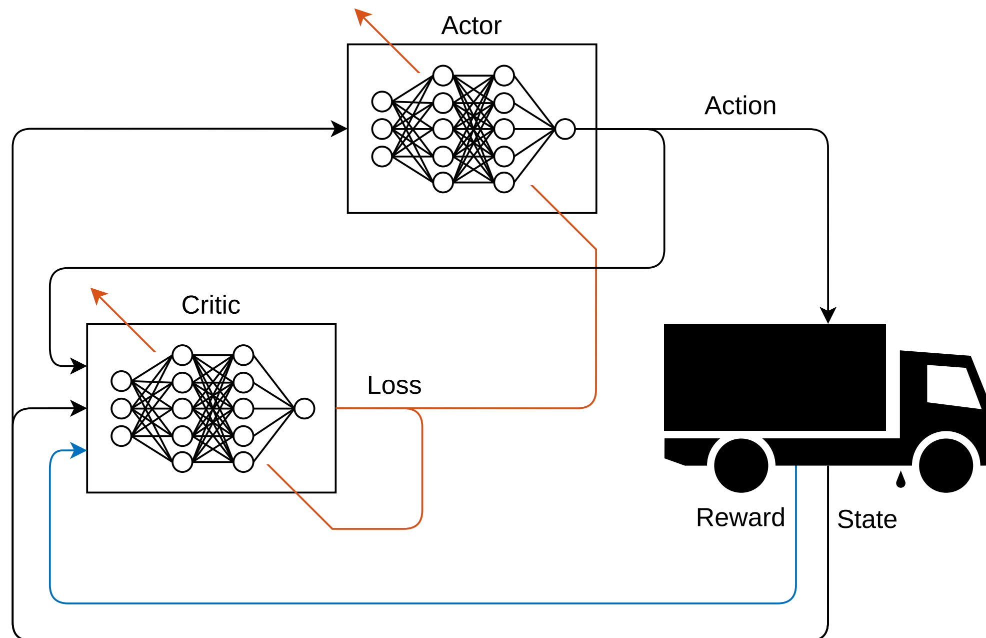
CHALMERS
UNIVERSITY OF TECHNOLOGY



Adaptive Model-Free Control Applied to Truck Front Wheel Drive

Real time control with reinforcement learning utilising recurrent deterministic policy gradient

Master's thesis in Systems, Control and Mechatronics

OSKAR JOHANSSON
BENJAMIN LUNDGREN

DEPARTMENT OF MECHANICS AND MARITIME SCIENCE

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021
www.chalmers.se

Master's thesis 2021:39

Adaptive Model-Free Control Applied to Truck Front Wheel Drive

Real time control with reinforcement learning utilising recurrent deterministic policy gradient

Oskar Johansson
Benjamin Lundgren

Department of Mechanics and Maritime Sciences
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology
Gothenburg, Sweden 2021

Adaptive Model-Free Control Applied to Truck Front Wheel Drive
Real time control with reinforcement learning utilising
recurrent deterministic policy gradient

Oskar Johansson
Benjamin Lundgren

© Oskar Johansson, Benjamin Lundgren, 2021.

Supervisor: Martin Karlsson, CPAC Systems AB
Supervisor: Marcus Broberg, CPAC Systems AB
Supervisor: Ola Göök, CPAC Systems AB
Examiner: Peter Forsberg, Mechanics and Maritime Sciences

Master's Thesis 2021:39
Department of Mechanics and Maritime Science
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Structure of a actor-critic learning method with self-optimising artificial neural networks.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2021

Adaptive Model-Free Control Applied to Truck Front Wheel Drive
Real time control with reinforcement learning utilising
recurrent deterministic policy gradient

Oskar Johansson
Benjamin Lundgren

Department of Mechanics and Maritime Science
Chalmers University of Technology

Abstract

This thesis investigates how reinforcement learning methods can be used to achieve adaptive and model-free control of a hydraulic front-wheel drive system. First, an existing sub-optimal controller is emulated by an artificial neural network using supervised learning. The continuous action- and state space reinforcement learning method "Recurrent Deterministic Policy Gradients" (RDPG) is then modified to work continuously in real time and implemented to improve the performance of the network and make it adaptive.
The emulating network performed similarly, albeit somewhat worse, compared to the original controller. Using RDPG with the emulating network against a simple model of the hydraulic system showed that the network adapted and further improved the performance from the sub-optimal starting point. However, applying the RDPG algorithm against a real system was infeasible with the selected hyper-parameters and would require further investigation for the algorithm to converge.
The conclusion is that using RDPG for adaptive model-free control can be feasible for non-linear dynamic system that exhibit slow, gradual changes and that first emulating a sub-optimal controller can enable learning on a system where learning from the beginning is not desired or possible.

Keywords: Reinforcement Learning, Machine Learning, Transfer Learning, Recurrent Deterministic Policy Gradients (RDPG), Adaptive control, Model-Free Control.

// Acknowledgements

We would like to thank *CPAC Systems AB* for providing resources, equipment and ideas that made it possible to write this thesis. We would like to thank our supervisors at *CPAC*, Marcus Broberg, Martin Karlsson and Ola Göök for their time and their help with the implementation in the truck and the test runs. We would also like to thank our examiner Peter Forsberg for sharing his insights in the field of machine learning and constructive discussions regarding this thesis.

Oskar Johansson and Benjamin Lundgren, Gothenburg, June 2021

Thesis advisors: Marcus Broberg, Martin Karlsson and Ola Göök, CPAC Systems AB
Thesis examiner: Peter Forsberg, Mechanics and Maritime Sciences

Contents

List of Figures **xi**

List of Tables **xiii**

Acronyms **xv**

1 Introduction **1**
1.1 Background 2
1.2 Purpose 3
1.2.1 Goals 3
1.2.2 Limitations 3

2 Theory **5**
2.1 Artificial Neural Network 5
2.1.1 Activation functions 7
2.1.2 Normalisation of input data 8
2.1.3 Training, gradient descent and backpropagation 8
2.1.4 Supervised Learning 10
2.2 Reinforcement Learning 11
2.2.1 Markov Decision Processes 11
2.2.2 Q-Learning 12
2.2.3 Deep Q-Learning 13
2.2.4 Deep Deterministic Policy Gradient 13
2.2.5 Recurrent Deterministic Policy Gradients 15

3 Methods **17**
3.1 Emulating the original controller 17
3.2 Reinforcement Learning 18
3.2.1 Continuous, real time RDPG 20
3.2.2 Transfer learning 22
3.3 Simulation 22
3.3.1 Tests against rudimentary model 23
3.4 Tests against truck 24
3.4.1 Drive cycle 24

4 Results **25**
4.1 Emulating the original controller 25

4.1.1 Driving straight forward 25
4.1.2 Driving forward while turning 26
4.1.3 Shifting gear 29
4.2 Reinforcement learning 31
4.2.1 Tests against model 31
4.2.1.1 Critic learning rate 33
4.2.1.2 Pre-training critic 33
4.2.2 Tests against the truck 33

5 Discussion **35**
5.1 Emulating a controller with ANN 35
5.2 Reinforcement learning against the model 36
5.3 Reinforcement learning against the truck 37
5.4 Implementational challenges 38

6 Conclusion **39**

Bibliography **41**

List of Figures

1.1 Schematic overview of the hydraulic system. 2

2.1 Structure of a small ANN with three inputs, two hidden layers and two outputs. 5
2.2 Composition of a single FNN neuron. 6
2.3 Composition of a single RNN neuron. 6
2.4 Composition of a single LSTM neuron. 7
2.5 The *ReLU*, *tanh* and *sigmoid* activation functions. 8
2.6 Simple neural network with labeled connection weights. 10
2.7 Unfolding of RNN cell from left with resulting unfolded structure on the right. 10
2.8 Illustration of how the reward is used to get a better approximation of $Q(s, a)$. The MSE between the two estimators over a batch is used to update the critic network with backpropagation. 14

3.1 Structure of the ANN used for control. The listed properties for each layer is layer type, layer size($\times$sequence length) and activation function. 18
3.2 Division of log data, with states s and actions a, into frames input vector x of length T and ground truth y of length 1. 19
3.3 Two parallel processes to run RDPG in real-time on a PC. 19
3.4 Comparison between model output, using sampled data as inputs, and the true system output from the same log. 23

4.1 ANN emulating the original controller from sampled data. 25
4.2 Actual and requested pressure for original controller and ANN driving forward in first gear. 26
4.3 Actual and requested pressure for original controller and ANN driving forward in second gear. 27
4.4 Actual and requested pressure for original controller and ANN driving forward in third gear. 27
4.5 Actual and requested pressure for original controller and ANN driving forward in fourth gear. 27
4.6 Actual and requested pressure for original controller and ANN driving forward and turning in first gear. 28
4.7 Actual and requested pressure for original controller and ANN driving forward and turning in second gear. 28

4.8 Actual and requested pressure for original controller and ANN driving forward and turning in third gear. 28
4.9 Actual and requested pressure for original controller and ANN driving forward and turning in fourth gear. 29
4.10 Actual and requested pressure for original controller and ANN during a gear shift from first to second gear. 30
4.11 Actual and requested pressure for original controller and ANN during a gear shift from second to third gear. 30
4.12 Actual and requested pressure for original controller and ANN during a gear shift from third to fourth gear. 30
4.13 Reward during training with RDPG against the system model. 31
4.14 Regulation performance of the closed-loop system after varying amounts of reinforcement learning against the system model. 32
4.15 Reward during training with varying critic learning rates against the system model. 33
4.16 Reward during training with varying amounts of critic pre-training against the system model. 34
4.17 Reward during training with varying critic learning rates against the truck. 34
4.18 Moving average over requested and actual pressure against the truck. 34

List of Tables

2.1 Representations of policies and expected reward in common reinforcement learning methods. DV is short-hand for discrete valued while CV is short-hand for continuous valued. 11

4.1 RMSE and RMSPE over 1600 time steps driving straight forward for the original controller and the ANN 26
4.2 RMSE and RMSPE over 1600 time steps driving forward and turning for the original controller and the ANN 27
4.3 RMSE and RMSPE over the first 150 time step after a gear shift for the original controller and the ANN. 29

Acronyms

ANN Artificial Neural Network. xi–xiii, 1, 3, 5, 7, 8, 10, 11, 13–15, 17–19, 22–30, 35–37, 39

BPTT Back-Propagation Through Time. 9, 16, 20, 21

CAN Controller Area Network. 24

DDPG Deep Deterministic Policy Gradient. 11, 13–15
DQN Deep Q-Learning. 11, 13, 15

ECU Electronic Control Unit. 18, 24

FNN Feedforward Neural Network. xi, 5, 6, 9, 15

LSTM Long Short-Term Memory. xi, 1, 5–7, 17, 35

MDP Markov Decision Process. 11–13, 15
MSBE Mean Square Bellman Error. 14
MSE Mean Squared Error. xi, 11, 14, 18, 25, 35

PC Personal Computer. xi, 3, 19, 24
PI Proportional-Integral. 2, 3
POMDP Partially Observable Markov Decision Process. 15

RDPG Recurrent Deterministic Policy Gradient. xi, xii, 11, 15–22, 24, 31, 33, 36–39
ReLU Rectified Linear Unit. 7
RMS Root Mean Square. 25
RMSE Root Mean Square Error. xiii, 25–27, 29, 35
RMSPE Root Mean Squared Percentage Error. xiii, 25–27, 29, 35
RNN Recurrent Neural Network. xi, 1, 3, 5, 6, 9, 10, 15, 20

1 Introduction

The process of solving control problems is traditionally divided in three parts: modelling of the plant system to be controlled, design of the control law or policy and lastly analysis of the interconnected controlled system, with regards to stability and regulation performance. While this method provides a clear approach, there are limitations as to what can be achieved. The controller design and analysis steps both hinge on the accuracy of the plant model. Furthermore, many of the controller design and most of the analysis tools become irrelevant if the plant has significant non-linearities [1]. While there are some methods that often give satisfactory results on non-linear systems such as backstepping control and non-linear model-predictive control [1], they are even more dependant on accurate modelling of the plant model.

In related fields, Artificial Neural Networks (ANNs) have been successfully deployed to solve a wide variety of non-linear and dynamic tasks. Recurrent Neural Networks (RNNs) incorporating e.g Long Short-Term Memory (LSTM) cells have been used for stock market predictions and natural language processing after training with supervised learning.

Deep learning ANNs have also been heavily utilised in various reinforcement learning methods [2–4], extending the limited domain that the traditional reinforcement learning methods such as Q-learning [5] can handle. These novel methods combines the exploration aspects from reinforcement learning with deep learning to train ANNs without any prior knowledge of the task that should be performed. Compared with traditional control methods, these can accomplish adaptive control of non-linear systems in a model-free setting.

Despite being model-free, these reinforcement learning methods are almost exclusively used to train ANN in virtual environments. This not only reduces the training time, it also removes the hassles concerning real world testing and potential damage caused to a physical system by a poorly trained ANN. However, a model of the system is still required to build the virtual environment.

This thesis examines methods to combine supervised learning based pre-training with reinforcement learning in a model-free setting with the aim of creating a controller without requiring a system model.

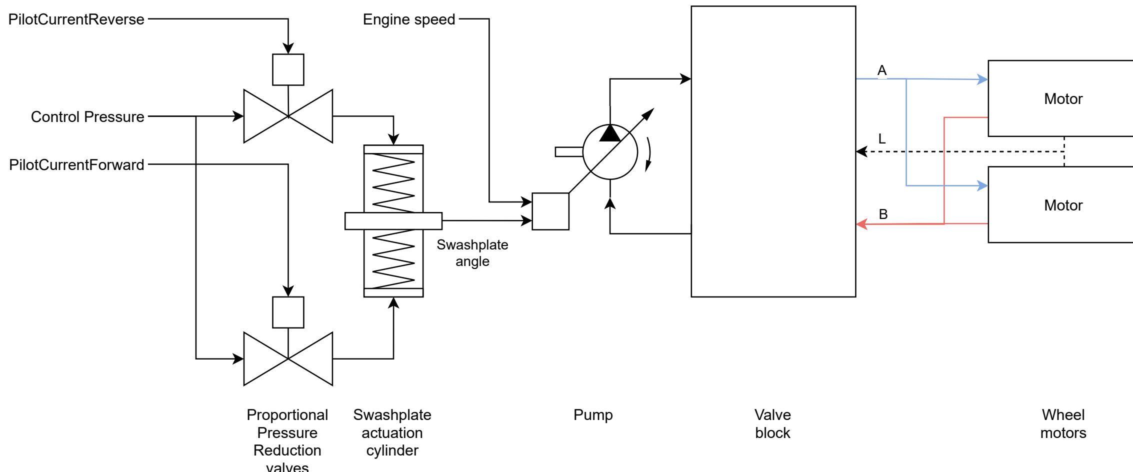


Figure 1.1: Schematic overview of the hydraulic system.

