ice,min} \leq \omega_{ice} \leq \omega_{ice,max} \tag{2.11}$$

$$\omega_{em,min} \leq \omega_{em} \leq \omega_{em,max} \tag{2.12}$$

$$0 \leq T_{ice} \leq T_{ice,max}(\omega_{ice}) \tag{2.13}$$

$$T_{em,min}(\omega_{em}) \leq T_{em} \leq T_{em,max}(\omega_{em}) \tag{2.14}$$

$$SoC_{min} \leq SoC \leq SoC_{max} \tag{2.15}$$

The only torque value, which can be negative, is T_{em}, where a negative value means that the battery is recharged, either by the combustion engine or during braking.

To ensure that the vehicle model stays within those bounds in this model a so called *control converter* is used. This is a control instance, which tests if the vehicle state stays within these bounds, when using the split the control strategy computed in the next simulation step. If this is not the case the control converter computes a new split, for which the vehicle state is admissible if possible.

This is not the only task of the control converter, which is illustrated in Figure 2.4, its other purpose is to convert the split computed by the control strategy into the torque values, as the vehicle model expects torque values. The vehicle is simulated every 0.02 seconds, so the control converter has to provide a new torque distribution every 0.02 seconds, too. The split used to compute this distribution remains fixed for one second, which reduces the computational effort.

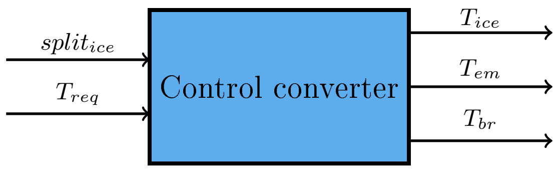

Figure 2.4: Control converter as implemented for the OASys project

How the torque distribution is computed exactly is described in detail in the following. After $split_{ice}$ is computed, how exactly depends on the control strategy and is described for the strategies, which are used in this thesis, in Section 2.2.2, the control converter gets $split_{ice}$ and T_{req} as input and tests if Equations 2.11 to 2.15 hold when this split is used. If not, a new split has to be computed, in this case the control converter computes the new split, depending on the condition which did not hold. Then the control converter calculates the torque values. If $T_{req} \geq 0$, which means the car is driving, the following formulas are used:

$$T_{ice}(t) = split_{ice}(t) \cdot T_{req}(t) \tag{2.16}$$

$$T_{em}(t) = (1 - split_{ice}(t)) \cdot T_{req}(t) \tag{2.17}$$

$$T_{br} = 0 \tag{2.18}$$

As the goal is to minimize the fuel consumption T_{br} should be always kept minimal and therefore is just not zero, when the car has to brake, but in this case $T_{req} < 0$ holds. So T_{br} does not depend on the split value, when the vehicle is not braking, as it would be a waste of fuel to brake and use the engines to drive the vehicle at the same time, which is therefore impossible.

If $T_{req} < 0$ the car is braking and therefore the torque values are computed differently. While braking the combustion engine can just be used to additionally charge the battery, but actually T_{ice} is mostly zero. Whenever the battery is not completely charged braking should be used to recharge the battery as much as possible, so T_{em} is negative. When charging the battery the condition in Equation 2.15 has to be considered. In this case the following formulas are used to calculate the torque values:

$$T_{ice}(t) = 0 \tag{2.19}$$

$$T_{em}(t) = max(T_{req}(t), T_{em,min}) \tag{2.20}$$

$$T_{br}(t) = -(T_{req}(t) - T_{em}(t)) \cdot r_i \tag{2.21}$$

To compute T_{br} from T_{req} the gear ratio is necessary, because T_{req} is the torque requested at the crankshaft and not the one at the wheels, but the brakes affect the wheels directly. So the remaining T_{req}, when recharging the battery as much as possible, has to be converted to the remaining $T_{wh,req}$, which has to be provided by the brakes. As $T_{wh,req} = T_{req} \cdot r_i$ applies, follows $T_{br}(t) = -(T_{req}(t) - T_{em}(t)) \cdot r_i$.

So the control converter distributes for each time step t the $T_{req}(t)$ to the engine torques according to Equations 2.16 to 2.21, and then returns T_{ice}, T_{em} and T_{br} as calculated above.

It is possible to turn the control converter off, which is useful to examine the changes in the vehicle state when driving with a certain control strategy. This is relevant for the evaluation of control strategies using neural networks later. When turning off the control converter it is important to convert the split to the engine torques for the simulated vehicle, which is usually done by the control converter. Without using the control converter, the simulated vehicle always uses the torques computed with the split the control strategy has calculated, even if this would cause for example a completely recharged battery. This way it can be examined, if two simulated vehicles using different control strategies just behave similar because of the control converter, or if the strategies are similar, which has to be determined, when comparing a strategy using an artificial neural network with the strategy the network was supposed to learn. In the next section first artificial neural networks are introduced.

2.3 Artificial neural networks

Artificial neural networks have a wide field of application, amongst other also to solve optimization problems. In this thesis artificial neural networks should be used to learn a control strategy for a hybrid vehicle, to reduce the computation time. In this section the basics of artificial neural networks are explained. First, the structure of artificial neural networks is introduced, then the training process is explained and the subsequent test phase. Finally, two libraries for artificial neural networks are presented.

An *artificial neural network (ANN)* is constructed similar to a biological neural network. It consists of layers of neurons and connections between those neurons. There is an input layer, an output layer and additional hidden layers between those. Here are only feed forward networks considered, those have just connections in direction from the input layer to the output layer, and from each neuron just to neurons from the respectively next layer. They can be regarded as directed, acyclic graphs. It is also possible to construct networks with connections in both directions but those networks are way more complex, difficult to train, and sometimes show chaotic behaviour, as explained in [18].

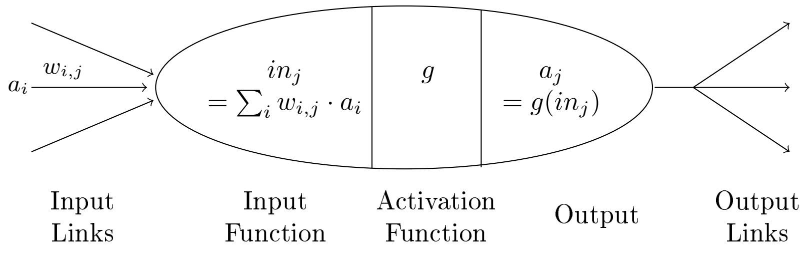

Figure 2.5: Mathematical model for a neuron j as in [18]

The ANN calculates output values according to the given input values, based on its weights and the network topology. To understand how exactly this is computed, it must be explained, how a single neuron works. The structure of a neuron is shown in Figure 2.5. As can be seen a neuron j gets different input values a_i from the neurons in the previous layer via the input links, and an additional input value $a_0 = 1$ called bias, whose weight $w_{0,j}$ is used to simplify the activation function. Each of those links has a weight, where the weight of the link from neuron i to neuron j is called $w_{i,j}$. The input in_j of neuron j with n inputs is then the weighted sum of its inputs $in_j = \sum_{i=0}^{n} w_{i,j} \cdot a_i$. The output a_j of neuron j is calculated using the activation function g, so $a_j = g(in_j)$. This output is then forwarded to the neurons in the next layer via the output links. This way the output of the whole network is computed.

Which function is computed by the network depends on several settings. First, the topology of the network, how many neurons it includes and how these are arranged in layers and connected with each other. Furthermore the weights of these connections and the activation functions of the neurons.

There are different activation functions, which are usually non-linear functions, for example step functions, the sign function or sigmoid functions, which are scaled and shifted hyperbolic tangent functions, illustrated in Figure 2.6. An important criterion for choosing an activation function later is the differentiability of it, which is necessary for several training algorithms. The sign function for example is not differentiable in zero and has everywhere else the derivative zero, which would prevent the weight adjustment by some training algorithms.

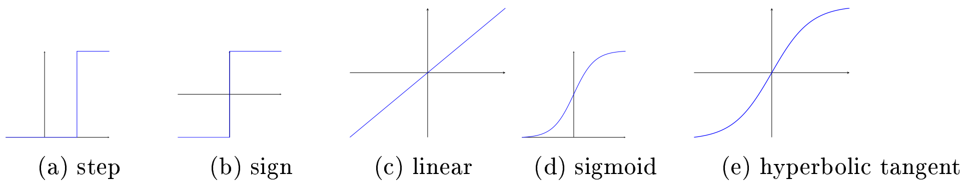

Figure 2.6: Activation functions

The input of a neuron should be in the intervals $[0, 1]$ respectively $[-1, 1]$, depending on the activation function, for the neuron to have different output values, e.g. if a neuron has the sign function as activation function and all possible input values are positive, the output is always the same. This can also occur when using an activation function which has continuous output values, for example a sigmoid function, if all possible input values are much larger than one, the output is always nearly one, which makes it difficult to detect differences in the inputs.

For the ANN to be able to detect a difference in various inputs, it is important that the weights match to the intervals in which the input values are, because the input of each neuron is the sum of the weighted inputs and has to be in the interval $[0, 1]$ respectively $[-1, 1]$, depending on the activation function, for the neuron to detect a difference. Scaling the input data can simplify finding appropriate weights. The benefit of scaling the input of the ANN is explained in the next section.

2.3.1 Scaling

In this section it is described how the input and output data of an ANN can be scaled and how this can improve the training process, which is explained in the next section. The output values have to be scaled in most cases, because they are the outputs of the neurons in the output layer and as such computed by an activation function, which usually only assumes values in $[-1, 1]$. So if the expected output values are in a different interval, they have to be scaled. In this case it is necessary to scale the expected output already in the training data, because while training the output of the network is compared with the expected output in the training data for which it is necessary that those values are comparable.

A possible scaling function to scale a value $x \in [x_{min}, x_{max}]$ to a value $x_{scaled} \in [min, max]$ is the following linear function:

$$x_{scaled} = min + \frac{x - x_{min}}{x_{max} - x_{min}} \cdot (max - min) \tag{2.22}$$

The input values do not have to be scaled, as the ANN can compensate different orders of magnitude in the inputs by weight modification. Nevertheless it is sensible to scale the input values to a uniform interval, since the learning process otherwise can be slowed down, especially when the input values differ by many orders of magnitude. This is the case, as the initial weights have to be set and it is difficult to find suitable initial weights for non-scaled inputs in different orders of magnitudes. If all weights are in the same order of magnitude many iterations are necessary to adjust the weights to compensate the different orders of magnitude in the inputs. So a better performance is expected, if all input values affect the ANN equally strong, see [2].

To scale the input values, the range in which all possible input values are included has to be known from the beginning, which is not given for every application. Another disadvantage is that because of some outliers the normal input signal could be mapped to just a small share of the target interval. This can be prevented by the elimination of outliers before scaling. The target interval depends just on the used activation function.

In the next section the training process, which should be speeded up by scaling the input, is explained in detail.

2.3.2 Training phase

In this section the training phase of an ANN is explained, including the corresponding decision about the topology, the activation functions and the initial weights, which have to be made before starting the actual training process. Then it is continued with a concrete training algorithm.

Before trying to learn a function using an ANN, it has to be examined, which topology is suited for learning this function, as not every topology offers the same abilities. A perceptron network, which has no hidden layer, just the input and output layers, is only able to learn linearly separable functions, according to [18] for example AND and OR, but not XOR, which is illustrated in Figure 2.7. An ANN with one hidden layer, which has to be sufficiently large, is already able to represent any continuous function of its inputs with any desired accuracy. An ANN with two hidden layers can even represent discontinuous functions, but the necessary number of neurons in the hidden layers grows exponentially with the number of inputs, as mentioned in [18]. For different network structures it is harder to investigate which functions can or can not be represented.

