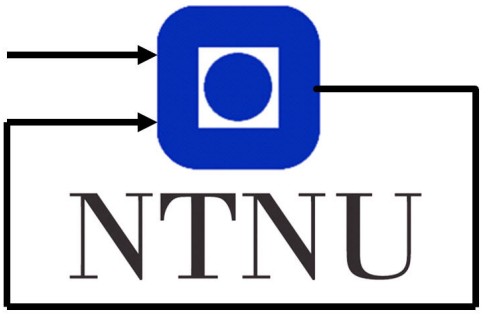
Physics-Informed Neural Networks: a basic introduction, and how they can be applied with Model Predictive Control

Jonas Kittelsen

December 2021



Department of Engineering Cybernetics

Summary

This report gives an introduction to Physics-Informed Neural Networks (PINNs), a method used to train neural networks to estimate dynamic systems. By utilizing the known differential equation of the system during training of the neural network, the need for training data is drastically reduced. Traditionally, when solving ordinary differential equations, these PINNs have a time input and outputs the state estimate. The performance of the PINNs usually degrades quickly when increasing the time horizon of the simulation, and they are not applicable to optimal control as the initial condition and control inputs are fixed. This report goes on to show how these two problems can be solved by adding the initial condition and control inputs as inputs to the PINN, as proposed by [1]. The resulting neural network is trained for a shorter horizon, but can be applied iteratively to obtain long range simulations with less degradation than traditional PINNs. It is also applicable to optimal control, due to the added inputs. The method is demonstrated by creating a neural network that reproduces the solution of the Van der Pol oscillator for a longer time horizon than traditional PINNs, and using this neural network with model predictive control to obtain similar control performance as a Runge-Kutta numerical integration scheme.

Contents

Chapter 1

Introduction

The research area of using Neural Networks (NNs) to solve differential equations by utilizing the underlying known physical laws and minimal amounts of simulation/measurement data has been going strong ever since it was introduced as Physics-Informed Neural Networks (PINNs) by [2][3]. The major strength of this method is that the physics knowledge of the system can be utilized to enforce a certain structure on the solution of the NN, for instance that it satisfies the differential equation of the system. But this physics constraint can sometimes be hard to enforce on a NN, especially when trying to simulate a system for a long time period.

Most current work on PINNs is concerned with solving initial value problems; for a given differential equation and initial condition, a NN is trained to find the solution. These traditional PINNs have the time as input and outputs the solution of the initial value problem. For a change in initial condition, a new PINN has to be trained.

However, for use with optimal control, we would like a model that can make predictions from any given current state, also considering a control input. This can be solved by adding the initial condition and control input as inputs to the NN, as proposed by [1]. The NN is trained to estimate the system on a short time horizon, where the control input is considered constant. The resulting model can then be used iteratively to obtain longer simulations, by feeding the output state of one iteration back as the initial condition of the next iteration, looping for as long as necessary to obtain the desired simulation length. The performance of this approach degrades less with increasing simulation times than training a traditional PINN to simulate the entire horizon. Also, at each iteration, a different control input may be applied, opening opportunities for use with optimal control. The article demonstrates the concept using Model Predictive Control (MPC) on two example system: the Van der Pol oscillator and the four-tank system. This method has also been applied to control a multi-link robot manipulator [4].

Even though there has been published a lot of research within the PINN field recently, few learning resources on the subject are available. I have for myself experienced that the subject can be hard to grasp, and would have welcomed better learning resources. In this report, I aim to give a detailed introduction to PINNs intended for newcomers to the field, including the extensions for use with optimal control. Using the Van der Pol oscillator as an example system, I show that the PINN extended for control is able to simulate the system with changing control input, and that it outperforms the traditional PINN framework for long range simulations. Further, I show how this PINN extended for control can be used with MPC on the example system, obtaining the same control performance as a Runge-Kutta numerical scheme.

The main components of this report are:

- A detailed introduction to PINNs is given in section 2.3.
- How the PINN framework can be extended for use with optimal control is described in section 2.4
- Some useful implementation details of the NNs in TensorFlow, along with how these models can be transferred to CasADi for use with MPC in chapter 3.
- Show that the resulting PINN extended for control can:
    - Estimate the Van der Pol oscillator with changing control input in section 4.1.
    - Outperform traditional PINNs for long range simulation in section 4.2.
    - Be used with MPC to control the example system in section 4.3.