1.1 Background

To improve traction, especially in harsh conditions with heavy loads, trucks can be equipped with front wheel drive capabilities [6]. Examples of drive mechanisms include mechanical, hydraulic or electric. A mechanical solution is relatively simple and proven, but requires a lot of space and adds considerable weight [6]. Electrical solutions are typically efficient and provide flexible power transfer, though batteries and charging systems can be costly. The system under investigation for control in this thesis is a hydraulic front wheel drive. The hydraulic solution provides flexible power transfer with less weight and a smaller footprint than the other solutions. Its main drawback is that it only works at low speeds [6].

The hydraulic system consists of a variable flow hydraulic piston pump, mounted on the main engine power take-off. The volumetric flow of the pump depends on the engine speed, which is controlled by an external controller (truck driver), and the stroke length of the pump, which can be controlled by adjusting the angle of the swashplate, a disc in the pump that guides the stroke of the pistons. By tilting the swashplate in either direction, the flow can be continuously varied between forward, no flow and reverse. The swashplate is actuated by a double acting, hydraulic cylinder with spring return to the centre, no flow, position. The cylinder is in turn actuated by two electrically controlled hydraulic valves, with control signals *PilotCurrentForward* and *PilotCurrentReverse* as seen in fig. 1.1. Setting any of these signals will cause the swashplate to tilt and thus a torque will be produced by the wheel motors in either the forward or reverse direction when the engine is running. The existing current controller for the two hydraulic valves is a Proportional-Integral (PI) controller with static feed-forward.

Due to pretensioning of the springs in the hydraulic cylinder, there is a control signal threshold that must be overcome before the swash plate is affected. This is important to consider as small control signals will have no effect on the system, a phenomenon known as a deadband.

1.2 Purpose

This thesis will investigate using an ANN for model-free control. The objective is to replace a sub-optimal PI controller and instead utilise an adaptive ANN controller.

1.2.1 Goals

- Emulate an existing, dynamical controller with a recurrent neural network and supervised learning.
- Utilise reinforcement learning to improve the performance of the recurrent neural network against a physical system.

1.2.2 Limitations

The ANN will control a physical system, a hydraulic front wheel drive system in a truck, and must be able to execute in real-time on a Personal Computer (PC). Since the existing controller already has a determined, fixed cycle rate of 125 Hz, the same will be used for the ANN-based controller.

The physical system is dynamic and all states are not measured. RNN is a type of artificial neural network with the ability to store and remember information between time steps, making it beneficial to use in a dynamic process. Due to this ability the ANN will only consist of RNN layers together with dense layers. Further options will not be investigated in this thesis.

The hydraulic system has two controllable valves, one for forward and one for reverse and braking. This thesis will only focus on control of the forward valve. There are two main reasons for this. Firstly, it can be argued that controlling one or the other are equivalent tasks. Secondly, the existing controller that provides a baseline and starting point for the work in this thesis is a work-in-progress and only the forward valve control is fully implemented.

2
Theory

This section serves to introduce the key topics used throughout the report. First, an introduction of ANNs will be given, with focus on the structure of the networks, different activation functions and the function of neurons in Feedforward Neural Networks (FNNs), RNNs and LSTMs. Then the topic of reinforcement learning is introduced and several methods which extend each other are described, leading up to the method used in this thesis.

2.1 Artificial Neural Network

In general, ANNs are a type of function approximators that are loosely based on biological neural networks [7]. They are composed of interconnected neurons that take one or more values as inputs and give a single output. The neurons are typically grouped in layers, where the outputs of one layer is used as the input for the next, as can be seen in fig. 2.1. The number of neurons in the first and last layers, known as the input and output layers, determines the number of inputs and outputs of the network. The number and shape of the layers between these, which are known as hidden layers, on the other hand determines the complexity of the neural network and how well it can be trained to approximate the desired function or behaviour.

In each neuron, the input vector is reduced to a scalar value by calculating a weighted sum $a = Wx + b$, where x is the input vector, W the weights and b a bias. In a traditional FNN, this weighted sum is passed to the output of the neuron, either directly $y = a$, or through a non-linear function known as an activation function,

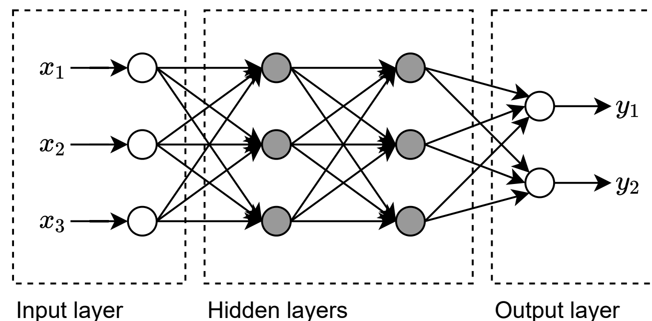


Figure 2.1: Structure of a small ANN with three inputs, two hidden layers and two outputs.

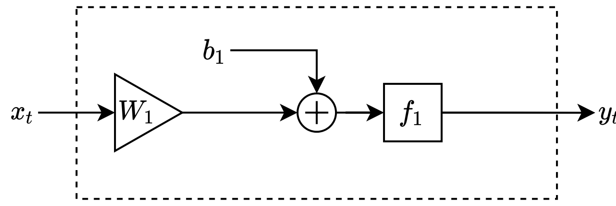


Figure 2.2: Composition of a single FNN neuron.

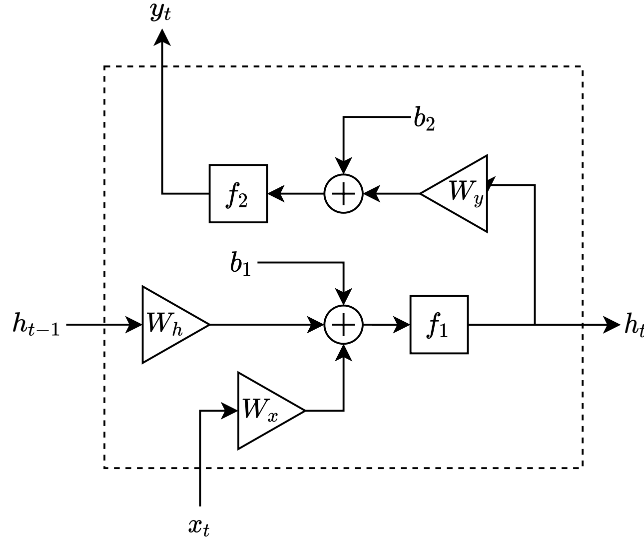


Figure 2.3: Composition of a single RNN neuron.

$y = f(a)$. The composition of a single neuron can be seen in fig. 2.2.

In a RNN [8], the neuron is given memory by introducing a hidden state that retains its value between time steps. For the simplest type of RNN, the value of this hidden state is updated as $h_t = f_1(W_x x_t + W_h h_{t-1} + b_1)$, where h_t is the hidden state, W_x and W_h are the weighting factors for the inputs and hidden state and f_1 is the activation function. The hidden state is then weighted and biased and passed through another activation function, yielding the output $y_t = f_2(W_y h_t + b_2)$. The composition of a RNN neuron can be seen in fig. 2.3.

RNNs suffers from difficulty of expressing long time dependencies and data further back in time become less and less significant. LSTMs [9] solves this by introducing a memory cell that can hold a value for an arbitrarily long time. In addition the memory cell c, LSTM uses a forget gate f_t, input gate i_t and output gate o_t to control the memory and the output. The composition of a LSTM neuron can be seen in fig. 2.4.

The forget gate is controlled by $f_t \in (0, 1)$ and decides if the memory cell state c_{t-1} should be forgotten. c_{t-1} is multiplied with f_t, $c'_{t-1} = c_{t-1} f_t$, where the forget gate is $f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$. If f_t is close to zero the memory c_{t-1} will be forgotten and if f_t is close to one the memory wont be affected.

The input gate is controlled by $i_t \in (0, 1)$ and decides if the new input $\tilde{c}_t$ should be

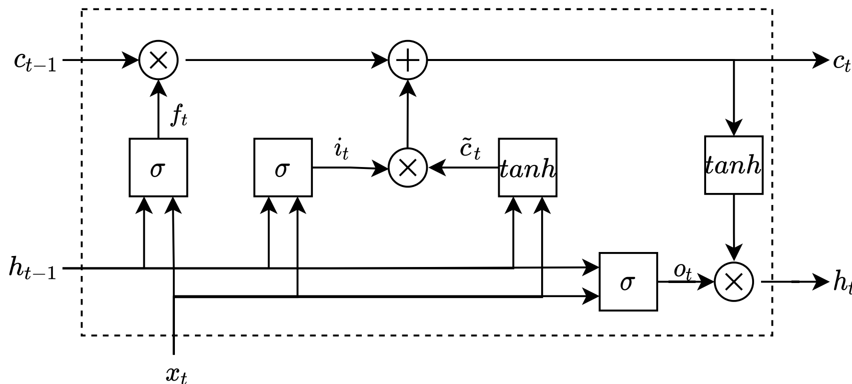


Figure 2.4: Composition of a single LSTM neuron.

remembered by the memory cell $\tilde{c}'_t = i_t \tilde{c}_t$. Where $i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$ and $\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$. The new input is added to the memory cell which become $c_t = c'_{t-1} + \tilde{c}'_t$.

The output gate is controlled by $o_t \in (0, 1)$ and decides if the memory cell c_t should be the output $h_t = o_t \tanh c_t$, where $o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$. The new memory c_t and output h_t is then passed to the next LSTM neuron.

2.1.1 Activation functions

An activation function f is often applied to the output from an ANN neuron. Without this the neuron will only output a linear combination of the input. Adding successive linear layers is then meaningless as the whole network will still be linear and could thus be condensed into a single, linear, layer. By applying a non-linear activation function, the output y becomes non-linear with regards to the input and hence the ANN also becomes non-linear. There exist many types of activation functions for ANNs, where some common are Rectified Linear Unit (ReLU), *tanh* and *sigmoid*. ReLU is rather simple and gives a partially linear output where all negative values are clamped to zero, hence $y = f(a) = \max(0, a)$. The output is then left-bounded by $y \in [0, \infty)$ and can be seen in fig. 2.5. The second activation function *tanh*, shown in (2.1), is the hyperbolic function of *tan* where the output is bounded by $y \in (-1, 1)$ and can be seen in fig. 2.5.

$$y = f(a) = tanh(a) = \frac{e^a - e^{-a}}{e^{-a} + e^a} \tag{2.1}$$

Another common activation function is the *sigmoid* function, shown in (2.2), which is similar to *tanh* but is bounded by $y \in (0, 1)$ and can be seen in fig. 2.5. The boundary is between zero and one is therefore useful for binary operations.

$$y = f(a) = \sigma(a) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

The activation function can also be fully linear, corresponding to not have an activation function. The output is no longer bounded in any direction and can obtain any real value $y \in (-\infty, \infty)$.

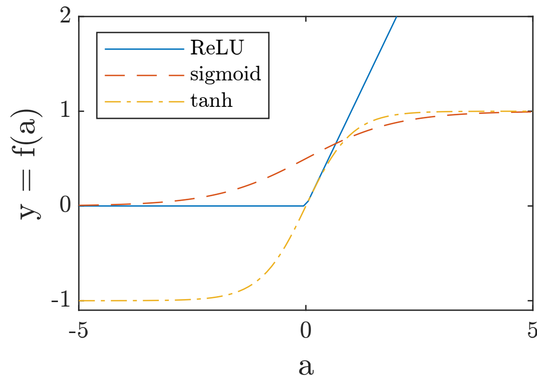


Figure 2.5: The *ReLU*, *tanh* and *sigmoid* activation functions.

2.1.2 Normalisation of input data

If the input is unbounded $x \in (-\infty, \infty)$ while training, the loss gradients could become arbitrarily large during backpropagation for some inputs and vanish for others. Different methods of normalising the data are available and should be used to increase the stability and performance of the training [10]. There are many types of normalisation methods which often aim to scale the input between some predefined values (often between zero and one). One problem with normalising data from continuous time series from a real system is that the signal bounds may not be known when the network is trained. Another approach which is more statistical and does not consider the bounds is to scale and bias the training data to achieve zero mean and variance of one.

2.1.3 Training, gradient descent and backpropagation

In order for a neural network to approximate the desired function it must be configured to do so. For a given network structure, this amounts to selecting the weights and biases that yields the desired input-output relationship.

The most common method for finding these weights leverages gradient descent to adjust the weights and biases so that a loss function is minimised. The weights and biases of the network are first joined in a parameter vector, $\theta = \begin{bmatrix} W_1 & \dots & W_n & b_1 & \dots & b_k \end{bmatrix}$. Then a loss function J is formulated to measure the network performance. This function should be continuous and differentiable in θ, and decrease as the desired performance increases. Lastly, the parameters θ are iteratively updated according to equation (2.3). The update consists of taking steps in the parameters θ along the negative direction of the gradient of the loss function J, which is the direction of greatest decrease.

$$\theta^{+} = \theta - \alpha \nabla_{\theta} J \tag{2.3}$$

The size of the update is determined not only by the gradient, but also by a learning rate factor, α. In general, higher learning rates tend to increase the rate of convergence, but decreases the stability of the training and may cause the solution to diverge. However if the learning rate is too low, the ANN may only learn the training data and not to generalise, thus overfitting the training data [11].

In networks with many layers, calculating the partial derivative of the loss with respect to a weight in an early layer involves applying the chain rule for every successive layer and activation function. The computational complexity thus increases rapidly with the number of layers. The backpropagation algorithm significantly reduces the number of computations by calculating weights from the back of the network first and storing intermediate values that can be reused in calculations as the algorithm works its way through the layers. In this section, backpropagation refers to the efficient algorithm for calculating the gradient of neural networks. However, it should be noted that backpropagation may also refer to the entire concept of using gradient descent with backpropagation to train a network.

