Parameter Estimation of PID Controlled Cruise Control System

Jesper Seelk Petersen
jespe21@student.aau.dk

Tomasz Pleśniak
tplesn21@student.aau.dk

Kristian Walstrøm Petersen
kwpe18@student.aau.dk

Aalborg University
Institute of Computer Science

October 27, 2023

Contents

1 Summary

This thesis aims to assess and compare three approaches for estimating controller parameters in a PID-controlled cruise control system using observational data from the plant. The methods include numerical techniques, deep neural networks, and a combined approach.

To apply numerical solutions, the system has been formulated as a parameterized initial value problem (IVP) by integrating the plant and controller into a system of ordinary differential equations.

IVP's comprise a set of differential equations describing the system and initial conditions. A solution to an IVP is a function that satisfies these conditions and effectively describes the system's behavior over a defined interval. This solution, known as a trajectory, can be approximated using numerical methods called ODE solvers.

Single shooting methods are among these numerical solutions. However, single-shooting approaches for parameter estimation can be computationally expensive. Methods like training a deep neural network (DNN) or using numerical discretization-based estimation (DBE) from observed data can provide suitable initial parameter guesses without directly solving the IVP.

DNNs however need to be tuned and trained. If not done properly, they can suffer from various issues. One of these being saturation.

Addressing the issue of saturation can be done by normalizing the data-set used for training. Min-max normalization was chosen for this, both for the input and target, due to its ability to standardize the range and improve training stability. While normalization reduced saturation, models trained on normalized data increased the total loss.

It was also observed that the data-set used contained regions of constant features, these provide no additional information for the network to learn from. To address this, these regions were removed to emphasize the distinct characteristics of different parameter configurations.

However, issues of overfitting arose with models trained on this data. This was fixed by lowering the complexity of the DNN trained.

Finally, results showed that DNNs, when trained on normalized data with excluded regions of constant features, yielded quick results, close to the target pa-

rameters, but with higher loss than numerical methods.

Also shown is the DNNs potential as use as an initial guess estimator. While it was unable to outperform the DBE initial guess estimator on noiseless data, without further optimization, it outperformed DBE on noisy data, both in terms of accuracy and computation time.

2 Abstract

In this thesis, our objective is to investigate methodologies for predicting the parameters of a PID-controlled cruise control system. Specifically, we delve into an examination and comparison of three distinct approaches; one involving numerical solutions, the second utilizing deep neural networks (DNN), and finally a combination of the two.

We present several possible approaches for modeling the system as an ordinary differential equation (ODE), a prerequisite for utilizing numerical solutions. We outline the challenges of this process, as well as address issues unique to the explored system. We also address the critical concern of selecting an appropriate ODE solver, in addition to exploring optimization methods thereof. Additionally, we investigate hyper-parameter optimization of neural networks using Weight and Biases and delve into approaches for identifying and mitigating issues related to model saturation and overfitting.

The ultimate goal is to compare these different methods. This comparison encompasses an assessment of prediction accuracy, computation speed, and an analysis of the respective strengths and weaknesses of the different approaches, with the intent of finding the one best suited for our task.

3 Introduction

This thesis aims to address the inverse problem of estimating parameters within a cruise control system. The primary objective is to thoroughly assess and compare three different approaches for estimating the parameters of said system, with the aim of identifying the most effective solution. These approaches include employing numerical solutions and/or utilizing a well-trained deep neural network (DNN), which will be methodically compared within the scope of this task. To clarify, the task at hand is predicting the parameters of the PID controller governing the cruise control system, based on observed data, comprising an ongoing sequence of speed measurements as the cruise controller approaches its desired speed.

The motivation for this study stemmed from an initial desire to accurately predict the execution of the PID-controlled cruise control system. However, predicting parameters, particularly for non-linear systems, can pose significant challenges. As such, we set out to compare the three approaches, starting with a thorough examination of traditional and well-established numerical methods. Recognizing the criticism of high computational costs associated with numerical solutions, we expanded our investigation to include other approaches; training a DNN model to

estimate the system's parameters and a combination of the two.

The different approaches will be thoroughly explored, addressing a range of associated challenges. These include issues related to modeling a closed-loop system as an initial value problem (IVP), a prerequisite for using numerical solutions, as well as selecting appropriate ODE solvers and optimizers. Our study also tackles common neural network optimization issues like hyper-parameter tuning, saturation, and overfitting. Finally, we investigate using DNNs to provide an initial guess for the ODE solvers, aiming to reduce the computational cost of numerical solutions.

The thesis concludes with a thorough comparative analysis. This comparison focuses on the respective computational speed and predictive accuracy of the methods, in addition to a comparison of the advantages and disadvantages inherent to each.

4 Motivation

Our interest in parameter estimation was brought about by an earlier problem. That is anomaly detection in observational data from a simulation of a PID-controlled cruise control system, being subjected to neutron radiation. These anomalies could disrupt the normal behavior of the system, appearing as deviations in the trace that would have been produced by a nominal execution.

In our previous attempts, we did not find a satisfactory solution to the problem of anomaly detection in the data-set of traces produced by this neutron-radiated system. Pivoting focus to solve the issue of parameter estimation is an indirect attempt at solving the original issue of anomaly detection. The behavior of the cruise control system is deterministic, meaning that any deviation from normal behavior must be caused by an error. By estimating the parameters of the system, it would be possible to determine the nominal speed value at any given time-step. Consequently, assuming the estimations could be made accurate enough, anomalies could then be identified by how much they deviated from the traces generated by the system.

We investigated traditional parameter estimation methods, which involve modeling and defining the system as an initial value problem. However, this can be a challenging task due to the computational cost associated with the repeated simulation of the system and often requiring a good initial guess. Thus we also investigated DNN for parameter estimation, analyzing how they perform as a stand-alone solution, or as a way to mitigate the previously mentioned issues. Our reasoning

for using a DNN for this purpose is that neural networks have shown promising results with similar problems, and should be able to create a mapping of a nominal trace generated by the system, to a particular set of parameters.

5 Background

In this section, the preliminary information pertaining to the thesis will be covered.

5.1 Ordinary Differential Equation

Differential equations are mathematical equations that involve derivatives of a function with respect to one or more independent variables. In practical terms, differential equations describe how physical phenomena change over some factor like space or time. Because of their ability to model change, differential equations play a fundamental role in describing dynamic processes in various fields.

Ordinary Differential Equations (ODEs) are specific types of differential equations that deal with the functions of a single independent variable.

ODEs can be classified based on their order, which indicates the highest derivative of the function involved in the equation. The order of an ODE has a significant impact on its complexity and the methods used to solve it.

The general form of ODE n'th-order given a function F of x, y, and derivatives of y is as follows:

$$F(t, y, y^{'}, ..., y^{(n)}) = 0 \tag{1}$$

where y is a function of $t, y^{'} = dy/dt$ is the first derivative with respect to t, $y^{(n)} = d^n y/dt^n$ is the n'th derivative with respect to t.

Every higher-order ODE can be represented as a system of first-order ODEs, with the number of ODEs equal to the order of the original system. To convert an nth-order ODE into a system of first-order ODEs, new variables are introduced to represent the derivatives, such that $y_1 = y, y_2 = y', ..., y_n - 1 = y^{n-1}$. This results in the following system:

$$
\begin{aligned}
{y_1}' &= y_2 \\
{y_2}' &= y_3 \\
&\vdots \\
{y_{n-1}}' &= F(t, y_1, y_2, ..., y_{n-1}) = 0
\end{aligned}
\tag{2}
$$

The system in Equation 2 of first-order ODEs is a coupled system, meaning the derivative of each y_i depends only on the values of the variables $t, y_1, y_2, ..., y_n$.

ODEs can be solved analytically or numerically, depending on the complexity of the equation. Analytic solutions provide exact mathematical expressions for the solution function, while numerical solutions use approximation methods to obtain a numerical solution to the equation.

5.2 Control systems

Control systems are a fundamental concept in engineering, designed to manage and regulate the behavior of various systems in which the state of the system changes continuously over some factor, usually time. The general form of control systems is:

$$y' = f(y, u) \tag{3}$$

Where y' is the rate of change of the system, f is an unknown function, y is the state of the system and u is the control variables. Control variables can be state-dependent $u(y)$, time-dependent $u(t)$, or both $u(y, t)$.

Closed-loop control systems, also known as feedback control systems, are a type of control system where the output of the system is fed back as input for comparison, with the desired target state known as the set-point. The key components of a closed-loop control system are a controller and a plant.

5.2.1 PID Controller

A proportional–integral–derivative (PID) controller is a type of feedback control method. The PID controller calculates a correction to the measured process variable based on its distance from the desired set-point at each time-step. This difference between measured variable $y(t)$ and set-point y_r is referred as error signal; $e(t) = y_r - y(t)$.

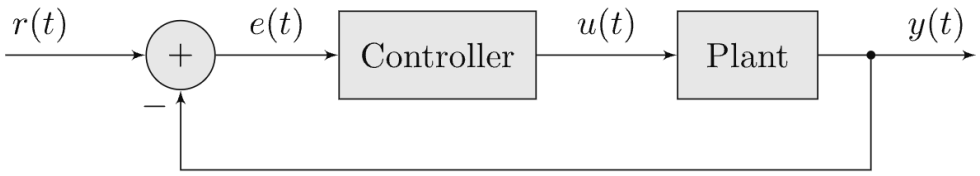


Figure 1: Block diagram of a closed-loop system. The output of the plant $y(t)$ is subtracted from the set-point $r(t)$, and this error $e(t)$ is fed to the controller, which produces the control signal $u(t)$ that is sent to the input of the plant, in an attempt to drive the error to zero. [4]

The control signal produced by a PID controller is the sum of three terms, weighted with proportional gain, integral gain, and derivative gain, which we will refer to as K_P, K_I, and K_D, respectively, from hereon. These parameters adjust the controller's response to the error signal.

The proportional term defined as $u_P = K_P \cdot e(t)$ acts as a "spring" that guides the state toward the set-point. A higher K_P value will result in a faster response.

The integral term $u_I = K_I \cdot I(t) | I(t) = \int_0^t e(\tau) d\tau$ is designed to counteract steady state errors. Steady-state error is the residual difference between the desired and actual values in a stable system under a constant input. In a cruise control system, steady-state error is the difference between the current and set-point speed once the system has settled, as indicated by the error signal.

The derivative term $u_D = K_D \cdot e'(t)$ helps the controller anticipate changes in the error signal and reduce overshooting. It is designed to accommodate second-order systems, that can oscillate under u_P and u_I unless they are dampened. The derivative term is rarely used in first-order systems particularly because of the difficulty in calculating error derivatives. Thus In first-order systems, $e'(t)$ is approximated using finite difference; $e'(t) \approx \frac{e(t) - e(t - dt)}{dt}$

These individual terms are summed for the complete control signal (correction value), as given by, as shown in Equation 4.

$$u_{PID} = K_P \cdot e + K_I \cdot I + K_D \cdot e' \tag{4}$$

5.3 ODE Solvers

ODE solvers are numerical techniques used to approximate the solutions of ODEs over a specified range of values, often referred to as the solution interval, given an initial state y_0.

They accomplish their task by first discretizing the continuous domain into a set of discrete time steps, where the smaller the step size, the more accurate the approximation, but at the cost of increased computational effort.

ODE solvers use numerical integration techniques to update the approximation of the state at each step. This process is iterative and repeats until the solver reaches the target time-step or another termination condition is met.

Because ODE solvers perform numerical integration, their general form can be represented as in Equation 5.

$$ {}_hS_{t_0}^{t_N} \approx \int_{t_0}^{t_N} f(t, y(t)) \cdot dt \tag{5} $$

Where f is an unknown function, t_0 and t_N are the beginning and target time-step of the solution interval respectively. y_0 is the initial state, and h is the step size.

5.3.1 Fixed-step Methods

Fixed-step size solvers maintain the same sized time-step through the entire integration process, meaning that the trade-off between speed and accuracy of the estimation is user-specified.

5.3.1.1 Euler's Method

The simplest ODE solver called forward Euler is based on a first-order approximation. Its definition can be seen in Equation 6.

$$ y_{n+h} = y_n + f(t_n, y_n) \cdot h \tag{6} $$

Euler's method approximates the integral by starting in state $y_0 = f(t_0)$ and repeatedly applying the rule Equation 7 N times.

$$ y_{i+1} \leftarrow y_i + f(t_i, y_i) \cdot h \tag{7} $$

This procedure gives a sequence $y_{0:N}$, in which each point y_i approximates the $y(t_0 + I \cdot h)$, and $y_n \approx y(t_f)$.

Because Euler's method only considers a single gradient, it produces an inaccurate estimation unless used with extremely small time-steps, which adds to the computational expense. Thus it is rarely used when accurate approximations are needed.

5.3.1.2 Runge–Kutta 4 Method

The Runge-Kutta 4th order method (RK4) is a more elaborate method than Euler. RK4 evaluates the function four times at each time-step, meaning that its approximation is more accurate although four times slower than Euler given the same step size.

It can be defined as in Equation 8.

$$\begin{aligned} y_{n+1} &= y_n + \frac{dt}{6} \cdot (k_1 + 2k_2 + 2k_3 + k_4) \\ k_1 &= f(t_n, y_n), \\ k_2 &= f(t_n + \frac{dt}{2}, y_n + dt \cdot \frac{k_1}{2}) \\ k_3 &= f(t_n + \frac{dt}{2}, y_n + dt \cdot \frac{k_2}{2}) \\ k_4 &= f(t_n + dt, y_n + dt \cdot k_3) \end{aligned} \tag{8}$$

5.3.2 Adaptive-step Methods

Adaptive-step solvers change the step size based on the estimated rate of change of derivatives in the region surrounding the current state. In particular, when the rates change slowly, adaptive-step solvers take larger steps speeding up calculations. On the other hand, when the rates change quickly those solvers decrease the size of their steps, achieving accuracy that could be only achieved with a small fixed step size. Because of these properties, adaptive-step solvers maintain a balance between speed and accuracy.

5.4 Optimization

One of the primary goals in optimization is to find a local minimum or maximum of a given objective function. For example, if the objective function represents a cost, then the aim is to find the value x which will yield the lowest possible cost. A general form of a minimization problem can be represented as in Equation 9.

$$\min_y f(y) \tag{9}$$

5.4.1 Gradient Descent

The idea behind Gradient Descent (GD) is to repeatedly calculate the gradient of the loss function with respect to the parameters of the function being optimized,

and then perform a small step in the direction of the greatest descent until either a global or local minimum is reached.

The update step for GD can be formulated as in Equation 10.

$$y_{y+1} = y_i - \gamma \cdot \nabla f(y_i) \tag{10}$$

Where γ represents a hyper-parameter called the learning rate, and $\nabla f(y)$ represents the gradient of an objective function f evaluated at y_i.

In many cases, it is impractical to use the entire data-set to calculate the gradient, often only randomly selected mini-batches are employed. This approach is referred to as stochastic gradient descent (SGD).

Because GD uses only first-order information, it necessitates small step sizes at each iteration so as not to step over a possible minimum, which in turn results in a relatively slow convergence speed. While having limited local information is beneficial when optimizing a large number of parameters, as encountered in neural networks, it becomes impractical in situations where the number of parameters is small.

5.4.1.1 Adam

The Adam algorithm [20] is an extended version of SGD as it combines ideas from momentum and the Root Mean Square Propagation (RMSprop) algorithms. It uses the squared gradients to scale the learning rate like RMSprop and it takes advantage of momentum by using the moving average of the gradient to accelerate the convergence towards the minimum. It can be represented by following pseudo-code Figure 2.

5.4.2 Newton's method

Newton's method, also called Newton–Raphson, is an unbound iterative optimization algorithm. Newton-Raphson uses both first and second-order information, specifically the gradient and Hessian matrix to create a local quadratic approximation of the objective function given some specific value. Then it performs a step in the approximated function to acquire the next value. This process repeats until the approximated minimum is reached.

The update step for Newton–Raphson can be formulated as in Equation 11.

$$y_{y+1} = y_i - \gamma \cdot H_i^{-1} \cdot \nabla f(y_i) \tag{11}$$

Algorithm 1 Adam

Require: α: step size
Require: $\beta_1, \beta_2 \in [0, 1)$: exponential decay rates for the moment estimates
Require: f: (stochastic) objective function
Require: x_0: initial parameter guess

$m_0 \leftarrow 0$ ▷ Initialize 1st moment vector
$v_0 \leftarrow 0$ ▷ Initialize 2nd moment vector
$t \leftarrow 0$ ▷ Initialize timestep
while x_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow (D\ f)(x_{t-1})$ ▷ Get gradients with respect to objective at timestep t
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ ▷ Update biased first moment estimate
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot (g_t \odot g_t)$ ▷ Update biased second raw moment estimate
    $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ ▷ Compute bias-corrected first moment estimate
    $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ ▷ Compute bias-corrected second raw moment estimate
    $x_t \leftarrow x_{t-1} - \eta \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)$ ▷ Update parameters
end while
return x_t

Figure 2: Adam optimizer pseudo-code [14]

Where H_i^{-1} represents the inverse of the Hessian matrix, $\nabla f(y)$ represents the gradient of an objective function f evaluated at y_i, and γ represents the learning rate.

Traditionally, the learning rate γ is set to 1, however, when dealing with non-convex functions, ill-conditioned problems, or when the Hessian matrix is not positive definite, line search algorithms are used to adjust the learning rate.

The benefit of using Newton's method is that it results in fast convergence. Additionally, Newton-Raphson exhibits quadratic convergence properties, meaning that the number of correct digits in the solution approximately doubles with each iteration when near the optimal solution.

The downsides of using Newton-Raphson are its sensitivity to initial guesses and the high computational cost of Hessian inversion.

5.4.3 BFGS

BFGS (Broyden-Fletcher-Goldfarb-Shanno) is an optimization algorithm used for unconstrained optimization problems. It belongs to the family of the quasi-Newton methods. These methods are designed to overcome the computational challenges associated with computing and storing the full Hessian matrix. Instead of computing the Hessian directly, quasi-Newton methods iteratively update an approximation to the inverse Hessian matrix at the end of each iteration.

The downside of BFGS is that it maintains and updates a full approximation to the inverse Hessian matrix at each iteration.

5.4.3.1 L-BFGS

L-BFGS (Limited-memory Broyden-Fletcher-Goldfarb-Shanno) is a modification of the BFGS method designed to address memory limitations associated with large-scale optimization problems. Instead of storing and updating the full inverse Hessian matrix L-BFGS maintains a limited memory history of previous gradients and parameter updates used for computation. The number of historical values is decided by a hyper-parameter.

5.4.4 Levenberg–Marquardt

The Levenberg–Marquardt (LM) algorithm merges elements from both gradient descent and the Gauss-Newton method. It introduces a damping parameter that governs the choice of optimization strategy. When the damping parameter λ is small, the algorithm applies a Gauss-Newton update, while larger values of λ prompt a shift toward gradient descent. Initially, the damping parameter λ is set to a large value, causing the initial updates to be conservative, taking small steps along the steepest-descent direction.