Chapter 2

Background

This chapter will start by giving a short description of the MPC formulation used in this project. Following this is a brief introduction to machine learning and neural network in general in section 2.2. Sections 2.3 and 2.4 introduces physics-informed neural networks in detail and how these can be used to solve ordinary differential equations, this is the main part of this chapter. Finally, everything is brought together in section 2.5 incorporating the neural network model with the MPC controller.

2.1 Model Predictive Control

We will start by having a look at the Nonlinear Model Predictive Control (NMPC) formulation that will be used in this project, for more information on MPC see for instance [5]. The NMPC formulation is:

$$\min \sum_{i=1}^{N} (\mathbf{x}[k+i] - \mathbf{x}^{ref}[k+i])^T \mathbf{Q} (\mathbf{x}[k+i] - \mathbf{x}^{ref}[k+i]) + \sum_{i=0}^{N_u-1} \Delta\mathbf{u}[k+i]^T \mathbf{R} \Delta\mathbf{u}[k+i] \tag{2.1a}$$

Subjected to:

$$\mathbf{x}[k+j+1] = \mathbf{F}(\mathbf{x}[k+j], \mathbf{u}[k+j]), \quad j = 0, \ldots, N-1 \tag{2.1b}$$

$$\mathbf{u}[k+j] = \mathbf{u}[k+j-1] + \Delta\mathbf{u}[k+j], \quad j = 0, \ldots, N_u - 1 \tag{2.1c}$$

$$\mathbf{u}[k+j] = \mathbf{u}[k+N_u-1], \quad j = N_u, \ldots, N-1 \tag{2.1d}$$

$$\mathbf{h}(\mathbf{x}[k+j], \mathbf{u}[k+j-1]) \leq 0, \quad j = 1, \ldots, N \tag{2.1e}$$

$$\mathbf{g}(\mathbf{x}[k+j], \mathbf{u}[k+j-1]) = 0, \quad j = 1, \ldots, N \tag{2.1f}$$

where k represents the current time step of the MPC, $x[k]$ is then the most recent measurement or estimate of the state and $u[k-1]$ is the last applied input. The

prediction horizon of the MPC is N steps, each step has a time length of T seconds. The quadratic cost function to be minimized is composed of two terms, one for the states and one for the change in inputs. The state deviations from the reference trajectory are weighted by the matrix **Q** for the entire prediction horizon, where **Q** is a square diagonal matrix of the same dimension as **x**. Note that the same **Q** matrix is used for all time steps in this formulation, the formulation could easily be changed to include a separate **Q** matrix for each time step. The second term of the cost function penalizes the change in input from one state to the next, weighted by the square diagonal matrix **R**. The choice of penalizing the change of input instead of the input value is to avoid steady state deviations in the control loop, also for some types of actuators, such as valves, it might be desirable to minimize the change in actuation to reduce wear on the actuator. The input is only considered changing for the first N_u steps, after this the input is constant and equal to the last optimized input $u[k+N_u-1]$, as implemented by constraint 2.1d. Constraint 2.1c ensures that each of the first N_u inputs are equal to the previous input plus the change in input for that time step.

Ensuring that the solution of the MPC problem satisfy the physical structure of the underlying system is enforced by constraint 2.1b. **F** is a mapping $(x[k], u[k]) \mapsto x[k+1]$, it maps the current state and input to the next state. For linear systems, this mapping can be implemented with a discrete linear model. For nonlinear models, this mapping can for instance be implemented by a linear approximation, which can either be calculated offline or online. For the offline case one or more linear models are calculated before deployment of the MPC, during control at every iteration the linear model closest to the current state is selected. For the online case, a new linear approximation is made at the current state. Other possibilities include using numerical integration of the nonlinear differential equation. The aim of this project is to implement this mapping with a neural network.

Equation 2.1e and 2.1f implements inequality and equality constraints of the system. One use of the inequality constraints is to limit the range of the input.