Consider the simple neural network in fig. 2.6 for an example of backpropagation. For the first layer, starting from the back, the loss gradient is calculated with respect to the last neuron update

$$a = \frac{\partial J}{\partial y_3}$$

With the chain rule and the previously stored result the gradients with respect to the weights in the last layer are calculated

$$\frac{\partial J}{\partial W_{3,1}} = \frac{\partial J}{\partial y_3}\frac{\partial y_3}{\partial W_{3,1}} = a\frac{\partial y_3}{\partial W_{3,1}} \quad , \quad \frac{\partial J}{\partial W_{3,2}} = \frac{\partial J}{\partial y_3}\frac{\partial y_3}{\partial W_{3,2}} = a\frac{\partial y_3}{\partial W_{3,2}}$$

Similarly for the next layer, the partial derivatives of the loss with respect to the neuron outputs are calculated. The intermediate result a from the previous step is used to reduce the complexity of the calculations

$$b_1 = \frac{\partial J}{\partial y_{2,1}} = a\frac{\partial y_3}{\partial y_{2,1}} \quad , \quad b_2 = \frac{\partial J}{\partial y_{2,2}} = a\frac{\partial y_3}{\partial y_{2,2}}$$

Lastly, the loss gradient with respect to the weights in the new layer are calculated. Due to the saved values from the intermediate steps, this calculation has the same complexity as for the first layer

$$\frac{\partial J}{\partial W_{2,1,1}} = b_1\frac{\partial y_3}{\partial W_{2,1,1}} \quad , \quad \frac{\partial J}{\partial W_{2,2,1}} = b_2\frac{\partial y_3}{\partial W_{2,2,1}}$$

$$\frac{\partial J}{\partial W_{2,1,2}} = b_1\frac{\partial y_3}{\partial W_{2,1,2}} \quad , \quad \frac{\partial J}{\partial W_{2,2,2}} = b_2\frac{\partial y_3}{\partial W_{2,2,2}}$$

Back-Propagation Through Time (BPTT) extends the concept of gradient descent with backpropagation to RNNs. This is achieved through a process called unfolding. In short, the recurrent input from an earlier time step is treated as if it was a regular input from a previous layer in the network. As illustrated in fig. 2.7, this process effectively removes the time dependency in an RNN cell by transforming the network into an equivalent FNN.

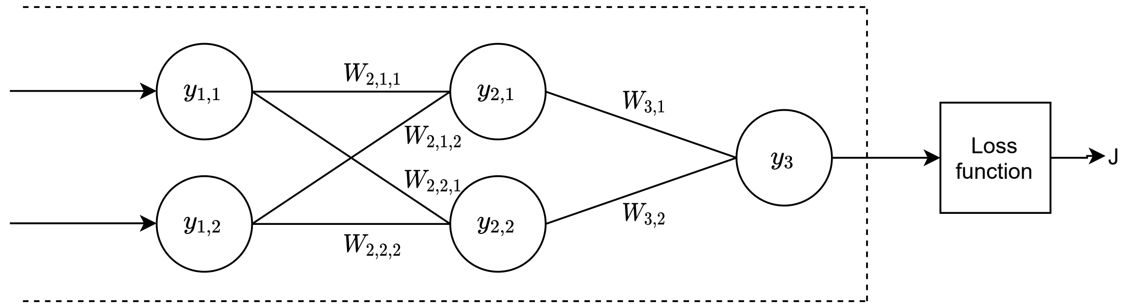


Figure 2.6: Simple neural network with labeled connection weights.

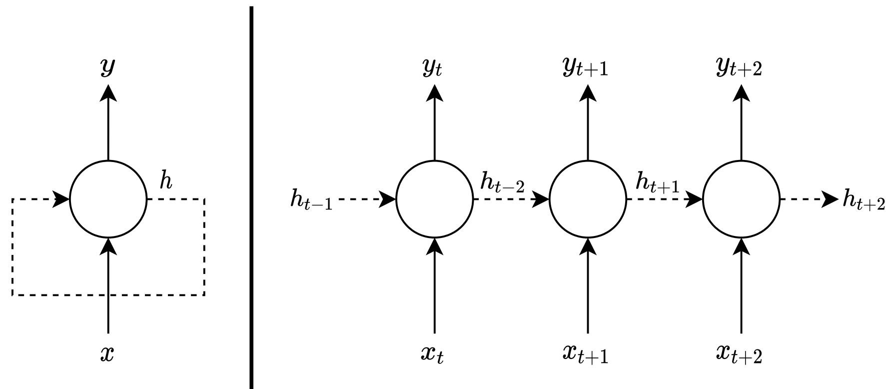


Figure 2.7: Unfolding of RNN cell from left with resulting unfolded structure on the right.

2.1.4 Supervised Learning

Supervised learning is a type of machine learning where a set of inputs and a set of corresponding desired output data is used to train a model. The training aims at extracting patterns from the training data that enables the model to use new input data to make predictions of the output.

The goal is that the network should generalise the patterns observed from the training data so that it is able to handle other, similar, data. This means that while the training is performed by optimising the ANN against a set of training samples, the performance is measured against a set of new samples, often called the validation set. Improving the performance against the training set does not guarantee improved performance against the validation set [12]. This can happen when the ANN is not learning the desired rule for making predictions but rather learning coincidental patterns caused by noise in the training data. While this may cause near perfect prediction fit against the training set it can perform inadequately on new data, a phenomenon known as overfitting [12]. There are many methods to reduce the risk of overfitting, including early stopping of the training process, extending the training set and training strategies such as dropout, where only a randomly selected subset of each layer is considered for each training batch, or weight regularisation which encourages small weights and therefore tends to achieve simpler models [13].

To apply gradient descent in supervised learning, a loss function is formulated where

the output from the network is compared to the desired output from the training data. For classification problems where the prediction of the network is either correct or incorrect it is common to use a method called categorical cross-entropy. However, for regression problems, where the prediction instead should be as close as possible to some numeric target, the Mean Squared Error (MSE) is often used. For loss J, prediction y' and desired output y, this is formulated as

$$J(y, y') = \frac{1}{N} \sum_{i}^{N} (y_i - y'_i)^2 \tag{2.4}$$

for a training batch with N examples consisting of features or inputs to the network $x_i,\ i = 1, 2 \ldots N$ which are used to find the prediction $y'_1,\ i = 1, 2 \ldots N$, and corresponding labels, or outputs $y_i,\ i = 1, 2 \ldots N$.

2.2 Reinforcement Learning

Reinforcement learning is a field in machine learning that aims to find how intelligent agents should behave in order to maximise the sum of immediate and future rewards that they receive [14].

Reinforcement learning has received much attention in the last few years, due to its ability to train an agent without having a model of the environment [15]. Instead the agent learns by exploring the environment and behaviour that leads to the desired results are encouraged by positive or negative feedback.

This section will cover some key theory in reinforcement learning and a few different methods together with their use cases and limits. To guide the reader through these methods, they are presented in order of complexity with each new method building on top of the previous ones. The methods are Q-learning, Deep Q-Learning (DQN) and Deep Deterministic Policy Gradient (DDPG)/Recurrent Deterministic Policy Gradient (RDPG). A comparison of key differences can be seen in tab. 2.1.

Table 2.1: Representations of policies and expected reward in common reinforcement learning methods. DV is short-hand for discrete valued while CV is short-hand for continuous valued.

Method	Expected reward	Policy	State- & Action-space
Q-Learning	Table Q(s,a)	$\arg\max_a Q(s, a)$	DV state, DV action
DQN	ANN Q(s,a)	$\arg\max_a Q(s, a)$	CV state, DV action
DDPG / RDPG	ANN Q(s,a)	ANN $\mu(s)$	CV state, CV action

2.2.1 Markov Decision Processes

Reinforcement learning algorithms are often based on Markov Decision Processes (MDPs), where the environment is modelled with states, actions and rewards. At every time step and non-terminal state s_t, an action a_t is taken. The action causes the environment to move to a new state s_{t+1} in a stochastic process $p(s_{t+1}|s_t, a_t)$

and to give a reward $r_t(s_t, s_{t+1})$ that describes how good that particular transition was. For a process to be described as a MDP it must have the Markov Property, meaning that given the current state, no previous states are needed to describe the next state, thus $P(s_{t+1}|s_t) = P(s_{t+1}|s_0, \ldots, s_{t-1}, s_t)$.

The main objective in reinforcement learning is to enable an agent to learn a policy $\pi(s)$ that at any state chooses the action that maximises the cumulative reward $r_t + \sum_{i=1}^{\infty} \gamma^i r_{t+i}$, which is a sum of the immediate reward and future rewards discounted by the factor $0 \leq \gamma < 1$. The discount factor sets a balance between whether longsighted and shortsighted behaviour is desired.

2.2.2 Q-Learning

Perhaps the most common reinforcement learning algorithm is Q-learning [5]. It works with discrete state and action spaces by creating and updating a table of Q-values, with states on one axis and action on the other. Every Q-value corresponds to a state-action pair and describes the expected cumulative reward resulting from choosing that action at that state. The table is updated incrementally by observing the rewards received after every transition.

The key idea behind Q-learning is closely related to the Bellman equation (2.5), which expresses the optimal expected reward $Q(s_t, a_t)$ recursively, as a sum of the immediate reward r and the future expected discounted cumulative reward $\gamma Q(s_{t+1}, a_{t+1})$ when following the policy that maximises the Q-value.

$$Q(s_t, a_t) = r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \tag{2.5}$$

This formulation also forms the basis of the Q-table update in equation (2.6). After each step, a new Q-value for the previous state and taken action is calculated with equation (2.5) from the received reward and the currently expected future reward. The Q-value is then update with a soft update, meaning that it is an average between the old and new value, weighted with a learning factor $0 \leq \alpha < 1$.

$$Q^+(s, a) = (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})) \tag{2.6}$$

While this implies that a Q-learning agent will start out with a bad policy, it will improve over time, and is even guaranteed to converge to the optimal policy as Watkins and Dayan have shown [5].

Though Q-learning can be used with continuous action or state spaces if they are discretised, the size of the Q-table grows fast with the resolution of the discretization. This does not only increases memory usage but also the time required to visit every state-action pair enough times to learn the Q-table. Another drawback with discretising the state and action space is that any underlying relations are lost, e.g that similar control signals typically will produce similar results. Q-Learning is also affected by the curse of dimensionality, meaning that the number of discrete states increases exponentially with the number of state dimensions.

2.2.3 Deep Q-Learning

Another reinforcement learning algorithm that is more suitable for continuous state space problems is DQN [2], where the Q-table is replaced by a neural network that estimates the Q-value of a given state-action pair. Using a function approximator to determine the value of a state-action pair allows the algorithm to determine the value of states that have not yet been visited, as long as similar, or nearby states have been. This inherent interpolation allows for learning in continuous, high dimensional state spaces. However, as the algorithm requires iterating over all possible actions to find the action that maximises the reward, it is not well suited for problems with continuous or high dimensional action spaces.

2.2.4 Deep Deterministic Policy Gradient

As mentioned previously, Q-learning can only handle discrete state and action spaces. DQN extends this domain by being able to handle continuous state spaces. For a system where both the action and state space are continuous, neither Q-learning nor Deep Q-learning are suitable [3]. An alternative approach is the DDPG algorithm developed by Google Deepmind [3] specifically for continuous state and action spaces. Similarly to Q-learning and DQN, DDPG is formulated around a MDP.

At its core DDPG works with two ANNs, called the actor and critic networks which work together to learn a task. The two networks have separate, yet clear tasks to perform. The actor network μ^θ expresses the current (control) policy $\mu^\theta(s)$, thus directly controlling the agent. It takes the current state as input and gives an action as output, essentially approximating the Q-value maximisation in Q-learning and DQN with an ANN. The critic network Q^ω, on the other hand, expresses the Q-values for the continuous state-action-space. In comparison with other reinforcement learning methods, the critic serves a similar purpose as the ANN in DQN and the Q-table in Q-learning.

To some extent DDPG can be seen as a further extension of the DQN method, as the critic network performs the same task as the ANN in DQN, but the policy $\arg\max_a Q(s, a)$ from Q-learning and DQN has been approximated by a neural network. The two main benefits of this are that the action space does not have to be discretised and that the policy evaluation does not grow in complexity with the dimension of the action space.

In DDPG the agent learns by iteratively updating the weights θ and ω of the actor and critic networks respectively. As in Q-learning and DQN, the objective is to maximise the expected cumulative reward. Once again the recursive expression of the cumulative reward given by the Bellman equation in equation (2.5) is used. A loss function for the critic network is derived from the Bellman equation and is presented in equation (2.7). It is clear that when critic network is optimal, and thus fulfils the Bellman equation (2.5), the loss will also be zero. The loss is thus a metric of how far from optimal the critic is for a given state-action combination.

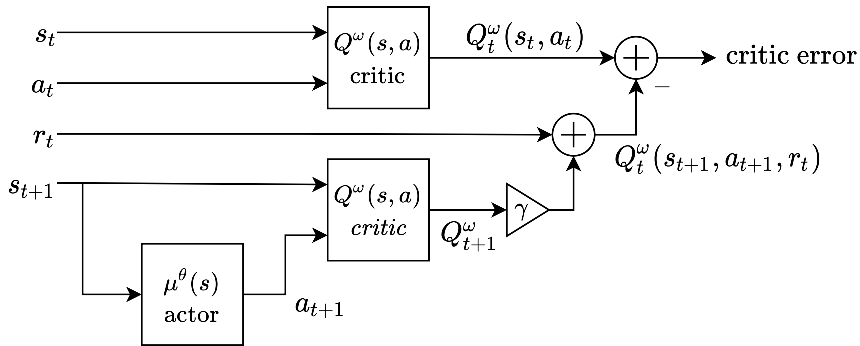


Figure 2.8: Illustration of how the reward is used to get a better approximation of $Q(s,a)$. The MSE between the two estimators over a batch is used to update the critic network with backpropagation.

$$
\begin{aligned}
L(\omega) &= \left[Q^\omega(s,a) - \left(r + \gamma \max_{a_{t+1}} Q^\omega\left(s_{t+1}, a_{t+1}\right)\right)\right]^2 \\
&\approx \left[Q^\omega(s,a) - \left(r + \gamma Q^\omega\left(s_{t+1}, \mu^\theta\left(s_{t+1}\right)\right)\right)\right]^2
\end{aligned}
\tag{2.7}
$$

This loss formulation is known as a temporal difference error, as it is the square error between two cumulative reward estimations from successive time steps. The loss can also be motivated by the fact that the second estimation is made after-the-fact and is based on the received reward, as illustrated in fig. 2.8. It is therefore generally a better estimation than the one made without information about the reward.