5.5 Adjoint Method

The adjoint sensitivity method, also known as the adjoint method, is a mathematical technique used to compute gradients of an objective function with respect to the parameters in systems governed by differential equations, particularly ODEs. It calculates gradients by using the state and the sensitivity of the loss with respect to said state, and then solving the augmented ODE backward in time. An augmented ODE is an extension of the original ODE that incorporates additional state variables. These additional variables are introduced to help with the sensitivity analysis and parameter estimation problems.

The pseudo-code for computing gradients using the adjoint method can be seen in Figure 3:

5.6 Deep Neural-Networks

A deep neural network (DNN) is a type of neural network (NN) that has one or more hidden layers between the input and output layers. The term "deep" refers to the presence of these multiple layers.

A NN can be represented as a tuple $N = (L, T, F)$, where:

Algorithm 1 Reverse-mode derivative of an ODE initial value problem

Input: dynamics parameters θ, start time t_0, stop time t_1, final state $\mathbf{z}(t_1)$, loss gradient $\partial L / \partial \mathbf{z}(t_1)$

$s_0 = [\mathbf{z}(t_1), \frac{\partial L}{\partial \mathbf{z}(t_1)}, \mathbf{0}_{|\theta|}]$ ▷ Define initial augmented state

def aug_dynamics($[\mathbf{z}(t), \mathbf{a}(t), \cdot], t, \theta$): ▷ Define dynamics on augmented state

  return $[f(\mathbf{z}(t), t, \theta), -\mathbf{a}(t)^\mathsf{T} \frac{\partial f}{\partial \mathbf{z}}, -\mathbf{a}(t)^\mathsf{T} \frac{\partial f}{\partial \theta}]$ ▷ Compute vector-Jacobian products

$[\mathbf{z}(t_0), \frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}] = \text{ODESolve}(s_0, \text{aug\_dynamics}, t_1, t_0, \theta)$ ▷ Solve reverse-time ODE

return $\frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}$ ▷ Return gradients

Figure 3: Adjoint Method pseudo-code [8]

$$L = L_k \mid k \in 0, \ldots, K$$

$$T \subseteq L \times L$$

$$F = F_k \mid k \in 1, \ldots, K$$

L is a set of layers. T is a set of connections between each of the fully connected layers and F a set of functions, one for each non-input layer. In a NN, the first layer L_0 is the input layer, the last layer L_K is the output layer, and all layers in between are called hidden layers. Each layer L_k consists of n_k neurons, also known as nodes.

5.7 Weights & Biases

To aid in the comparison and optimization of hyper-parameters, "Weights & Biases" was used. Weights & Biases is among other things a logging and visualization tool specifically built for machine learning problems. [31]

For this thesis, the "sweep" functionality of Weights & Biases has been utilized in hyper-parameter optimization. A "sweep" is a term used for describing a series of experiments, in which a number of models with different hyper-parameters are compared. In the Weights & Biases framework, a sweep works by defining several ranges or sets of values. Hyper-parameters are then selected from the sets and/or ranges and the results from the resulting model are then logged. The hyper-parameter selections is made either at random, sequentially going through every possible combination, or selected through Bayesian search. After training a series of models, Weights & Biases aids in selecting optimal hyper-parameters by visualizing the results.

5.8 Normalization

Normalization refers to the process of adjusting the scale of a variable or a data-set to bring it within a specific range or standardize it in a way that facilitates comparison or improves convergence in certain algorithms. It is often beneficial for training neural networks, as it can add stability during training by reducing the effects of outliers. It can also provide consistency and comparability, by scaling features to have the same range, ensuring features with larger numerical values won't dominate the loss when training a model. [2, 18]

Two of the most common normalization methods are 'min-max' normalization, often just referred to as "normalization", and "ZScore" normalization, also referred to as "standardization". Min-max normalization scales a set of values to be between 0 and 1, or -1 and 1. The function for min-max normalization can be seen in Equation 12.

$$f(x) = \frac{x - min}{max - min} \tag{12}$$

Where x is the value being scaled, min is the minimum value in the set and max the maximum value in the set. Min-max can help deal with outliers and extreme values, in addition to scaling all features to the same range to reduce biases during model training.

ZScore normalization scales a set of values such that the mean is 0 and the standard deviation is 1. The function for ZScore normalization can be seen in Equation 13.

$$f(x) = \frac{x - \mu}{\sigma} \tag{13}$$

Where x is the value being scaled, μ is the arithmetic mean of the set, and σ is the population standard deviation of the set. Scaling in this manner can benefit data analysis, making it simpler to understand the values and their relative deviation within the set, which can accelerate training.

In general, there are three different approaches to applying normalization; feature, sample, and global normalization. Sample normalization involves scaling each individual sample in a data-set independently. The normalization is applied across the features for a specific sample. This means that each sample has its own scale based on its characteristics.

Feature normalization involves scaling the values of each feature in the data-set independently. For each feature, the normalization is applied across all samples.

This ensures that each feature contributes to the learning process equally, regardless of its original scale.

Global normalization applies a common scale factor to the entire data-set. The transformation is performed collectively across all samples and features, taking into account the combined values of the entire data-set. This ensures that all features and samples share a similar scale.

6 Problem definition

Given observational data from the plant of a PID-controlled cruise control system, the aim of this thesis is to examine and compare the use of numerical methods, deep neural networks (DNNs) and a combined approach for the task of estimating controller parameters, namely proportional gain K_P, integral gain K_I, and derivative gain K_D. The comparison will be based on the quantitative measurements of the loss on each parameter, as well as computation speed.

In order to employ numerical solutions, a prerequisite is to formulate the system as a parameterized Initial Value Problem (IVP), which requires combining the plant and the controller into a system of ordinary differential equations (ODE).

7 Related Work

The literature on solving the initial value problem (IVP) for models of ordinary differential equations spans various scientific communities and approaches. Generally, approaches for solving an IVP can be divided, among others, into numerical and analytical.

Many of the most popular numerical techniques for solving IVPs are explored in detail by Jonathan Claver in [7] and Howe, Nikolaus Harry Reginald in [15]. .

Numerical methods for parameter estimation in ODEs often focus on approximating the solution of differential equations by simulating the model trajectories, however, since this is done with each iteration of the optimization process they often have high computational cost. These methods are referred to as shooting approaches and have been extensively studied in works such as [26], [5], [28] and [25]. Moreover, shooting approaches rely heavily on initial guesses, especially when combined with gradient-based optimizers. For that purpose, it is common to use methods for estimating parameters without the need to simulate the solution trajectory. Methods for attaining an initial guess commonly involve matching the approximating gradients derived from the observed data using finite differences to the gradients obtained from the ODE when using observed data as a state input. These methods are presented in works such as [30], [24], and [32].

Neural networks can also be used for the task of estimating the parameters of ODEs. Vivek Dua in [10] decomposes the problem of parameter estimation into two sub-problems. The first sub-problem creates a neural network that maps the relationship between time-steps and the state of the system. The second sub-problem uses the neural network model to obtain an estimate of the parameters which can later be used as an initial guess for a numerical solution. Another two-stage solution for parameter estimation was proposed in [6], where neural ordinary differential equations are used to estimate state derivatives, which are then used to estimate the parameters of a more interpretable model. In [17] Vivek Dua and Elnaz Jamili propose a technique where the approximation of the state variable and model parameters are done simultaneously. .

Other notable work includes the approximation of reconstruction maps for model parameter estimation proposed as in [29].

DB Column	timestamp	can_id	high_word	low_word	flux	mem_sum	off_nominal
Description	Unix timestamp from logging computer when the CAN message was logged	Descriptive name associated with ID field of CAN packet	Contents of high 32-bits of CAN data as signed int	Contents of low 32-bits of CAN data as signed int	Neutrons / cm^2 estimate calculated once per second	Sum of byte array in program memory used to increase sensitivity to soft errors	1 = behavior has differed from the nominal run 0 = behavior has been identical to the nominal run
Type	numeric	char	int	int	int	int	Boolean (1 or 0)

Figure 4: Table schema

8 Data-set

The data-set consists of time-series observational data collected from a simulated cruise control system, where the system's speed is controlled by a PID controller. In this section, we will provide a brief overview of the data-set and its structure.

Each simulation had the aforementioned PID controller and an associated plant that exchanged messages with each other. The simulations ran for 15 seconds, and the communication between the controller and plant was synchronized to ensure determinism in the order of packets. Additionally, the experiments were deterministic in their execution, given a set of parameters, meaning that if a set of simulations shared parameters, their execution would be identical.

The data was stored in tables according to the schema shown in Figure 4. The values stored in certain columns depended on the value of the can_id column, as explained in Figure 5. The trace from a simulation was stored in separate tables and saved to a file for later analysis. Additionally, each simulation was associated with a unique ID, which consisted of six parts that identified the parameter values of the simulation in question, following the schema shown in Figure 6. Only nominal files are used in this thesis. As mentioned in section 4, the original problem involved anomaly detection, which is why there was a need to denote whether data was anomalous or not.

We focused on the entries in the tables with the can_id of PLANT_INFO_VehicleSpeed and used the high_word entry, which represents the velocity. The order of the time-steps, in the time-series, are determined based on the sequence, and each subsequent data point records the velocity at 0.01-second intervals following the preceding one.

Hex ID	Dec ID	Descriptive Name in DB	Sender	Purpose	High Word Meaning	Low Word Meaning
100	256	PLANT_CMD_ResetController	Plant	Reset the Controller when the Plant is initialized	NA	NA
130	304	PLANT_INFO_SimulationTime	Plant	Report the simulation time steps, beginning at zero	Simulation time in "steps"	NA
132	306	PLANT_INFO_VehicleSpeed	Plant	Report the current speed from the Plant (vehicle)	Current velocity	NA
200	512	CTRL_STATUS_Wakeup	Controller	Report that the Controller has started	NA	NA
210	528	CTRL_INFO_TorqueCommand	Controller	The controller command to set the torque	Calculated torque value	NA
211	529	CTRL_INFO_VehSpd_ECHO	Controller	Echo back the speed used in the torque calculation	Current velocity	NA
220	544	CTRL_STATUS_ResetByPlant	Controller	Report 0x100 message received, Controller reset	NA	NA
230	560	CONFIG_PARAMS	Command PC	Send run configuration parameters to both Plant and Controller	Byte 3: Kp Byte 2: Ki Byte 1: Kd Byte 0: Sp	NA

Figure 5: can_id meanings

	kp<X>	ki<X>	kd<X>	sp<X>	nominal I experiment	<N>
Description	<X> = Kp PID parameter	<X> = Ki PID parameter	<X> = Kd PID parameter	<X> = initial speed	nominal = beam off experiment = beam on	<N> = count of the nominal or experiment run
Range	100 <= X <= 1000	2 <= X <= 15	1 <= X <= 3	0 <= X <= 20		N >= 0

Figure 6: Simulation id schema

9 Noise Generation

During the experiments, the examination was expanded to include an investigation into the impact of noise on the loss of the various methods. The cruise control system however, as previously mentioned in section 8, is deterministic producing no noise. Consequently, in order to examine the effect of noise, said noise had to be introduced. The decision was made to introduce it artificially.

An advantage of artificially created noise is knowing the distribution of said noise. This eliminates the need to make assumptions or approximations of the real noise distribution. With this knowledge, we can easily compare the effect of different noise distributions on our prediction accuracy.

The choice was made to introduce the noise into the system during its simulation. The system consists of a plant, the simulated vehicle, and the PID controller. The noise is introduced at the plant when the current velocity of the vehicle is updated and relaid back to the controller, the reason for introducing the noise here is to simulate an inaccurate sensor. The noise introduced is a fluctuating error, a changing variable, it modifies velocity, and the amount is chosen at random within a specified range at each time-step. The formula for the noisy velocity calculation is described in Equation 14.

$$f(y) = y + \mathcal{N}(0, E) \tag{14}$$

Where $f()$ is the function that generates the noisy velocity and y represents the true velocity. $\mathcal{N}$ represents a normal distribution, where "0" is the mean and E, representing the fluctuating error, is the standard deviation of the distribution.

After the erroneous velocity has been calculated it is sent to the controller, which then uses the noisy result for its calculations. These simulations are then used to generate new traces, which can then be used as a data-set for testing and evaluation.

By introducing the noise during the simulation, rather then simply modifying the existing trace data, we are able to achieve more 'real-to-life' data, or 'to-simulation' in our case. The reason being that the controller output will be influenced by the result of the previous time-step, which was likewise effected by the noise. This results in the errors of previous steps compounding and effecting one another, unlike if it had been introduced after the fact.

10 Evaluation Metrics

We evaluate the performance of the DNN and numerical method predictions through quantitative methods, calculating the loss of each parameter separately. We have chosen a straightforward approach, evaluating the absolute differences. We deliberately decided not to use methods that combined the results into a single value. Our intention here was to preserve as much information as possible for potential model evaluation. The issue with a combined value is models with different prediction losses on individual parameters may have an identical combined loss. Consequently, distinguishing between these two models becomes impossible, even if they have notable differences. The equation for calculating the loss can be seen on Equation 15.

$$|P_{pred} - P_{target}| = P_{loss} \qquad (15)$$

$$\left| \begin{bmatrix} 200 \\ 5 \\ 2 \end{bmatrix} - \begin{bmatrix} 230 \\ 3 \\ 2 \end{bmatrix} \right| = \begin{bmatrix} 30 \\ 2 \\ 0 \end{bmatrix} \qquad (16)$$

Where P_{pred} and P_{target} are vectors containing the prediction and targets respectively, both include values corresponding to the K_P, K_I and K_D parameters. P_{loss} is a vector of absolute differences for each parameter. An example application can be seen on Equation 16.

In addition to the absolute difference another metric we will be using is the population standard deviation. This metric is useful when doing quantitative testing as it allows a measure of the general dispersion and deviation between multiple measurements. Standard deviation is calculated as seen in Equation 17.

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2} \qquad (17)$$

Where σ is the standard deviation, x_i represents an individual data-point, μ is the arithmetic mean and N is the total number of data-points in the population.

As the final measuring metric used in this thesis is the evaluation time, representing the mean time it takes to make a single prediction based on a set of observations. These time measurements will be performed on the same computer to have a consistent testing environment. These measurements will be performed on a "MacBook Air M1, 2020", 3.2 GHz CPU, 1.28 GHz GPU, 8gb of ram and 256gb SSD storage.

11 Initial Value Problem

In this thesis, for our numerical solutions, we consider the model of the system formulated as a parameterized initial value problem (IVP). The IVP consists of model derivatives specified by a system of ODEs, namely the PID-controlled cruise control system and initial conditions specifying the initial state of the model as defined in Equation 18.

$$\begin{aligned} y'(t) &= f(t, y(t), p) \\ y(0) &= y_0 \\ t &\in (0, T) \end{aligned} \tag{18}$$

Where $y(t)$ is the state vector of a single dimension, containing the velocity at a given time-step. t is a time-step in T, where T is the set of all time-steps. p is a constant vector of model parameters K_P, K_I, and K_D. y_0 is the initial condition; the velocity at time t_0, and f is the cruise control model responsible for generating model derivatives of a system at time-step t.

The goal of solving a parameterized IVP is to find a function that satisfies the differential equation while taking into account the specified parameters and initial conditions. In simpler terms, an IVP aims to find the function that describes how a system behaves over a given interval, starting from an initial point with specific parameter values. This result, known as a trajectory, can be approximated by utilizing numerical methods called ODE solvers, which simulate the system's state from the initial time t_0 to a designated final time.

For this problem, we know the initial state as we are given a trace. Therefore, the solution to the IVP essentially equates to determining the correct model parameters so that the simulated trajectory from the known starting time to the final time closely matches the observed trajectory of the original system.

12 System modeling

As mentioned in section 11 in order to solve the parameterized IVP, we need to define the cruise control system as an ODE. In order to accomplish this the plant and the controller must be combined. Such combination results in the first-order ODE seen in Equation 19.

$$\begin{aligned} y' &= \begin{cases} \frac{K_P \cdot e + K_I \cdot \int e + K_D \cdot e' - CdA \cdot y^2}{m} & \text{if } y > 0 \\ \frac{K_P \cdot e + K_I \cdot \int e + K_D \cdot e' + CdA \cdot y^2}{m} & \text{otherwise} \end{cases} \\ e &= y_r - \lfloor y \rfloor \end{aligned} \tag{19}$$

Where CdA is a static value representing opposing forces affecting the vehicle, and m is the static mass of the vehicle. e is a control error calculated as the difference between set-point y_r and floored velocity $\lfloor y \rfloor$ at time-step t. Flooring the velocity $\lfloor y \rfloor$ when calculating the error e is a feature in the system presented in this thesis, as in the original implementation of the cruise control system there were identified bugs with floating point truncation. Usually, the error is calculated by $e = y_r - y$, but because the goal was to model the system that produced the observed data, we included it in the model definition.

In first-order systems, the integral of the error is usually approximated using Riemann sum e.q $\int_0^t e(t) \approx \sum_{n=1}^{t} e(t) \cdot dt$ and the derivative of the error is approximated by finite difference $e'(t) \approx \frac{e(t)-e(t-dt)}{dt}$, where dt represents the step-size. Approximating those terms requires in-memory storage of past values and knowledge of the step-size at each integration.

When simulating the change in velocity of a PID-controlled cruise control system, the choice of an appropriate numerical solver plays a critical role in achieving accurate trajectory approximation. For the purpose of choosing the most suitable one for our task, we tested a selection of first-order, higher-order, and adaptive-step solvers, namely Euler, RK4, and dopri5 [9] respectively. To evaluate the performance of each, we simulated 100 trajectories with different initial conditions and parameters in order to compare those trajectories to the ones approximated using said solvers. As a metric of accuracy, we used the mean squared error (MSE) that measures the difference between approximated trajectories and the target trajectory. The results show that Euler's method is the most reliable choice, as it consistently provides accurate and stable results. We hypothesized that other solvers introduce errors and instability due to the fact that the derivative and integral terms are calculated using past values stored in memory. This could significantly influence the trajectory accuracy. These deviations could occur because the more complex solvers are sensitive to the influence of past values, leading to deviations from the target trajectory. In contrast, Euler's method maintains precision when past values are integral to the approximation, perhaps due to its relative simplicity. To test that hypothesis, we repeated the initial experiments with different combinations of proportional, integral, and derivative terms, as integral and derivative terms are influenced by past values contrary to the proportional term. The purpose of the test was to determine which terms causes the inaccuracies seen in the more complex solvers. The results shown in Table 1 indicate that the discrepancy in the approximation issue is mainly caused by the integral term, however, the derivative term also contributes to it, which aligns with the hypothesis that it is the past values causing the complex solvers to become inaccurate.