2.2 Machine learning and neural networks

Machine learning is the science of creating computer algorithms that learn through experience, instead of explicitly programming them. An instance of such an algorithm is usually called a model. We could, for instance, create a model that predicts if there is a cat or not in an image. The input of this model is an image, and the output is the prediction of whether there is a cat present in the image. During the learning process, we use images where we know if there is a cat present or not to train the model to recognize cats.

Neural networks

Many algorithms can be used with machine learning, here we will only focus on the one used in this project, Neural Networks (NNs) or sometimes called artificial neural networks. This section will give a short introduction to the functioning of NNs and how they are trained, emphasizing the parts that are important for the following sections. More in depth information about NNs can be found in [6], or pretty much anywhere online.

Figure 2.1 shows a densely connected neural network with two inputs, one output and two hidden layers with 5 units/neuron/nodes each. Here densely refers to the property that each node is connected to every node of the previous layer, only densely connected networks will be used in this report.

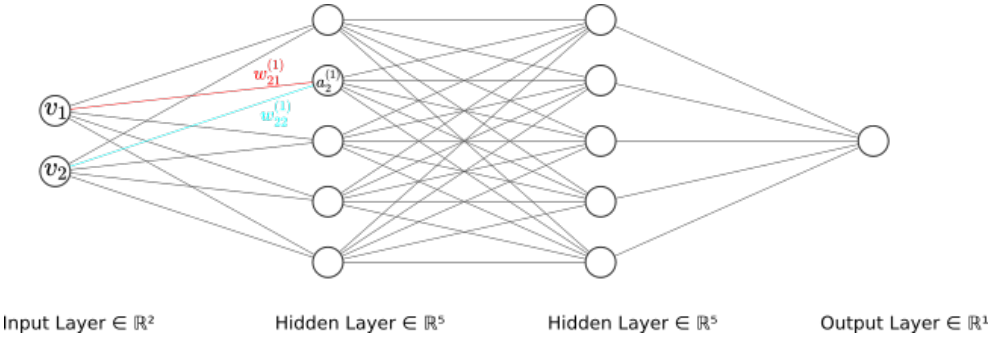


Figure 2.1: Densely connected neural network with 2 inputs, 2 hidden layers with 5 neurons each and 1 output. Densely refers to the property that each neuron is connected to all neurons of the previous layer. $w_{21}^{(1)}$ and $w_{22}^{(1)}$ are the weights associated with the second neuron of the first hidden layer, they are used together with the input activations, v_1 and v_2, and a bias term to calculate the activation of the neuron $a_2^{(1)}$. The figure was created using [7].

Prediction, forward propagation

When using the NN to make a prediction, either during training or after deployment, the data we wish to predict on is applied to the input of the NN. The input is then propagated through the network, by first calculating the activation of the neurons in the first hidden layer, then the second hidden layer and so on until we reach the output layer. These calculations going from input to output is called forward propagation or a forward pass. For the 'cat detection in image' example, an image is applied to the input layer, and the calculated output could be the probability of a cat being present in the image.

The activation of a neuron is calculated as a weighted linear combination of the activations of the neurons in the previous layer plus a bias term, which is passed through a nonlinear function called an activation function. As an example, consider the activation $a_2^{(1)}$, the second neuron in the first hidden layer in figure 2.1.

Superscript (1) refers to the neuron being part of the first hidden layer, and subscript 2 referring to that it is the second neuron of this layer. The bias term of this neuron is $b_2^{(1)}$. The activation is calculated as:

$$a_2^{(1)} = g(w_{21}^{(1)} v_1 + w_{22}^{(1)} v_2 + b_2^{(1)}) \tag{2.2}$$

where v_1 and v_2 are the inputs, or in general the activation of the neurons of the previous layer. $g(\cdot)$ is the nonlinear activation function. The forward propagation formula can be generalized to vector form as:

$$\mathbf{a}^{(l)} = \mathbf{g}(\mathbf{W}^{(l)} \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}) \tag{2.3}$$

where $\mathbf{a}^{(l)}$ is a vector of the activations in layer l, $\mathbf{W}^{(l)}$ is a matrix with all the weights for layer l and $\mathbf{b}^{(l)}$ is a vector of biases for layer l. $\mathbf{g}(\cdot)$ applies the activation function $g(\cdot)$ element wise.