Using this loss to update the critic network directly puts the learning at risk to diverge [3] as the network is only updated with regards to the latest experience. To improve stability, DDPG uses a replay buffer $\mathcal{R}$ to store a certain number of past experiences (s, a, r, s_{t+1}, d). During the learning step, this buffer is sampled randomly in batches of many experiences over which the loss is averaged, forming the Mean Square Bellman Error (MSBE) in equation (2.8). This loss is used to update the weights of the critic network through backpropagation.

$$
L(\omega, \mathcal{R}) = \underset{(s,a,r,s_{t+1},d)\sim\mathcal{R}}{\mathrm{E}} \left[\left(Q^\omega(s,a) - \left(r + \gamma \max_{a_{t+1}} Q^\omega\left(s_{t+1}, a_{t+1}\right)\right)\right)^2\right] \tag{2.8}
$$

As the loss is calculated by the same ANN that will be updated from the loss, it is likely to diverge for numerical reasons [3]. The solution in DDPG is to introduce target networks for both actor and critic. The target networks, $\mu^{\theta'}$ and $Q^{\omega'}$ are copies of the actor and critic networks but the weights are updated more slowly with a soft update, hence $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$ and $\omega' \leftarrow \tau\omega + (1-\tau)\omega'$ with $\tau \ll 1$. This greatly improves the stability, thus may slow the convergence rate [3].

The update step for the actor is simple compared to that of the critic network. Using the same batch as for the critic update, the actor predicts actions to take based on the starting state in each experience. These state-action pairs are then given a Q-value by the critic network which is used as negative loss with backpropagation to update the actor weights.

There are several parameters and functions that may be changed in DDPG in order to affect the learning process and the behaviour of the trained agent. Perhaps the most important is the reward function, $r(s, a)$ which returns a scalar value to the agent after every time step based on how good the outcome was. The reward can be arbitrarily calculated and is up to the designer of the reinforcement learning setup to choose. For control problems a suitable reward could be based on a strictly negative following error, e.g $-e^2$. As the objective of the algorithm is to maximise the reward, it will thus strive to minimise the error.

In order for the agent to explore and learn new strategies, noise is added to the actions given by the actor. This noise allows for the otherwise deterministic ANN to try different actions in the same situation. The outcome is then observed by the critic which should guide the actor towards the better strategy. By changing the amplitude of this noise, a balance is achieved between exploration (high noise) and exploitation (low noise).

In addition to the reward and exploration noise there are a number of hyperparameters which may be used to tune the learning process. These are the update factor τ, the learning rates α and β for the ANNs and the size of the replay buffer $\mathcal{R}$. In contrast to the reward which affects the behaviour of a successfully trained agent, these mostly affect the stability and rate of convergence of the actual training process.

2.2.5 Recurrent Deterministic Policy Gradients

Many systems do not fulfil the observability condition of the MDP. A simple example of this may be a mechanical system where only position is measured but not the velocity. These system can instead be modelled as Partially Observable Markov Decision Processes (POMDPs). As Q-Learning, DQN and DDPG are all formulated around a MDP, they are not valid reinforcement learning methods for systems that are only POMDP. The RDPG algorithm which is an adaptation of DDPG was developed at Google Deepmind [4] to work specifically with POMDPs. The most concrete difference is that RDPG uses RNNs rather than FNNs for both actor and critic networks. The RDPG algorithm can be seen in algorithm 1.

This is relevant as not all states are measured in the truck which can therefore not be modelled as a MDP and controlled using DDPG. Instead, it is modelled as a POMDP and controlled using RDPG.

Algorithm 1: RDPG Algorithm by Heess et. al [4]

Initialise actor network $\mu^{\theta}(h_t)$ with weights θ
Initialise target actor network $\mu^{\theta'}$ with weights $\theta' \leftarrow \theta$
Initialise critic network $Q^{\omega}(h_t, a_t)$ with weights ω
Initialise target critic network $Q^{\omega'}$ with weights $\omega' \leftarrow \omega$
Initialise empty replay buffer R
for *M episodes* **do**
  Initialise empty history h_0
  for *T transitions* **do**
    Observe current state o_t
    Calculate the reward r_t
    Append h_{t-1} with a_{t-1}, o_t and r_t
    $h_t \leftarrow (h_{t-1}, a_{t-1}, o_t, r_t)$
    Predict new action $a_t = \mu^{\theta}(o_t) + \epsilon$, ϵ: exploration noise
    Execute a_t to the environment
  end
  Store all transitions $(o_1, a_1, r_1, \ldots, o_T, a_T, r_T)$ in R
  Sample a mini-batch of N episodes $(o_1^i, a_1^i, \ldots, a_T^i, r_T^i)_{i=1,\ldots,N}$ from R
  Construct histories $h_t^i = (o_1^i, a_1^i, \ldots, a_{t-1}^i, o_t^i)$
  Compute target action from target actor $a_{t+1}^{i'} = \mu^{\theta'}(h_{t+1}^i)$
  Compute target value using target critic

$$y_t^i = r_t^i + \gamma Q^{\omega'}(h_{t+1}^i, a_{t+1}^{i'})$$

  Compute critic update using BPTT

$$\Delta\omega = \frac{1}{NT} \sum_i \sum_t \left(y_t^i - Q^{\omega}(h_t^i, a_t^i) \right) \frac{\delta Q^{\omega}(h_t^i, a_t^i)}{\delta \omega}$$

  Compute actor update using BPTT

$$\Delta\theta = \frac{1}{NT} \sum_i \sum_t \frac{\delta Q^{\omega}\left(h_t^i, a_t^i = \mu^{\theta}(h_t^i)\right)}{\delta a} \frac{\delta \mu^{\theta}(h_t^i)}{\delta \theta}$$

  Update actor and critic weights using Adam
  Update the target networks weights

$$\omega' \leftarrow \tau\omega + (1-\tau)\omega'$$
$$\theta' \leftarrow \tau\theta + (1-\tau)\theta'$$

end

3

Methods

This section describes the methods used to investigate how an adaptive ANN can be used for model-free control. The first step aims to emulate the behaviour of the original controller using supervised learning and determine the suitability of using an ANN as a controller. The second step aims to investigate how the RDPG algorithm can be utilised to improve the performance and make the ANN adaptive.

3.1 Emulating the original controller

The ANN is implemented using Tensorflow [16] and Keras [17]. These tools are commonly used and simplifies the creation and training of an ANN. The architecture consists of dense and LSTM layers. However one limitation with Keras' LSTM is that the activation function must be set to *tanh* in order to execute on the GPU [17]. If the activation function is set to anything else the execution time will increase. Since execution time is important, only *tanh* will be used for LSTM layers.

To determine a network architecture; different input sequence lengths, depths and widths of the network are tested. The performance is measured by the prediction loss and by manual optimisation a relatively small architecture was obtained which is presented in fig. 3.1. The advantage of a small architecture is decreased learning time and most importantly, a shorter prediction time which is important as the network must be able to predict within the 8 ms cycle. An architecture could also be derived by Neuroevolution [18], which is a method to find an optimised ANN architecture through a process inspired by natural selection. However the method is computationally heavy and requires running a pool of concurrent networks, which all must be able to interact with the environment. This is impractical against the real system.

Data were collected from the truck using the original controller and driving at all relevant gears and speeds. In total, 86000 samples, about 11 minutes of driving data were collected. From this data, the inputs and outputs of the original controller were extracted to enable the use of supervised learning to emulate the original controller. The inputs (states) s and output (action) a were

$$s = \begin{bmatrix} \text{Requested pressure} \\ \text{Actual pressure} \\ \text{Wheel speed} \\ \text{Engine speed} \end{bmatrix}, \quad a = \begin{bmatrix} \text{Valve pilot current} \end{bmatrix}. \tag{3.1}$$

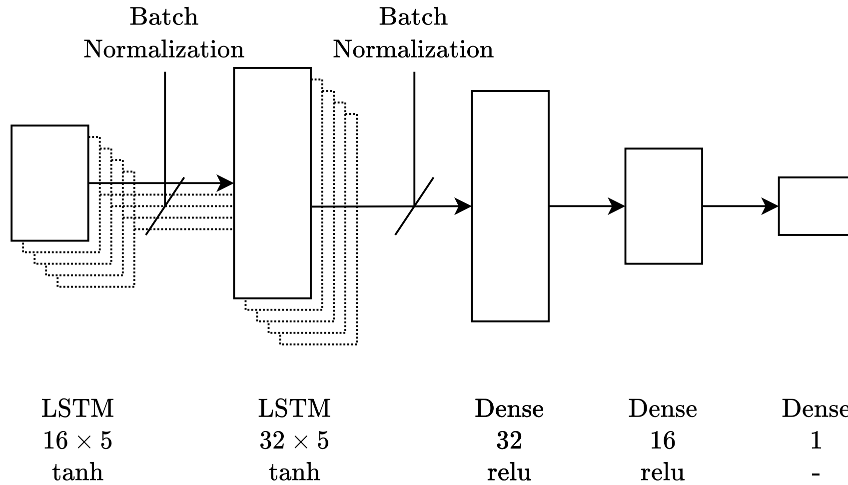


Figure 3.1: Structure of the ANN used for control. The listed properties for each layer is layer type, layer size(×sequence length) and activation function.

The sampled data were normalised using predefined scaling. The scaling factors were the maximum value that the signals can take and were determined by the permissible operating range of the hardware. The normalised sampled data were divided into frames with length $T = 5$, the input sequence length to the network. All T states were included, but only the action at the last time step was included. This resulted in a frame, seen in fig. 3.2, where the input vector x consists of all previous states from $s(t-T)$ up until the current state $s(t)$ and the ground truth y_n is the output from the original controller. The frames overlapped with 3 samples. The loss was calculated as the MSE between the predicted action $a(t)$ and the original controller's action. All these frames were randomly divided so that 90% were used for training and 10% were used for validation.

3.2 Reinforcement Learning

In general reinforcement learning is used for an untrained agent that explores the environment and iteratively becomes better. However, here the agent must perform, at least, good enough when the algorithm starts to not damage the truck.

As the algorithm runs against a real system it must execute at the same pace. The Electronic Control Unit (ECU) in the truck is running at a fixed cycle rate of 125 Hz and the prediction must thus be performed in less than 8 ms. However, the RDPG algorithm is relatively slow due to the heavy computations during backpropagation. Therefore, the algorithm is divided in parallel tasks as displayed in fig. 3.3.

The first task observes the state, predicts an action and outputs it to the truck. Additionally, this task stores the state, action, reward and resulting state to the replay buffer. This ensures that an action is output to the truck shortly after every new state observation. The algorithm can be seen in algorithm 2.

The second task samples the replay buffer and uses these samples to perform the

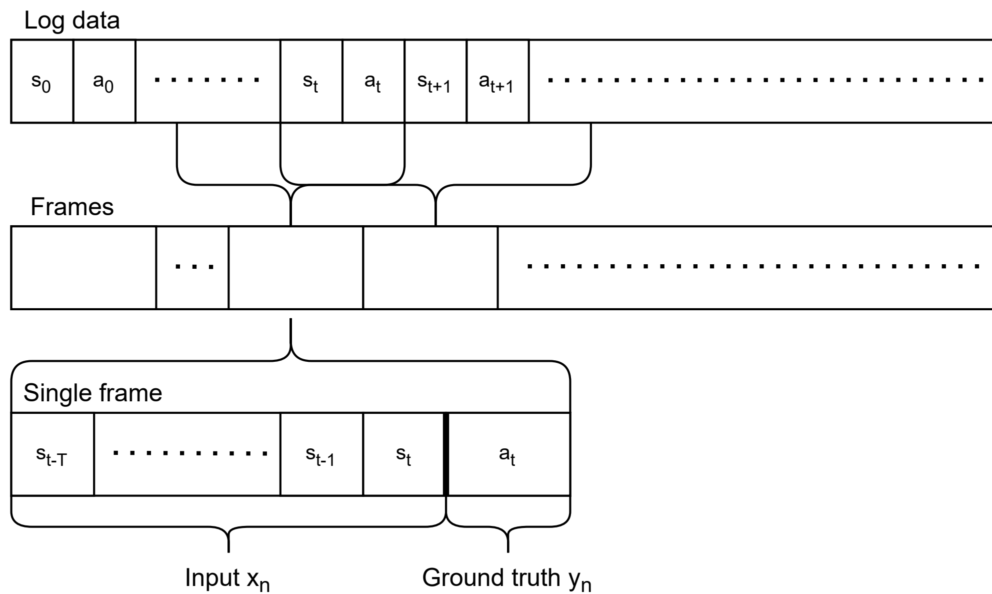


Figure 3.2: Division of log data, with states s and actions a, into frames input vector x of length T and ground truth y of length 1.

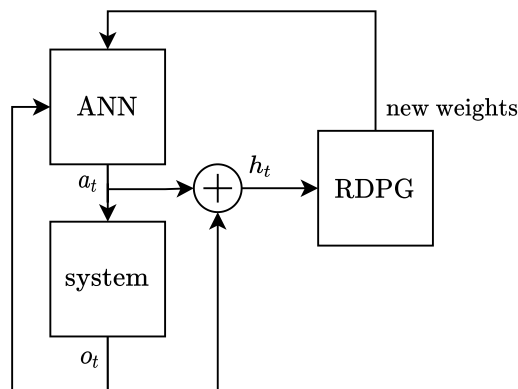


Figure 3.3: Two parallel processes to run RDPG in real-time on a PC.

update step in RDPG for actor, critic and the target networks. The weights of the updated actor network is then sent to the first task to updates a local copy of the actor network used for making predictions. This separation allows the RDPG algorithm to run with an independent sample rate from the truck. The algorithm can be seen in algorithm 3.

In this implementation, the replay buffer is 16384 samples long and 512 samples are used in every batch of the RDPG algorithm. In total, this implementation requires 5 ANNs due to the local copy in the first task.

The main objective of an reinforcement learning algorithm is to maximise the cumulative reward. To minimise the pressure regulating error, the reward is defined as the negative square of the pressure error, $reward = -error^2 = -(requested_pressure - actual_pressure)^2$. A lower error thus corresponds to a higher reward and the re-

inforcement learning algorithm will aim to decrease the error.