Mean MSE for trajectory comparison			
Metrics	Euler	RK4	Dopri5
Mean MSE u_{PID}	0.309 {0.039}	18354.199 {26145.486}	3966.444 {8186.335}
Mean MSE u_P	0.331 {0.013}	0.294 {0.138}	0.294 {0.138}
Mean MSE u_{PI}	0.300 {0.038}	18144.556 {27817.927}	18144.556 {27817.927}
Mean MSE u_{PD}	0.346 {0.013}	0.476 {0.197}	0.476 {0.197}

Table 1: Mean MSE for trajectory comparison

In order to avoid the degradation of accuracy in complex solvers caused by past values, attemps were made to model the system in such a way, as to avoid storing previous values in memory for calculating the derivative and integral terms, so that the complex solvers could potentially be used for accurate trajectory approximation. Modeling the system as such was initially attempted by increasing the order of the system by taking the derivatives of both sides, a popular technique in stability analysis. To do this, it is essential that the set-point y_r lies at the origin $y_r = 0$ before the derivative increases. After the increase in derivatives, we ended up with the system shown in Equation 20

$$y'' = \begin{cases} \frac{K_P \cdot e' + K_I \cdot e + K_D \cdot e'' - 2CdA \cdot y' y}{m} & \text{if } y > -y_r \\ \frac{K_P \cdot e' + K_I \cdot e + K_D \cdot e'' + 2CdA \cdot y' y}{m} & \text{otherwise} \end{cases} \tag{20}$$
$$e' = -\lfloor y' \rfloor$$

The idea behind this technique is that by increasing the derivatives, the integration required for calculating the integral term is eliminated, as now it can be approximated by e. Equation 20 also shows that despite transformation there is still a need to approximate the derivative term using finite difference, as solving e'' requires knowledge thereof, thus only partially eliminating the models dependency on past values for calculation. If the order of the system was not equal to the order of the derivative term, this would not be the case. This is one of the reasons why most of the first-order systems are controlled by a PI controller without the derivative term.

When comparing the trajectory of the second-order model with the re-centered target trajectory, there is a significant difference between the two. This discrepancy is caused by the previously mentioned flooring of the velocity. This comparison be seen in Figure 7.

A different approach was attempted to eliminate the need for storing past values, achieved by augmenting the state of the ODE. This is done by adding additional variables to the state of the ODE. The augmented variables are integrated alongside the standard state variables. In the case of Equation 19, one can include

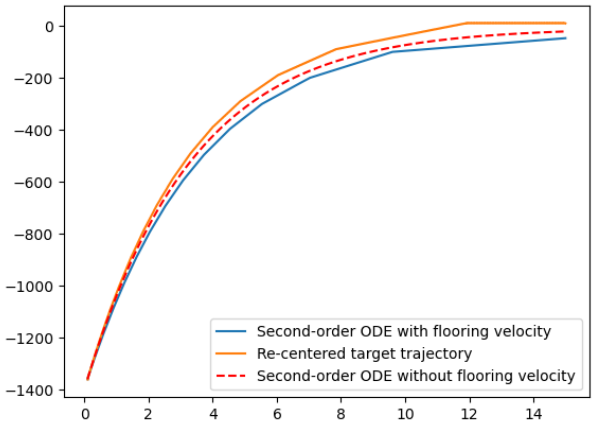


Figure 7: Comparison between re-centered target trajectory and one approximated by higher-order ODE with and without floored velocity

the error variable in the state to calculate the integral term at time-step. This augmentation could be thought of as increasing the order of the system. Its application can be seen in Equation 21.

$$y' = \begin{cases} \frac{K_P \cdot e + K_I \cdot E + K_D \cdot e' - CdA \cdot y^2}{m} & \text{if } y > 0 \\ \frac{K_P \cdot e + K_I \cdot E + K_D \cdot e' + CdA \cdot y^2}{m} & \text{otherwise} \end{cases} \tag{21}$$
$$E' = e$$

Using this technique the integral terms reliance on past values can be eliminated as its derivative e is known at each state. However the derivative term is still reliant on knowledge of the step-size and past values.

The results of testing this model against 100 simulated trajectories can be seen in Table 2. As shown by the results, this model achieves lesser accuracy than Equation 19 for all of the solvers.

Mean MSE for trajectory comparison using augmented state model			
Metrics	Euler	RK4	Dopri5
Mean MSE u_{PID}	567.579 {765.697}	559.840 {762.847}	560.0254 {762.58136}

Table 2: Mean MSE for trajectory comparison using augmented state model

As the attempts to eliminate the models dependency on past values did not yield better results, the decision was made to use the first-order ODE model, Equation 19, with Euler solver.

13 Single Shooting

This section explores the application of single shooting methods [3], as a numerical solution, to finding the set of system model parameters such that the solution of the IVP best fits the observed data, as defined by an appropriate objective function. We explain the idea behind the Incremental Single Shooting (ISS) [25] approach. We emphasize the importance of careful consideration of initial parameter estimation, optimization algorithms, and loss functions. Additionally, highlighting the challenges associated with the adjoint method when working with systems where historical values play a significant role. After considering the results in this section, we conclude that the best-explored solution for estimating parameters in the PID-controlled cruise control would be to utilize the Levenberg-Marquardt optimization algorithm, with the initial guess calculated by the Numerical Discretization-based Estimation.

Single shooting methods are iterative techniques. In each iteration of the optimization process, ODE solvers are utilized to estimate the solution of the IVP. This estimation involves the initial state and a predicted set of parameters across the entire interval of interest. The obtained solution, along with observed data, is then used to evaluate the objective function. In this thesis, we employed a variant of the Single Shooting method known as Incremental Single Shooting (ISS). The fundamental concept behind ISS revolves around the iterative adjustment of system parameters while assessing the system's anticipated behavior against the observations. In simpler terms, the trajectory interval is divided into smaller sub-intervals, with each one approximated sequentially while optimizing the same set of parameters. ISS offers the distinct advantage of updating parameters in an incremental manner. Instead of making abrupt, disruptive changes to the control system, it facilitates a gradual, more stable transition. In our specific case, ISS can acquire knowledge of the parameters before the control system reaches its set-point, generating data traces that significantly influence the optimization process.

Parameter estimation in systems of ODEs is commonly done using maximum likelihood estimation (MLE) [11]. This is because MLE tries to maximize the likelihood of parameters resolving a case, which is particularly useful when the estimated trajectory does not fit the observed data exactly. However, to perform MLE one needs to know or make assumptions about the probability distribution of the data.

Given observed data and assuming a normal distribution with the same variance across all errors, parameter estimation for ODEs, be represented as non-linear least squares represented problem as Equation 22.

$$\min_{p} \sum_{i=1}^{n} (\hat{y}(t_i) - y(t_i, p))^2, \text{ such that y(t,p) satisfies Equation 18} \tag{22}$$

Where n is the number of data points in the trajectory, $\hat{y}(t_i)$ is the observed velocity at time t_i, $y(t_i)$ is the predicted trajectory at time t_i with parameters p.

For solving the problem as defined in Equation 22 with a single shooting approach, we define the objective function as the sum of squared errors:

$$O(p) = \sum_{i=1}^{n} (\hat{y}(t_i) - y(t_i, p))^2 \tag{23}$$

13.1 Initial Guess

A common criticism of estimating parameters using a single-shooting approach, especially in combination with gradient-based optimizers, is that repeated simulation of the system's trajectory is computationally expensive, especially if the ODE is complex. In such cases, optimization tends to diverge or converge to local optima without sufficiently good initial guesses p_0, thus methods that use observed data to obtain suitable p_0 without approximating the solution of the IVP can be employed.

One way to obtain a good initial guess is to train an NN to estimate parameters directly from the observations. Another set of methods is based on fitting approximated derivatives from the observed data, usually by finite difference to the derivatives generated directly by the ODE model using observed values as state inputs. Numerical discretization-based estimation (DBE) [32] is one such method and it can be formulated as the following least squares objective function Equation 24.

$$\min_{p} \sum_{i=1}^{n} (\frac{\hat{y}(t_{i+1}) - \hat{y}(t_i)}{t_{i+1} - t_i} - f(t_i, \hat{y}(t_i), p))^2 \tag{24}$$

Where $\hat{y}(t_{i+1}$ and $\hat{y}(t_{i+1}$ are observed states at time t_{i+1} and t_i respectively. f is the ODE function and p is the set of parameters of said function.

To test the DBE's application in finding a good initial guess for the PID-controlled cruise control system, we conducted two experiments, one involving noise-free data and the other data with normally distributed noise with a standard deviation of 10. For both experiments, we used Levenberg-Marquardt (LM) optimizer with

the initial values set to 1 for each parameter. the results of those experiments can be seen in Table 3 and Table 4.

Initial guess using DBE	
Metrics	DBE with LM
Mean Iterations	11.000 {0.000}
Mean Time [s]	0.058 {0.004}
Mean distance K_P	2.433 {3.491}
Mean distance K_I	0.099 {0.140}
Mean distance K_D	0.564 {0.298}

Table 3: Results of parameter estimation with DBE and LM

Results of parameter estimation with DBE and LM with noise	
Metrics	DBE with LM
Mean Iterations	11.000 {0.000}
Mean Time [s]	0.061 {0.013}
Mean distance K_P	124.130 {71.892}
Mean distance K_I	3.981 {2.554}
Mean distance K_D	66.546 {18.671}

Table 4: Results of parameter estimation with DBE and LM with noise

13.2 Optimization Algorithms

The choice of the optimization algorithm significantly impacts the precision, efficiency, and convergence of the parameter estimation process. Given the nature of the IVP and the associated system, four specific optimization algorithms were selected for comparison: Adam [19], L-BFGS [22], Newton-Raphson, and Levenberg-Marquardt [21]. This section explains the reasoning behind this selection.

Adam is chosen for its adaptability and efficiency in handling noisy data. The optimizer's adaptive learning rates adjust individually to each parameter which aligns with the adaptive nature of the IVP and the need for robustness.

L-BFGS is included for its memory-efficient quasi-Newton approach, finding a balance between computational efficiency and convergence speed.

The classical Newton-Raphson method is selected for its potential rapid convergence when the initial guess is sufficiently close to the optimal parameters.

Levenberg-Marquardt is chosen for its balance between gradient descent and Newton-Raphson methods, offering versatility and stability. Its damping factor adapts the optimization process based on the curvature of the objective function, making it suitable for non-linear least squares problems.

This diversity allows for a comprehensive exploration of the parameter space.

13.2.1 Optimization Experiments

For each optimizer, two experiments were performed, one with noiseless data and one with normally distributed noise with a standard deviation of 10. Each experiment was conducted over a set of randomly selected trajectories from the data-set. The initial value for the parameters was set to the mean value of their respective ranges.

We used the PyTorch implementation of Adam. We set the number of epochs to 1000, with early stopping, and terminating optimization if the absolute difference between five consecutive losses was less than 5-4e. The initial learning-rate was set to 5, multiplied by 0.1 every 100 epochs to address slow convergence.

Newton-Raphson was implemented using PyTorch-minimize, with the learning-rate set to 1.0 as per default, and the maximum number of epochs set to 50.

We also used the L-BFGS implementation from the PyTorch library. We set the learning-rate to 1.0 and the maximum number of epochs to 50 as with Newton-Raphson.

Levenberg-Marquardt was implemented using the Scipy library without changing any of its default parameters.

For experiments with PyTorch and PyTorch-minimize, the sum of squared errors was used as an objective function. Moreover, for all experiments, we used Euler's method with a step-size set to 0.01 as that was the step-size used to generate target trajectories.

As can be seen from the results Table 5, Table 6, Table 7 and Table 8, all of the optimizers performed well in both scenarios.

The only outlier in terms of the loss of the predicted parameter was the Adam optimizer. It is possible that the performance of Adam could be further adjusted by fine-tuning the learning-rate, amongst others, but at the potential cost of increasing the computational cost. Further analyzing the data provided by experiments with Adam we yield several conclusions. Firstly, the computational cost for convergence is significant, as indicated by the high average number of iterations. Secondly, this

Results of solving IVP using l-bfgs and Newton-Raphson		
Metrics	L-BFGS	Newton-Raphson
Mean Iterations	4.800 {3.765}	6.811 {5.699}
Mean Time [s]	28.940 {6.465}	44.323 {9.047}
Mean distance K_P	0.854 {0.585}	0.840 {0.736}
Mean distance K_I	0.144 {0.177}	0.250 {0.354}
Mean distance K_D	0.567 {0.134}	0.680 {0.296}

Table 5: Results of solving IVP using L-BFGS and Newton-Raphson

Results of solving IVP using Adam and Levenberg-Marquardt		
Metrics	Adam	Levenberg-Marquardt
Mean Iterations	445.790 {90.746}	102.700 {34.516}
Mean Time [s]	161.829 {33.032}	8.196 {2.690}
Mean distance K_P	2.651 {10.529}	0.326 {0.216}
Mean distance K_I	0.962 {3.952}	0.260 {0.272}
Mean distance K_D	2.448 {6.433}	0.820 {0.242}

Table 6: Results of solving IVP using Adam and Levenberg-Marquardt

Results of solving IVP using L-BFGS and Newton-Raphson with added noise		
Metrics	L-BFGS	Newton-Raphson
Mean Iterations	4 {3.197}	5.699 {1.828}
Mean Time [s]	32.553 {12.821}	46.334 {7.211}
Mean distance K_P	2.826 {2.828}	3.581 {3.037}
Mean distance K_I	1.075 {1.473}	1.577 {1.398}
Mean distance K_D	1.347 {0.759}	1.168 {1.025}

Table 7: Results of solving IVP using l-bfgs and Newton-Raphson with added noise

Results of solving IVP using Adam and Levenberg-Marquardt with added noise		
Metrics	Adam	Levenberg-Marquardt
Mean Iterations	576.333 {49.195}	153.9 {45.328}
Mean Time [s]	205.350 {17.862}	13.196 {4.368}
Mean distance K_P	19.528 {14.618}	3.310 {2.832}
Mean distance K_I	2.824 {2.459}	1.637 {1.423}
Mean distance K_D	23.876 {24.241}	1.637 {1.434}

Table 8: Results of solving IVP using Adam and Levenberg-Marquardt with added noise

approach relies heavily on having an accurate initial guess, as evidenced by the standard deviation of iterations. These challenges arise from the inherent nature of gradient descent optimizers. They operate by approximating the objective function linearly at each step with limited local information, necessitating small step sizes at each iteration. While this is beneficial when optimizing a large number of parameters, as seen in neural networks, it becomes impractical in situations with a small number of parameters, as in this case.

Analyzing the rest of the optimization methods we can see that Levenberg-Marquardt performed the fastest, however with a larger amount of function evaluations than L-BFGS and Newton-Raphson. On the other hand, Newton-Raphson achieved the highest number of function evaluations with the highest average computation time excluding Adam.

After performing the aforementioned experiments, we determined that Levenberg-Marquardt is the best suited for our case due to the fact that it consistently achieved the fastest convergence speed across all experiments while achieving results comparable to those of L-BFGS and Newton-Raphson.

13.3 Comparing Parameter Estimation Methods

Selecting the most suitable parameter estimation method is crucial when dealing with ODE systems. The choice of an appropriate loss function can have a profound impact on the precision and dependability of parameter estimates. In this section, we compare two widely used approaches for parameter estimation: Least Squares and Huber Loss[16].

Our findings reveal that in situations characterized by noise-free data, the Least Squares method emerges as the most effective option. However, in scenarios characterized by the presence of noisy data, the Huber Loss proves to be a more robust alternative, offering parameter estimates of increased reliability.

To conduct an evaluation, we performed experiments involving both Least Squares and Huber Loss on the data featuring normal noise with a standard deviation of 100. These experiments were executed using the L-BFGS optimizer and their outcome can be seen in Table 9.

The rationale behind the observed outcomes can be attributed to the inherent characteristics of each of the loss functions examined. Least Squares optimization centers on the minimization of the sum of squared errors, a computational approach that balances efficiency but with a sensitivity to outliers and assumptions

Comparison between Huber and Least Squares		
Metrics	Least Squares with noise	Huber without noise
Mean Iterations	30.38 {2.756}	31.58 {5.308}
Mean Time [s]	9.562 {9.905}	2.402 {8.711}
Mean distance K_P	1.514 {1.257}	1.276 {1.105}
Mean distance K_I	0.497 {0.383}	0.491 {0.346}
Mean distance K_D	1.477{1.294}	1.336 {1.111}

Table 9: Comparison between Huber and Least Squares

about the noise distribution. In contrast, the Huber Loss stands out for its resilience in the face of outliers, a feature achieved by combining the Mean Absolute Error (MAE) and Mean Squared Error (MSE) loss functions. This adaptability is further enhanced by the inclusion of a hyper-parameter; δ, which governs the transition point from quadratic to linear behavior. These characteristics render Huber Loss highly versatile and suitable for a wide array of scenarios, particularly those featuring noisy data or the presence of outliers.

While these loss functions can be used in many scenarios, they do not necessarily work in every scenario. A possible solution is to employ Maximum Likelihood Estimation (MLE). MLE seeks parameters that maximize the likelihood of observing the given data and can be often represented. This likelihood maximization can often be represented as a specific loss function based on the negative log-likelihood. This however requires knowledge of the probability distribution of the noise in the data.

Taking into consideration the aforementioned insights, we conclude that Least Squares is the most suitable choice for our particular problem. This decision is substantiated by the notably minimal error evident in the data originating from the cruise control system.

13.4 Adjoint experiments

In this section, we present the idea behind the adjoint method for computing the gradient with respect to the parameters, and why it is not suitable in cases where loss is dependent on the variables stored and updated in memory.

The adjoint method streamlines the computation of parameter gradients within an ODE system, resulting in a substantial reduction in computational resources compared to conventional finite-difference approaches. This computational efficiency also extends to memory usage, as the method calculates gradients without

requiring explicit storage and backpropagation throughout the entire ODE trajectory.

Given the advantages outlined above, experiments to assess the performance of the adjoint method in the context of parameter prediction using Adam and L-BFGS optimizers were conducted. Initially, those experiments were performed using the same configuration as the one seen in subsubsection 13.2.1 .

The results using the Adam optimizer show that utilizing the adjoint method increases the time of the optimization and leads to parameter estimates that deviate farther from the target compared to cases where the method is not utilized.

When conducting experiments with the L-BFGS optimizer, challenges associated with exploding gradients were encountered, which subsequently led to unsuccessful optimization attempts. To mitigate this issue, we implemented gradient clamping and reduced the learning rate. However, even with these adjustments, the results continued to follow a similar pattern, where the integration of the adjoint method extended the computational time of the optimization process and led to parameter estimates that deviated more significantly from the target values in contrast to cases where the method was not utilized. The results of the aforementioned tests can be seen in Table 10.

We hypothesized a potential connection between the issue of exploding gradients and the storage and updates of historical values. Such values could introduce dependencies that the adjoint is not aware of and cannot account for, but which significantly impact the loss function. To test our hypothesis, we conducted experiments with different combinations of proportional, integral, and derivative terms in our controller. This is because integral and derivative terms are calculated using historical values. The results confirmed our theory as only the controller with proportional term did not show signs of exploding gradients. The exploding gradient can be shown by the evolution of the loss function, rapidly increasing, transitioning to a nearly vertical ascent. The evolution of loss across these experiments is visually represented in Figure 8,Figure 9 and Figure 10.