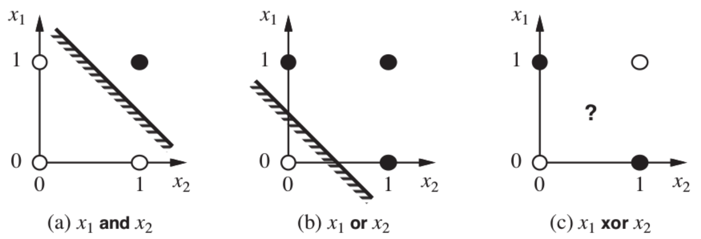

Figure 2.7: Linearly separable functions, taken from [18]

So neural networks are useful to learn for example a continuous function for which just some values are given, but not the function itself, or a preferably continuous function which is given. Learning a given function with a neural network is just useful, if the output of the function has to be computed in real-time, but the function itself is difficult to compute. Such a function could be approximated by the network, which would reduce the computational complexity and the time required to determine function values.

To construct a network, which computes a given function, first the topology of the network has to be determined. There is no known rule for determining how many layers, neurons and connections between those neurons a network should optimally have, depending on the function to be computed. So it is necessary to test different topologies and choose the one, which works best. For a certain topology activation functions for the neurons have to be selected, then a network is created with initial weights and this activation functions. The initial weights can be random values in an interval, which should be chosen according to the possible values of the input data, so that the input of the activation function is approximately between -1 and 1, and the network is able to detect a difference in the input, as explained before. Another possibility to set the initial weights is to use the Algorithm 1 developed by Widrow and Nguyen [14], which depends on the smallest and largest possible input value, so we assume here that the input values are between -1 and 1. This algorithm is developed for a network with just one hidden layer, but it can be applied recursively to neural networks with more hidden layers.

```
function INIT_WEIGHTS(number of hidden neurons $h$, number of input values $n$)
    initialize all weights with random values
    $w_i \leftarrow$ weights of neuron $i$
    $\beta \leftarrow 0.7 \cdot h^{\frac{1}{n}}$
    for all hidden neurons $i$ do
        for all weights $w_{j,i}$ do
            $w_{j,i} \leftarrow \frac{w_{j,i}}{\|w_i\|} \cdot \beta$
        end for
    end for
end function
```

Algorithm 1: Initialize weights from [14]

The resulting network does not compute the correct function, yet. To compute the correct function, the weights have to be adapted to the problem. This is achieved by training the network.

A network is trained with some training data, a given set of data, according to the function that should be learned. This training data consists of data tuples, which each contain possible input values for the function and the corresponding function values. During training, the network gets some of these input data and computes the output with the initial weights, according to the difference between this output values and the function values from the training data the weights are adapted, for example with the back-propagation algorithm for learning in multilayer networks in Algorithm 2.

The training depends on several factors:

- the initial weights
- the selection of the training data
- the training algorithm, which itself can depend on:
  - how frequently the weights become adapted, e.g. after each input pattern or after the whole data set
  - the learning rate
  - the error function
  - other individual factors (e.g. an increase factor or a weight decay shift)

```
function BACK_PROP_LEARNING(training data, network with $L$ layers, initial
weights $w_{i,j}$ and activation function $g$)
    $\Delta$ vector of errors, indexed by nodes
    repeat
        for all tupel $(x,y)$ in training data do
            for all nodes $i$ in input layer do
                $a_i \leftarrow x_i$
            end for
            for $l = 2$ to $L$ do
                for all nodes $j$ in layer $l$ do
                    $in_j \leftarrow \sum_i w_{i,j} \cdot a_i$
                    $a_j \leftarrow g(in_j)$
                end for
            end for
            for all node $j$ in output layer do
                $\Delta[j] \leftarrow g'(in_j) \cdot (y_j - a_j)$
            end for
            for $l = L - 1$ to $1$ do
                for all node $i$ in layer $l$ do
                    $\Delta[i] \leftarrow g'(in_j) \sum_j w_{i,j} \cdot \Delta[j]$
                end for
            end for
            for all weight $w_{i,j}$ do
                $w_{i,j} \leftarrow w_{i,j} + \alpha \cdot a_i \cdot \Delta[j]$
            end for
        end for
    until stopping criterion is satisfied
return network
end function
```

Algorithm 2: Back-propagation for learning in multilayer networks as in [18, 12]

- the stop criteria
- the number of iterations

2.3.3 Testing phase

The training phase should be followed by a testing phase, to test whether the ANN correctly learned the function. This is not always given, sometimes the training data contain too less data tuples for the number of neurons in the network, in this cases so called overfitting is performed, as described in [18]. This means the ANN just memorizes the given data tuples, but does not manage to compute the correct output values for different input data. It is also possible to train a too simple network with complex training data, whose relation is complicated to learn, in this cases the ANNs simply do not manage to learn the correct functions as well, as they are too complex. So it is necessary to check if the ANN works as desired.

To achieve an ANN which successfully learned a given function, it is often necessary to have several training and testing phases. It is started with a topology and activation and training functions which are assumed to produce an appropriate ANN. The training of this network is followed by a testing phase and if the result is not good enough, the training becomes repeated with different parameters, for example more or less neurons, another activation function or a different training function. After this the resulting ANN is tested again, if the result is still not good enough then the procedure becomes executed again until the outcome is sufficiently good.

In the process the changes of the parameters are not just arbitrary. For example if less neurons cause an even worse result, the next time again more neurons are used. It is also recommendable to train and test not just one network with the same parameters, as the resulting ANN is not always the same and sometimes it might seem to be a worse result than before, but in fact the average result would be better.

The testing phase requires some test data, which consists of data tuples. Those data tuples need to have exactly the same number of input and output values as the ones in the training data. The testing process provides the most useful and realistic results, when the test data inputs are different from the training data inputs. Nevertheless the test data outputs in each tuple have to be the function values of the inputs in the same tuple, for the function which should have been learned.

While testing the ANN, it gets the input values of each tuple in the test data as inputs and has to compute the corresponding outputs. This outputs should be the same as the ones in the test data, as those are the function values, which should be computed by the ANN. The larger the difference between those values, the worse the function was learned.

The error of each output can be defined using different metrics. It should be noted that a small difference to the expected output could be due to rounding, as the values are calculated by a computer. To compute the distance between an output x of the network and the corresponding expected output x^* according to the test data for example the following metrics can be used:

- Discrete metric: $d(x, x^*) = \begin{cases} 1 \text{ , } if \ |x - x^*| > \epsilon \\ 0 \text{ , } else \end{cases}$
- Manhattan metric: $d(x, x^*) = |x - x^*|$
- Euclidean metric: $d(x, x^*) = (x - x^*)^2$

For multidimensional outputs the sum of the distances of the individual entries is calculated and divided by the number of outputs. If the output is either 1 or -1 respectively 0, but no other value, the fault can be simply measured with the discrete metric, where $\epsilon = 0$.

The total error of a set of test data can be computed as the sum of the errors of all output values of each test data tuple, using a fixed metric: $e = \sum_x d(x, x^*)$. When using the discrete metric for this, the total error corresponds to the number of errors in the set of test data. But especially if there are several sets of test data with different length the total error is not comparable, because it contains no information regarding the individual error size anymore. The share of mistakes on the number of tuples tested, would be a more suitable comparison value. So the average error would be better comparable. The average error can be calculated from the total error by dividing by the number of test data. When computing the average error also the number of output values for each tuple should be considered by dividing by the number of outputs per input tuple as well. In the next section two concrete implementations of artificial neural networks are introduced.

2.3.4 Implementation

There are several different implementations for artificial neural networks. As the neural network should be used for learning a control strategy for a hybrid vehicle and if possible be used itself to implement such a control strategy, it should be implemented in the same programming language as the control strategy. The programming language used for the existing strategies and the vehicle model is C++. Therefore FANN and Open NN, which are open source neural network libraries available in C++, come into consideration.

FANN

FANN [15] is the abbreviation for Fast Artificial Neural Network, which is a library for neural networks developed mainly by Steffen Nissen as a graduate project. As the name says, the library is supposed to be as fast as possible. The following information in this section are taken from the reference manual for FANN [16].

With the FANN library it is possible to create networks with connections from every neuron to every neuron in the respectively next layer, but also networks with less connections or with connections not just to the next layer but to all following layers. Also several activation functions are implemented, for example the following, where s is the steepness, a parameter to further adapt the functions, x is the sum of the weighted input values and y is the function value:

- SIGMOID: $y = \frac{1}{1+exp(-2 \cdot s \cdot x)}$
  $\rightarrow$ range of values: $0 < y < 1$
- SIGMOID_SYMMETRIC (hyperbolic tangent): $y = tanh(s \cdot x)$
  $= \frac{2}{1+exp(-2 \cdot s \cdot x)} - 1$
  $\rightarrow$ range of values: $-1 < y < 1$
- LINEAR_PIECE_SYMMETRIC (bounded and linear): $y = x \cdot s$
  $\rightarrow$ range of values: $-1 < y < 1$

Those are the activation functions, which are used in this thesis. Because as mentioned in [1] a network with three layers, when using the sigmoid activation function, is already able to learn all functions a network with different activation functions could learn. In FANN are however more activation functions implemented.

Furthermore the library offers two different error functions for the training, a linear and a hyperbolic tangent error function. There are five different training algorithms available:

- INCREMENTAL: a standard backpropagation algorithm, weights are updated after each input pattern, see Algorithm 2
  $\rightarrow$ sometimes fast, but does not train well for advanced problems
- BATCH: a standard backpropagation algorithm, weights are updated after calculating the mean square error for the whole data set
  $\rightarrow$ slower than incremental, but better solutions for some problems
- RPROP: resilient backpropagation, a batch training algorithm, where each weight adaptation depends on the last weight change and the sign of the gradient of the error function, detailed in [10]
  $\rightarrow$ faster than backpropagation, but can converge to local minima

- QUICKPROP: a batch backpropagation training algorithm, where each weight adaptation depends on the last weight change and the gradient of the error function for the last and the current step, for more information see [3]
  $\rightarrow$ faster than backpropagation, but can be chaotic while learning because of large step size
- SARPROP: RPROP with simulated annealing, noise is added to the update value and a weight decay term to the error function, as described in [19]
  $\rightarrow$ converges faster and more often than RPROP

Those algorithms are all first-order training algorithms, as they use the first derivation of the error function. They can be combined while training, it is possible to use one algorithm in the beginning, until a certain stop criteria is reached and then continue with another one.

FANN provides a further training algorithm, the so called cascade training algorithm, as explained in [17], but this one generates a network with connections not only between successive layers. This algorithm starts with an ANN without hidden layer, where all inputs are connected to all outputs. Then neurons are added one by one to the network. Before a neuron is added several candidate neurons, initialized with different random weights, are trained separately. A candidate neuron has trainable connections to the input neurons and all previously added neurons but no output links yet. It receives still the error from the output neurons for training. The input links of a candidate are trained while the other weights in the ANN are fixed. The purpose of the training is to adjust the input weights of the candidate neuron in order to maximize the covariance S between the candidates output c_p and the error $e_{k,p}$ at the output neuron k for the input in the training data tuple p. The covariance S is defined as in [17]:

$$S = \sum_{k=0}^{K} \sum_{p=0}^{P} (c_p - c)(e_{k,p} - e_k) \tag{2.23}$$

where K is the number of output neurons, P is the number of tuples in the training data, c is the average output of the candidate over all tuples in the training data and e_k is the average error at output neuron k over all tuples in the training data. When all candidates are trained, the candidate with the largest covariance S is added to the ANN by fixing its input connections and adding output connections to all output neurons, whose weights are initialized with small random values. When a new neuron is added all output connections are trained again. This is repeated until the maximum number of neurons is reached or the ANN is trained good enough.