Measuring the performance of the model (loss function)

During training, we do a forward pass on some training data to calculate the NN output. This output is then compared with the desired output (which we need to know for the training data), the deviation is used to calculate some measure of how good the predictions are, usually referred to as the loss of the model. This loss is then a function of the NN output and the desire output. There are several choices for this loss function, in this project the mean square error between output and desire output is used. The important part is that the value of the loss function is a scalar and should decrease as the model performance increases.

Derivatives, backward propagation

The training of the NN can then be formulated as an optimization problem where the goal is to minimize the loss function, given some training data, by updating the weights and biases of the NN. Most training algorithms updates the weights and biases by utilizing the gradient of the loss function, the loss differentiated with respect to all the weights and biases. These derivatives are calculated by propagating backward though the NN, this is usually called backward propagation, backprop or automatic differentiation. The details of how this works is not presented her (see for instance [6] for details), the important takeaway is that we can efficiently obtain the derivatives of the loss function (or NN output) with respect to any model parameter. Later we will use this to obtain the derivative of the NN outputs with respect to the NN inputs.

Training

During training of the NN we will use the training data to calculate the loss and make a step using an optimization algorithm, this is repeated for as long as neces-

sary. One of these passes through all the training data is usually called an epoch, although for L-BFGS this is commonly called an iteration. There are several optimization algorithms available for training of NNs. The two algorithms discussed in this work will now be presented.

Adam

Adam [8] is a first-order method which utilizes the gradient of the loss function to update the model weights and biases. Compared with the gradient descent algorithm, Adam has some smart additions like adjusting the step length (usually called learning rate). When using Adam, the initial step length must be set, other parameters are possible to tune, but this is usually not necessary. Adam is a very popular algorithm for training NN in general. Implementations are available in both of the most popular NN frameworks, PyTorch and TensorFlow.

L-BFGS

L-BFGS or limited-memory BFGS is a quasi-Newton method, meaning that it utilizes both the gradient and an approximation of the hessian matrix (second order derivatives) [9]. The algorithm implements a line search to find a good step length at every iteration, so unlike Adam we do not need to specify the step length (learning rate). PyTorch has an implementation of L-BFGS ready to use on NNs, whereas TensorFlow does not. General implementations are found in both the TensorFlow Probability and the SciPy libraries, these can be applied to NNs using some programming magic [10].

Validation and test data

As the model is trained to fit the training data, the performance on this dataset will be optimistic compared with utilizing the model on new data. For this reason, it is normal to set aside some of the training data for a more realistic performance measure. This data is then not used for training the model, only for evaluating the performance. Typically, a validation and a test set is created. The validation set is used during training to select for instance the NN structure and how long to train the model. The test set is only used after the final model is obtained, the performance on this dataset is the most realistic for new data.

2.3 Physics-informed neural networks

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}), \quad \mathbf{y}(0) = \mathbf{y}_0 \tag{2.4}$$

Consider the Ordinary Differential Equation (ODE) 2.4, if we would like to approximate the solution $\mathbf{y}(t)$ using a NN, one approach would be to obtain measurement or simulated data and use this for training the NN. Another approach

is to utilize the physics knowledge of the system, the ODE, together with the initial condition to train the NN, this results in a Physics-Informed Neural Network (PINN) as introduced by [2][3]. Most current work around PINNs is concerned with solving Partial Differential Equations (PDEs), a good introduction to this is given in [2][3]. Here we will only consider ODEs, which simplifies the process a bit.

The goal is to create a mapping $\boldsymbol{N}$, implemented by a NN, which maps the input t to the output $\mathbf{y}(t)$

$$\mathbf{y}(t) = \boldsymbol{N}(t), \quad t \in [0, T] \tag{2.5}$$

that satisfies the ODE 2.4 on the time interval $[0, T]$, using only the known initial condition and the ODE. Figure 2.7 shows a schematic illustration of this setup.

Figure 2.2: PINN structure for solving an ODE. The NN maps the time t to the state at this time: $\mathbf{y}(t)$.