Based on the empirical results presented above, we conclude that the use of the adjoint method is not suitable when historical values are updated and stored in memory, which is the case with the modeled system of PID-controlled cruise control.

Results of solving IVP using Adam and L-BFGS with adjoint		
Metrics	Adam	L-BFGS
Mean Iterations	550.45 {81.938}	28.53 {74.412}
Mean Time [s]	351.056 {128.611}	31.351 {96.470}
Mean distance K_P	25.272 {16.687}	196.643 {1121.221}
Mean distance K_I	4.099 {2.838}	4.335 {2.981}
Mean distance K_D	77.823 {52.593}	2.932 {11.560}

Table 10: Results of solving IVP using Adam and adjoint

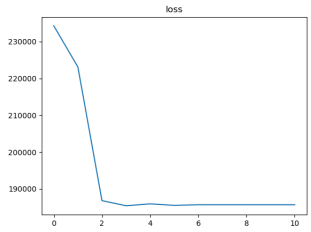


Figure 8: The evolution of loss with proportional term u_P.

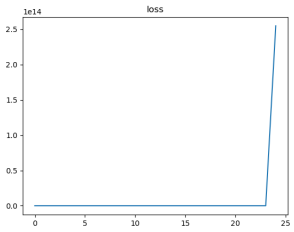


Figure 9: The evolution of loss with proportional and integral terms u_PI.

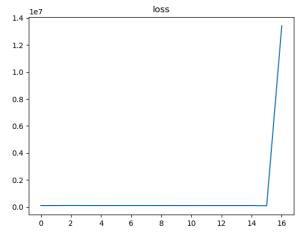


Figure 10: The evolution of loss with proportional and derivative terms u_PD.

14 Deep Neural-Networks

This section goes over how a final DNN model trained achieved a summed total loss of less than 12. Describing how hyper-parameters were tuned, and how issues of saturation and overfitting were dealt with.

14.1 Hyper-Parameter Tuning

The DNN implementations were done in Python, using the TensorFlow/Keras [1] libraries.

To train and test the models, sets of 1000 traces were sampled randomly drawn from the data-set section 8, with 70% being used for training and 30% for testing.

Hyper-parameters are settings or configurations that are not learned from the data but are set prior to the training process. They control the overall behavior of the neural network and influence the learning process. Adjusting the hyper-parameters of a neural network determines the overall architecture, and can be a challenging task, especially without the assistance of specialized tools. Adjusting hyper-parameters is crucial for tailoring the network to a specific problem. Some of these hyper-parameters include the number of layers, the number of nodes in

said layers, the activation function of those nodes, and amount of epochs. Due to all these being tuneable, neural networks can have a near infinite number of unique configurations. In an attempt to get better prediction results from the DNNs trained, the tool "Weights and Biases" was used to log and compare the test results of each configuration.

Weights and Biases can be provided with a set of values or ranges to test within. Opting for specific sets of values reduced the amount of possible combinations, while still being able to explore a wider numerical range. Thus enabling a measured approach to how much of the state space is explored. Additionally, opting to split certain hyper-parameters into separate sweeps, further reducing the overall amount of combinations and the run-time of the experiment. Figure 11 shows the parameter values Weights and Biases was initially setup to train models within.

- Number of runs: 54
- Data-set size: 1000
- Layer amount: [2, 4, 8, 12, 16, 24]
- Nodes per layer: [64, 128, 256, 512, 1024, 2048]
- Epochs: 16
- batch size: [1, 2, 4, 8, 16, 32]
- Vector size: 1500
- Activation function: swish
- Loss function: mean squared error
- Optimizer: Adam
- Learning rate: 1e-3

Figure 11: Initial Weights and Biases experimental setup.

To clarify some of the values/sets of values in Figure 11 that merit it; "number of runs: 54", is the number of different models trained, with a random sampling, from the hyper-parameter values listed. Settling on 54 as that would roughly equate to an exploration of 25% of the possible hyper-parameter combinations, believing that this would be a large enough exploration to make an informed decision, without reaching an infeasible run-time for the experiments. "layer amount", "nodes per layer" and "batch size" having sets of values, indicates that these are the param-

eters being tested, during this first sweep. The choice was made to group "layer amount" and "nodes per layer" hyper-parameters as they are highly dependent on one another, for example, if the network has few layers, it may require a large number of nodes in each layer, for the network to have the required complexity to be able to relate a certain input to a target, and vise versa. Batch size was additionally included to spare the additional sweep.

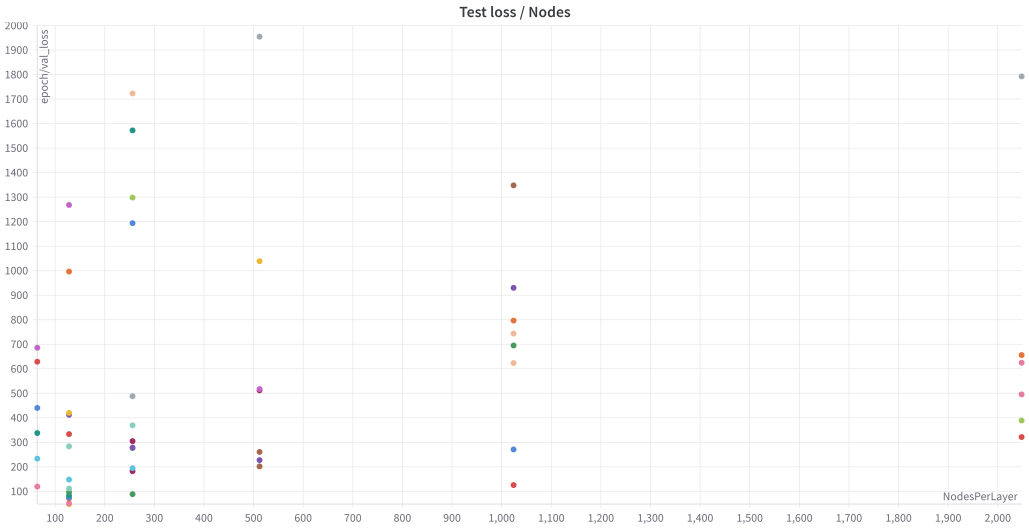
"Epochs" and "learning rate" were optimized in the second sweep, as they like nodes and layers are dependent on one another. For example, with a low number of epochs a larger learning rate may be preferable to ensure the network is able to get close to a minima in the amount of epochs given, with the risk that it will overshoot said minima, and not be quite able to reach it. With a large number of epochs, a lower learning rate may be preferable, as there is more time to converge, and a lower rate may avoid issues of overshooting, however at the cost of training potentially taking significantly longer.

"Vector size: 1500", was chosen as it represents all the time-steps of a trace. The initial hypothesis for this choice was that providing the neural network with as much information as possible would yield more accurate predictions.

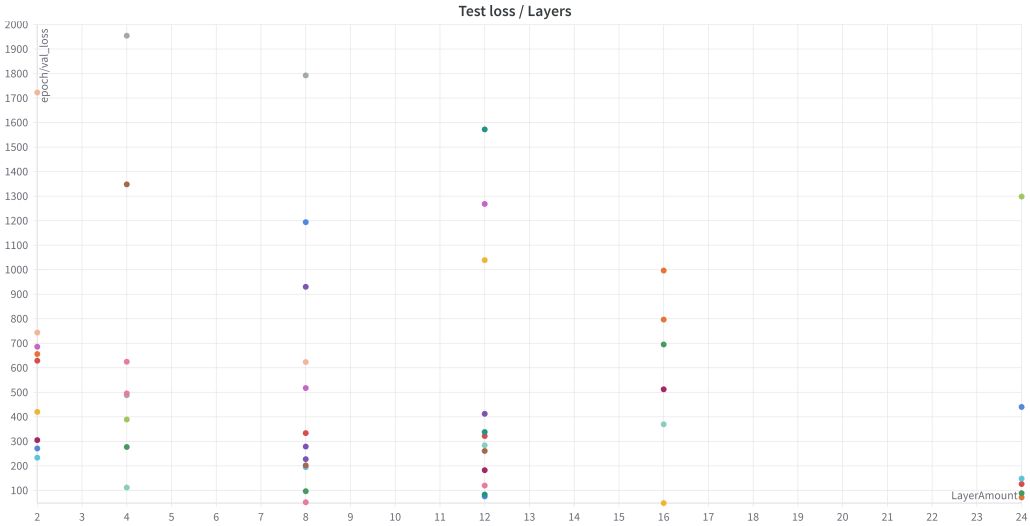
As can be seen in Figure 20, the results from the initial experiments with Weights and Biases were vague and difficult to draw solid conclusions from, as the different hyper-parameter values often had many outliers in their test loss values. Although, from the values in the set of "nodes per layer", 128 seemed to consistently perform well. 24 layers also performs well in several of the test runs. The choice of these hyper-parameter values were based on the fact that both hyper-parameters tended to get losses that clustered low in terms of test loss. Lastly batch size was the most difficult hyper-parameter to make a good decision on, however, out of the set of options, a "batch size" of 8 was chosen, as it had low clustering of losses, like the previous two mentioned, albeit with more outliers than them.

As previously explained, after settling on hyper-parameter values for "nodes per layer", "layer amount", and "batch size", the focus was on finding the best "epochs" and "learning rate". To do this, Weights and Biases was again used, the difference this time being that the entire state space was explored, as there was only two hyper-parameters to optimize. The specific test values can be seen in Figure 13. It is important to note that any values not mentioned remained unchanged from the previous experiment.

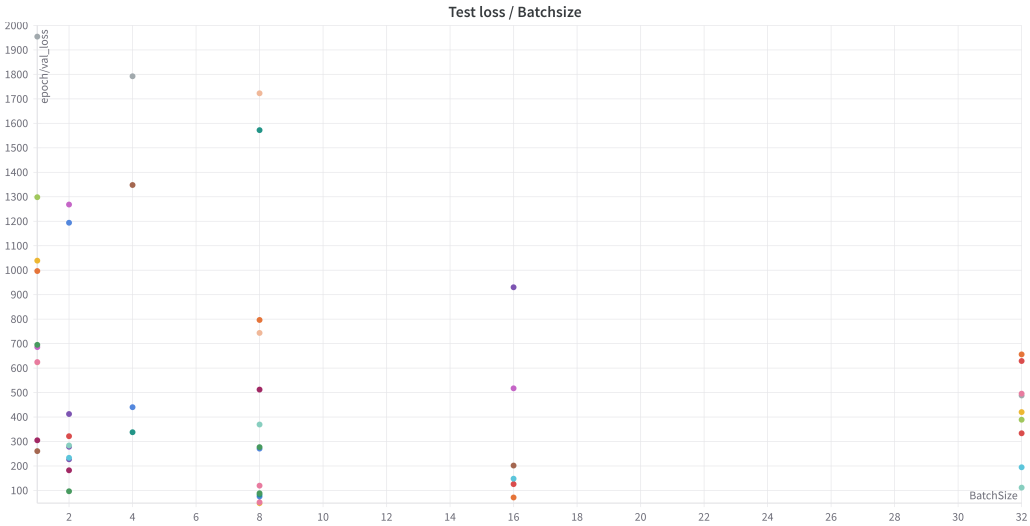
The specific values for "epochs" and "learning rate" were easier to decide upon than the previous three hyper-parameters. Settling on 256 "epochs", as it had a tight grouping at a low test loss, for the models trained. For the "learning rate" 1e-4 was chosen, as the best performing experiment was done with this.



(a) Test loss for differing amounts of "nodes per layer".



(b) Test loss for differing "layer amounts".



(c) Test loss for differing "batch sizes".

Figure 12: Test loss's of initial Weights & Biases experiments.

- Number of runs: 36
- Layer amount: 24
- Nodes per layer: 128
- Epochs: [8, 16, 32, 64, 128, 256]
- batch size: 8
- Learning rate: [1e-2, 5e-3, 1e-3, 5e-4, 1e-4, 5e-5]

Figure 13: Second experimental setup with Weights and Biases.

These two experiments were then repeated, trying to find an optimal amount of layers, "nodes per layer" and "batch size" with the "epochs" and "learning rate" settled on. Followed by another experiment on "epochs" and "learning rate" again, in an attempt to catch any interdependence between the two. The results of these intermittent experiments can be found in subsection 16.1.

The final parameter selection can be seen in Figure 14

- Layer amount: 4
- Nodes per layer: 64
- Epochs: 512
- batch size: 32
- Learning rate: 1e-4

Figure 14: Final values for DNN model hyper-parameter with an input vector size of 1500.

Then a set of 10 DNN models were trained with these hyper-parameters, in an attempt to account for random variance, with the mean and standard deviation of loss on each parameter, along with the run-time of a single prediction listed in Table 11.

The results seen in Figure 20, reveals a consistent pattern; when testing various model parameter configurations with Weights and Biases, clusters often emerge in the test loss, however, with a notable presence of outliers. Investigating the outliers further, by training stand-alone models, and an otherwise random sampling of parameters, revealed that the models trained would often produce the same output regardless of input, a likely culprit of the many outliers observed.

Results of predictions using models trained on an input vector of 1500	
Metrics	Mean and std of parameter prediction test loss
Mean prediction time [s]	0.001 {0.0002}
Mean test loss K_P	7.613 {6.114}
Mean test loss K_I	4.407 {2.905}
Mean test loss K_D	1.956 {1.382}

Table 11: Mean and standard deviation of prediction test loss for models trained on the entire simulation trace with hyper-parameters optimized through Weights and Biases.

14.2 Saturation Measurements

While investigating the issue of models consistently producing identical outputs regardless of input, we considered that this might be attributed to a phenomenon called "saturation" in the trained models. This condition arises when many neurons in a neural network tend to operate within a narrow range of their activation function, often at the extreme ends. To illustrate, take the ReLU activation function. In this case, the neurons only output zero values, regardless of the negative inputs provided. Practically, this leads to a situation where neurons struggle to differentiate between different inputs. They fail to make full use of the available range of activation values. Consequently, this results in the models producing uniform outputs regardless of the input provided.

The Swish activation function is a smooth, non-monotonic function that was introduced as an alternative to activation functions like ReLU [27]. When the input is positive, ReLU allows the gradient to flow directly through, alleviating the vanishing gradient problem [13] that can occur with functions like sigmoid or tanh when they receive a large positive value, causing them to become saturated.

As an example, the sigmoid activation function Equation 25.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{25}$$

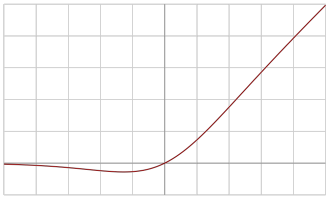
When the sigmoid function receives a very small or large value, the output of the function approaches zero or one, respectively. Causing the gradients in the network to become very small, i.e. the vanishing gradient problem. When the gradients are small, the weights in the network are updated slowly, making it difficult for the network to learn.

Swish shares the feature with ReLU of allowing positive values to flow trough. However, ReLU can cause neurons to become inactive, output zero, for negative input, leading to a "dying ReLU" problem [23], a similar problem to when sigmoid

and tanh approaches their respective extremes. For swish, the contribution of negative values is reduced, but it's not entirely removed, as in ReLU. Swish combines the linearity of the identity function with the non-linearity of the sigmoid function. It is defined as seen in Figure 15a.

$$f(x) = x \cdot \sigma(x)$$

(a) Swish Equation



(b) Swish Function Plot

Figure 15: Swish Activation Function

Where σ is the sigmoid function.

To investigate potential saturation-related issues, a measure was implemented, taking inspiration from [12], which graphed activation function outputs over a set of ranges. Given that swish lacks an upper limit for positive inputs, the issue was assumed to stem from the negative activations. Specifically, values between 0 and -0.3 were examined, as the lowest observed activation value in the testing data was -0.273. This range was then divided into 10 segments, each indicating the percentage of activations within that range. Table 12 displays the mean and standard deviation of saturation across 10 models.

Saturation of models trained on an input vector of 1500	
Metrics	Mean and std of saturation
0 - -0.03	55.195 {1.962}
-0.03 - -0.06	0.016 {0.001}
-0.06 - -0.09	0.008 {0.001}
-0.09 - -0.12	0.008 {0.003}
-0.12 - -0.15	0.006 {0.001}
-0.15 - -0.18	0.009 {0.003}
-0.18 - -0.21	0.007 {0.001}
-0.21 - -0.24	0.007 {0.002}
-0.24 - -0.27	0.011 {0.001}
-0.27 - -0.3	0.012 {0.002}

Table 12: Mean and standard deviation of saturation within ranges for models trained on the entire simulation trace with hyper-parameters optimized trough Weights and Biases.

As can be seen in Table 12, the majority of nodes have activations in the 0 - -0.03 range, a sign of saturation as mentioned. This was believed to be the

primary culprit of the model sometimes not being able to differentiate between inputs and simply outputting the same value, regardless of input.

In addressing this problem, the initial approach involved normalizing the data-set used for training the models. Both the model input and the targets underwent normalization, albeit with differences in the application.

14.3 Normalization

To improve the performance and alleviate issues with saturation in the DNN, normalization was employed on the data-set [18]. Various normalization techniques were evaluated including ZScore and min-max normalization, which were the most common among the ones tested. Testing and comparing the techniques on the loss achieved did not reveal a consistently superior method. The less common techniques were rarely an improvement, and in some cases, under-performed notably. The decision was made to solely use min-max normalization for both the input and target parameters.

This decision was driven firstly by the need to standardize the range of the target parameters. In the data-set, the K_P parameter for the controller had a wide numerical range, spanning from 100 to 1000, whereas the K_I and K_D parameters ranged from 2 to 15 and 1 to 3, respectively. Given the substantial numerical difference between K_P and the other two parameters, it biased the training in favor of accurately predicting K_P. By applying min-max normalization to scale all three parameters within a range from 0 to 1, this bias was eliminated, ensuring equal priority for each parameter. Furthermore, min-max normalization can enhance training stability by scaling the input vector. Reducing large numerical values to a lesser scale helps mitigate training volatility, addressing concerns such as overshooting and potentially accelerating the convergence process.

The decision was made to not utilize ZScore normalization, despite it achieving similar results to min-max when tested. This choice was due to the fact that ZScore normalization relies on the mean and standard deviation of a given data-set to scale the values. However, in the context of the problem at hand, these measures offer no valuable information to the system. To clarify, the input to the DNN consisted of a time-series of velocities, representing the vehicle speed changing over time as the cruise controller reached the target speed. In this context, neither the mean nor standard deviation of the time-series provides any insight, as to how how a specific set of parameters alter the values of said time-series. Similarly, the mean and standard deviation offer no meaningful insights when it comes to presenting the target parameters. The data-set contains every possible combination of K_P,

K_I, K_D and initial speed. Given that this represents a uniform distribution across the state space, with no combination more prevalent than another, the mean or standard deviation of these parameters is not useful.