Possible stop criteria are the mean square error and the number of output values, which differ more than a selected bit fail limit from the function value in the training data. If the network does not reach the required error values another stop criterion is a maximum number of epochs.

Altogether FANN is fast and offers many possibilities to adapt a network and its training process.

Open NN

Open NN ([7]) was developed by Roberto López as part of his PhD thesis [5], so as a PhD thesis is more complex than a graduate project, it might be better suitable than FANN.

The Open NN library is structured different than the FANN library, according to [5]. The implemented type of ANNs is called multilayer perceptron and is built by perceptrons, which correspond to the neurons described earlier. In Open NN the topology and the activation functions can not be adapted as diverse as in FANN. A multilayer perceptron as implemented in Open NN is a feed-forward network in which every neuron has connections to all neurons in the next layer. It can have multiple layers: one input, several hidden and one output layer, as mentioned in [6]. There are five different activation functions available: threshold, symmetric threshold, logistic, hyperbolic tangent and linear functions, for more details see [8]. In the hidden layers every neuron has a sigmoid activation function as default value, in the output layer they have linear ones.

Each multilayer perceptron has an objective functional assigned, which defines the task the network should solve and provides a method to measure how well the network represents the function it should learn. The concept of an objective functional is a replacement for the concept of an error function and enables to extend the learning tasks, which is not relevant in this thesis, because an error function is already sufficient. For example the sum of squares error is an objective functional.

According to this concept the training process is to minimize the objective functional. An objective functional is improved by a training algorithm. To optimize the objective functional the training algorithm adjusts the weights in the network. Like in FANN, there are, according to [5], different training algorithms available for the initial weight vector $w^{(0)}$, learning rate $\lambda^{(i)}$, gradient vector $g^{(i)}$ and Hessian matrix $H^{(i)}$ as defined in [5]:

- RandomSearch: tries out randomly distributed weight vectors, zero-order, global optimization method
  $\rightarrow$ extremely slow convergence in most cases, only used to obtain good initial guess for more efficient methods

- EvolutionaryAlgorithm: described in more detail in [5], zero-order, global optimization method
  $\rightarrow$ used for problems, which are difficult to solve with common methods

- GradientDescent: $w^{(i+1)} = w^{(i)} - \lambda^{(i)} \cdot g^{(i)}$, first-order, local optimization method
  $\rightarrow$ suitable learning rate necessary, sometimes requires many iterations, slow convergence

- ConjugateGradient: $w^{(i+1)} = w^{(i)} - \lambda^{(i)} \cdot h^{(i)}$, where $h^{(i)}$ is the train direction vector with $h^{(0)} = -g^{(0)}$ and $h^{(i+1)} = g^{(i+1)} + \gamma^{(i)} \cdot h^{(i)}$ and $\gamma^{(i)}$ is a parameter which can be updated in several ways as described in [5], first-order, local optimization method
  $\rightarrow$ combination of GradientDescent and NewtonMethod, more effective than the single methods in isolation

- NewtonMethod: $w^{(i+1)} = w^{(i)} - \lambda^{(i)} \cdot H^{-1(i)} \cdot g^{(i)}$, second-order, local optimization method
  $\rightarrow$ high computational complexity to compute the Hessian matrix and its inverse

- QuasiNewtonMethod: $w^{(i+1)} = w^{(i)} - \lambda^{(i)} \cdot G^{(i)} \cdot g^{(i)}$, first-order, local optimization method
  $\rightarrow$ like NewtonMethod, but H^{-1} is approximated by G to reduce the computational complexity

The zero-order algorithms in Open NN are mainly useful to find a good initial guess for problems which are difficult to solve. The first-order algorithms are similar to the algorithms available in FANN and the second-order algorithms are just used to find an even better solution than the first-order ones if possible.

A training strategy can include initialization, main and refinement training algorithms, according to [6]. This is useful for difficult problems, where just one algorithm does not yield satisfactory results. The initialization algorithm, which is usually a zero-order algorithm, should bring the network near the optimum, with global optimization. The main algorithm, on which the strategy mainly relies, is used to find a nearly optimal solution, here often a first-order algorithm is used. Last, for even more accuracy, the refinement algorithm is used; second-order algorithms can perform best here, as they require the most exact information.

Some of the stopping criteria in Open NN are the following, as described in [5] and in [6]:

- maximum number of epochs reached
- maximum computation time exceeded
- evaluation of the objective functional is minimized to a goal value
- performance improvement in one epoch is less than a certain value
- norm of objective function gradient is below a goal value

Altogether Open NN offers as FANN many possibilities to adapt a network and the training process, but the training process is slower than FANN. Using FANN has the advantage, that the effect of changes in the other parameters can be tested faster, because the training process in Open NN, especially when the evolutionary training algorithm is involved, needs a lot more time to reach a demanded error than FANN. This would be an acceptable disadvantage, as the final network has to be trained just once, and using the network to compute an output is still fast. Open NN also has some functions, which are not implemented in FANN and vice-versa. For example the evolutionary algorithm, which might be useful. So the final decision, which library is better suited and is used to learn a control strategy for a hybrid vehicle, is discussed later.

Chapter 3

Learning control strategies

In this chapter the generation of training data, as well as the decisions about the topology, activation functions and training algorithms for the network, which is supposed to learn a control strategy for a hybrid vehicle, are described.

There are several control strategies implemented, basic control strategies as there would be rule-based strategies (ICE, EM) and strategies based on optimal control either non-predictive (ECMS) or with prediction horizon (DP, as in [4]) and more complex learning based strategies (GA, as described in [13]), which use a set of basic control strategies and choose in each situation the currently best split.

To train a network training data is required. This is generated by the function, which should be learned by the network. The number of training data grows exponentially with the number of dimensions which are considered as input, as for all admissible ranges of the inputs the output behaviour has to be learned. Therefore strategies which use a prediction horizon like DP and learning based strategies like GA, which use basic control strategies with a prediction horizon, are not learned here, as they need for each time step in the prediction horizon a gear and a requested velocity as additional input dimensions. To keep the number of inputs small in this thesis the strategy, which is learned, is an ECMS. Which training data is used to learn an ECMS and how this training data is generated is the topic of the next section.

3.1 Generation of training data

To train an ANN a set of training data as described in Section 2.3.2 is required. Training data consist of tuples, which each contain a possible combination of input values and the corresponding output values. When using the ECMS the output depends on the gear, the requested velocity and the actual velocity, for the PECMS also on the battery state of charge and for the PIECMS additionally on the sum of the past battery states of charge. In the following the PECMS is used, as for the PIECMS one input value more is necessary. For the ECMS even one input value less would be necessary, but that means the battery state of charge is not taken into account when distributing the torque on the engines, which is not reasonable, because if the battery is nearly discharged using the electric motor is impossible and when it is nearly completely charged it is useless to charge it using the combustion engine. The only output that is computed by a control strategy is the split, so an ANN, which learns a control strategy, has also only this one output value. Altogether a tuple in the training data specifies five values: $(v_{req}, v_{act}, gear, SoC, split_{ice})$, where $split_{ice}$ is the only output value.

The training data have to be generated by using the function that should be learned, in this case the PECMS. Before generating the data, a recapitulation of the vehicle model implemented within the OASys project is useful. When a vehicle driving a certain driving cycle is simulated, not always the split calculated by the control strategy is used. After the control strategy computed a split for the current conditions, the control converter examines whether the computed split is feasible. If one of the constraints in Equations 2.11 to 2.15 is not fulfilled when using this split, the split is discarded and replaced by another split the control converter computes. It has to be decided, if the network should be trained with the split values PECMS computes, or the ones, which are in fact used, because of the control converter.

In PHEVs the combustion engine can be used to recharge the battery, when it runs with a torque greater than the requested torque. In these cases the split is larger than one. Using the combustion engine to recharge the battery comes into effect either to enable the combustion engine to operate in its optimal rotational speed ranges with a high efficiency, or when the battery has nearly reached the lowest permitted state of charge. If in the latter case, the control strategy computes a split, with which the electric motor would be used anyway, the control converter changes this split, as Equation 2.15 would not hold, to a split value which is not necessarily in $[0, 1.5]$, but rather greater. So both, the strategies and the control converter, have this so called 'boost' option, but in the control strategies implemented in the OASys project the split is restricted to $[0, 1.5]$. As the control converter has no such restriction, the split the control converter computes can have values far above 1.5. This would cause problems when scaling the split values to a certain interval. Therefore the strategy itself without the control converter is learned. This causes no problem, because the control converter is used during the simulation, when using a strategy with a neural network, as well. This way the comparison between the split values the PECMS computes and the ones the neural network strategy computes is more sensible. Additionally it can be evaluated how often the control converter changes the split, when used during the simulation.

Next it has to be decided, whether the training data needs to be scaled. As mentioned in Section 2.3.1, at least the output data have to be scaled to the range of the activation functions in the output layer. In this thesis the hyperbolic tangent is used as activation function, therefore the output values have to be scaled to the interval $[-1, 1]$. The input values can be scaled to the interval $[-1, 1]$ too, as this could improve the training of the network. As the orders of magnitude of the input values do not differ too much, the results might be good even without scaling. As the initial weights are set using Algorithm 1 of Widrow and Nguyen, but generalized for training data in an arbitrary interval, the input values are first not scaled.

It still has to be decided, which data tuples represent the strategy the best. As a reminder: each data tuple contains the four input values v_{req}, v_{act}, SoC and $gear$ and the corresponding split as output value. The input values in the training data should cover the whole domain of possible input data when using the trained network. So a first approach would be to generate data tuples for homogeneously distributed grid points covering the possible intervals of all input values. For each of those input combinations the corresponding output $(split_{ice})$ is computed using the PECMS. These computations involve the simulation of the vehicle behaviour with the demanded v_{act}, v_{req}, $gear$ and SoC. The vehicle is simulated beforehand until the velocity v_{act} is reached. The state of charge SoC is set, then the PECMS is called with the requested velocity and gear. When using the C++ implementation this vehicle is required for the computation of the equivalent consumptions for different split values, but when using the Matlab implementation the vehicle is only used to convert the inputs for the network to the corresponding inputs for the PECMS and not for the computation itself. Because the Matlab PECMS gets the requested torque at the wheels, the angular velocity and the fuel rate instead of the actual and requested velocities as input. These values are computed by the vehicle model. After this computation the inputs SoC, $gear$, v_{act}, and v_{req} as well as the split value computed by the PECMS are written in the training data file.

In this training data there might still be data tuples, which are unrealistic, for example for $gear = 1$ a relatively high velocity of $v_{req} = 40\frac{m}{s}$ is not feasible. To train the network with unrealistic training data is not useful, because when the network is used in a control strategy itself, it will not have to compute split values for such inputs. Moreover, training the network with such unrealistic inputs might hinder learning and lead to unnecessarily bad output values for realistic inputs too. Therefore it is better to generate the training data just with realistic inputs and not with all possible combinations of input values.