3.2.1 Continuous, real time RDPG

The RDPG algorithm proposed by DeepMind [4] was first used for solving tasks with a chaotic nature, such as video games, where the first few control actions often have a lasting effect on the state. To enable learning even if a poor initial action is made, RDPG will only train for a limited amount of time before the environment is reset to the initial conditions.

However, the hydraulic system being controlled in this thesis does not have such a dependency on earlier actions. While it certainly is a dynamic system, it is stable, and the influence of past control signals on the current state decays over time.

Furthermore, as the aim is to run the algorithm against a physical system, as well as being adaptive in normal driving scenarios, it is not feasible to recurrently reset the environment as this would require the driver to manually bring the truck to a stop or another reproducible state.

Therefore, we propose a modified, continuous RDPG algorithm where the training is contained within one continuous episode of arbitrary length. As the system has an input response of limited time, the histories h_i^t will also be of a fixed, limited length, containing only the last T states and actions. This is also motivated by the fact that RNN networks in general have relatively short-term memory cells, meaning that storing an excessive amount of past inputs will only marginally, if at all, improve the performance. As explained in [19], this can be attributed to the fact that in the BPTT algorithm, the loss gradients tend to decrease as the algorithm progresses further back in time and eventually vanishes, essentially removing the ability to learn from too old inputs. Limiting the history length is also required for practical purposes, as the replay buffer would otherwise grow without bounds.

Algorithm 2: Modified RDPG Algorithm Part I

Initialise thread p running algorithm 3
Initialise actor network $\mu^\theta(h_t)$ with pre-trained weights θ
for *always* **do**
  Initialise empty history h_0
  while *t is busy* **do**
    Observe current state o_t
    Calculate the reward r_t
    Append h_{t-1} with a_{t-1}, o_t and r_t
    $h_t \leftarrow (h_{t-1}, a_{t-1}, o_t, r_t)$
    Predict new action $a_t = \mu^\theta(o_t) + \epsilon$, ϵ: exploration noise
    Execute a_t to the environment
  end
  Store all transitions $(o_1, a_1, r_1, \ldots, o_T, a_T, r_T)$ in R in thread p
  Load updated actor weights calculate by algorithm 3
end

Algorithm 3: Modified RDPG Algorithm Part II

Initialise actor network $\mu^{\theta}(h_t)$ with pre-trained weights θ
Initialise target actor network $\mu^{\theta'}$ with pre-trained weights $\theta' \leftarrow \theta$
Initialise critic network $Q^{\omega}(h_t, a_t)$ with weights ω
Initialise target critic network $Q^{\omega'}$ with weights $\omega' \leftarrow \omega$
Initialise empty buffer R
while do

Sample a mini-batch of N episodes $(o_1^i, a_1^i, \dots, a_T^i, r_T^i)_{i=1,\dots,N}$ from R
Construct histories of finite length $h_t^i = (o_{t-T}^i, a_{t-T}^i, \dots, a_{t-1}^i, o_t^i)$
Compute target action from target actor $a_{t+1}^{i'} = \mu^{\theta'}(h_{t+1}^i)$
Compute target value using target critic

$$y_t^i = r_t^i + \gamma Q^{\omega'}(h_{t+1}^i, a_{t+1}^{i'})$$

Compute critic update using BPTT

$$\Delta\omega = \frac{1}{NT}\sum_i \sum_t \left(y_t^i - Q^{\omega}(h_t^i, a_t^i)\right)\frac{\delta Q^{\omega}(h_t^i, a_t^i)}{\delta\omega}$$

Compute actor update using BPTT

$$\Delta\theta = \frac{1}{NT}\sum_i \sum_t \frac{\delta Q^{\omega}\left(h_t^i, a_t^i = \mu^{\theta}(h_t^i)\right)}{\delta a}\frac{\delta\mu^{\theta}(h_t^i)}{\delta\theta}$$

Update actor and critic weights using Adam
Update the target networks weights

$$\omega' \leftarrow \tau\omega + (1-\tau)\omega'$$
$$\theta' \leftarrow \tau\theta + (1-\tau)\theta'$$

end

3.2.2 Transfer learning

Transfer learning is a concept where a network trained for a specific task is used again for another, related task. The main purpose of using transfer learning is to improve the learning with the new task. Both the learning rate and the final performance can improve compared to if the network hadn't been trained for the previous task [20].

In this thesis, transfer learning refers to how the ANN trained using supervised learning is leveraged in the reinforcement learning setup. This is implemented by using the ANN from the first step to initialise the actor in the RDPG algorithm. There are several reasons for why this is done. Firstly, it is anticipated that it will decrease the time required to converge during reinforcement learning. Secondly, the pre-trained actor should start out with, and hopefully remain, having a policy that is good enough to run the reinforcement learning algorithm without catastrophic results in the physical system. Lastly, as the control signal must exceed a certain threshold in order to control the system, an untrained actor would require relatively large exploration noise in order to receive any meaningful feedback.

It should be noted that in this setup, only the actor is pre-trained and the critic will be initialised with random weights. It is assumed that this might cause an initial deterioration of the actors performance before the critic has learned enough. To remedy this, there will be an investigation of how the critic could be pre-trained and how, if the behaviour is changed.

3.3 Simulation

It is crucial that the output of the ANN and the reinforcement learning does not cause any undesired behaviour or damage to the truck. Therefore, a rudimentary model of the hydraulic pump is used for verification before testing on the real system. Such a model also serves as a substitute for the physical system when testing the implementation of the reinforcement learning algorithm and if the physical system is unavailable. Even though this model does not capture the full behaviour of the real system, it is similar enough to give some idea about the performance and tendencies of the reinforcement learning algorithm.

The model should describe the hydraulic system behaviour. It uses a static relationship between engine speed, wheel speed and valve pilot current to estimate the resulting differential pressure over the hydraulic motors. The model output is low-pass filtered to account for the rise time caused by inertia in the physical system.

In fig. 3.4, the model is compared to the real system. The inputs to the model, as well as the true system's pressure are recorded during a test drive with the original controller. As is evident from the graph, the model output does not follow the true system pressure particularly well. Nevertheless, it shows some degree of similarity and more so when looking at the transient parts of the simulation. It should also be noted that this difference presents an interesting possibility. As the actor network will be pre-trained to emulate an existing controller, the results of reinforcement learning against the model will to some extent be an indication of the ability to

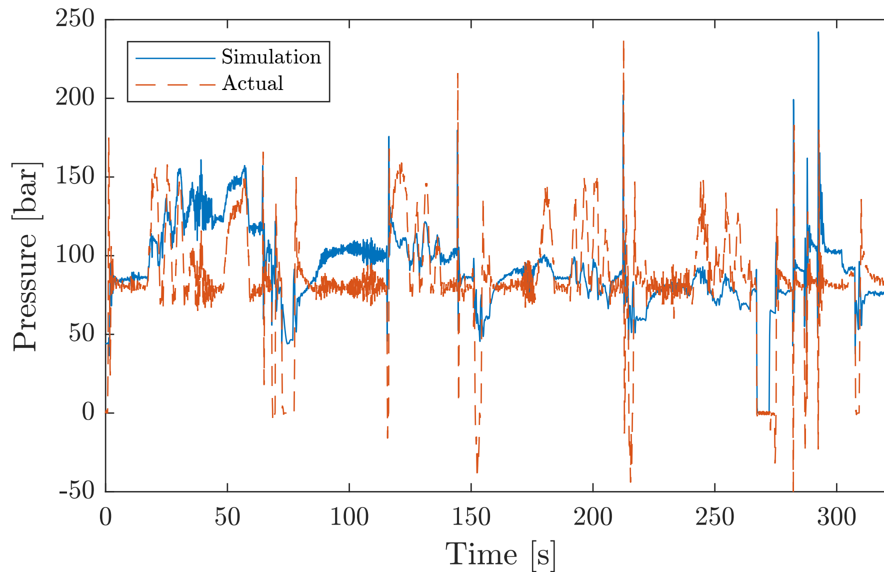


Figure 3.4: Comparison between model output, using sampled data as inputs, and the true system output from the same log.

adapt to a change in the system.

When the model is used to simulate the system it only predicts the pressure. The valve pilot current will be given by the controller and the rest of the inputs come from log files.

3.3.1 Tests against rudimentary model

Though the objective was to investigate a model-free approach, there are multiple benefits of having a model. One being that the effect of different parameters on the behaviour of the reinforcement learning algorithm can be evaluated without extensive real-world testing.

There are several important features the reinforcement learning algorithm should have, where the first is long term stability. Since the ANN will control a real system, the prediction must stay stable and deteriorating or oscillating performance is undesired. The second is the learning time where it is favourable that the algorithm quickly trains the network to perform better. The performance will be evaluated by observing the convergence rate, oscillations and long term stability of a moving average of the following error.

It should be noted that some of these criteria may affect each other as there often is a trade-off to be made between stability and convergence rate. Both are strongly affected by the choice of the learning rate for the actor and critic networks. A low learning rate leads to a long training time, while a high learning rate can affect the stability. By testing different values and ratio between the actor and critic learning rates, optimal parameters for some optimality criterion may be found.

Pre-training of the critic network is another interesting topic that will be evaluated. Since the actor is pre-trained using supervised learning, it is not far-fetched

to assume that pre-training the critic could also benefit the performance of the reinforcement learning algorithm. If the critic is untrained and the actor is trained, the critic could predict wrong cumulative reward and move the actors weights in wrong direction, causing an initial decrease of performance. During the pre-training phase, the modified RDPG algorithm will run with locked actor networks for a certain time so that only the critic and target critic networks will be updated. Ideally, this should improve initial learning.

3.4 Tests against truck

The performance of the ANN was tested against the real system by running inference in real-time in the truck. This is implemented as a python script running on a PC that communicates with the ECU in the truck via a Controller Area Network (CAN) interface. At every time step, the ECU outputs the measured quantities to the CAN bus, which are then fed into the ANN. The output of the ANN, which is the valve pilot current, is then returned to the ECU via the CAN bus.

The ECU is operating at a fixed cycle rate of 125 Hz and the ANN must be able to predict a new output within one cycle, thus limiting the size of the network. The real-time requirement also restricts the implementation of the RDPG algorithm which must not interrupt the inference for more than $1\,\mathrm{s}/125\,\mathrm{Hz} = 8\,\mathrm{ms}$.

3.4.1 Drive cycle

The major benefit of the truck over the rudimentary model is the diversification in the data. The rudimentary model is using sampled data that is looped. The agent will then experience the same input indefinitely, with the exception of the actual pressure that the model is predicting. However in the truck the probability to generate the same input state is very low.

An optimal drive cycle would include all possible scenarios that the truck will encounter, such as different temperatures, inclinations and speeds. However the objective is to investigate the usage of adaptive model-free control, limiting the need of an optimal drive cycle as the performance of the algorithm is the main objective rather than the performance of the hydraulic system in every possible situation. The test location used for real-world testing has physical limitation such as the absence of hills. The drive cycle will thus only include three common scenarios, driving straight forward, driving forward while turning and gear shifts.

4

Results

4.1 Emulating the original controller

The ANN was trained using supervised learning with 86000 samples, about 11 minutes of driving data from the truck using the original controller. Fig. 4.1 shows how the ANN is emulating the original controller from the sampled data. The ANN manages to predict the output with a scaled validation MSE loss of 215 mA^2, corresponding to a Root Mean Square Error (RMSE) of 14.7 mA. This means that the network predicts with an average error, in the Root Mean Square (RMS) sense, of 14.7 mA. To contrast this, the Root Mean Squared Percentage Error (RMSPE) is a mere 1.9 %.

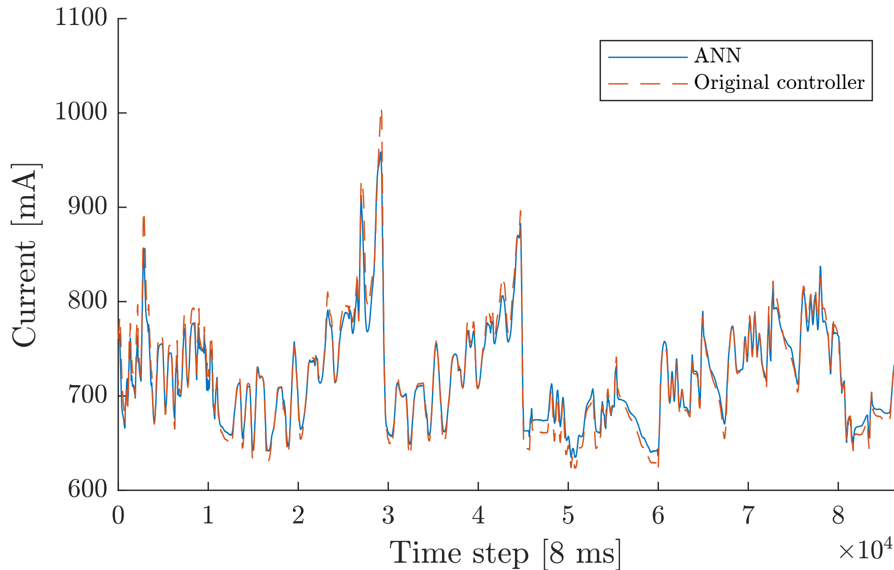


Figure 4.1: ANN emulating the original controller from sampled data.

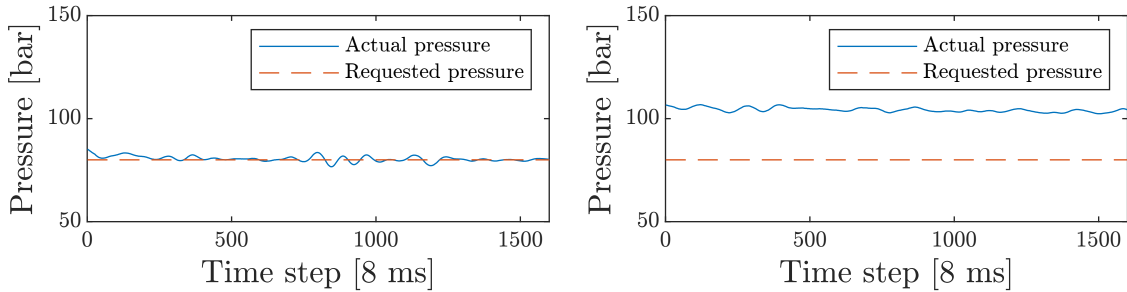
4.1.1 Driving straight forward

The performance was measured by observing the the requested and actual pressure in the hydraulic system. The result is presented as the RMSE and RMSPE between the requested and actual pressure and can be seen in table 4.1 while driving straight forward.