Despite deciding to use min-max normalization for both the input and target parameters, the application differed. As discussed in subsection 5.8 there are three different methods for applying normalization, feature, sample, and global normalization.

In the case of input data, a global normalization approach was utilized. The input represents a time-series where each subsequent value depends on the preceding one. Consequently, to ensure the time-steps remain comparable and meaningful comparisons can be made, the time-steps must all be scaled equally, which makes feature normalization unsuitable. For similar reasons, sample normalization is likewise unsuitable for the data-set. As each sample may have varying initial and top velocities, scaling the data utilizing sample normalization, would lead to samples being scaled individually. As a result, making meaningful comparisons between samples becomes difficult, this would impede the learning process's ability to identify common patterns. To avoid these inconsistencies between features and samples, global normalization was applied to the input data, ensuring a uniform scale is applied equally and maintaining comparability.

For target parameters, the choice was made to use feature normalization instead. The primary benefit of employing min-max normalization for the target parameters ensures a common numerical scale for all parameters, preventing any single parameter from being favored during the training process. This necessitated individual normalization of each feature, thus feature normalization was required.

Table 13 and Table 14 show the results and saturation of 10 models trained with normalized data.

Results of predictions using models trained on a normalized input vector of 1500	
Metrics	Mean and std of parameter prediction test loss
Mean prediction time [s]	0.001 {0.0001}
Mean test loss K_P	16.138 {16.743}
Mean test loss K_I	2.354 {1.847}
Mean test loss K_D	0.679 {0.446}

Table 13: Mean and standard deviation of prediction test loss for models trained on globally min-max normalised input and feature min-max normalized targets.

Saturation of models trained on a normalized input vector of 1500	
Metrics	Mean and std of saturation
0 - -0.03	2.098 {0.312}
-0.03 - -0.06	2.242 {0.865}
-0.06 - -0.09	2.423 {0.931}
-0.09 - -0.12	2.092 {0.524}
-0.12 - -0.15	2.413 {0.646}
-0.15 - -0.18	4.662 {0.162}
-0.18 - -0.21	8.440 {0.942}
-0.21 - -0.24	8.760 {0.371}
-0.24 - -0.27	9.482 {0.824}
-0.27 - -0.3	7.444 {1.506}

Table 14: Mean and standard deviation of saturation within ranges for models trained on globally min-max normalized input and feature min-max normalized targets.

As can be seen in Table 13, the mean test loss on K_P, increased when models were trained on min-max normalized data. Summing the K_P, K_I, and K_D losses for models trained on non-normalized data and comparing them with those trained on normalized data, it becomes shows that the models trained on normalized data experienced an increase in total loss. When K_P is not normalized, the loss for predicting the parameter wrong is likely to be much larger, as the range of possible values for it is wider, causing the network to prioritize it. Bringing the K_P parameter within the same range of values as K_I and K_D means the model trained will treat the loss on each equally, and the gain achieved on K_I and K_D might not be enough to compensate the likely increase on K_P.

However, the saturation improved, as shown in Table 14, where it can be seen that the activations are no longer concentrated in a single range, close to zero. A benefit of the normalized models is, that they are less likely to produce the same output regardless of input and likely being better at generalizing than their non-normalized counterparts, as they use a larger range of their activation function.

14.4 Constant Feature Regions

To further improve the models trained, an examination of the data from the cruise control system was done. It was observed that the target speed was consistently reached around the 500th time-step. Once the cruise control system reached its set-point, it maintained a steady speed, resulting in a portion on the data where

features are constant across all data-points, the last 1000 entries, and as [2] states, this provides no additional information for the network to learn from. In such instances, the neural network might struggle to discern meaningful patterns, potentially affecting the model's performance. Removing these regions of constant features would emphasize the unique characteristics that distinguishes how different parameter configurations reached the desired speed. This adjustment could potentially make it simpler for a model to associate a specific input with a particular target, potentially reducing the test loss.

To test this hypothesis, hyper-parameters were again optimized, reasoning that they might have different ideal settings for different input sizes. The last iteration of this can be seen in Figure 16. The final hyper-parameters selection can be seen in Figure 17.

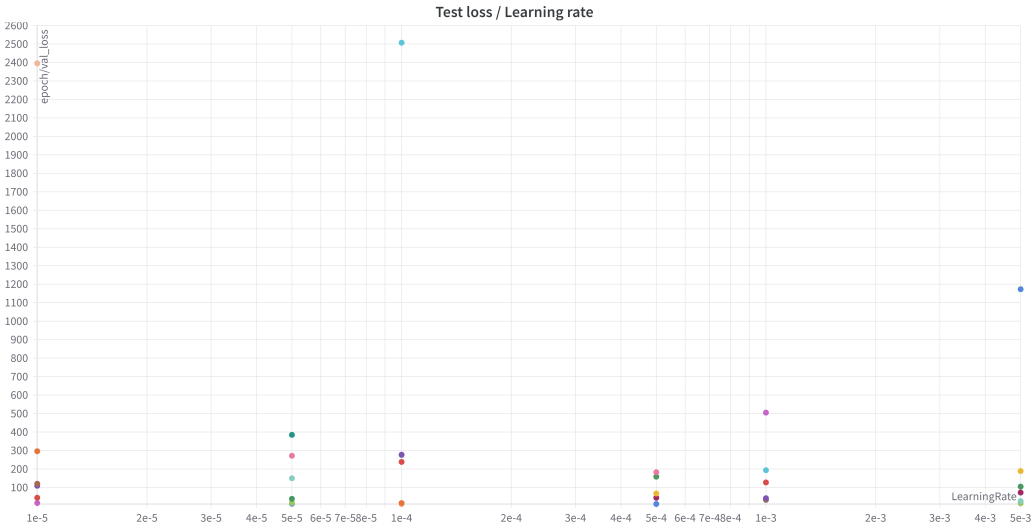


Figure 16: Test loss for differing amounts of "learning rates".

- Layer amount: 16
- Nodes per layer: 1024
- Epochs: 256
- batch size: 32
- Learning rate: 5e-4

Figure 17: Final values for DNN model hyper-parameter with an input vector size of 500.

In Figure 16, an improvement is evident in the number of outliers in the trained models. This indicates that they did not suffer from the issue of producing the same output regardless of input, as previously mentioned about the models trained with a "vector size" of 1500. Furthermore, the hyper-parameters selected in Figure 17 resulted in a much more complex network, with both a higher number of "layer amount" and "nodes per layer".

Results of predictions using models trained on an input vector of 500	
Metrics	Mean and std of parameter prediction test loss
Mean prediction time [s]	0.001 {0.0002}
Mean test loss K_P	25.395 {39.252}
Mean test loss K_I	3.689 {2.521}
Mean test loss K_D	1.131 {0.788}

Table 15: Mean and standard deviation of prediction test loss for models trained on the first five-hundred entries of simulation traces with hyper-parameters optimised through Weights and Biases.

Results of predictions using models trained on a normalized input vector of 500	
Metrics	Mean and std of parameter prediction test loss
Mean prediction time [s]	0.003 {0.0005}
Mean test loss K_P	42.696 {56.337}
Mean test loss K_I	3.416 {1.774}
Mean test loss K_D	0.676 {0.437}

Table 16: Mean and standard deviation of prediction test loss for models trained on globally min-max normalised input and feature min-max normalized targets.

However, as can be seen in Table 15 and Table 16, the results actually worsened, when compared to Table 11 or the normalized version thereof Table 13. This initially disproved the hypothesis that removing the regions of constant features region of the data would reduce the test loss of the models trained. Examining the training loss of the models trained on the data-sets with regions removed however revealed that the training losses were consistently lower than the testing loss, indicating that the models trained were being over-fitted.

14.5 Overfitting

As stated previously, the models trained with a "vector size of 500" exhibited signs of overfitting, whereas these signs were absent in the models trained with a vector size of 1500. This is likely because of the more complex network, in conjunction with the many epochs they were trained for, allowing them enough complexity to

create a model that maps the training data completely and enough epochs to do so. In contrast, the models with a vector size of 1500 were relatively simpler, forcing the models to learn general patterns instead.

To test this hypothesis, another round of Weight and Biases experiments were conducted, focusing on picking lower values for the neural networks trained. The selection settled on can be seen in Figure 18.

- Layer amount: 4
- Nodes per layer: 512
- Epochs: 512
- batch size: 4
- Learning rate: 0.0001

Figure 18: Low value focus for DNN model hyper-parameter with an input vector size of 500.

While Weights and Biases actually found that an even higher amount of "epochs" were needed, an lower "learning rate" was needed, this might be due to the difficulty of finding a minima in this network that produces the least loss on all of the training data given. "Layer amount" and "nodes per layer" were however reduced. Models trained with this configuration of hyper-parameters yielded the results seen in Table 17 and Table 18.

Results of predictions using models trained on an input vector of 500 with a less complex network	
Metrics	Mean and std of parameter prediction test loss
Mean prediction time [s]	0.001 {0.0002}
Mean test loss K_P	5.029 {5.592}
Mean test loss K_I	3.201 {2.376}
Mean test loss K_D	0.745 {0.458}

Table 17: Mean and standard deviation of prediction test loss for models trained on the first five-hundred entries of simulation traces with hyper-parameters optimised trough Weights and Biases.

As the results show, these models out-competes the models trained with a "vector size" of 1500, to some extend proving the hypothesis that removing the regions of constant features would emphasize the distinct characteristics that distinguished how different parameter configurations reached the desired speed. Like with the 1500 models, these "vector size" 500 models, when trained on non-normalized data have a high saturation in the 0 - -0.03 range, 84.574% to be exact, and normalizing

Results of predictions using models trained on a normalized input vector of 500 with a less complex network	
Metrics	Mean and std of parameter prediction test loss
Mean prediction time [s]	0.003 {0.0005}
Mean test loss K_P	10.040 {8.634}
Mean test loss K_I	1.187 {1.078}
Mean test loss K_D	0.704 {0.444}

Table 18: Mean and standard deviation of prediction test loss for models trained on globally min-max normalised input and feature min-max normalized targets.

the data drops the activations in this range to 31.162. Still a rather high value, when compared to the results seen when normalization was applied to the "vector size" 1500 models, indicating that the network could benefit from further refinement. This is because the activations in this range are close to an output of zero, meaning that any input that was entered into the node was essentially nullified. This is referred to as a "dead node", and could indicate that the network could be reduced in complexity even further.

Another approach to handling overfitting is adding noise to the training data. Noisy data introduces variability and outliers into the training set, forcing the model to learn more general patterns, rather than memorizing specific examples. Furthermore, adding noise helps simulate real-world scenarios, where noise is prevalent.

However, adding noise to the training data, to the same degree as was done in subsubsection 13.2.1, with a standard deviation of ten, was not enough to have an significant impact on the performance of models trained with the hyper-parameter setup in Figure 18. The results were nearly identical to those obtained when the models were trained on noiseless data. Furthermore, using Huber loss for loss calculations on noisy data showed no improvement either, as it did for the numerical solutions subsection 13.3.

15 Conclusion

As mentioned in the problem definition section 6, the goal of this thesis was to evaluate and compare DNNs, numerical solutions and a combined approach, to find which is most suitable for the task of predicting the parameters of the controller, given observational data from the plant in a cruise control system.

Examining the results in Table 19, shows that employing a model trained on

Table of DNN and numerical solution results			
Metrics	DNNs	Numerical solution	Numerical solution with DNN initial guess
Mean prediction time [s]	0.003 {0.0005}	5.793 {1.9}	9.277 {2.935}
Mean test loss K_P	10.040{8.634}	0.556 {0.490}	0.879 {0.784}
Mean test loss K_I	1.187 {2.905}	0.098 {0.104}	0.159 {0.150}
Mean test loss K_D	0.704 {0.444}	0.469 {0.227}	0.372 {0.212}

Table 19: Mean and standard deviation comparisons of evaluation matrices of the DNNs, numerical solutions and the combined approach.

normalized data with regions of constant features excluded for parameter prediction can yield results close to the actual targets. However, these results, while generally good, have a higher loss than numerical methods, specifically the Levenberg-Marquardt single shooting approach, with the DBE as an initial guess.

An advantage of DNNs however, lies in their ability to make predictions faster. A disadvantage then is that DNNs require a training phase. For this, an appropriate training data-set and the fine-tuning of hyper-parameters, etc. are needed.

The con of numerical methods are then that they require an accurate model of the system, not something that is always available. Additionally, the selection of a suitable solver is pivotal in accurately approximating the original trajectory, and thereby, target parameters.

A combined approach was then attempted, with the DNN model employed as a method for an initial guess for the Levenberg-Marquardt solver. As can be seen in Table 19 this combined approach's accuracy and computing speed is roughly equivalent as when using DBE for initial guess estimation. However in this case it is likely not worthwhile to employ DNNs as an initial guess estimator, due to it being significantly more difficult to implement. However, this conclusion could change if the DNNs prediction accuracy improved further.

Table 20 shows the results of previously mentioned approaches predicting on noisy data. As can be seen, the DNN model was not affected significantly by the degree of noise used, unlike the numerical solution. Despite this, the conclusions between the two approaches on noisy data remain the same, with the numerical solution having a lower loss, with the trade-off of computation speed. This is likely due to the initial guess from the DBE estimator performing poorly on noisy data. Notably however on noisy data the numerical solution using the DNN model for an initial guess, bests the numerical with the DBE initial guess estimator. Outper-

Table of DNN and numerical solution results on noisy data			
Metrics	DNNs	Numerical solution	Numerical solution with DNN initial guess
Mean prediction time [s]	0.002 {0.0004}	12.685 {3.919}	11.620 {3.395}
Mean test loss K_P	10.250 {8.762}	2.934 {1.629}	1.590 {1.197}
Mean test loss K_I	1.795 {1.303}	1.449 {1.167}	0.699 {0.589}
Mean test loss K_D	0.669 {0.471}	1.529 {1.899}	1.372 {0.980}

Table 20: Mean and standard deviation comparisons of evaluation matrices of the DNNs and numerical solutions on noisy data

forming it both in terms of accuracy and with a marginal computing time decrease.

While both DNNs and single shooting approaches have their respective merits and drawbacks, their efficacy hinges on distinct prerequisites and constraints. DNNs offer swifter predictions but necessitate a robust training phase. At the same time, single shooting approaches excel with an accurate model of the system but are contingent on careful selection of solvers and step sizes. DNNs might be preferable in scenarios where few computing resources are available, fast/frequent predictions are required, or in cases where a model of the system is not available. The numerical solution might be preferable when time is not as critical, but accuracy is. Finally, it is possible that a DNN model could be optimized further which may change some of the conclusions we have made thus far. We did not achieve this though, and leave that as potential future work.

References

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[2] Walid Hasen Atomi. "The Effect of Data Preprocessing on the Performance of Artificial Neural Networks Techniques for Classification Problems". University Tun Hussein Onn Malaysia, 2012, pp. 17–18.

[3] Yonathan Bard. "Nonlinear parameter estimation". In: *(No Title)* (1974).

[4] *Block diagram of a closed-loop controller - Pieter P*. URL: https://tttapa.github.io/Pages/Arduino/Control-Theory/Motor-Fader/PID-Controllers.html.

[5] Hans Georg Bock and Karl-Josef Plitt. "A multiple shooting algorithm for direct solution of optimal control problems". In: *IFAC Proceedings Volumes* 17.2 (1984), pp. 1603–1608.

[6] William Bradley and Fani Boukouvala. "Two-Stage Approach to Parameter Estimation of Differential Equations Using Neural ODEs". In: *Industrial & Engineering Chemistry Research* 60.45 (2021), pp. 16330–16344. DOI: 10.1021/acs.iecr.1c00552. eprint: https://doi.org/10.1021/acs.iecr.1c00552. URL: https://doi.org/10.1021/acs.iecr.1c00552.

[7] Jonathan Calver. "Parameter Estimation for Systems of Ordinary Differential Equations". In: 2019.

[8] Tian Qi Chen et al. "Neural Ordinary Differential Equations". In: *Neural Information Processing Systems*. 2018.

[9] John R Dormand and Peter J Prince. "A family of embedded Runge-Kutta formulae". In: *Journal of computational and applied mathematics* 6.1 (1980), pp. 19–26.

[10] Vivek Dua. "An Artificial Neural Network approximation based decomposition approach for parameter estimation of system of ordinary differential equations". In: *Comput. Chem. Eng.* 35 (2011), pp. 545–553.

[11] Ronald A Fisher. "On the mathematical foundations of theoretical statistics". In: *Philosophical transactions of the Royal Society of London. Series A, containing papers of a mathematical or physical character* 222.594-604 (1922), pp. 309–368.

[12] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[13] Sepp Hochreiter. "The vanishing gradient problem during learning recurrent neural nets and problem solutions". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.02 (1998), pp. 107–116.

[14] Nikolaus Harry Reginald Howe. "Learning neural ordinary differential equations for optimal control". In: (2021). URL: https://papyrus.bib.umontreal.ca/xmlui/handle/1866/26529.

[15] Nikolaus Harry Reginald Howe. "Learning neural ordinary differential equations for optimal control". In: (2022).

[16] Peter J Huber. "Robust estimation of a location parameter". In: *Breakthroughs in statistics: Methodology and distribution*. Springer, 1992, pp. 492–518.

[17] Elnaz Jamili and Vivek Dua. "Parameter estimation of partial differential equations using artificial neural network". In: *Computers & Chemical Engineering* 147 (2021), p. 107221. ISSN: 0098-1354. DOI: https://doi.org/10.1016/j.compchemeng.2020.107221. URL: https://www.sciencedirect.com/science/article/pii/S0098135420312643.

[18] Daehyon Kim. "Normalization methods for input and output vectors in backpropagation neural networks". In: *International Journal of Computer Mathematics* 71.2 (1999), pp. 161–171. DOI: 10.1080/00207169908804800. eprint: https://doi.org/10.1080/00207169908804800. URL: https://doi.org/10.1080/00207169908804800.

[19] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[20] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[21] K Levenberg. "Method for the solution of certain problems in least squares siam". In: *J Numer Anal* 16 (1944), 588–A604.

[22] Dong C Liu and Jorge Nocedal. "On the limited memory BFGS method for large scale optimization". In: *Mathematical programming* 45.1-3 (1989), pp. 503–528.

[23] Lu Lu et al. "Dying relu and initialization: Theory and numerical examples". In: *arXiv preprint arXiv:1903.06733* (2019).

[24] Benn Macdonald and Dirk Husmeier. "Gradient matching methods for computational inference in mechanistic models for systems biology: a review and comparative analysis". In: *Frontiers in bioengineering and biotechnology* 3 (2015), p. 180.

[25] Claas Michalik, Ralf Hannemann, and Wolfgang Marquardt. “Incremental single shooting—a robust method for the estimation of parameters in dynamical systems”. In: *Computers & Chemical Engineering* 33.7 (2009), pp. 1298–1305.