Even more realistic data would be the combinations of input values that occur in sample driving cycles. So another approach is to consider those data tuples which occur while simulating a vehicle model using the PECMS on a certain driving cycle. This is probably less useful than the first approach described above, because in this way the training data would not contain all different realistic input values and it cannot be guaranteed that the whole possible input domain is fully exploited, which is important for the adaptation of the weights in the ANN. Another disadvantage is that the used driving cycles cannot be used to test the result, as the test data should be different from the training data. A further problem might be that the ANN simply memorizes the driving cycle, which was used to generate the data, which is not the aim, as the network should replace the strategy in general and not just for one certain driving cycle. Therefore the first approach for the data generation is used.

When using the first approach it has to be decided, which input values are realistic. The PECMS always keeps the SoC, which is independent of the other parameters, between 0.5 and 0.7. However, because the control converter is applied to the PECMS output, sometimes values slightly below 0.5 occur. Therefore when generating training data, SoC values in the range $[0.49, 0.7]$ are considered. As mentioned before, the velocities and the gear are not independent. For each gear there is an optimal driving range that should be considered. Furthermore the assumption is made that the gear is changed from one time step to the next by a maximum of one. So for each gear only those requested and actual velocities are relevant, which are in the optimal driving range of either the gear itself or the following or the previous gear. The optimal velocities for each gear in the used vehicle model and the resulting intervals for the training data are listed in Table 3.1. The decision on how many training data are generated, that is how large the distance between the input tuples in the training data is, is made later.

When the training data finally are available, it is still not possible to start with the training of an ANN yet. Before this is possible some more decisions have to be made. For example which activation functions are the best suited, how many neurons are necessary or which training algorithm leads to the best result. How those decisions are made is described more detailed in the next sections.

gear	optimal velocities in m/sec	training interval in m/sec
1	0 - 3	0 - 8
2	3 - 8	0 - 14
3	8 - 14	3 - 20
4	14 - 20	8 - 45
5	20 - 45	14 - 45

Table 3.1: Feasible velocity ranges

3.2 Selecting the neural networks parameters

In this section the decisions about the parameters of the used ANN are explained, starting with the activation functions over the training algorithm to the corresponding parameters of the ANN.

First the activation functions of the neurons have to be chosen. The neurons in the input layer have always a linear activation function. As mentioned in [1] a network with three layers, when using the sigmoid activation function in the hidden layer, is already able to learn all functions a network with different activation functions could learn. So in the following the used networks have sigmoid activation functions in the hidden layers. For the output layer, in Open NN linear activation functions are used by default. However the linear activation function is not useful for the output layer in our case, as the output interval should be bounded, but with linear activation functions it would be possible to get output values, which are not in the admissible interval. Therefore we use sigmoid activation functions also in the output layer.

Next the decision which training algorithm is used has to be made. To test if a zero-order algorithm like the evolutionary algorithm or a second-order algorithm improve the result, Open NN has to be used, but in FANN apart from different first-order backpropagation algorithms also the cascade training algorithm is implemented, which might lead to better results. So Open NN and FANN offer different possibly useful modification options for the network.

Altogether different configurations are tested in this thesis: one configuration uses a first-order training algorithm which both FANN and Open NN offer, so that this configuration enables to compare the libraries regarding the running time and the resulting trained networks. Furthermore two library-specific configurations are tested, to improve the results. Those configurations are described in Chapter 5.

Next we need to specify the learning rate, the desired error, and the number of iterations. The learning rate has a default value of 0.7 in FANN, which seems to work best for our problem too, when comparing the result for fixed parameters and different learning rates. During the first training iterations, the error usually shrinks. However, when a certain error is reached, it starts to grow again in the following iterations. Thus, we obtained the sweet spot of the error value by using a very small desired error. Afterwards, we started the training again using the observed minimal error value. If a certain error has to be reached, the number of epochs must be set accordingly large. In general, a large number of epochs is useful, as the chance to terminate the learning process because the desired error is reached increases. Therefore, we set the maximum number of iterations to 100.000.

Furthermore, some decisions about the topology have to be made. For example the number of hidden layers and the number of neurons have to be decided. The decisions about the topology are explained in the next section.

3.3 Topology of the neural network

In this section the remaining decisions regarding the topology of the network are discussed; the number of layers, the number of input values and the corresponding number of networks[1].

As a network with three layers is able to represent all continuous functions according to [1], a network with three layers should be sufficient and just the number of neurons must be adapted, if the PECMS is continuous. Otherwise, a network with two hidden layers is sufficient to represent the PECMS. But as mentioned in [11] not just the representability of a function, but also its learnability has to be considered. Functions which are difficult to learn might be learned better when using ANNs with more layers. But in general ANNs with more layers are more difficult to train, therefore we first consider an ANN with only one hidden layer. If the result is not sufficient then ANNs with two or even more layers are tested.

Next the number of inputs has to be determined. The networks should get the same information as the PECMS, to learn how to calculate the split like the PECMS does. That information covers only on the state of charge of the battery, the gear, the actual velocity, and the requested velocity. Therefore the network has to get this four values as input.

It has to be tested whether it would improve the result, if multiple networks with fewer input values would be used. To make this decision, the training data have to be examined. To estimate whether an ANN is able to learn a function, represented by a set of data, a plot of the data, which should be learned helps, as there can be seen, if there are 'jumps' in the represented function, which makes it difficult to learn. This irregularities might be reduced, by using fewer input parameters and more networks.

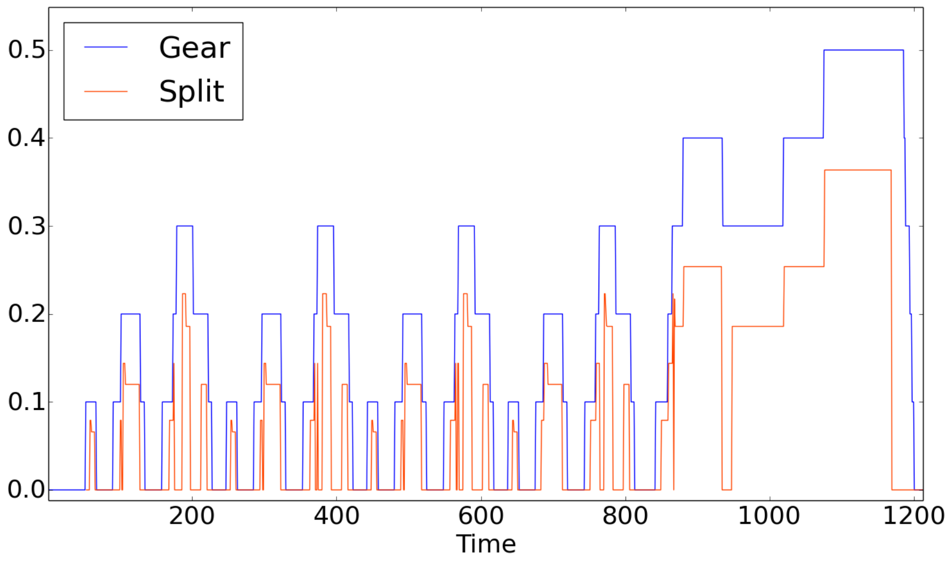

Figure 3.1: PECMS used on the NEDC driving cycle

[1]Please note, that for a function $f : A \times F \to B$ with A, B, F being sets and f being finite, we can either train a single network to learn f using inputs from $A \times F$ or we can try to learn $f(x) : A \to B$ by a separate network for each $x \in F$

When plotting the split and the gear for the driving cycle NEDC using the control strategy PECMS, see Figure 3.1, it can be seen that the split values are discontinuous when the gear is switched. As the gear can just take on 5 different values, it would be possible to train five different networks, one for each gear, instead of a single one with the gear as parameter. If for each gear one network is trained, those do not have to get the gear as parameter anymore and therefore get one input value less. So an approach to simplify the learning process is to train five networks, with just three input values: actual and requested velocity and the battery state of charge. The number of outputs is simply one, as only the split has to be computed.

So there are the following options:

- one ANN with four input values: $gear$, v_{act}, v_{req}, SoC
- five ANNs, for each gear one, with three input values: v_{act}, v_{req}, SoC

For the second option other training data than described in Section 3.1 have to be generated. In this case five training data files, one for each gear, are necessary, which do not contain values for the gear.

Which of these two possibilities is better has to be tested. Therefore the testing criteria have to be defined, which is the topic of the next chapter. Then the training and testing have to be executed, to be able to compare these alternatives.

Chapter 4

Neural network-based control strategy

This chapter is about the evaluation of the created ANNs. There are various aspects, which can be evaluated. In this thesis the focus is on the comparison of the input/output behaviour of the ANN with that of the PECMS and on using the ANNs as a control strategy and comparing this strategy to the PECMS when it is used in a simulated vehicle on a driving cycle. In this chapter first it is described how the trained ANNs can be used as a control strategy for a simulated vehicle. This is necessary to compare the final states of simulated vehicles on a certain driving cycle, one of them using the PECMS and another one using the trained ANNs. Later it is explained, how the input/output behaviour of the ANN and the PECMS can be compared directly.

For testing the created ANNs by using them as a control strategy and comparing this strategy to the PECMS two things have to be explained: how the ANNs are used as a control strategy and what has to be considered when using different strategies with the intention to compare them. It is started with the explanation how the ANNs are used as control strategy.

In Chapter 3 are two approaches considered, training one network with four inputs, or five networks with three inputs, for each gear one. When using the first approach, the trained network can simply be used as control strategy by giving it the corresponding inputs, which the PECMS would get as well, and using its output to distribute the torque. When the second approach is implemented, only all five networks combined can be used as control strategy. In this case it depends on the current gear which network gets the remaining three values as input, for example if the vehicle is driving in the third gear the ANN, which was trained for gear three, gets the input and only the output computed by this ANN is used to distribute the torque. So in each step only one network is used.

To examine such a neural-network-based control strategy, it is used in a simulation. For a simulation, a vehicle is simulated on a certain driving cycle. Therefore the simulation rate has to be set, a value which determines how often the car becomes simulated per second. To be able to compare this strategy to the PECMS, the simulation environments have to be the same, so this value has to be the same in all simulations. Also how often a new split becomes computed should be equal in all simulations, as computing the split more or less often leads to different results. In the

Matlab model the split value is fixed for one second, but the vehicle is simulated 50 times per second. To be able to compare the Matlab PECMS implementation to the neural-network-based strategy in C++, these settings are also used in the C++ simulation. So the control strategy computes a new split once every second for the present state of the vehicle model. During the simulation on a certain driving cycle, the simulated vehicle computes amongst other values also the absolute fuel consumption for the whole driving cycle.

When the trained ANNs are used as a control strategy, for example the drivability and functionality of this strategy can be evaluated, which is the topic of the next section, but the basic intention is to compare the network strategy to the PECMS. This is done, as explained above, by executing simulations using different control strategies. After the simulation are completed, the final states of the vehicle models can be compared, especially the absolute fuel consumption, as the main intention of the PECMS was to minimize this absolute fuel consumption and not for example the CO_2 emission. Additionally, also the final SoC has to be taken into account, as a higher SoC might compensate a slightly higher fuel consumption.