Throughout this section, the first order system

$$\dot{y} = -y, \quad y(0) = 0.5 \tag{2.6}$$

will be used as an example to illustrate the concepts. The analytic solution is

$$y(t) = 0.5\, e^{-t}, t \geq 0 \tag{2.7}$$

The training of a PINN consists of two parts, the initial (and boundary for PDEs) condition, and the physical structure of the ODE. Let's start with the initial condition.

Imposing the initial condition

As the initial condition is a single point in the state-space, only one training example is needed to enforce this condition on the NN. The training example consists of the input, $t = 0$, and the desired output $\hat{\mathbf{y}} = \mathbf{y}(0)$. The error/loss of the NN is calculated as the Mean Square Error (MSE) of the output of the NN compared with the desired output $\hat{\mathbf{y}}$

$$MSE_y = \frac{1}{N_y}\sum_{i=1}^{N_y}(y_i(0) - \hat{y}_i)^2 \tag{2.8}$$

where N_y is the dimension of $\mathbf{y}$. Training the NN using only this loss term will enforce the initial condition.

Figure 2.3 shows a comparison of the analytic solution 2.7 of the example system in blue, and the output of a partially trained NN in orange. The deviation between the desired initial condition $\hat{y} = 0.5$ and the NN output $y(0) \approx 0.63$ is marked in the upper left corner of the figure. Squaring this deviation results in the loss MSE_y from equation 2.8. By minimizing this loss, updating the weights and biases of the NN, the NN can be trained to approach the desired output $\hat{y}$.

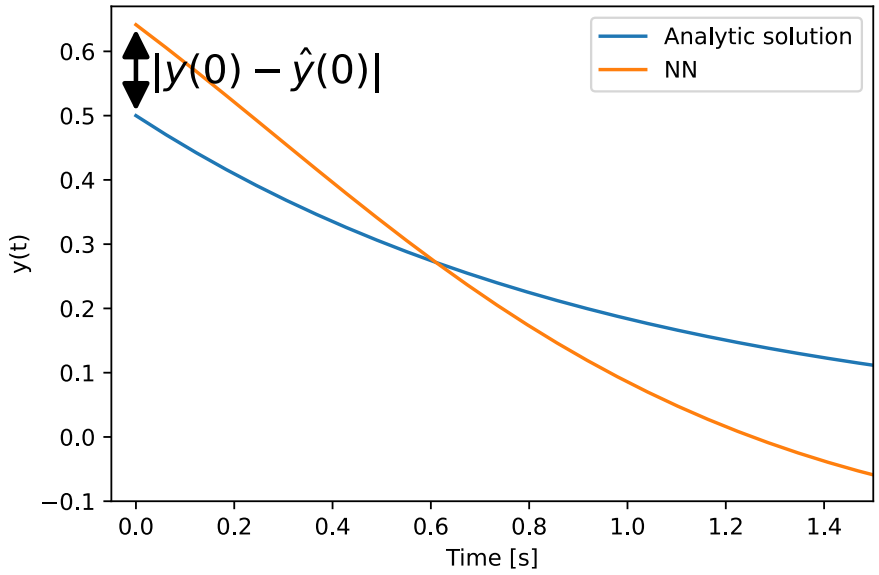


Figure 2.3: Comparing the analytic solution (in blue) of the example system (2.7) with a neural network (in orange) that is partially trained to satisfy the ODE and initial condition in equation 2.6. The black arrow highlights the deviation between the NN output for $t = 0$ and the desired output, the initial condition $\hat{y}(0) = 0.5$.

Next, we will consider how to impose the physical structure of the system using the ODE.

Imposing the underlying physics using collocation points

We want our NN to satisfy the ODE for $t \in [0, T]$. As we cannot work with a continuous interval, we will sample N_f number of points within the interval

$t^k, k = 1, \ldots, N_f$, usually with a random uniform distribution. These points are called collocation points. At each of these points we will impose the ODE on the output of the NN. To approximate the solution well, N_f must be large enough to cover the time interval well. However, it should not be too large, as this may use excessive time- and memory resources in the learning algorithm during training. This is usually not an issue when there is only one input to the NN, but when the number of inputs increases (as it will later in this chapter) this is something to take into consideration.