The RMSE is larger for the ANN compared to the original controller for all gears

Table 4.1: RMSE and RMSPE over 1600 time steps driving straight forward for the original controller and the ANN

Gear	Original (RMSE)	ANN (RMSE)	Original (RMSPE)	ANN (RMSPE)
1	2.1 bar	24.3 bar	2.6 %	30.4 %
2	7.4 bar	18.2 bar	9.2 %	22.7 %
3	10.0 bar	13.3 bar	12.5 %	16.7 %
4	5.5 bar	8.2 bar	6.9 %	10.3 %



(a) Original controller. **(b)** ANN controller.

Figure 4.2: Actual and requested pressure for original controller and ANN driving forward in first gear.

while driving straight forward. For the first gear, seen in fig. 4.2, the original controller's error is close to zero for the complete episode, while the ANN has a large bias.

In the second gear, seen in fig. 4.3, the original controller has larger error compared to the first gear. However the ANNs performance is better compared with the first gear, though with some more oscillations.

In the third gear, seen in fig. 4.4, the error has decreased for the ANN compared to the first and second gear. The RMSE for the ANN is only 30 % higher than the original controller at this gear.

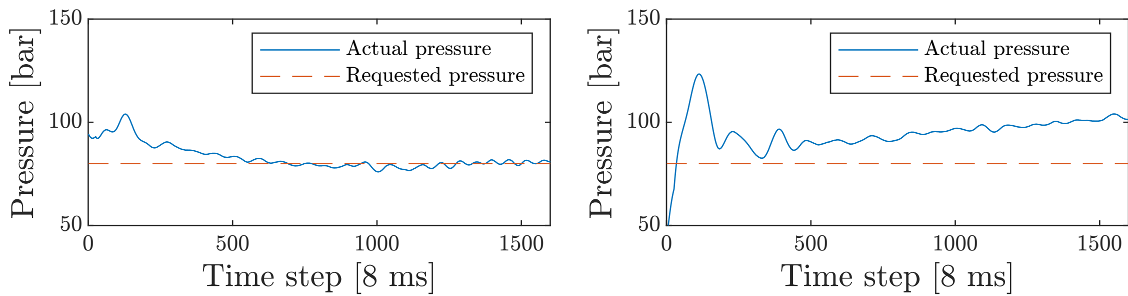
At the fourth gear, seen in fig. 4.5, the ANN has the best performance for all gears. However the RMSE is still larger for the ANN compared to the original controller.

4.1.2 Driving forward while turning

When driving forward while turning, the requested pressure will increase as the steering angle increases. The results are presented as the RMSE and RMSPE between the requested and actual pressure and can be seen in table 4.2.

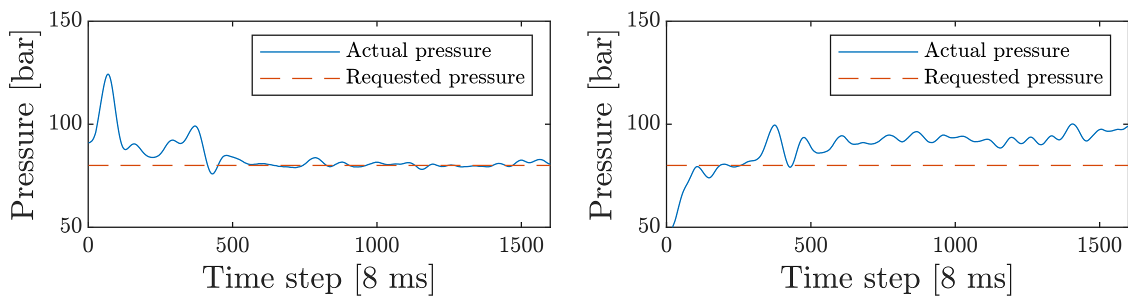
Similar to when driving straight forward the RMSE is larger for the ANN compared to the original controller for all gears. At the first gear the bias is still present for the ANN, however it is able to follow the dynamics as seen in fig. 4.6.

At the second, third and fourth gear, seen in fig. 4.7, 4.8 and 4.9, the performance is quite similar. At the third and fourth gear the original controller only performs slightly better.



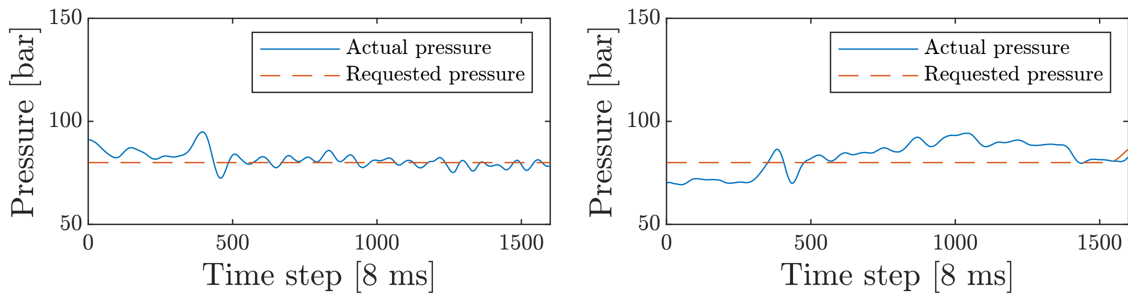
(a) Original controller. **(b)** ANN controller.

Figure 4.3: Actual and requested pressure for original controller and ANN driving forward in second gear.



(a) Original controller. **(b)** ANN controller.

Figure 4.4: Actual and requested pressure for original controller and ANN driving forward in third gear.

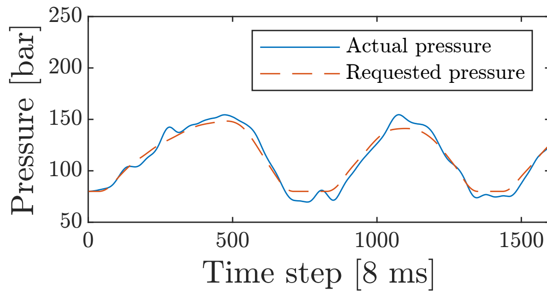


(a) Original controller. **(b)** ANN controller.

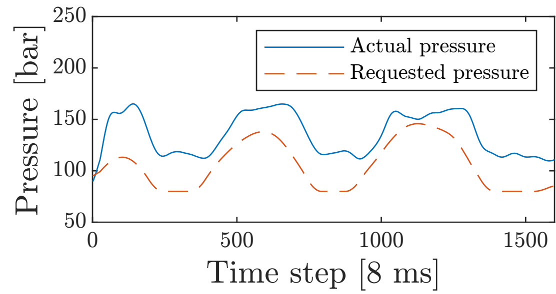
Figure 4.5: Actual and requested pressure for original controller and ANN driving forward in fourth gear.

Table 4.2: RMSE and RMSPE over 1600 time steps driving forward and turning for the original controller and the ANN

Gear	Original (RMSE)	ANN (RMSE)	Original (RMSPE)	ANN (RMSPE)
1	6.7 bar	30.6 bar	6.5 %	35.0 %
2	5.4 bar	11.7 bar	4.9 %	14.1 %
3	9.1 bar	11.5 bar	8.4 %	14.7 %
4	10.1 bar	12.7 bar	10.1 %	16.1 %

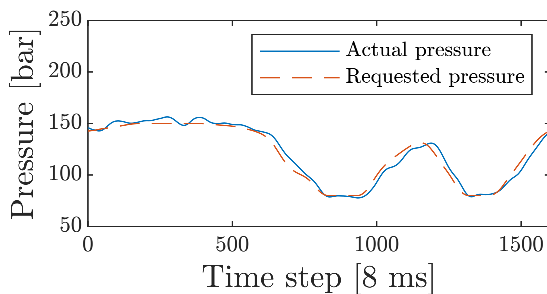


(a) Original controller.

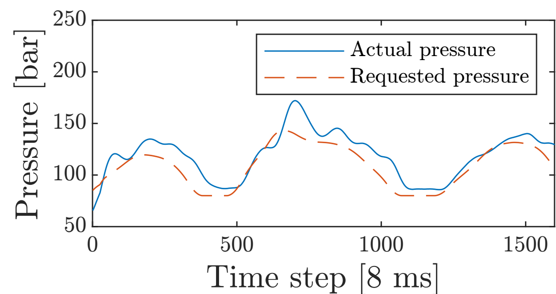


(b) ANN controller.

Figure 4.6: Actual and requested pressure for original controller and ANN driving forward and turning in first gear.

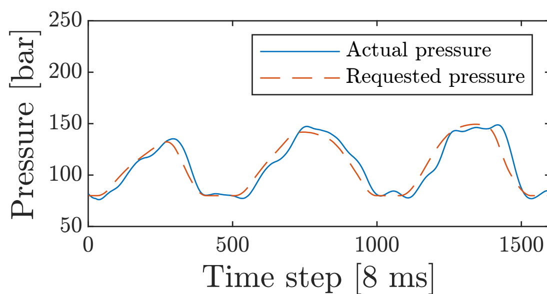


(a) Original controller.

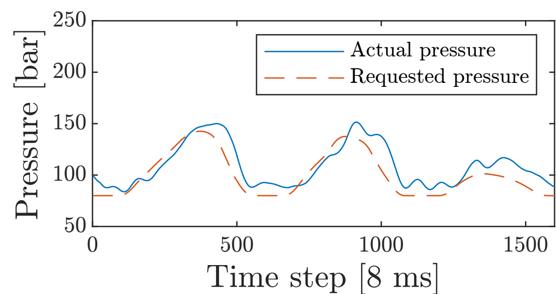


(b) ANN controller.

Figure 4.7: Actual and requested pressure for original controller and ANN driving forward and turning in second gear.

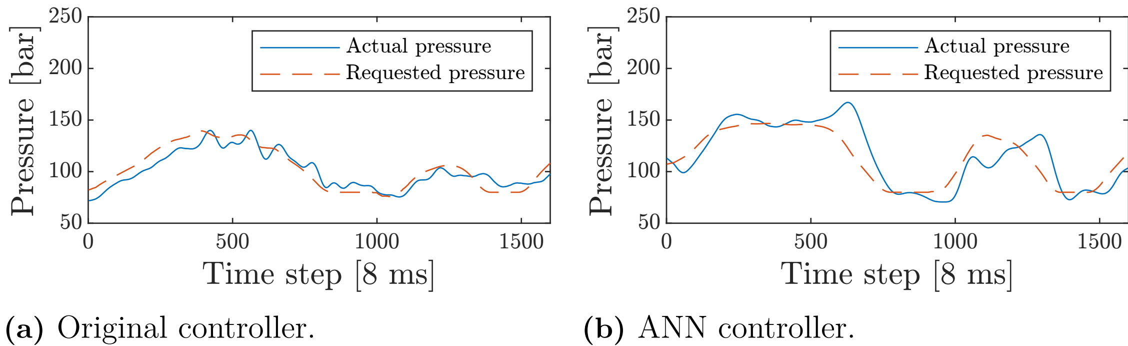


(a) Original controller.



(b) ANN controller.

Figure 4.8: Actual and requested pressure for original controller and ANN driving forward and turning in third gear.



(a) Original controller.

(b) ANN controller.

Figure 4.9: Actual and requested pressure for original controller and ANN driving forward and turning in fourth gear.

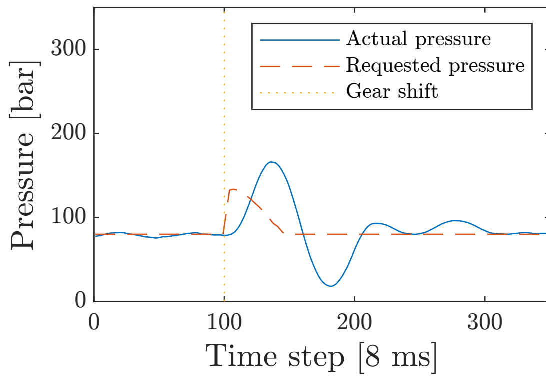
Table 4.3: RMSE and RMSPE over the first 150 time step after a gear shift for the original controller and the ANN.

Gear	Original (RMSE)	ANN (RMSE)	Original (RMSPE)	ANN (RMSPE)
1 to 2	31.0 bar	31.1 bar	35.2 %	35.7 %
2 to 3	44.3 bar	39.5 bar	43.8 %	39.8 %
3 to 4	60.4 bar	44.5 bar	59.4 %	40.0 %

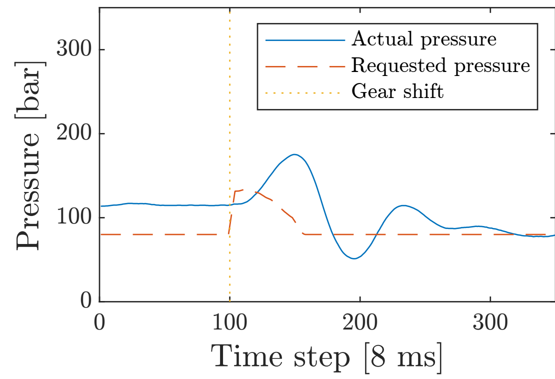
4.1.3 Shifting gear

During a gear shift the requested pressure will increase for a short time and then decrease. The results are presented as the RMSE and RMSPE between the requested and actual pressure for the first 100 time steps after a gear shift and can be seen in table 4.3. The RMSE is lower at all gear shifts for the ANN, however the RMSE is larger at the gear shifts scenarios compared to both driving straight and turning.

For all gear shifts a noticeable difference is present between the original controller and the ANN. The first overshoot is quite similar for both controllers in all gears, however the second overshoot is much larger for the original controller. The performance can be seen in figure 4.10, 4.11 and 4.12.

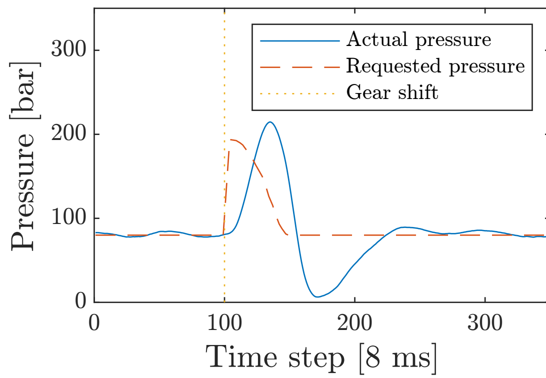


(a) Original controller.