As a reference value the so called ICE strategy can be used, which is the strategy, which always returns the split value 1, and therefore the vehicle uses solely its combustion engine, like a conventional vehicle. The fuel consumption should be reduced, by using a hybrid vehicle, so all strategies should achieve a lower absolute fuel consumptions than this strategy. Apart from the absolute fuel consumption also the functionality of the network strategy is relevant to decide if it is a suitable replacement for the PECMS. Which criteria are used to examine this, is described in the following section.

4.1 Evaluation of control strategies

To evaluate a control strategy, several criteria have to be examined. A control strategy is useful only if the requested velocities are reached timely. This is guaranteed here by the control converter, independent of the control strategy itself, as the strategy just computes the distribution of the requested torque and not the concrete torque values for each engine. So the actual velocity is for every strategy, also those using ANNs, close to the requested velocity. As this would be the case for every control function, it is no measure for the quality of a control strategy. But as the network should have learned the PECMS, it should be compared, how often the control converter changes the split values, when using the PECMS and the network strategy, to measure how often the split value computed by the network is invalid.

Another criterion could be the drivability of the vehicle, when using the network strategy. But this would just evaluate the function relating the input with the output and not how well a network learned the PECMS, as the PECMS does not consider the drivability when computing the split values. A measurement for the drivability would be for example the sum of the absolute differences between successive split values. A good drivability would be achieved, when the split values do not change abruptly and rapidly, as the engines have to react according to this split values and in reality they are not able to react immediately and sound.

Finally, it is important that the computations are executed in real-time, as the intention for creating a strategy using an ANN is to achieve a fast strategy. To prove that this is achieved, the running times of both strategies can be compared; if the network strategy is at least as fast as the PECMS, it is real-time capable. This can be done by simulations on a given driving cycle, once using the PECMS, the other time using the network strategy and measuring both running times. When the above stated characteristics are similar in both strategies the network strategy would be a suitable replacement for the PECMS.

So far only the results of the simulations on different driving cycles are considered, but not yet the input/output behaviour. The examination of the simulation results enables to compare for example the driving behaviour and the fuel consumption and evaluate, if the strategy using neural networks is a suitable replacement for the original strategy on different driving cycles, but the examination of the input/output behaviour is well suited to evaluate the quality of the trained network. Therefore this is analysed in the next section.

4.2 Testing the neural networks

In this section the testing process for an ANN, which learned the PECMS, is described. To test the ANNs themselves, which means to compare their input/output behaviour with that of the PECMS, it has to be decided, which testing data and which error functions will be used to estimate, if the ANNs properly learned the PECMS. Possible test data are grid data in the input intervals. Grid-based test data can be generated exactly like grid training data, which is described in Section 3.1. If both training and testing data are generated grid-based, the grid for the test data needs to be shifted, as the test data should not be identical to the training data, to test if the ANN not just memorized the training data. Another possibility is to use a driving cycle for testing. Therefore, the PECMS is simulated on a concrete driving cycle and for each time step the actual and requested velocities, the battery state of charge, the gear and the computed split value are written in a file, which then can be used as test data. For testing this is the better alternative, as this way it is tested, if the network is able to compute the correct split for realistic input values.

By using the ANNs as a control strategy also the computed split values can be compared, but the errors are cumulative, as a different split value causes a slightly different state of the simulated vehicle, which means the ANN gets other inputs in the next step than PECMS, which in turn leads to a different split. However, if the network is trained well, the deviation from the PECMS should be small. Comparing the split values during a simulation is also useful to see which influence deviations have on the absolute fuel consumption, which can be examined by comparing the final states of the vehicles, as described in Chapter 4.

When plotting the split values the PECMS computes and the ones the network strategy computes on the same driving cycle, the current states of the vehicle can be plotted too, to see for which input values the difference of the output values is too large and to determine what could be changed in the training process to achieve a better result. Therefore it would be useful to prevent subsequent errors. This can be done by computing the output of the PECMS during the simulation using the network strategy. That means, we use the neural network strategy result as split values in the simulation, but for each input we additionally compute as reference values also the splits determined by the PECMS strategy, so different splits do not cause subsequent errors.

To estimate, if an ANN properly learned the PECMS, a measurement for the error of an ANN has to be defined. The testing process for an ANN in general was already described in Section 2.3.3, there are also metrics defined, which are used for determining the average error in the output values. The mean square error defined there by the Euclidean metric and the testing criteria defined in this chapter are used in the following chapter to evaluate the created networks.

Chapter 5

Evaluation

In this chapter the trained networks are evaluated. It is also explained which problems occurred and how they are tried to be solved. For being able to evaluate the created networks, in Chapter 4 testing criteria have been defined. In the following only the testing criteria defined there are considered. Using those criteria the neural-network-based strategies are compared to the PECMS they should have learned and to the ICE strategy. In the next section the best result is evaluated more detailed.

The results for ICE in Table 5.1 provide a lower bound for the absolute fuel consumption. In this chapter the following abbreviations are used: DC for driving cycle, CC for the number of splits which had to be changed by the control converter, and MSE for mean square error. Furthermore, $cons_f$ is the absolute fuel consumption, SoC is the final SoC in a simulation, and $\Delta split_{ice}$ is the sum of the differences of successive split values.

DC	$cons_f$	SoC	CC	$\Delta split_{ice}$	running time
NEDC	429.077	0.70	342	0	11.37
FTP_75	725.771	0.70	248	0	17.44
FTP_HIGHWAY	380.8	0.70	97	0	7.22

Table 5.1: ICE results

There are different ANNs, which have to be tested, trained using different training algorithms, as mentioned in Section 3.2. For comparability, the configurations of training algorithms the RPROP algorithm in FANN and the QuasiNewtonMethod in Open NN are used. Actually, QUICKPROP is more similar to the QuasiNewton-Method, but as RPROP is a first-order training algorithm leading to significantly better results for this problem, RPROP is used instead. The libraries are still comparable, because both use one first-order training algorithm, but QUICKPROP and the QuasiNewtonMethod are not exactly the same training algorithms either.

Additionally two library-specific configurations are tested. In FANN the cascade training algorithm is used, to test if this type of training algorithm improves the result. In Open NN a training strategy, consisting of the evolutionary algorithm, the quasi-Newton method and the Newton method, is used.

From here on the corresponding control strategies have the following names:

- FANN: using the RPROP algorithm

- CASCADE: using the cascade training algorithm implemented in FANN
- OPEN_NN: using the QuasiNewtonMethod
- OPEN_NN_OPT: using a training strategy with three different training algorithms

All four training algorithms can be used for training either one network with four inputs, or five networks with three inputs, as explained in Section 3.3. It also has to be examined, which influence the training data have. It is always trained with grid data, but the number of data tuples in the training data can be varied, by adjusting the distance between individual data tuples.

Due to time constraints not all possible combinations of settings can be tested in this thesis. As it is computationally intensive and therefore time-consuming to generate many training data and train ANNs, just the two FANN configurations are used to determine, which settings lead to better results, as FANN is faster than Open NN. We start with a small amount of training data to make the decision if one or five networks lead to better results. How many neurons are necessary is tested for each configuration separately, for the networks trained using the RPROP algorithm one hidden layer is used.

ANNs	DC	MSE of FANN	MSE of CASCADE
1	NEDC	0.0134	0.00287
	FTP_75	0.0359	0.0147
	FTP_HIGHWAY	0.117	0.0307
5	NEDC	0.00281	0.00566
	FTP_75	0.00404	0.00691
	FTP_HIGHWAY	0.00692	0.0225

Table 5.2: Results for one and five networks, trained with a total of 6372 training data tuples

In Table 5.2 it can be seen that the control strategies with five networks lead better results, only the CASCADE strategy on the driving cycle NEDC is better with one network. This was tested for networks which are trained with training data with scaled inputs and some with unscaled inputs. The results are approximately equally good, while training with training data with scaled inputs and outputs the demanded error is just reached faster. Therefore in the following we analyse strategies using five networks, for each gear one and first training data with unscaled inputs.

Next it is tested whether using more training data improves the results, because when plotting the split values, it can be seen that the split the network strategies compute tends to be lower than the one computed by PECMS for the same input data, especially for the gears one to three, see Figure A.1. This could be due to the small amount of training data, if no tuple with a high split is in the training data, it is understandable that the network does not learn to return a higher split for inputs between the tuples in the training data. And for gear five there are some peaks in the split values the PECMS does not compute. Which cause these have has to be determined, too.

The results in Table 5.3 show that more training data do not improve the total result. To find the reason for this the mean square errors for the individual networks

strategy	DC	MSE
FANN	NEDC	0.00701
	FTP_75	0.00363
	FTP_HIGHWAY	0.0107
CASCADE	NEDC	0.00925
	FTP_75	0.00825
	FTP_HIGHWAY	0.0379

Table 5.3: Results for five networks trained with a total of 120210 training data tuples

are calculated, see Table A.1. It can be seen, that the network for the fourth gear provides worse results than the one trained with less data, see Table A.2. It is also noticeable that the results of CASCADE are for less as well as for more training data worse than the results of FANN, therefore the CASCADE strategy is not further pursued.

The worse results of the fourth network might be caused by the unscaled training data, which might slow down the training. But when training the networks for the fourth and fifth gear with scaled training data the results are not significantly better. As the training data are not completely continuous, it might help to smooth them, but this caused even worse results and is therefore discarded again.

To determine, why the results for the fourth gear is worse than with less training data, the split values on the NEDC driving cycle are plotted, see Figure A.2. The problem is that the network for gear four returns too often zero. To determine why this is the case the training data for gear four are plotted. As the training data have four dimensions, which is difficult to plot, the split depending on the requested and actual velocities for the fixed state of charge, for which the problem occurs, is plotted in Figure 5.1.

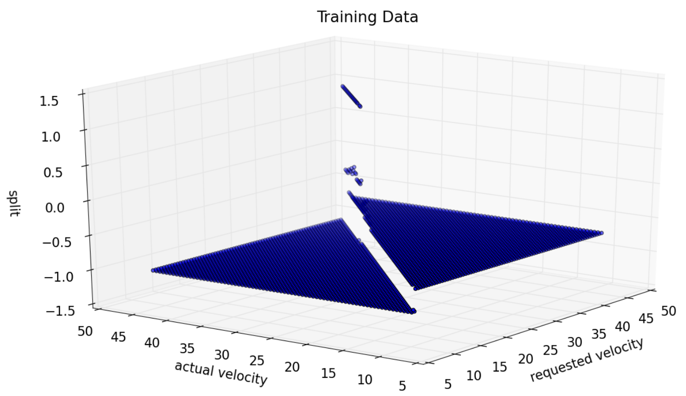

Figure 5.1: Training data for gear four, for the fixed state of charge $SoC = 0.5$, with scaled split values

It can be seen that the split is minimal, when the requested velocity is smaller than the actual velocity. When the requested velocity is noticeably larger than the actual one, the split is also constant but larger. But for actual and requested velocities, which are nearly the same, there are training data with even larger split values. When the network is trained with this training data, it tries to generalize those data, to be able to compute a reasonable output for input values in-between the training data tuples. This is quite difficult for training data like this, as they are not continuous. The reason for this is that in PECMS a case distinction is made whether the requested torque is greater or less than zero. So for input values for which the torque is just slightly greater than zero the behaviour of the ANN might be different than the PECMS's behaviour, as the output of inputs in-between the training data tuples depends on several neighbouring training data tuples, which might cause unexpected peaks or zeros. This might be averted by generating more training data for such discontinuous input areas. This is due to time-constraints not part of this thesis.