[26] Martin Peifer and Jens Timmer. “Parameter estimation in ordinary differential equations for biochemical processes using the method of multiple shooting”. In: *IET systems biology* 1.2 (2007), pp. 78–88.

[27] Prajit Ramachandran, Barret Zoph, and Quoc V Le. “Searching for activation functions”. In: *arXiv preprint arXiv:1710.05941* (2017).

[28] Antônio H Ribeiro and Luis A Aguirre. “Shooting methods for parameter estimation of output error models”. In: *IFAC-PapersOnLine* 50.1 (2017), pp. 13998–14003.

[29] Johann Rudi, Bessac Julie, and A. Lenzi. *Parameter Estimation with Dense and Convolutional Neural Networks Applied to the FitzHugh-Nagumo ODE*. Dec. 2020.

[30] James M Varah. “A spline least squares method for numerical parameter estimation in differential equations”. In: *SIAM Journal on Scientific and Statistical Computing* 3.1 (1982), pp. 28–46.

[31] *Weights & Biases documentation*. URL: `https://docs.wandb.ai`.

[32] Hulin Wu, Hongqi Xue, and Arun Kumar. “Numerical discretization-based estimation methods for ordinary differential equation models via penalized spline smoothing with applications in biomedical research”. In: *Biometrics* 68.2 (2012), pp. 344–352.

16 Appendix

16.1 Sweeps

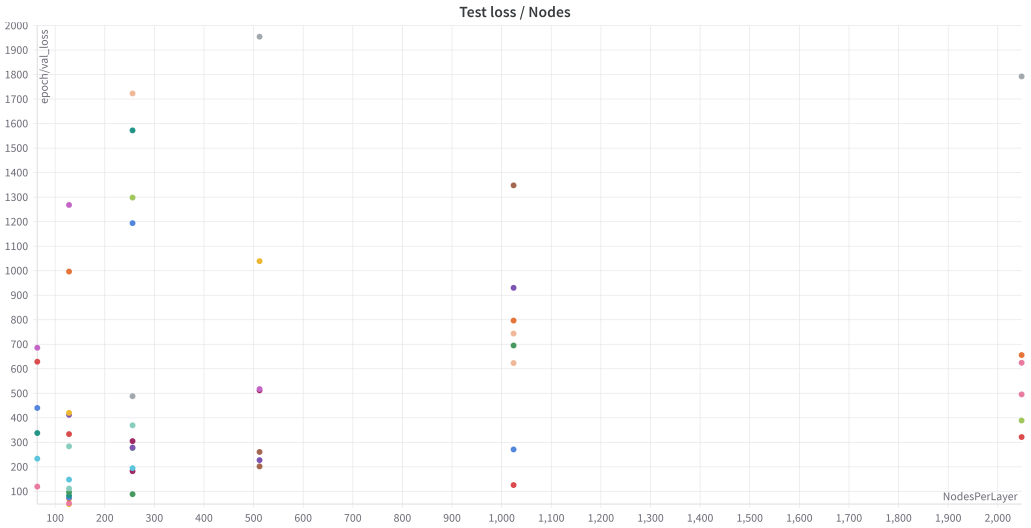
In the following, every sweep performed with Weights & Biases will be listed, with a full itemized list of hyper-parameter values/sets of values the experiments where performed within, and the resulting scatter plots.

16.1.1 Sweep 1

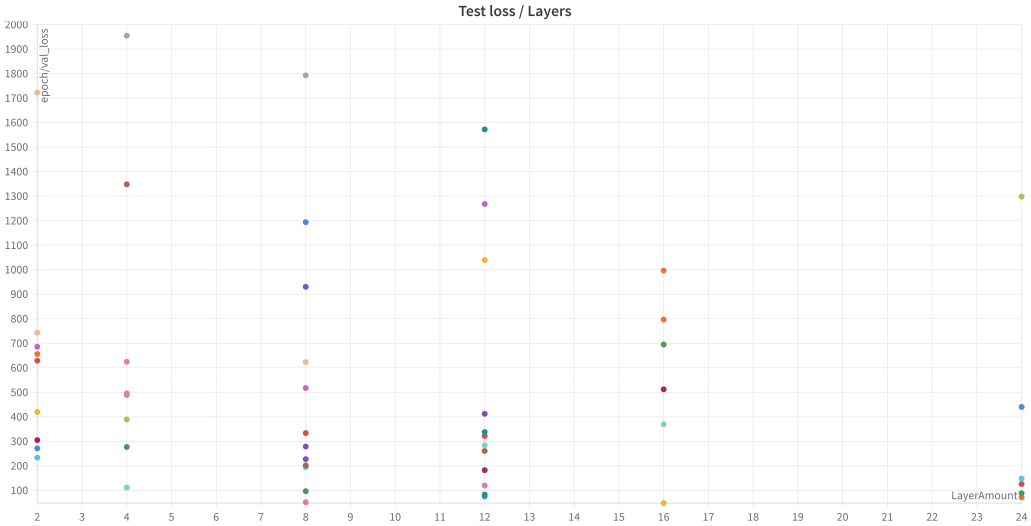
- Number of runs: 54
- Data-set size: 1000
- Layer amount: [2, 4, 8, 12, 16, 24]
- Nodes per Layer: [64, 128, 256, 512, 1024, 2048]
- Epochs: 16
- batch size: [1, 2, 4, 8, 16, 32]
- Vector size: 1500
- Activation function: swish
- Loss function: mean squared error
- Optimizer: Adam
- Learning rate: 1e-3

Figure 19: sweep 1 configuration

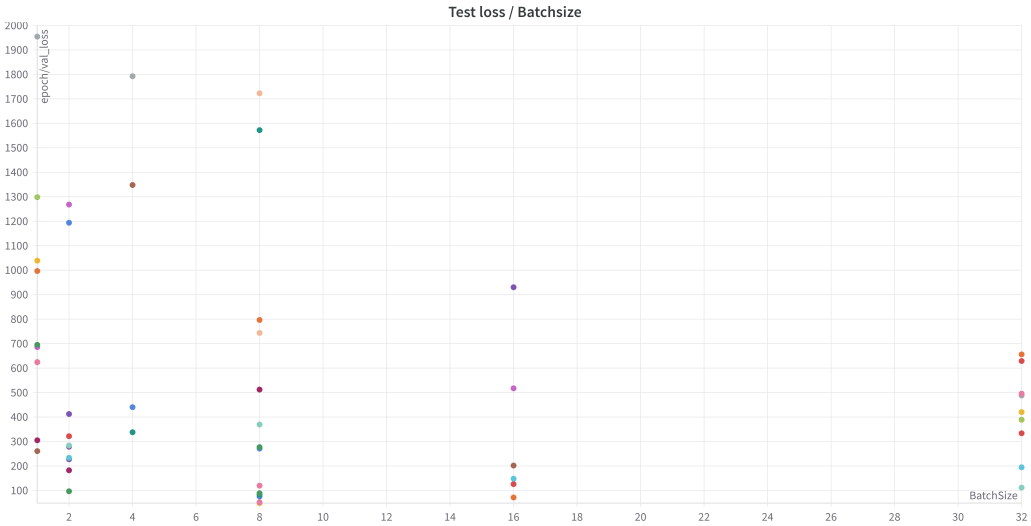
24 "layer amount", 128 "nodes per layer" and a "batch size" of 8 was chosen.



(a) Test loss for differing amounts of "nodes per layer".



(b) Test loss for differing "layer amounts".



(c) Test loss for differing "batch sizes".

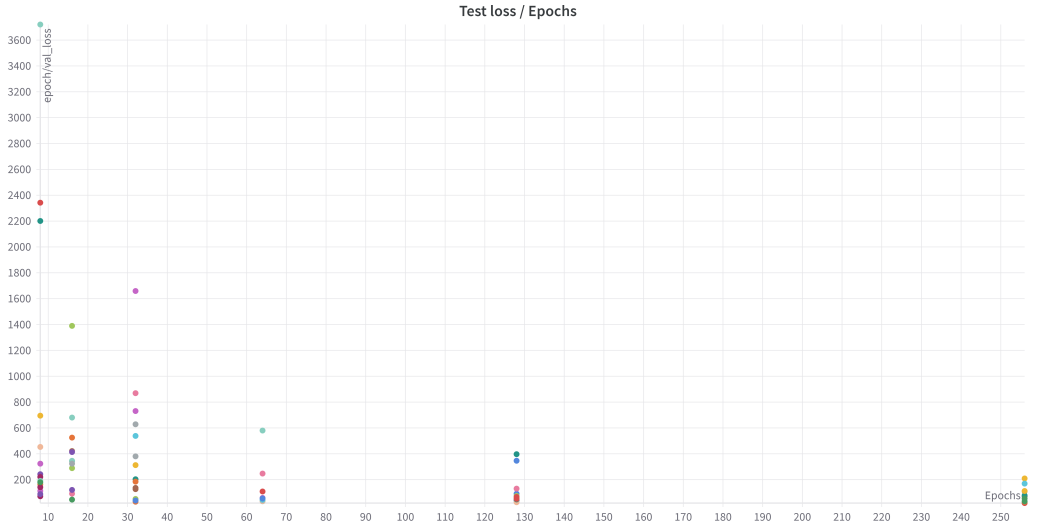
Figure 20: Sweep 1 results.

16.1.2 Sweep 2

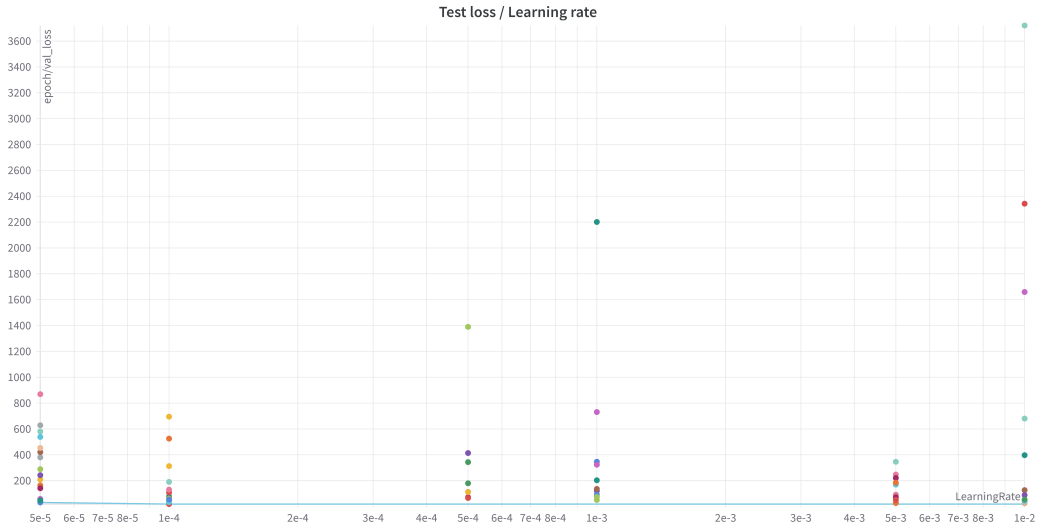
- Number of runs: 75
- Data-set size: 1000
- Layer amount: 24
- Nodes per Layer: 128
- Epochs: [8,16,32,64,128,256]
- batch size: 8
- Vector size: 1500
- Activation function: swish
- Loss function: mean squared error
- Optimizer: Adam
- Learning rate: [1e-2, 5e-3, 1e-3, 5e-4, 1e-4, 5e-5]

Figure 21: sweep 2 configuration

256 "epochs" and a learning rate of "1e-4" was chosen.



(a) Test loss for differing amounts of "Epochs".



(b) Test loss for differing "learning rates".

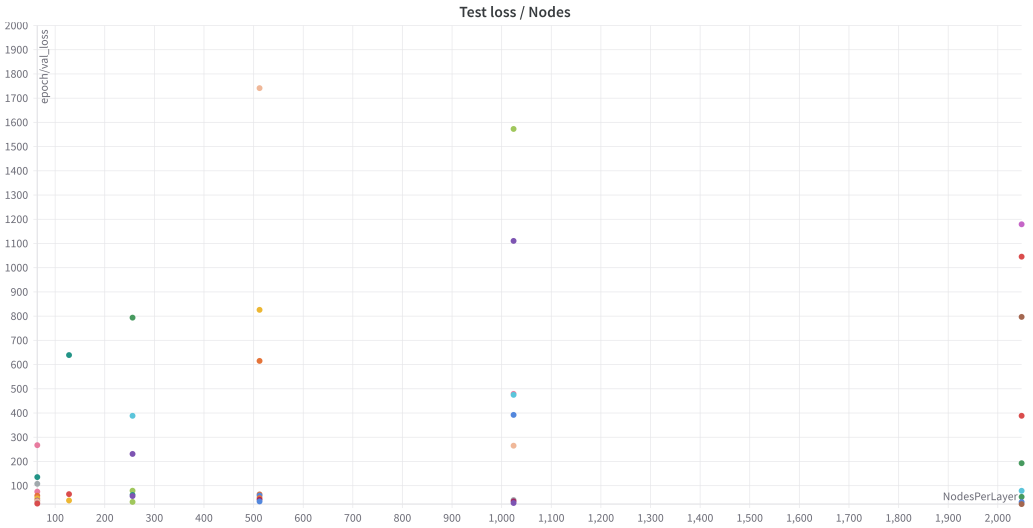
Figure 22: Sweep 2 results.

16.1.3 Sweep 3

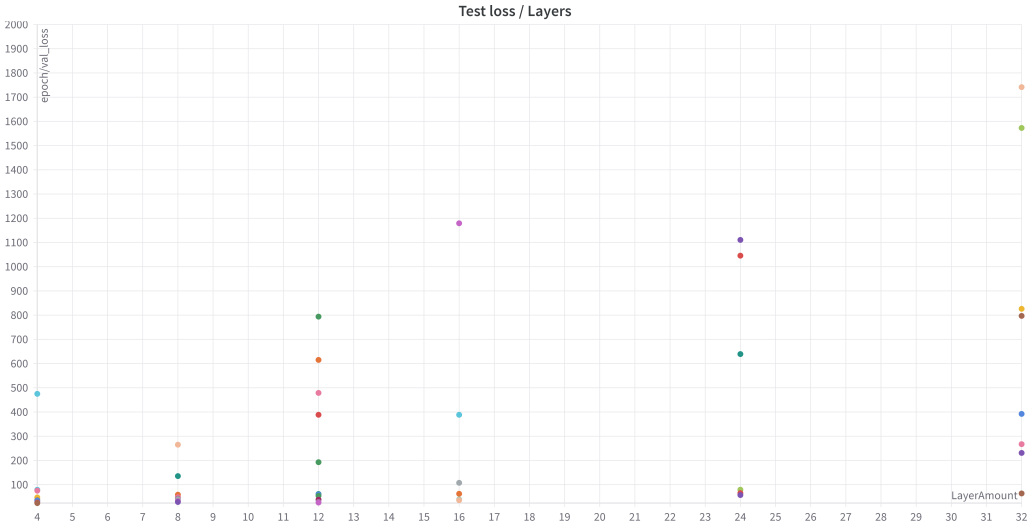
- Number of runs: 54
- Dataset size: 1000
- Layer amount: [4, 8, 12, 16, 24, 32]
- Nodes per Layer: [64, 128, 256, 512, 1024, 2048]
- Epochs: 256
- batch size: [1, 2, 4, 8, 16, 32]
- Vector size: 1500
- Activation function: swish
- Loss function: mean squared error
- Optimizer: Adam
- Learning rate: 1e-4

Figure 23: sweep 3 configuration

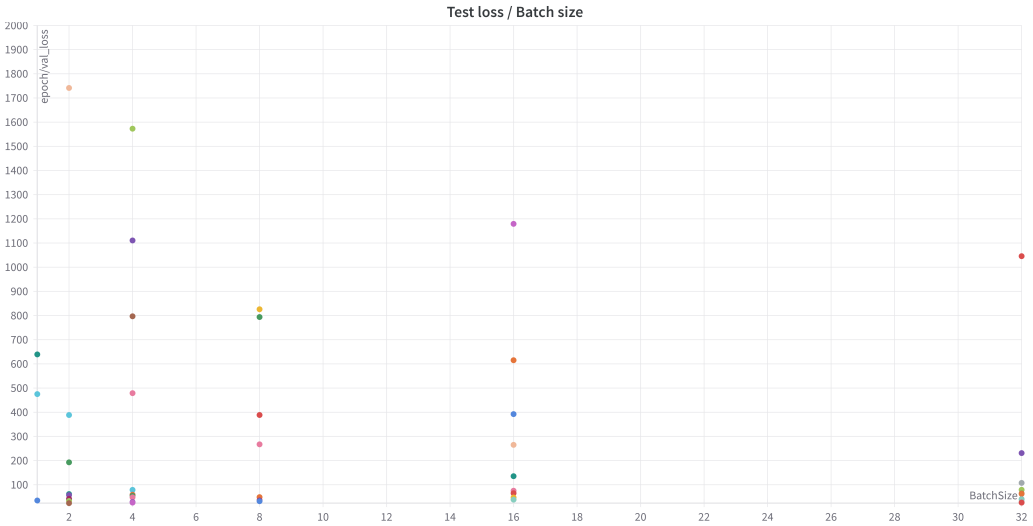
4 "layer amount", 64 "nodes per layer" and a "batch size" of 32 was chosen.



(a) Test loss for differing amounts of "nodes per layer".



(b) Test loss for differing "layer amounts".



(c) Test loss for differing "batch sizes".

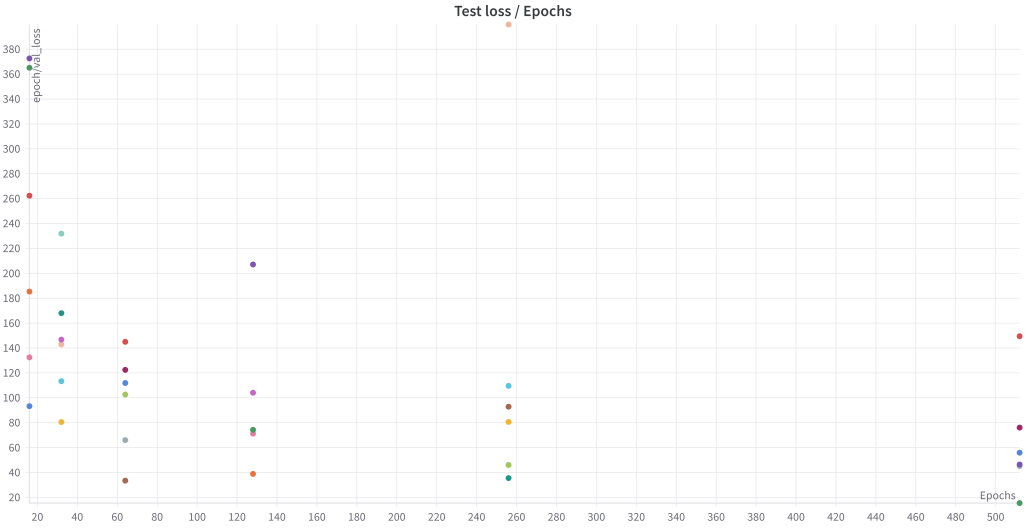
Figure 24: Sweep 3 results.