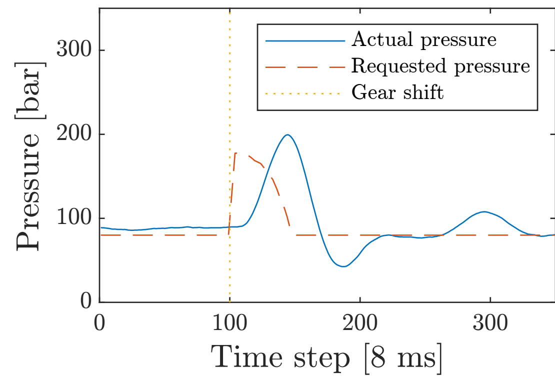


(b) ANN controller.

Figure 4.10: Actual and requested pressure for original controller and ANN during a gear shift from first to second gear.

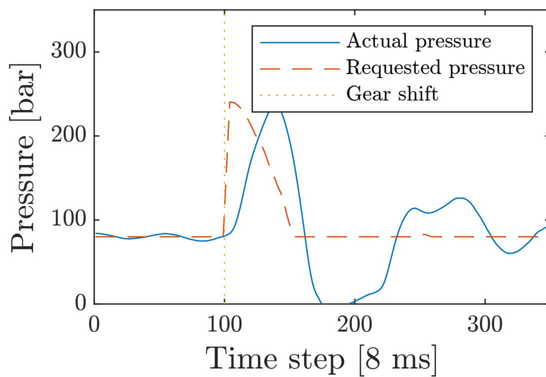


(a) Original controller.

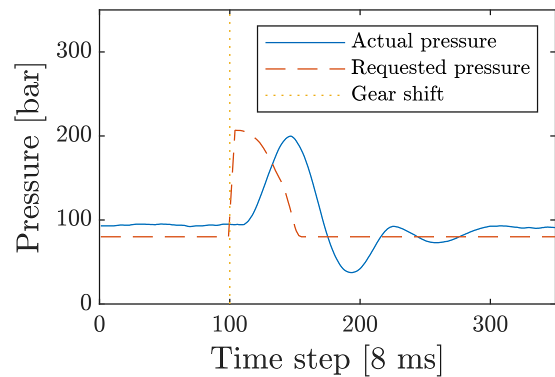


(b) ANN controller.

Figure 4.11: Actual and requested pressure for original controller and ANN during a gear shift from second to third gear.



(a) Original controller.



(b) ANN controller.

Figure 4.12: Actual and requested pressure for original controller and ANN during a gear shift from third to fourth gear.

4.2 Reinforcement learning

The reinforcement learning algorithm was first evaluated against a rudimentary model to be able to find how the hyperparameters affected the learning process. The results from these tests was then applied to the tests in the truck.

4.2.1 Tests against model

The RDPG was evaluated against the rudimentary model. The performance in terms of pressure regulation after different amount of training is presented in fig. 4.14. The results show that the pre-trained network does not manage to control the dynamics of the pressure model very well from the start. It does however manage to control the pressure to within a factor of two of the requested pressure. As time progresses, it is evident that the pressure regulation improves, and thus the objective of maximising $-(\text{RequestedPressure} - \text{ActualPressure})^2$ is approached as the error is minimised.

The performance trend is however, not monotonic. This is clearly shown in fig. 4.13, where the reward decreases in the beginning of the training process before eventually starting to increase steadily.

It is also interesting to note that the performance in some situations become worse in the first few hours of training, which can be seen in fig. 4.14 e.g between sample 0 and 5000 or sample 15000 and 20000. This does not go against the maximisation objective as it is evaluated over the replay buffer, thus allowing the performance to decrease in some situations as long as it increases more in others. Fig. 4.13 also shows how the average reward converges while the low reward outliers, which are visible in the graph for the shorter moving average, remain for quite some time.

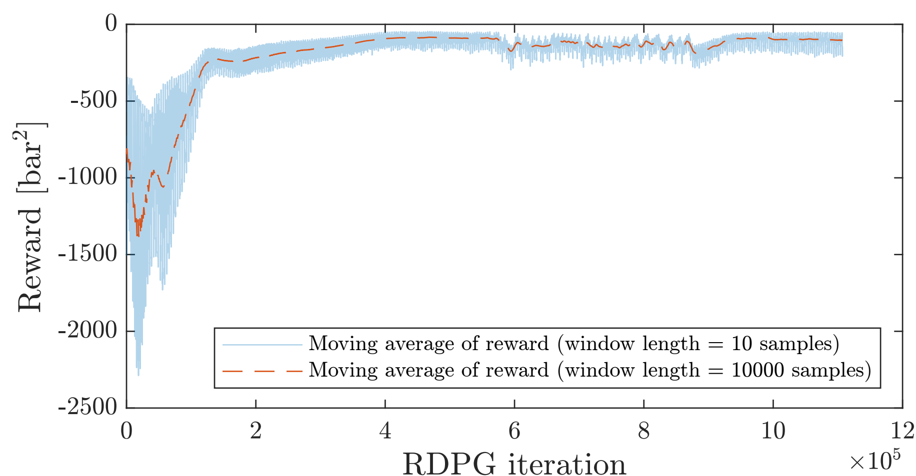


Figure 4.13: Reward during training with RDPG against the system model.

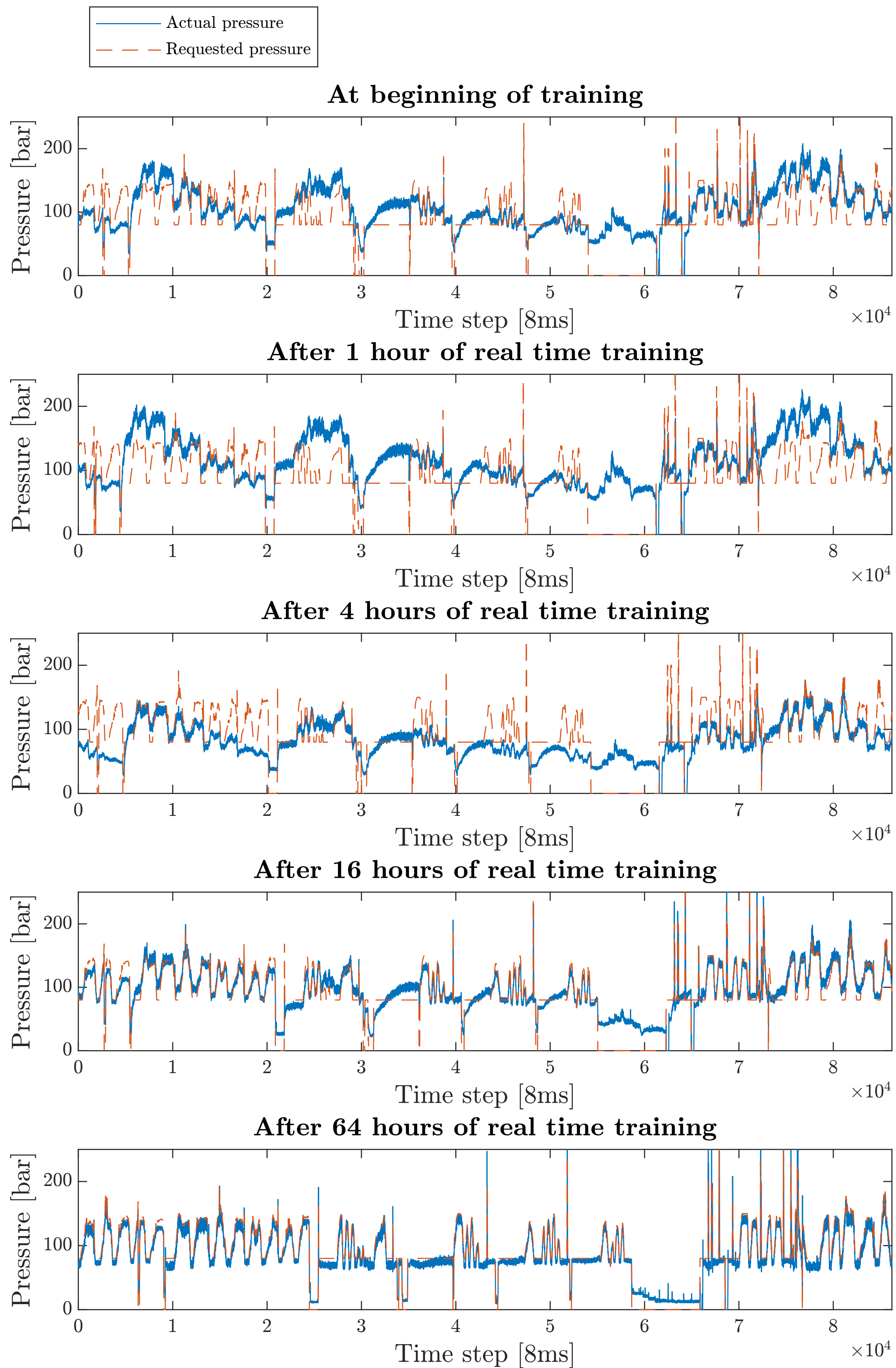


Figure 4.14: Regulation performance of the closed-loop system after varying amounts of reinforcement learning against the system model.

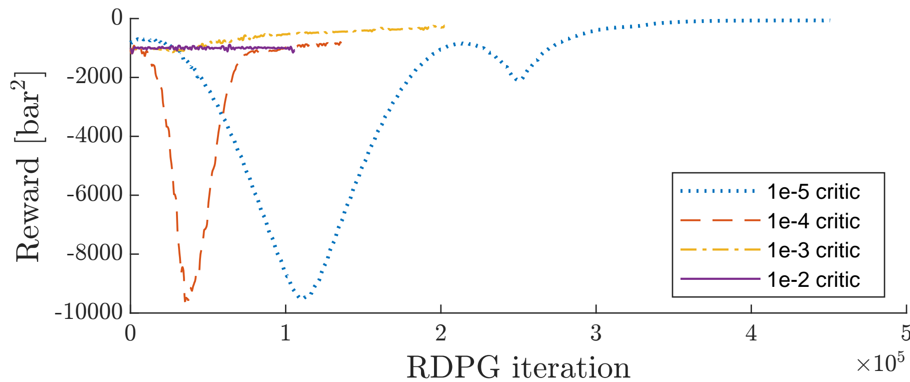


Figure 4.15: Reward during training with varying critic learning rates against the system model.

4.2.1.1 Critic learning rate

Varying the learning rate β of the critic network had a clear impact on learning process. A smaller learning rate caused the reward to drop significantly and a long recovery time as is evident for $\beta = 1 \times 10^{-5}$ in fig. 4.15. Increasing the learning rate to 1×10^{-4} yielded the same decrease in the reward, although with a faster recovery. Further increasing the reward to between 1×10^{-3} and 1×10^{-2} removed the initial drop in reward and none of the networks dropped below the initial performance from the supervised learning based pre-training. However, though the reward did not drop when training with the highest learning rate, 1×10^{-2}, the reward did not seem to increase either.

4.2.1.2 Pre-training critic

With a pre-trained actor, the critic was trained for a predefined number of iterations before the actor was updated by the RDPG algorithm. The critic networks were pre-trained for 0, 1000, 5000 and 20000 of iterations respectively. The reward after the pre-training episode is shown in fig. 4.16.

The algorithm only converged for the critic networks with 0, 1000 and 5000 pre-training iterations, while the critic that was pre-trained for 20000 iterations diverged. One distinct difference between pre-training for 0, 1000 or 5000 iterations is how much the reward decreases before it converges.

4.2.2 Tests against the truck

The tests against the truck were using a pre-training of 1000 iterations for critic and a critic learning rate of 1×10^{-3} and 1×10^{-4}. The reward can be seen in fig. 4.17. The two tests were aborted due to due reaching the limits of the hydraulic system.

During the test with a critic learning rate of 1×10^{-3} the RDPG algorithm drove the signal to high and the pressure in the system became to large and had to be aborted before reaching the maximum. The requested and actual pressure can be seen in 4.18a. The second test with a learning rate of 1×10^{-4} the RDPG algorithm drove the signal too low and into a deadband window where the pressure became

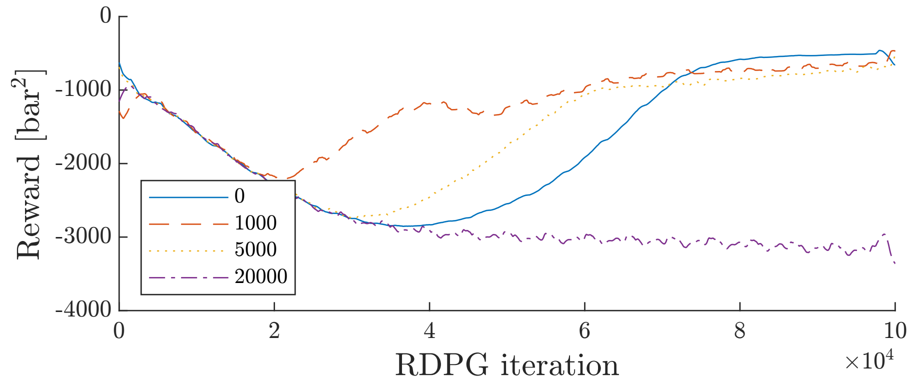


Figure 4.16: Reward during training with varying amounts of critic pre-training against the system model.

zero. The pressure can be seen in fig. 4.18b.