So far the best result is achieved with five networks and many training data, except for the network for gear four. Whether with Open NN a better result can be achieved still has to be examined. When evaluating the results that are achieved with Open NN, as listed in the Tables A.4 and A.3, it can be seen that the results achieved with more training data are better than those with less training data. Though there is no significant improvement noticeable when using a training strategy consisting of three different training algorithms.

When comparing the Open NN strategies with the FANN strategy, the FANN strategy is slightly better than the Open NN strategies. Therefore the best total result so far is achieved with FANN, using five networks, for each gear one, where those for gear one to three and five are trained with more training data and the one for gear four with less training data. The best strategy is examined in more detail in the next section, there also the drivability and the split value progression for a certain driving cycle is evaluated.

5.1 Final results

In this section the best achieved result is evaluated in more detail. For comparison the test results of PECMS on different driving cycles are used, see Table 5.4.

DC	$cons_f$	SoC	CC	$\Delta split_{ice}$	running time
NEDC	388.275	0.626	61	8.44	12.31
FTP_75	587.058	0.533	80	42.48	18.69
FTP_HIGHWAY	356.712	0.579	73	19.80	7.66

Table 5.4: PECMS results

The best result is achieved with FANN, as examined in the last section, with five networks that are trained with many data, only the network for gear four is currently trained with less training data. Simulation results for FANN as control strategy on different driving cycles are listed in Table 5.5. The fuel consumption and final state of charge of the battery are nearly the same as when using the PECMS. Also the difference of successive split values is similar to the one of PECMS, but the control converter has to change the split values way more often than when the PECMS is used. However, if we additionally compute the PECMS outputs during a simulation with FANN as a control strategy, the control converter would change the split the

PECMS computes more often, in about the same order of magnitude as when using FANN. The running time is even shorter than the one of the PECMS, so altogether we can confirm the drivability and functionality of the FANN strategy. So the FANN strategy is a suitable replacement for the PECMS.

DC	$cons_f$	SoC	CC	$\Delta split_{ice}$	running time
NEDC	388.095	0.626	4830	8.037	10.507
FTP_75	586.613	0.533	5806	44.77	16.063
FTP_HIGHWAY	356.021	0.579	3490	18.47	6.73

Table 5.5: Final states of simulated vehicles when using FANN as control strategy

Finally, the output error of the FANN strategy is determined using the mean square error as before. The total test results are presented in Table 5.6, the individual test results for each network in Table 5.7.

DC	MSE
NEDC	0.00102
FTP_75	0.00360
FTP_HIGHWAY	0.00699

Table 5.6: Total test results of the FANN strategy

	MSE				
DC	gear 1	gear 2	gear 3	gear 4	gear 5
NEDC	0.000302	0.00171	0.00212	0.000179	0.00195
FTP_75	0.00565	0.00162	0.00252	0.00627	0.00348
FTP_HIGHWAY	$7.14\cdot10^{-5}$	$8.42\cdot10^{-5}$	0.00199	0.00559	0.00811

Table 5.7: Individual test results of the networks used in the FANN strategy

The split value progression for those driving cycles is plotted in the Figures A.3, A.4 and A.5. Using the example driving cycle NEDC the result is evaluated more detailed in the following. In Figure 5.2 the split values for the start of the NEDC are plotted. The progression of the split values the FANN strategy and the PECMS compute matches approximately. The networks compute peaks, where PECMS computes them, only the height does not always match. Furthermore, FANN computes some peaks that the PECMS does not compute; this is probably due to the fact that the networks try to generalize the function their training data represents by applying extrapolation.

In Figure 5.3 a more detailed view on the end of the NEDC driving cycle is illustrated. It can be seen that the split values for gear four are a little higher than for PECMS, where the input values correspond to torque values just slightly greater than ; this situation was analysed earlier in this chapter. Probably for the same reason, the split values for the fifth gear are somewhat more uneven than those of the PECMS.

All in all, the resulting neural-network-based control strategy satisfies our needs, but there are still possibilities to further improve the result. These possibilities are explained in the next section.

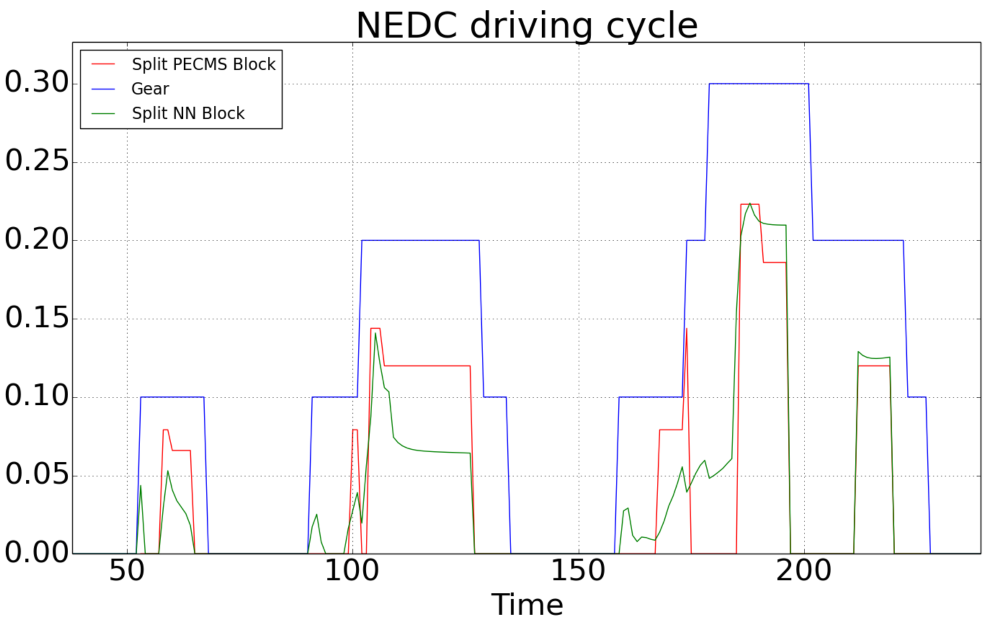

Figure 5.2: Split values FANN computes for the start of the NEDC driving cycle

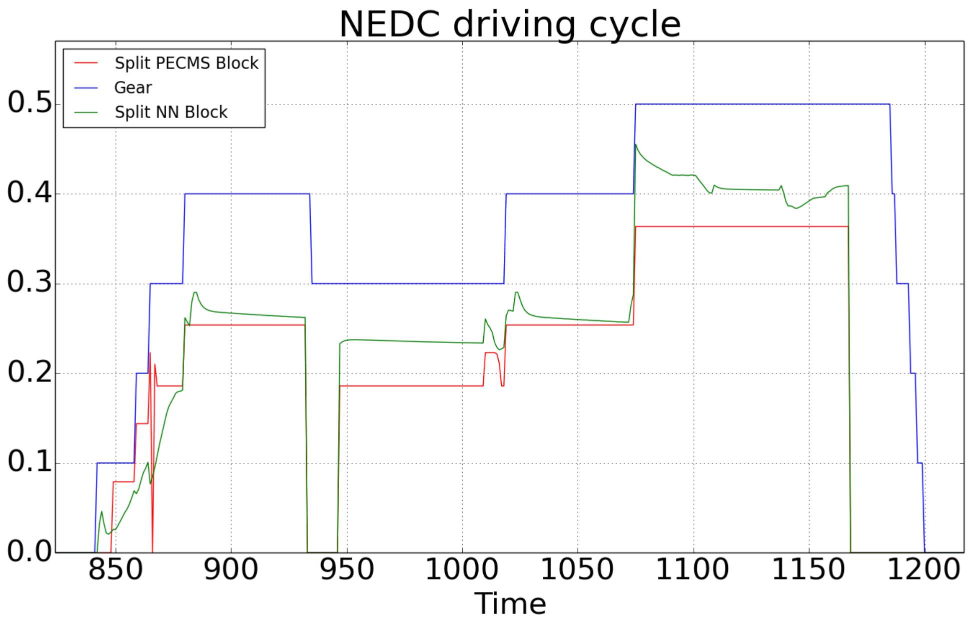

Figure 5.3: Split values FANN computes for the end of the NEDC driving cycle

Chapter 6

Conclusion

This thesis started with an introduction to the vehicle model developed in the context of the DFG project OASys and the corresponding control strategies. In the further proceeding the vehicle model was used to evaluate different control strategies by computing the absolute fuel consumption for different driving cycles, as the purpose of the considered control strategies is to minimize the fuel consumption.

Next, artificial neural networks were introduced and two different libraries for artificial neural networks were presented. Those libraries were applied to try learning the control strategy PECMS for hybrid electric vehicles using artificial neural networks. Several decisions were made about the training data, the network topology, other network parameters and the used training algorithm. At last as training data grid data were used, where the split values were computed directly by the PECMS and not by the control converter, which is a control instance of the implemented vehicle model.

Regarding the network topology it was proved to be better to use five networks, for each gear one, instead of one network with the gear as additional input. The comparison of the libraries FANN and Open NN showed that FANN leads to slightly better results for our problem.

Several testing criteria for the developed control strategies were used to evaluate them and to identify the best one. It was noticed that a larger amount of training data leads mostly to better results than a smaller amount. The best control strategy was evaluated in more detail also regarding aspects like driveability and functionality, with the result that the performance of a neural-network-based control strategy is equally good as the PECMS. This makes the neural-network-based strategy a suitable replacement for the PECMS.

Nevertheless there are still many possibilities to improve the results, which are described in the next section.

6.1 Future work

The achieved control strategy would be optimal, if for every possible input value its output would be the same output as PECMS computes for this input. This is not the case yet, so the result can still be improved. Due to time constraints in this thesis only a limited amount of training data could be generated. Also the training time and size of the used ANNs is limited. It is assumed that generating more training data would improve the result, and also letting the ANNs learn for a longer time should lead to improvements. This could not be tested in the course of this thesis, as it already took several days to generate the used training data and to train the ANNs. When generating more training data it might help to take into account where the deviations are larger than average and generate more data especially in the corresponding ranges.

It would also be interesting to apply different training strategies. At last it would be useful to generate a neural network that has learned the best available control strategy, i.e. the one with (i) the lowest energy consumption measured by absolute fuel consumption and battery state of charge and (ii) a good drivability measured by the split-differences. That is our motivation to train neural networks to learn the genetic-algorithm-based strategy GeneiAL as future work.

Bibliography

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

[2] D. Dichant. *Demand Planning mittels Neuronaler Netze.* Diplom.de, 2002.

[3] Scott E. Fahlman. An empirical study of learning speed in back-propagation networks. Technical report, Carnegie Mellon University, 1988.

[4] Sascha Geulen, Martina Josevski, Johanna Nellen, Janosch Fuchs, Lukas Netz, Benedikt Wolters, Dirk Abel, Erika Ábrahám, and Walter Unger. Learning-based control strategies for hybrid electric vehicles. In *Proceedings of the 2015 IEEE Multi-Conference on Systems and Control (MSC 2015)*, pages 1722–1728. IEEE, 2015.

[5] Roberto López González. *Neural Networks for Variational Problems in Engineering.* PhD thesis, Technical University of Catalonia, 2008.

[6] Roberto López González. Open NN manual, 2012. `http://libfann.github.io/fann/docs/files/fann_cpp-h.html`.

[7] Roberto López González. Open NN: An open source neural networks C++ library. `http://opennn.cimne.com/`, 2014.

[8] Roberto López González. Open NN user's guide, 2014. `http://opennn.cimne.com/docs/Flood3UsersGuide.pdf`.