16.1.4 Sweep 4

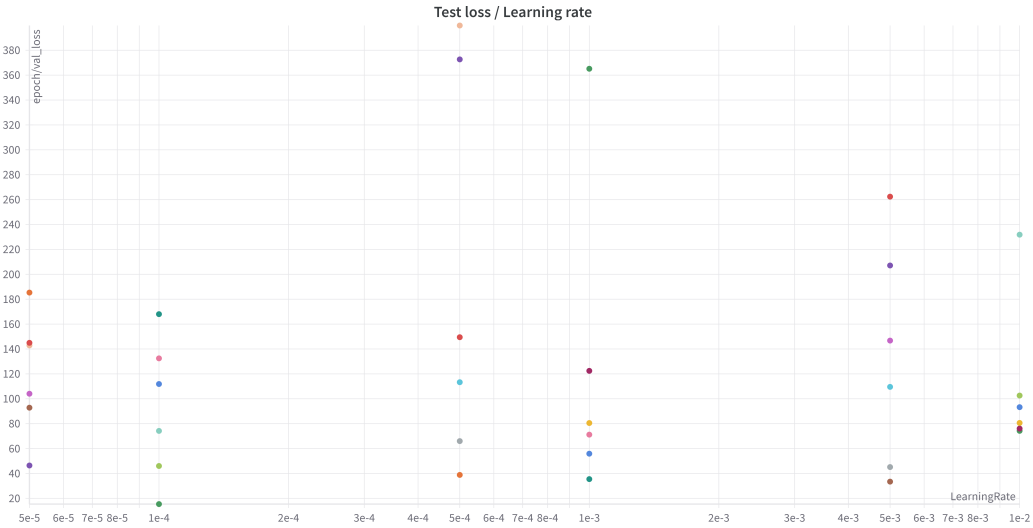
- Number of runs: 36
- Dataset size: 1000
- Layer amount: 4
- Nodes per Layer: 64
- Epochs: [16, 32, 64, 128, 256, 512]
- batch size: 32
- Vector size: 1500
- Activation function: swish
- Loss function: mean squared error
- Optimizer: Adam
- Learning rate: [1e-2, 5e-3, 1e-3, 5e-4, 1e-4, 5e-5]

Figure 25: sweep 4 configuration

512 "epochs" and a learning rate of "1e-4" was chosen.



(a) Test loss for differing amounts of "Epoch".



(b) Test loss for differing "Learning rates".

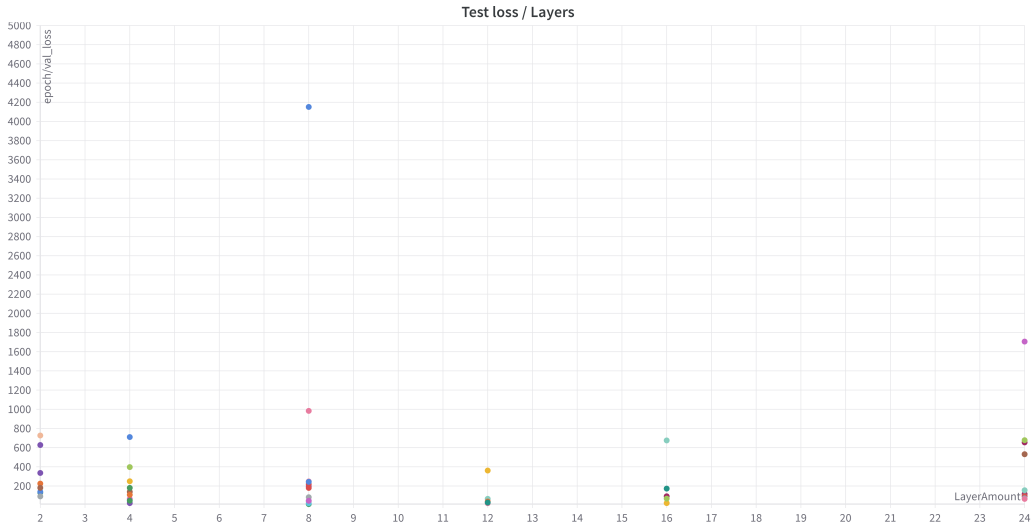
Figure 26: Sweep 4 results.

16.1.5 Sweep 5

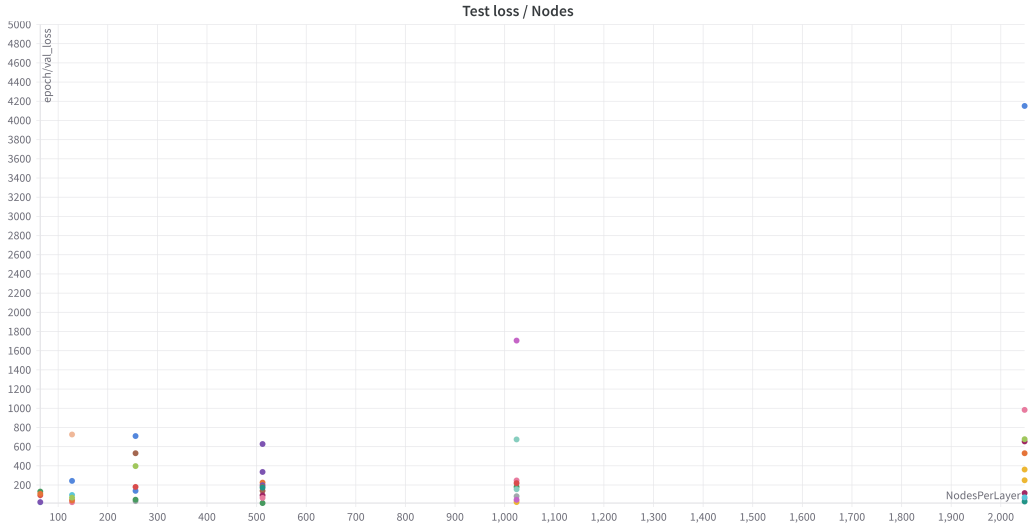
- Number of runs: 54
- Dataset size: 1000
- Layer amount: [2, 4, 8, 12, 16, 24]
- Nodes per Layer: [64, 128, 256, 512, 1024, 2048]
- Epochs: 16
- batch size: [1, 2, 4, 8, 16, 32]
- Vector size: 500
- Activation function: swish
- Loss function: mean squared error
- Optimizer: Adam
- Learning rate: 1e-3

Figure 27: sweep 5 configuration

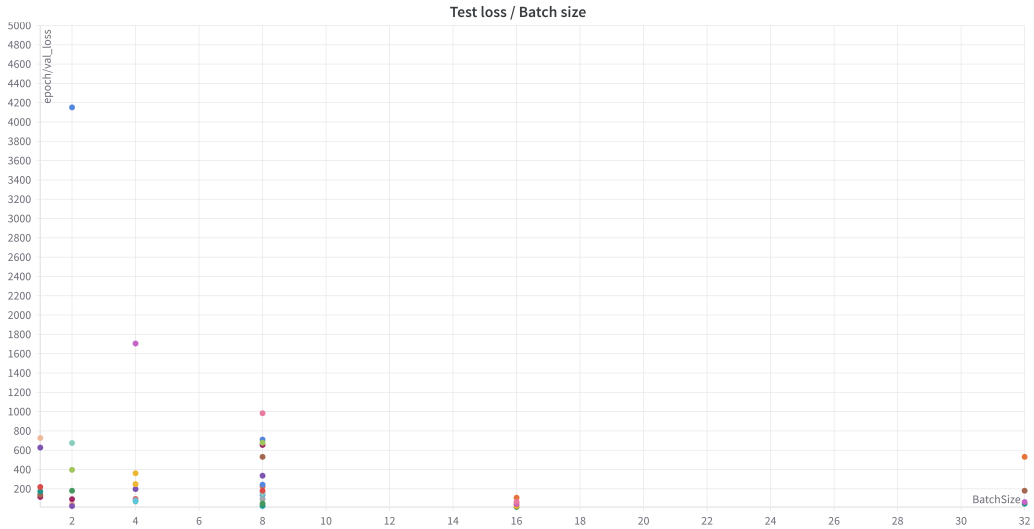
16 "layer amount", 1024 "nodes per layer" and a "batch size" of 4 was chosen.



(a) Test loss for differing amounts of "layers".



(b) Test loss for differing "nodes".



(c) Test loss for differing "batch size".

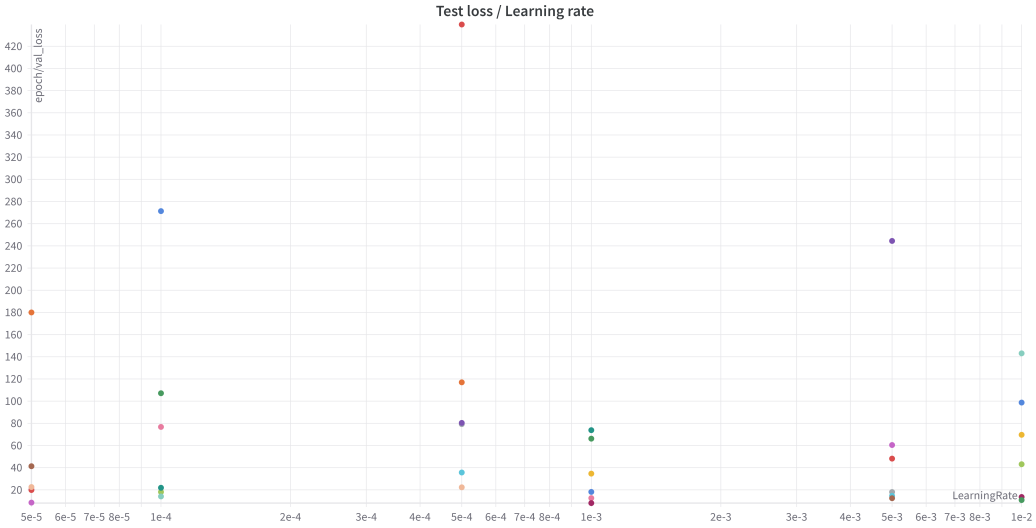
Figure 28: Sweep 5 results.

16.1.6 Sweep 6

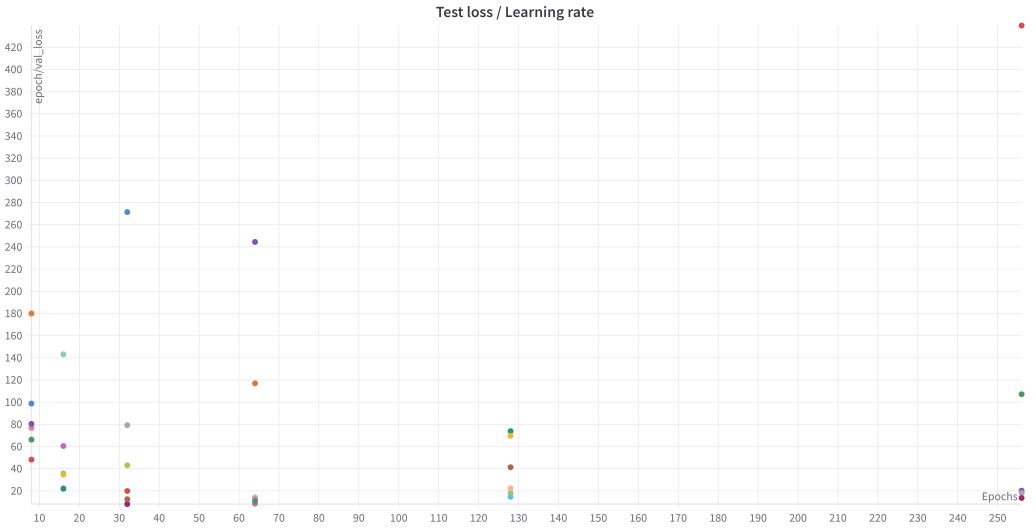
- Number of runs: 36
- Dataset size: 1000
- Layer amount: 12
- Nodes per Layer: 128
- Epochs: [8, 16, 32, 64, 128, 256]
- batch size: 8
- Vector size: 500
- Activation function: swish
- Loss function: mean squared error
- Optimizer: Adam
- Learning rate: [1e-2, 5e-3, 1e-3, 5e-4, 1e-4, 5e-5]

Figure 29: sweep 6 configuration

64 "epochs" and a learning rate of "1e-3" was chosen.



(a) Test loss for differing amounts of "learning rate".



(b) Test loss for differing "epochs".

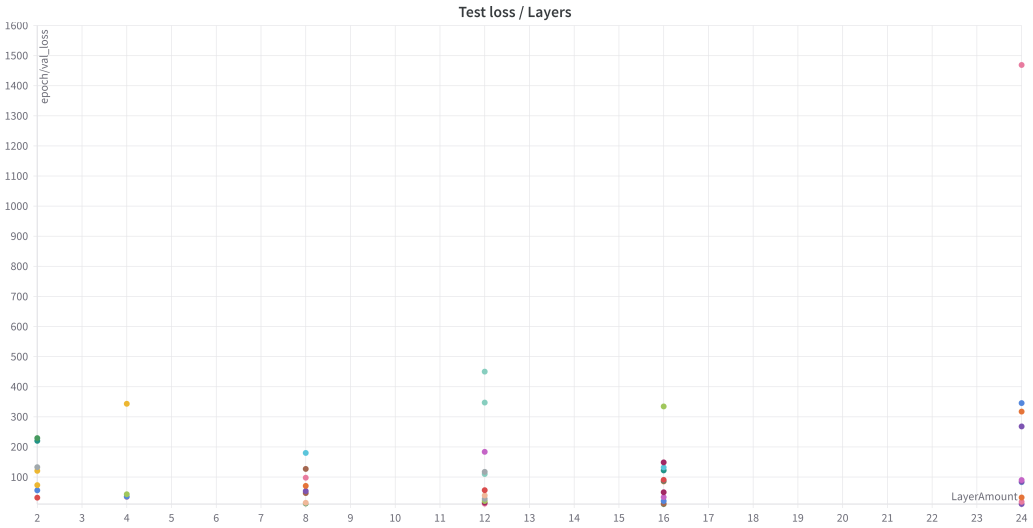
Figure 30: Sweep 6 results.

16.1.7 Sweep 7

- Number of runs: 54
- Dataset size: 1000
- Layer amount: [2, 4, 8, 12, 16, 24]
- Nodes per Layer: [64, 128, 256, 512, 1024, 2048]
- Epochs: 64
- batch size: [1, 2, 4, 8, 16, 32]
- Vector size: 500
- Activation function: swish
- Loss function: mean squared error
- Optimizer: Adam
- Learning rate: 5e-5

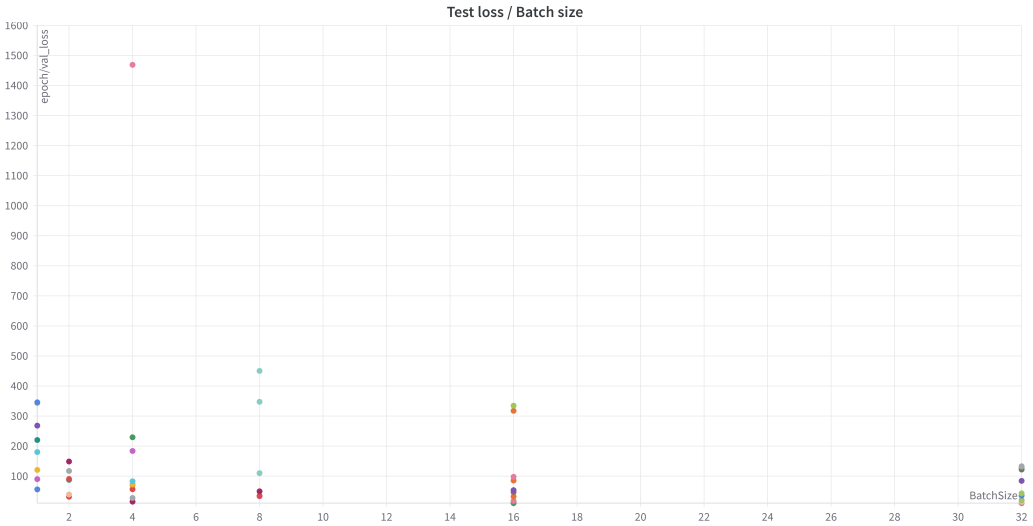
Figure 31: sweep 7 configuration

16 "layer amount", 1024 "nodes per layer" and a "batch size" of 32 was chosen.



(a) Test loss for differing amounts of "layers".

(b) Test loss for differing "nodes".



(c) Test loss for differing "batch size".

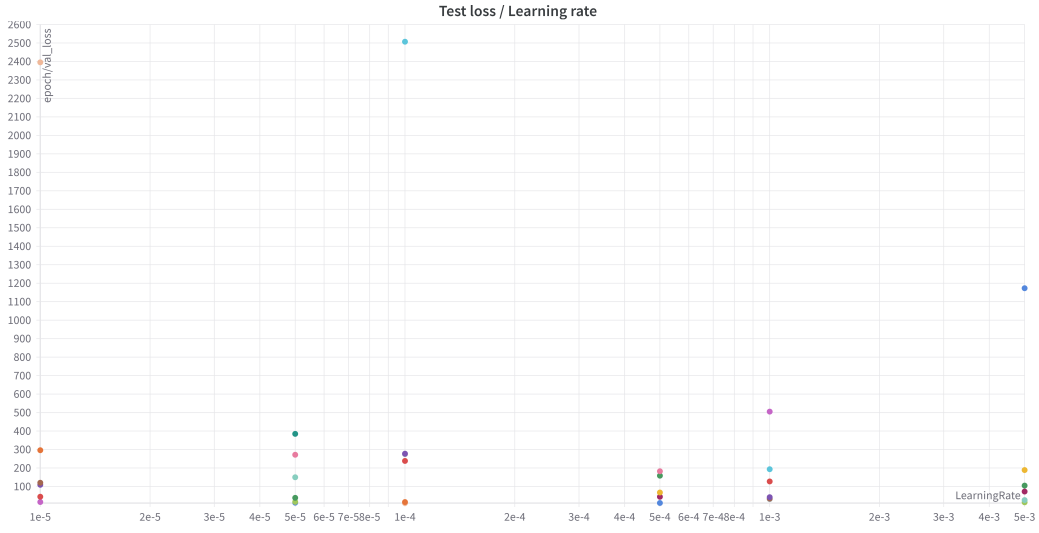
Figure 32: Sweep 7 results.

16.1.8 Sweep 8

- Number of runs: 36
- Dataset size: 1000
- Layer amount: 16
- Nodes per Layer: 1024
- Epochs: [8, 16, 32, 64, 128, 256]
- batch size: 32
- Vector size: 500
- Activation function: swish
- Loss function: mean squared error
- Optimizer: Adam
- Learning rate: [5e-3, 1e-3, 5e-4, 1e-4, 5e-5, 1e-5]

Figure 33: sweep 8 configuration

128 "epochs" and a learning rate of "5e-4" was chosen.