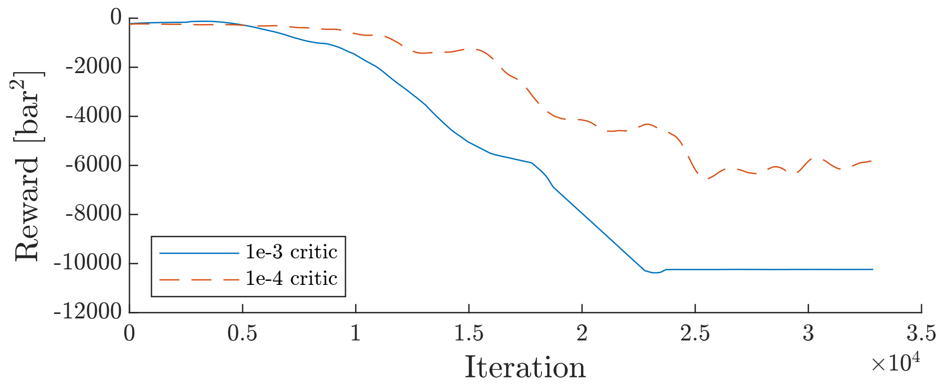
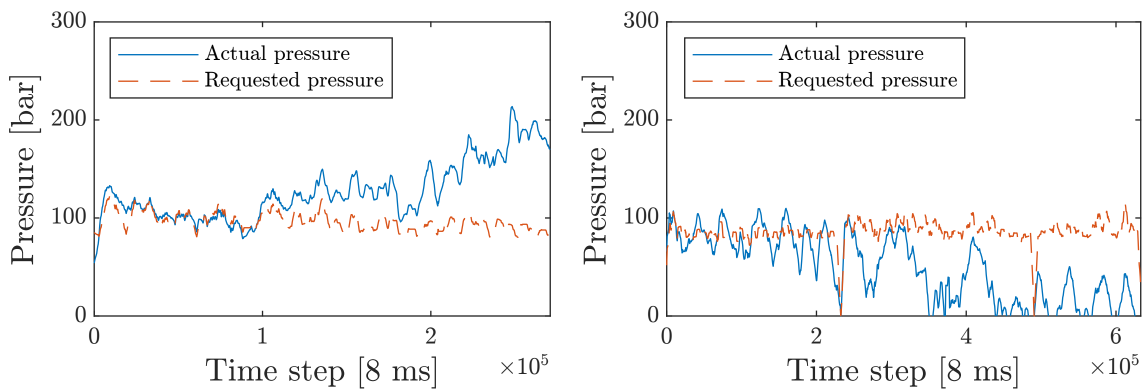


Figure 4.17: Reward during training with varying critic learning rates against the truck.



(a) Critic learning rate 1×10^{-3}.

(b) Critic learning rate 1×10^{-4}.

Figure 4.18: Moving average over requested and actual pressure against the truck.

5

Discussion

The purpose of this thesis was to investigate how an adaptive ANN controller can replace an existing controller. The goals were to first emulate the existing controller and thereafter utilise reinforcement learning to improve the performance and make the ANN adaptive.

5.1 Emulating a controller with ANN

An ANN with LSTM and dense layers was trained with 11 minutes of sampled driving data using supervised learning. The loss to be minimised was the MSE between the original controller's output and the ANN's output. The result can be seen in fig. 4.1 and the RMSE and RMSPE was only 14.7 mA and 1.9 % respectively. Because of this, the pressure error when using the ANN was expected to be only slightly worse compared to the original controller.

The test cycle included three drive scenarios, driving straight forward, turning and gear shifts. The performance of the two controllers were measured by the squared pressure error.

In the first scenario, driving straight forward, the ANN performed differently in different gears. When driving in the first gear, seen in fig. 4.2b, a large bias is present and the performance is poor for the ANN compared to the original controller. For the second, third and fourth gear the performance was also poor compared to the original control, however the performance increased for the higher gears.

The poor performance could be a consequence of insufficient training data for some gears or a change in the physical system between collection of training data and the test drive. The training data should include the entire span for the signals, such as all engine speed, wheel speed, gears and pressures. However the network should be able to generalise from the training data and not require every single combination, thus it is assumed that the data contains at least a sparse representation of the different scenarios. The poor performance could also be a consequence from the different conditions between the sampling and the testing. The data were sampled a few months before the testing and both the environment and the actual system have, at least slightly, changed. Another approach could have been to sample more data and train the ANN during the same test day.

The architecture could also affect if the ANN becomes over fitted from the training data, thus performing poor on new data. Since the validation data is sampled from

the same condition as the training data the result is biased. However since the tests against the real system were rare, trying different architecture was difficult and not feasible in this thesis. Another parameter that can be tuned is the input sequence length. The length was set to five meaning that the ANN knows the current state as well as the states from the last 32 ms. During training against sampled data the performance did not increase with longer sequence length, however it could be a parameter to investigate further to enable the network to increase its performance.

The second scenario was driving forward while turning which will create a requested pressure that follows the steering angle. In the first gear the bias in actual pressure is still present, though the ANN is able to follow the dynamic, which can be seen in fig. 4.6b. For the second and third gear the bias is much smaller, though still present. The fourth gear does not seem to have any bias, however the performance is poor compared to the original controller, which can be seen in fig. 4.9. Like in the scenario with driving straight forward the ANN has a poor performance at all gears while driving forward and turning.

However, when comparing the response time it seems that the ANN lags behind and has a small time delay when trying to follow the requested pressure compared to the original controller. As for the original controller the time delay is almost not noticeable. The time delay difference is noticeable while driving forward and turning in third gear and can be seen in fig. 4.8. However some delay is expected since the actual hydraulic system is slow and the pressure cannot be changed instantaneously.

The third scenario is gear shifts, where the ANN has the same or better performance than the original controller. The requested pressure is raised quickly and is hard to track for either controller. An explanation could simply be that the hydraulic system is slow, which makes it hard to control in fast dynamic scenarios. The most significant difference between the two controller is how deep the first dip becomes and for all gear shifts the ANN's dip is smaller.

5.2 Reinforcement learning against the model

The ANN trained with supervised learning was used in a reinforcement learning setting against a model of the hydraulic system. The input data to the model were 11 minutes of logged driving data which were repeated during the learning process. As the model did not accurately describe the real system it was expected that the agent should start with a relatively poor policy which would improve over time. The results were, however, that the policy became worse in the early stages of training before eventually improving. This goes in line with results described by Google DeepMind [4] when RDPG was first outlined and evaluated.

This initial deterioration was believed to be caused by an untrained critic network giving poor, or even detrimental, feedback to the actor network. This would in turn cause the actor network to unlearn parts of what it learnt during the pre-training phase. To some extent this hypothesis was strengthened by the results of pre-training the critic network which reduced the deterioration for small amounts of pre-training. Nevertheless, the deterioration still occurred and large amounts of

pre-training caused the agent to diverge.

Additional testing showed that letting the critic learn much faster, with a learning rate between 10^4 and 10^5 times higher than that of the actor yielded not only a faster convergence, but also significantly lower deterioration in the early stages of training. This also supports the hypothesis that an untrained critic network causes the actor network to unlearn some of its pre-training. This is an interesting topic for future work; are there other factors which cause this initial deterioration and how can it be remedied?

It should be noted that the reinforcement learning tests ran against the model are based on a repeated 11-minute driving cycle. With limited amount of training data, care has to be taken to avoid over training or over fitting, that is, learning patterns in the training data that are not desired, such as memorising input-output pairs instead of learning an behaviour viable for real-world data. Several methods are used to limit this risk, the first is intrinsic from the use of the model which ensures that some of the inputs (actual pressure) varies for every repetition of the logged data. Secondly, due to the exploration noise used in RDPG, the training data is further augmented in a similar fashion. Finally, the ANN structure is kept relatively small compared to the number of samples, which reduces the risk of over fitting as the number of weights to adjust is small relative to the number of training samples.

5.3 Reinforcement learning against the truck

The real tests in the truck were using the pre-trained ANN with an initial performance shown in tab. 4.1, 4.2 and 4.3.

Both tests had to be aborted due to limitations in the hydraulic system. Either the pressure became to high or the control signal became too low and the hydraulic system entered a deadband. The tests revealed some of the challenges with using the RDPG algorithm, which are the uncertainty of convergence and difficulty of tuning the hyper parameters. It is both uncertain that the algorithm will converge and how fast. While training against a model the physical limits are not present. Utilising a model also helps to iteratively change and optimise the hyperparameters that affects the learning considerably, which was shown in fig. 4.15 for different critic learning rate. These parameters have to be tuned for every system and must be evaluated by running the RDPG algorithm many times which can be impractical for physical systems. However if the system is accessible, robust and simple to test on, it should be possible.

For the second test the output from the network became too low and the hydraulic system entered a deadband. In this region, changing the control output does not affect the resulting pressure. This means that neither increasing, nor decreasing the control signal will result in a better reward. As RDPG explores by adding a noise to the control signal, it is hard to get out of a deadband if the noise is lower than the size of the deadband. On the other hand, as the noise limits the maximum achievable performance when converged, it is undesirable to have too large noise.

5.4 Implementational challenges

The driving data used for learning with RDPG is diverse and contains driving in all gears at a wide engine speed range which enables the network to learn a general control strategy that works in many situations. In real world application such diverse driving can not be taken for granted. Possible scenarios are that some gears are used extensively while others are only briefly used, that long periods are spent at constant speeds or that the engine speed is kept below a certain level. This kind of driving could result in a specialised controller that handles those specific driving conditions well but does not generalise to other conditions. Though the methods used in this thesis could be used to produce a fixed network by using the RDPG algorithm in a controlled environment with diverse driving, which is an interesting method in itself, this is not enough if the goal is to achieve adaptive control in real-world scenarios. However, as the RDPG is an off-policy method and does not have to train on the data generated by the current actor network, it could be possible to diversify the training data by selectively keeping parts of the replay buffer in memory. One possible method is to divide the buffer into bins, where each bin corresponds to a certain range in the input space, for example, which gear is selected or some combination of inputs. This would allow the learning process to take old experiences into account so that important aspects are not forgotten.

Implementing the adaptive algorithm within an embedded system in a truck raises new challenges. RDPG is an off-policy algorithm and does not have to execute in real-time, however the actor must predict a new action in 125 Hz. As long as the computational power is enough to meet this requirement the RDPG algorithm can execute in its own pace. However the convergence rate of the agent is dependent on both the learning rate and the number of RDPG iteration. A more powerful computer would be able to execute more iterations in a certain amount of time and thus converge faster. Therefore the computational requirements depends on the requirements of how fast the agent should be able to adapt.

Another requirement is the memory capacity of the embedded system. Both actor and critic has its weights and those are copied to their target networks in the RDPG algorithm. Depending on the resolution of the data types used for the weights, the memory requirements could vary. The resolution could be decreased, though the update step of the weights requires backpropagation with calculation of the gradients and a decrease in resolution of the weights would decrease the resolution of the gradients. The RDPG algorithm is utilising a replay buffer to store previous experiences and to not forget old experiences. The replay buffer is sampled with the batch size and cannot be too large. However it is important that the size is large enough to contain a diversity of scenarios. Therefore the size of the replay buffer would be a balance between learning from both old and recent experiences or only recent.

6
Conclusion

The purpose of this thesis was to investigate using an ANN for adaptive model-free control. The first goal was to emulate an existing controller with an ANN using supervised learning. The result showed that this was feasible, though with relatively poor performance. However, perfect emulation of a sub-optimal controller was not the intention, as the reinforcement learning algorithm should improve the performance of the ANN. Though it would be possible to improve the performance with more training data.

The second goal was to utilise reinforcement learning to adapt and improve the performance of the ANN, thus creating an adaptive model-free controller. This approach showed promising results against a rudimentary model of the hydraulic system, and the pre-trained network gradually learned to control the model. The learning rate for the untrained critic network had a large impact on the convergence of the algorithm and testing showed that a too low learning rate resulted in the actor unlearning parts of its pre-training in the early stages of training.

Another interesting result is how pre-training the critic seem to improve the performance of the RDPG algorithm. However the result indicated that to much pre-training made the algorithm diverge.

Testing the method against the physical system resulted in divergence of the learning. However, as extensive testing was not possible due to problems with the test bed, it was not possible to perform an exhaustive search for hyper parameters that would allow the system to converge. Further testing and tuning of parameters such as learning rates could lead to convergence. These results show how these methods, despite being model-free, can benefit greatly from having a simulation model, as it allows for fast testing and verification.

Bibliography

[1] E. F. Camacho and C. Bordons, "Nonlinear model predictive control," in *Model Predictive control.* Springer London, 2007, pp. 249–288. [Online]. Available: https://doi.org/10.1007/978-0-85729-398-5_9

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[3] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcment learning," *arXiv preprint arXiv:1509.02971*, 2019. [Online]. Available: https://arxiv.org/abs/1509.02971v6

[4] N. Heess, J. J. Hunt, T. P. Lillicrap, and D. Silver, "Memory-based control with recurrent neural networks," *arXiv preprint arXiv:1512.04455*, 2015. [Online]. Available: http://arxiv.org/abs/1512.04455

[5] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[6] E. Pettersson, "Improving and Evaluating Hydraulic Front Wheel Drive for Trucks," Master's thesis, Chalmers University of Technology, Sweden, 2012. [Online]. Available: https://hdl.handle.net/20.500.12380/170928

[7] S. Agatonovic-Kustrin and R. Beresford, "Basic concepts of artificial neural network (ann) modeling and its application in pharmaceutical research," *Journal of pharmaceutical and biomedical analysis*, vol. 22, no. 5, pp. 717–727, 2000.

[8] L. Mingxian, "Rnn," Dec 2018, accessed 2021-01-15. [Online]. Available: https://commons.wikimedia.org/wiki/File:RNN.png

[9] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 11 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[10] J. Sola and J. Sevilla, "Importance of input data normalization for the application of neural networks to complex industrial problems," *IEEE Transactions on Nuclear Science*, vol. 44, no. 3, pp. 1464–1468, 1997.

[11] L. N. Smith, "A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay," *arXiv*

preprint arXiv:1803.09820, 2018. [Online]. Available: https://arxiv.org/pdf/1803.09820.pdf

[12] T. Dietterich, "Overfitting and undercomputing in machine learning," *ACM computing surveys (CSUR)*, vol. 27, no. 3, pp. 326–327, 1995.

[13] X. Ying, "An overview of overfitting and its solutions," in *Journal of Physics: Conference Series*, vol. 1168, no. 2. IOP Publishing, 2019, p. 022022.

[14] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, 2nd ed. Cambridge, MA: MIT press, 2018.

[15] B. Recht, "A tour of reinforcement learning: The view from continuous control," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 2, no. 1, pp. 253–279, 2019. [Online]. Available: https://doi.org/10.1146/annurev-control-053018-023825

[16] M. A. et. al, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[17] F. e. a. Chollet, "Keras," 2015. [Online]. Available: https://keras.io

[18] R. Miikkulainen and et al., "Evolving deep neural network," *CoRR*, vol. abs/1703.00548, 2017. [Online]. Available: http://arxiv.org/abs/1703.00548

[19] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 06, no. 02, pp. 107–116, Apr. 1998. [Online]. Available: https://doi.org/10.1142/s0218488598000094

[20] L. Torrey and J. Shavlik, "Transfer learning," in *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI global, 2010, pp. 242–264. [Online]. Available: https://ftp.cs.wisc.edu/machine-learning/shavlik-group/torrey.handbook09.pdf

DEPARTMENT OF MECHANICS AND MARITIME SCIENCE
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se