[9] Lino Guzzella and Antonio Sciarretta. *Vehicle Propulsion Systems - Introduction to Modeling and Optimization.* Springer Science & Business Media, Berlin Heidelberg, 2012.

[10] Christian Igel and Michael Hüsken. Empirical evaluation of the improved Rprop learning algorithms, 2003.

[11] David Kriesel. *Ein kleiner Überblick über Neuronale Netze.* 2011. `http://www.dkriesel.com`.

[12] U. Lämmel and J. Cleve. *Künstliche Intelligenz.* Carl Hanser Verlag GmbH & Company KG, 2012.

[13] Johanna Nellen, Benedikt Wolters, Lukas Netz, Sascha Geulen, and Erika Ábrahám. A genetic algorithm based control strategy for the energy management problem in PHEVs. In *Proceedings of the Global Conference on Artificial Intelligence (GCAI 2015)*, pages 196–214. EasyChair, 2015.

[14] D. Nguyen and B. Widrow. Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. In *Proceedings of the 1990 International Joint Conference on Neural Networks, IJCNN*, pages 21–26 vol.3. IEEE, 1990.

[15] S. Nissen. FANN: Implementation of a fast artificial neural network library. `http://leenissen.dk/fann/wp/`, 2015.

[16] S. Nissen. FANN reference manual, 2016. `http://libfann.github.io/fann/docs/files/fann_cpp-h.html`.

[17] Steffen Nissen. Large scale reinforcement learning using Q-SARSA (λ) and cascading neural networks. *Masters thesis, Department of Computer Science, University of Copenhagen, København, Denmark*, 2007.

[18] Stuart Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach.* 2013.

[19] N. K. Treadgold and T. D. Gedeon. The SARPROP algorithm: A simulated annealing enhancement to resilient back propagation. In *Proceedings of the International Panel Conference on Soft and Intelligent Computing*, 1996.

Appendix A

Experimental results

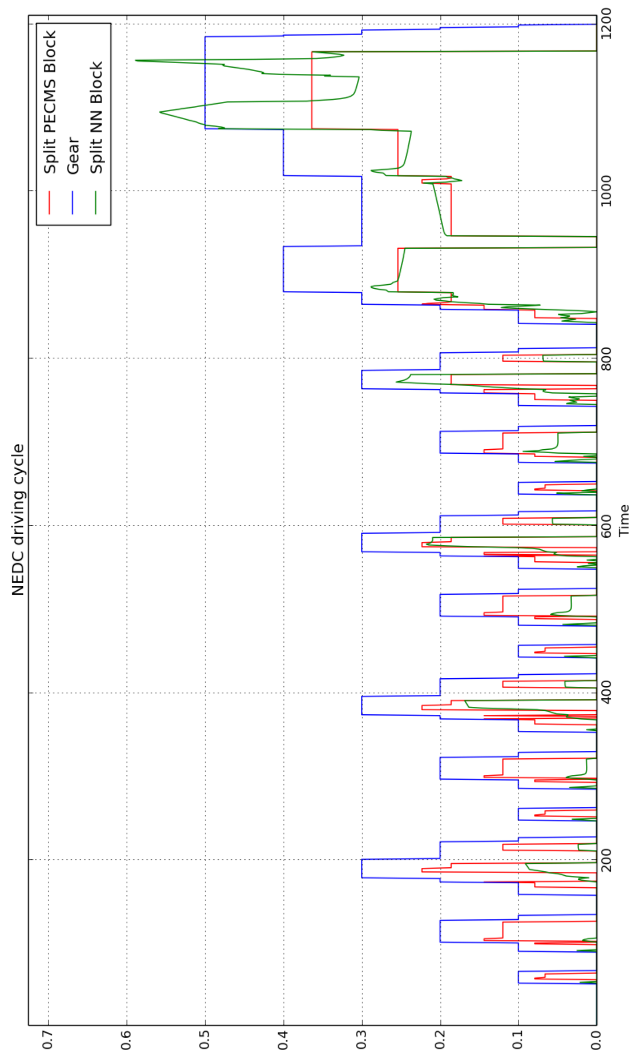

Figure A.1: FANN, using five networks trained with a total of 6372 training data tuples, used on the NEDC driving cycle

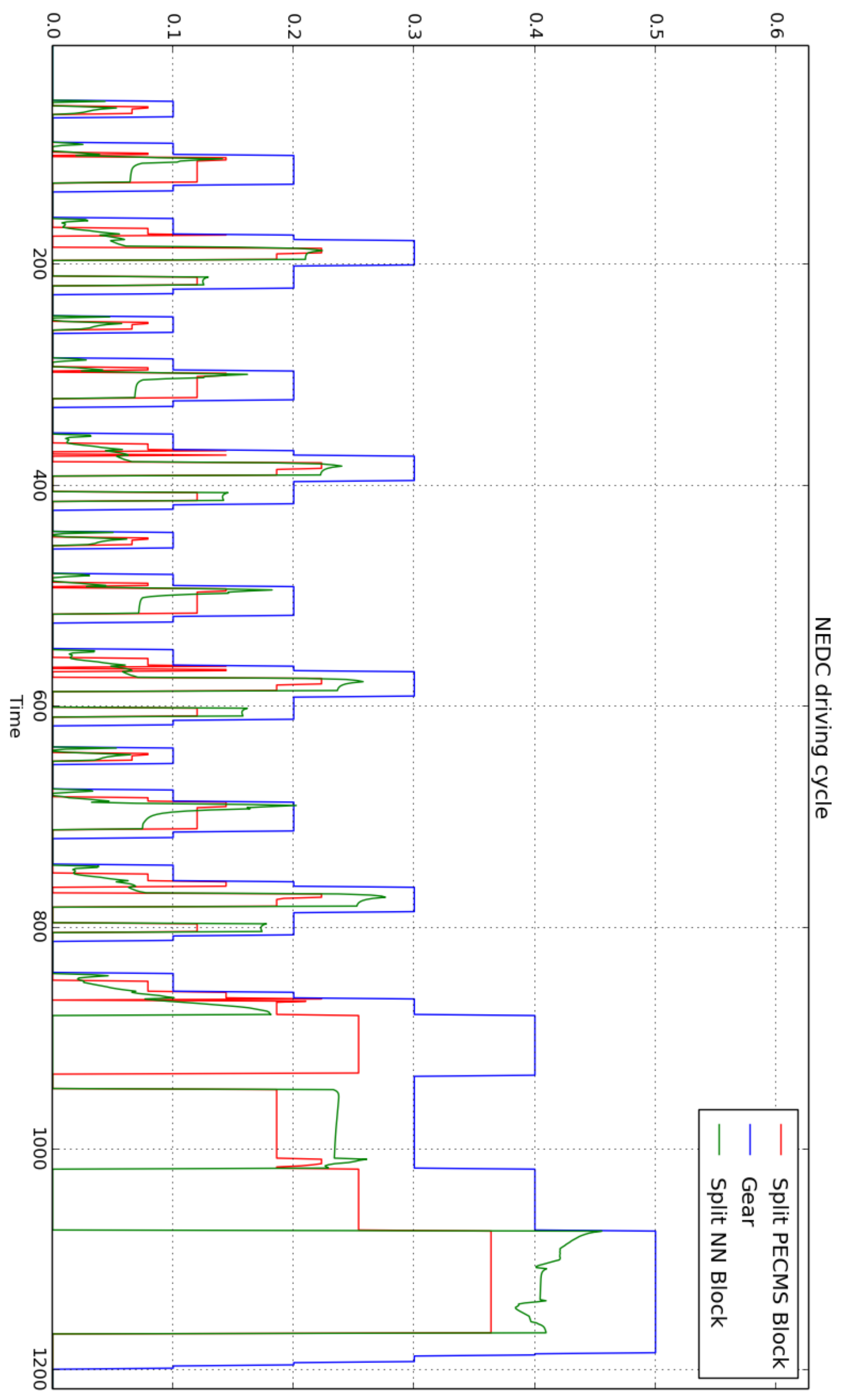

Figure A.2: FANN, using five networks trained with a total of 120210 training data tuples, used on the NEDC driving cycle

		MSE				
strategy	DC	gear 1	gear 2	gear 3	gear 4	gear 5
FANN	NEDC	0.000302	0.00171	0.00212	0.0621	0.00195
	FTP_75	0.00565	0.00160	0.00251	0.0161	0.00325
	FTP_HIGHWAY	$7.14 \cdot 10^{-5}$	$8.42 \cdot 10^{-5}$	0.00199	0.0437	0.00783
CASCADE	NEDC	0.000363	0.0050	0.00533	0.0308	0.0448
	FTP_75	0.00657	0.00297	0.00308	0.0114	0.0377
	FTP_HIGHWAY	0.000154	$2.74 \cdot 10^{-5}$	0.00577	0.0220	0.0458

Table A.1: Individual results when using a total of 120210 training data tuples for FANN and CASCADE

		MSE				
strategy	DC	gear 1	gear 2	gear 3	gear 4	gear 5
FANN	NEDC	0.000794	0.00555	0.00229	0.000182	0.0105
	FTP_75	0.00414	0.00134	0.00312	0.0120	0.0108
	FTP_HIGHWAY	$3.33 \cdot 10^{-5}$	$2.22 \cdot 10^{-5}$	0.00396	0.00725	0.00754
CASCADE	NEDC	0.000764	0.00400	0.00649	0.0166	0.0196
	FTP_75	0.00868	0.00192	0.00535	0.00968	0.0146
	FTP_HIGHWAY	0.000214	$8.92 \cdot 10^{-5}$	0.0103	0.0132	0.0263

Table A.2: Individual results when using a total of 6372 training data tuples for FANN and CASCADE

		MSE					
strategy	DC	total	gear 1	gear 2	gear 3	gear 4	gear 5
OPEN_NN	NEDC	0.061	0.000317	0.302	0.00340	0.0230	0.00991
	FTP_75	0.0165	0.0113	0.0497	0.00234	0.00820	0.0108
	FTP_HIGHWAY	0.0138	$1.28 \cdot 10^{-5}$	$3.85 \cdot 10^{-5}$	0.00179	0.00832	0.0166
OPEN_NN_OPT	NEDC	0.00696	0.000581	0.00402	0.0110	0.0106	0.0321
	FTP_75	0.00954	0.00753	0.00332	0.0109	0.0107	0.0240
	FTP_HIGHWAY	0.0267	$5.60 \cdot 10^{-5}$	$8.46 \cdot 10^{-5}$	0.010	0.0110	0.0320

Table A.3: Individual results when using a total of 120210 training data tuples for OPEN_NN and OPEN_NN_OPT

		MSE					
strategy	DC	total	gear 1	gear 2	gear 3	gear 4	gear 5
OPEN_NN	NEDC	0.623	0.132	1.420	1.5	0.214	0.129
	FTP_75	0.711	0.0393	1.146	1.433	0	0.125
	FTP_HIGHWAY	0.341	$9.96 \cdot 10^{-6}$	0.844	1.13	0.717	0.188
OPEN_NN_OPT	NEDC	0.754	0.101	1.421	1.017	1.37	1.429
	FTP_75	0.720	0.0954	1.147	1.009	0.713	1.146
	FTP_HIGHWAY	1.173	0.0914	0.844	0.808	0.883	1.29

Table A.4: Individual results when using a total of 6372 training data tuples for OPEN_NN and OPEN_NN_OPT