(a) Test loss for differing amounts of "learning rate".

(b) Test loss for differing "epochs".

Figure 34: Sweep 8 results.

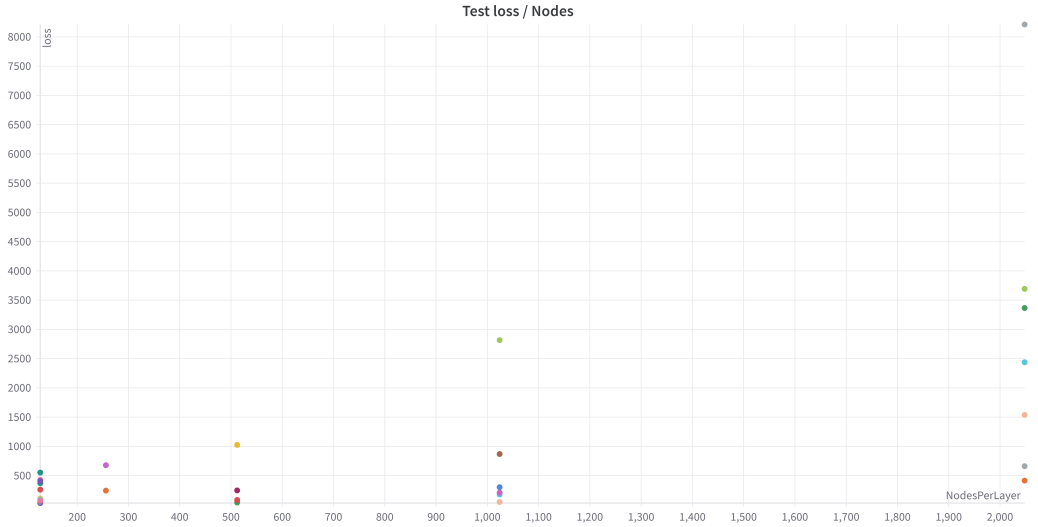
16.1.9 Sweep 9

- Number of runs: 32
- Dataset size: 1000
- Layer amount: [8, 12, 16, 20, 24]
- Nodes per Layer: [128, 256, 512, 1024, 2048]
- Epochs: 64
- batch size: [4, 8, 16, 32, 64]
- Vector size: 500
- Activation function: swish
- Loss function: mean squared error
- Optimizer: Adam
- Learning rate: 1e-3

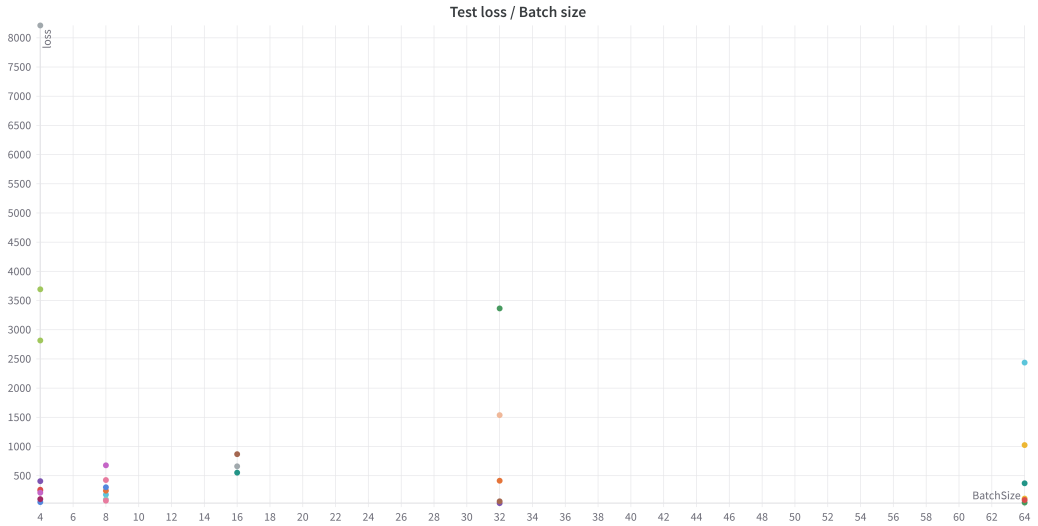
Figure 35: sweep 9 configuration

8 "layer amount", 128 "nodes per layer" and a "batch size" of 8 was chosen.

(a) Test loss for differing amounts of "layers".



(b) Test loss for differing "nodes".



(c) Test loss for differing "batch size".

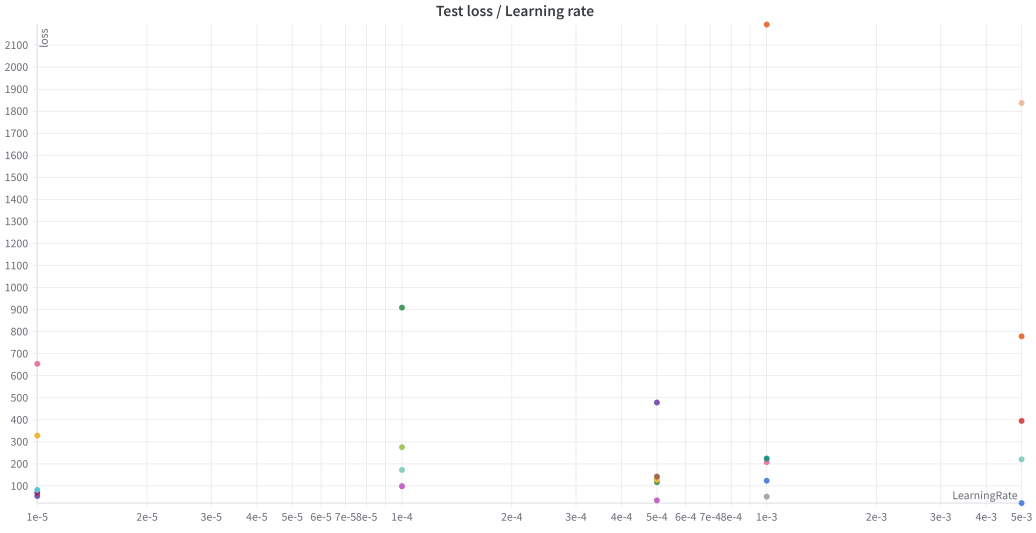
Figure 36: Sweep 9 results.

16.1.10 Sweep 10

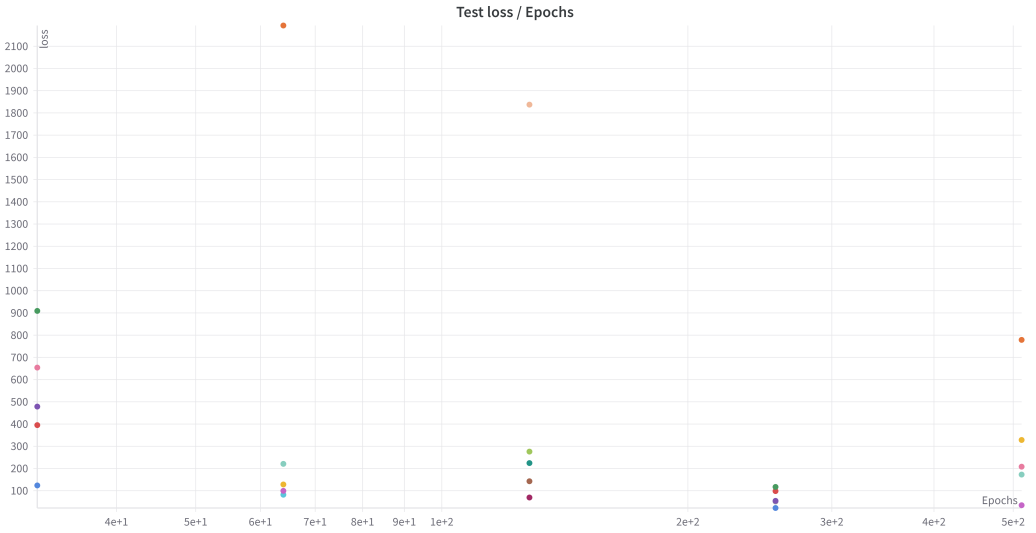
- Number of runs: 25
- Dataset size: 1000
- Layer amount: 8
- Nodes per Layer: 128
- Epochs: [32, 64, 128, 256, 512]
- batch size: 8
- Vector size: 500
- Activation function: swish
- Loss function: mean squared error
- Optimizer: Adam
- Learning rate: [1e-3, 5e-3, 1e-4, 5e-4, 1e-5, 1e-5]

Figure 37: sweep 10 configuration

256 "epochs" and a learning rate of "5e-4" was chosen.



(a) Test loss for differing amounts of "learning rate".



(b) Test loss for differing "epochs".

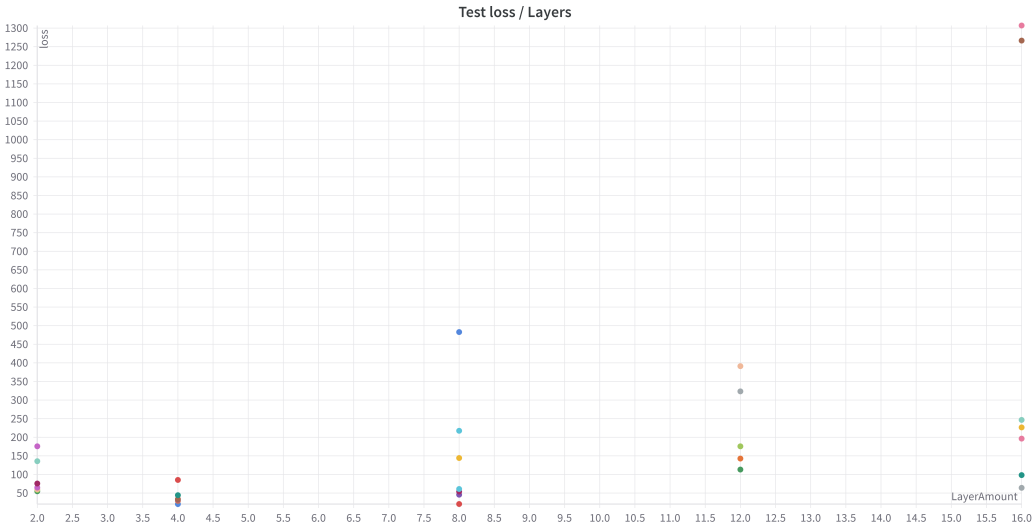
Figure 38: Sweep 10 results.

16.1.11 Sweep 11

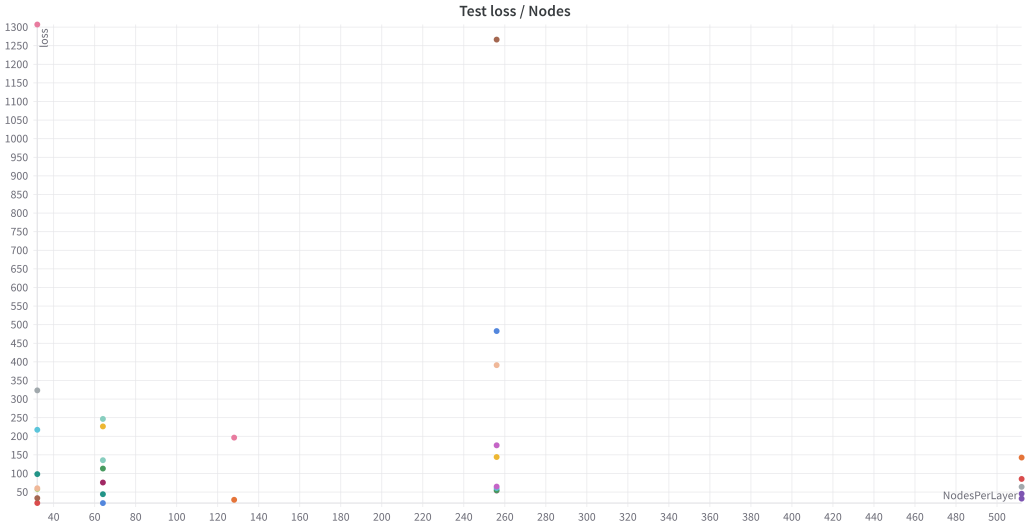
- Number of runs: 32
- Dataset size: 1000
- Layer amount: [2, 4, 8, 12, 16]
- Nodes per Layer: [32, 64, 128, 256, 512]
- Epochs: 256
- batch size: [2, 4, 8, 16, 32]
- Vector size: 500
- Activation function: swish
- Loss function: mean squared error
- Optimizer: Adam
- Learning rate: 5e-4

Figure 39: sweep 11 configuration

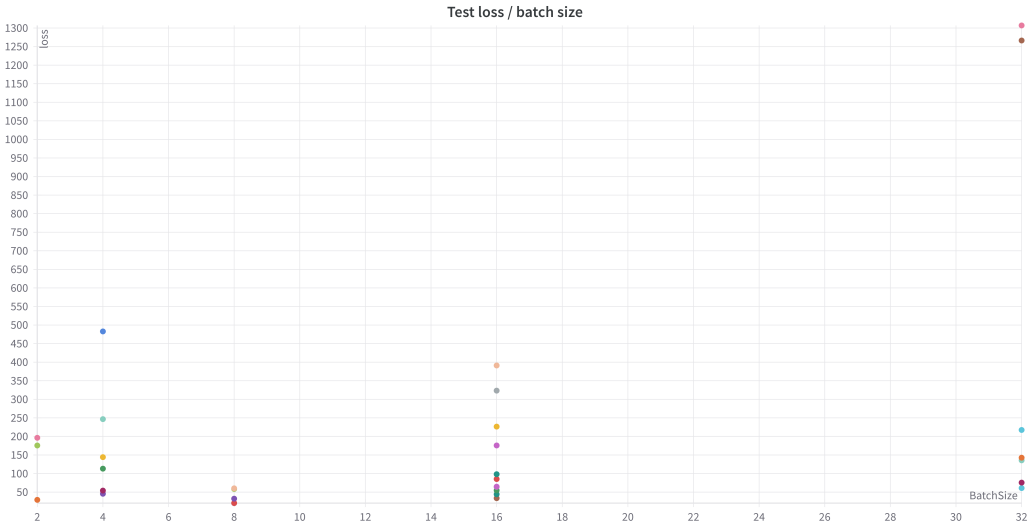
4 "layer amount", 512 "nodes per layer" and a "batch size" of 8 was chosen.



(a) Test loss for differing amounts of "layers".



(b) Test loss for differing "nodes".



(c) Test loss for differing "batch size".

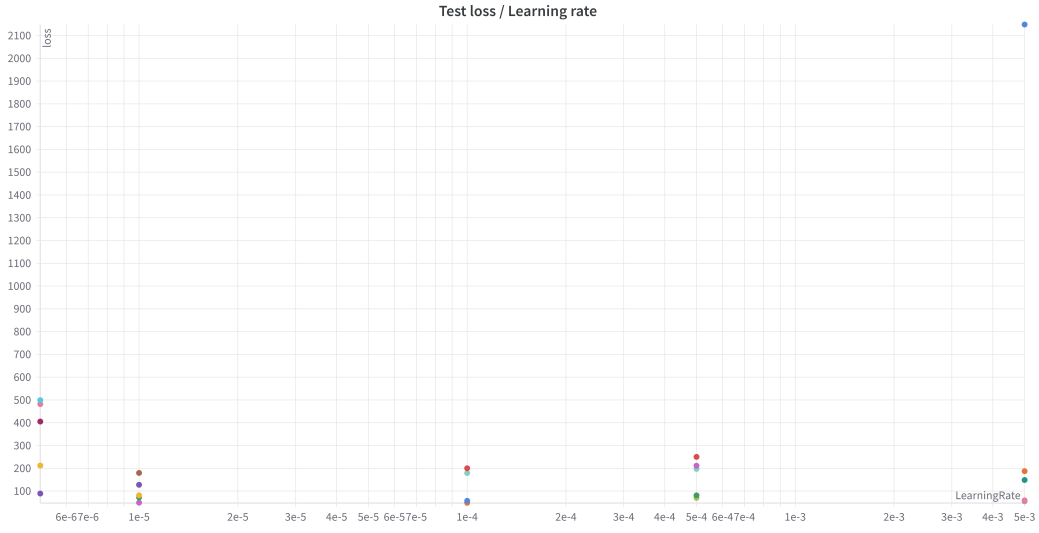
Figure 40: Sweep 11 results.

16.1.12 Sweep 12

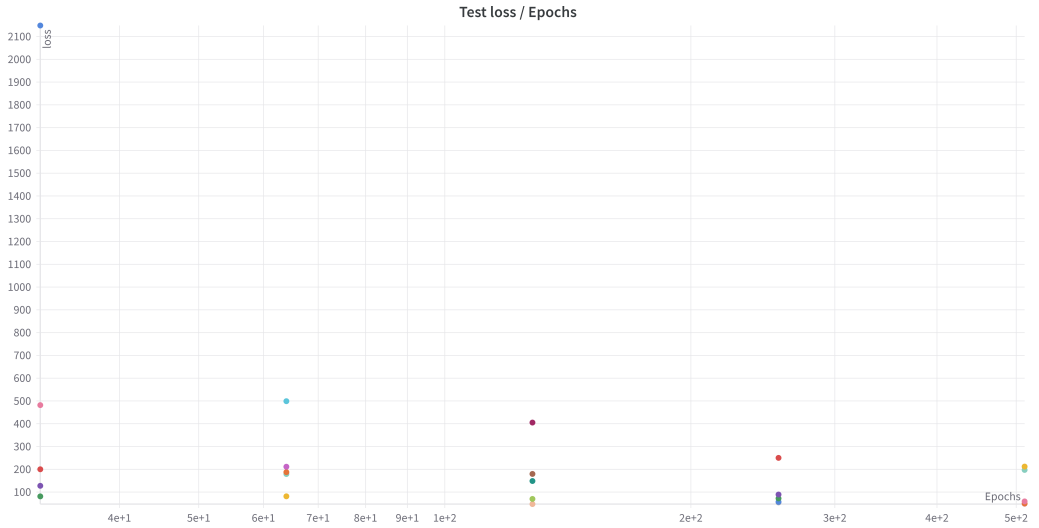
- Number of runs: 25
- Dataset size: 1000
- Layer amount: 4
- Nodes per Layer: 512
- Epochs: [32,64,128,256,512]
- batch size: 4
- Vector size: 500
- Activation function: swish
- Loss function: mean squared error
- Optimizer: Adam
- Learning rate: [5e-3, 1e-4, 5e-4, 1e-5, 5e-5, 1e-6]

Figure 41: sweep 12 configuration

512 "epochs" and a learning rate of "1e-4" was chosen.



(a) Test loss for differing amounts of "learning rate".



(b) Test loss for differing "epochs".

Figure 42: Sweep 12 results.