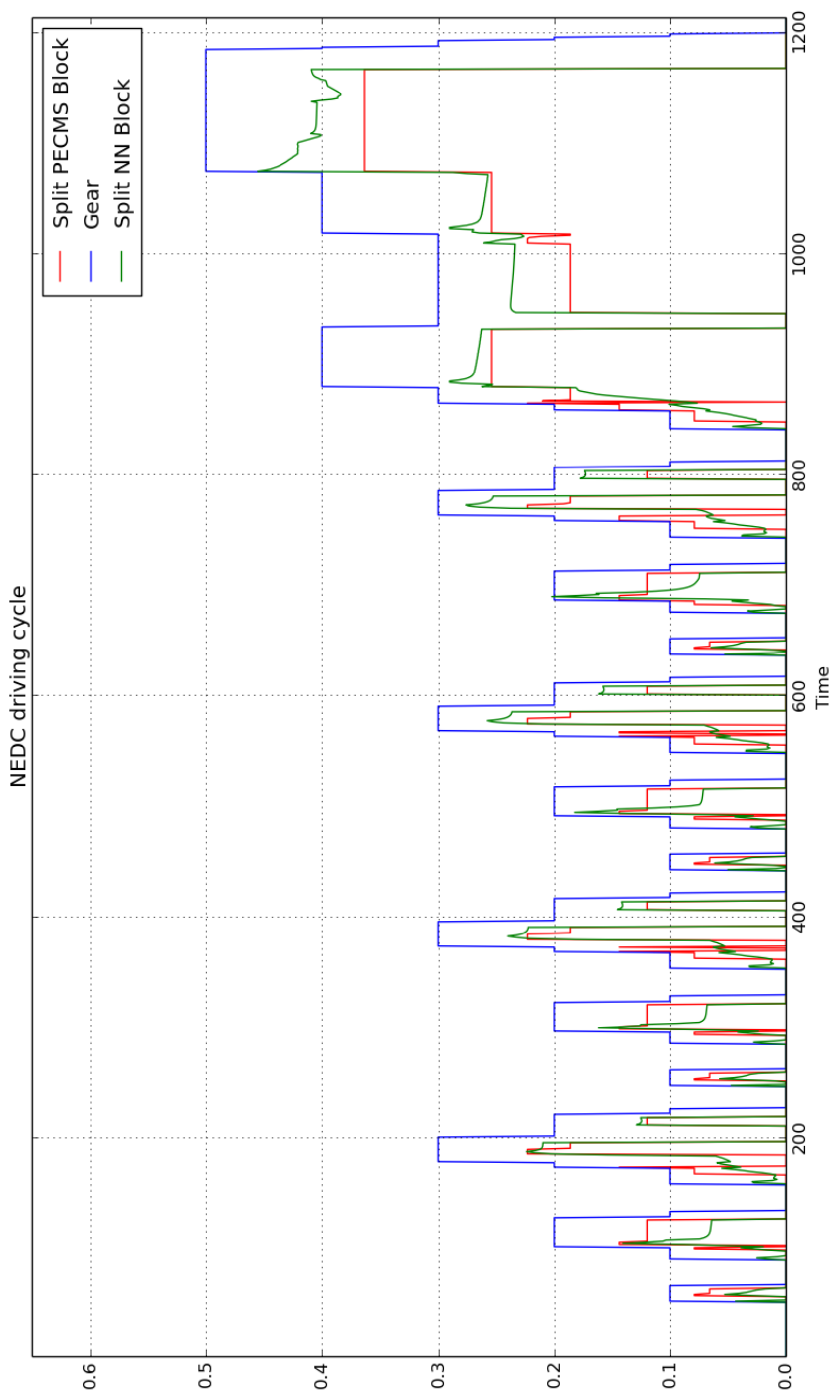

Figure A.3: Best result for FANN, used on the NEDC driving cycle

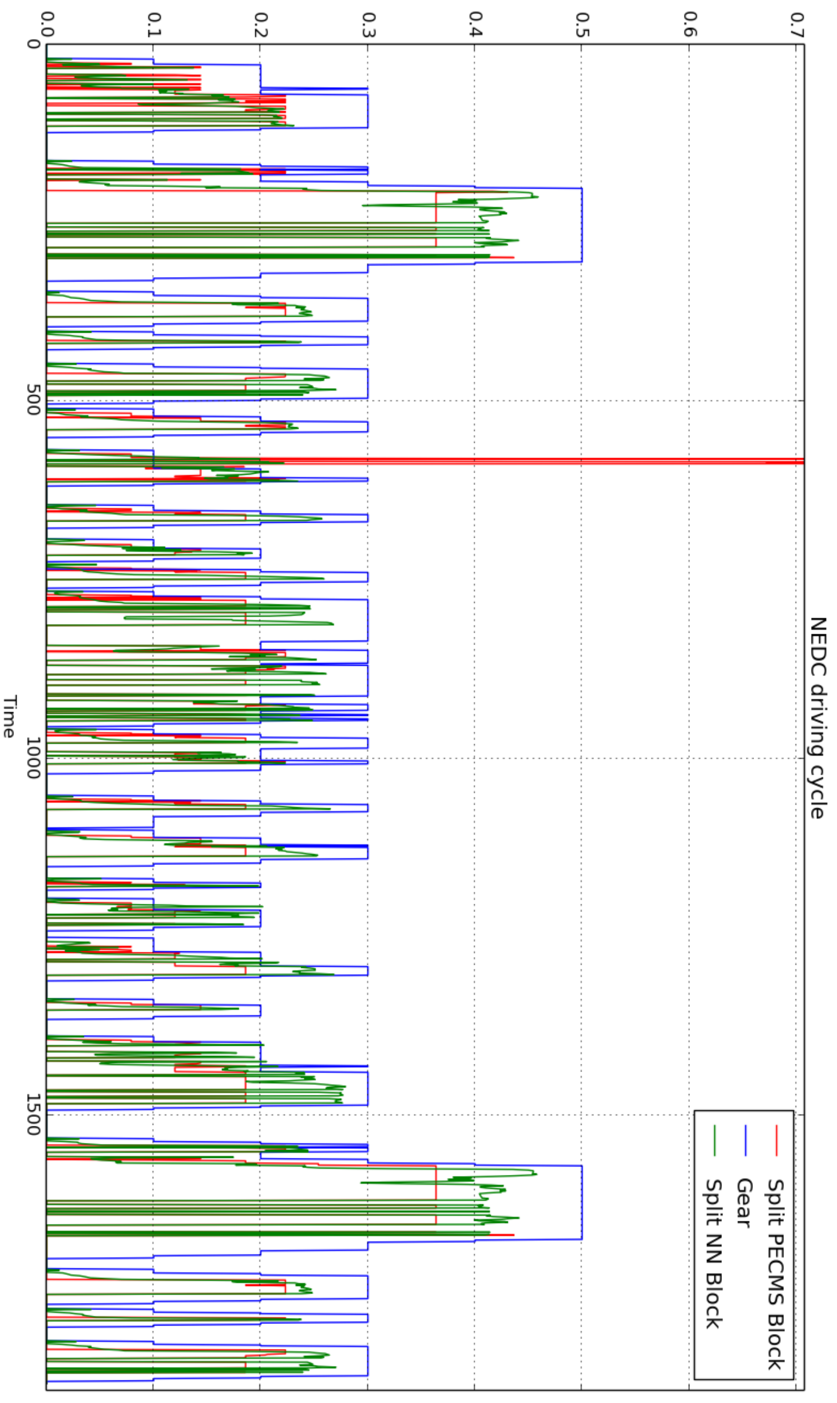

Figure A.4: Best result for FANN, used on the FTP_75 driving cycle

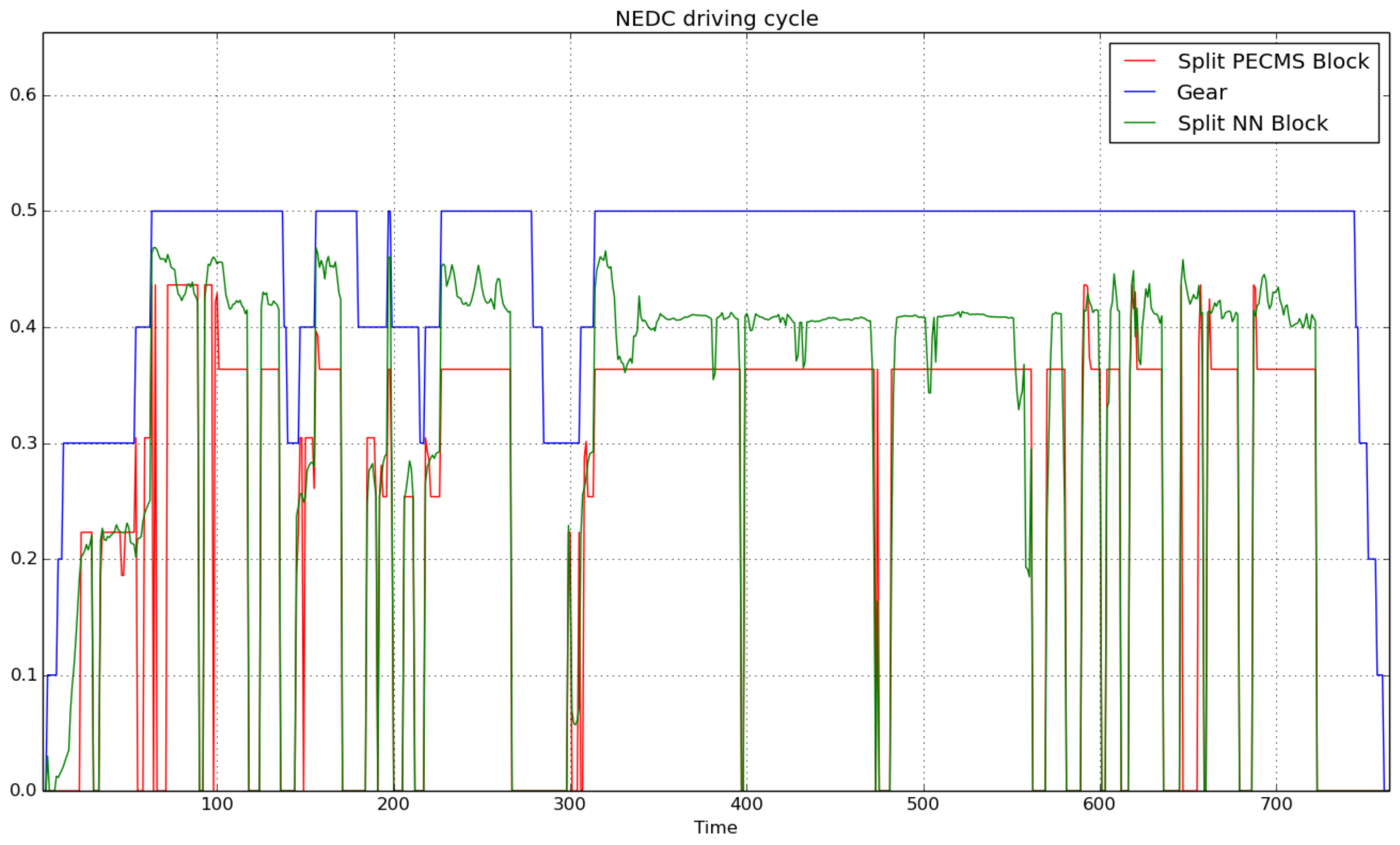

Figure A.5: Best result for FANN, used on the FTP_HIGHWAY driving cycle