A measure of how well the NN satisfies the ODE is found by moving the right-hand side of the ODE equation 2.4 to the left-hand side, this difference will be defined as the residual **F**:

$$\mathbf{F}(\mathbf{y}) := \frac{\partial \mathbf{y}}{\partial t} - \mathbf{f}(\mathbf{y}) \tag{2.9}$$

where $\mathbf{y}$ will be the output of the NN. If this residual is zero, the NN reproduces the system dynamics perfectly. The partial derivative symbol is use because the NN has several variables we can differentiate with respect to (weights, biases and later, other inputs). The derivative $\frac{\partial \mathbf{y}}{\partial t}$ will also be referred to as $\mathbf{y_t}$.

The residual function 2.9 is applied to each collocation point at every iteration of the training. For a single collocation point t^k, the output of the NN $\mathbf{y}(t^k)$ is calculated. The first term of the residual is the NN output $\mathbf{y}(t^k)$ differentiated with respect to the NN input t^k, which is obtained using automatic differentiation. The second term is calculated by inserting the NN output $\mathbf{y}(t^k)$ into the ODE (2.4), which returns $\mathbf{f}(\mathbf{y}(t^k))$, the desire value of $\frac{\partial \mathbf{y}}{\partial t}$.

Let's have a look at how this applies to our first order example system (2.6) for a collocation point $t^k = 0.8$, again comparing a partially trained NN with the analytic solution in figure 2.4. The black dot represents the output of the NN at this collocation point: $y(0.8) = 0.17$. This output is differentiated with respect to the input $t^k = 0.8$ using automatic differentiation, which results in $y_t(0.8) = -0.48$, illustrated with the orange dotted tangent line in figure 2.4. This is the first term of the residual function.

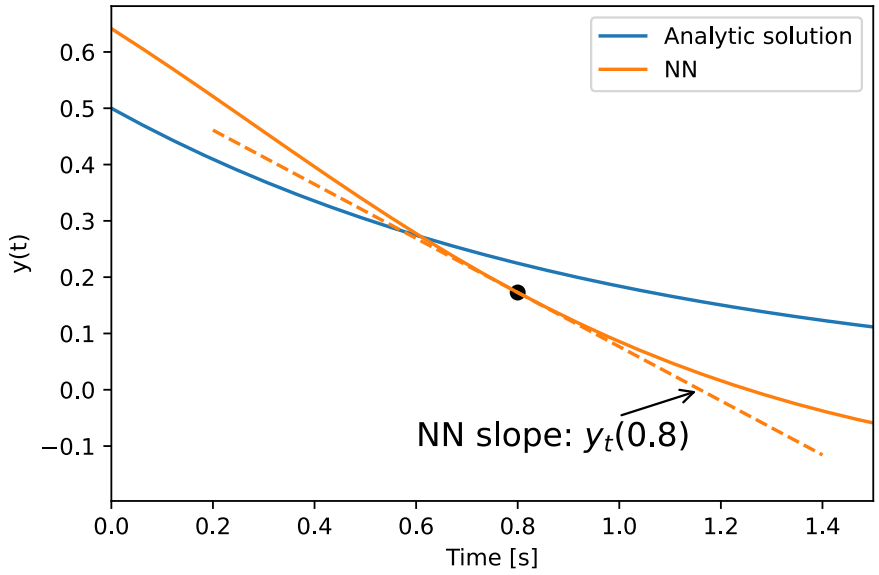


Figure 2.4: Comparing the analytic solution (in blue) of the example system (2.7) with a NN (in orange) that is partially trained to satisfy the ODE and initial condition in equation 2.6. The black dot indicates the NN output at a collocation point $t^k = 0.8$, which is $y(0.8) = 0.17$. The tangent line is the derivative of the NN output with respect to the time input, this represents the first term of the residual function in 2.9.

The second term of the residual is the desired value of the NN derivative for the output $y = 0.17$. This is calculated using the ODE (2.6):

$$f(y(0.8)) = f(0.17) = -0.17 \tag{2.10}$$

The dotted blue line in figure 2.5 illustrates this desired slope. Note that this line is not the tangent for the analytic solution at $t = 0.8$, but at the location where $y = 0.17$. From the figure we can see that the slope of the analytic solution at $t = 0.8$ is steeper than the calculated slope 2.10, but as we in general do not know the solution, we cannot easily calculate this slope.

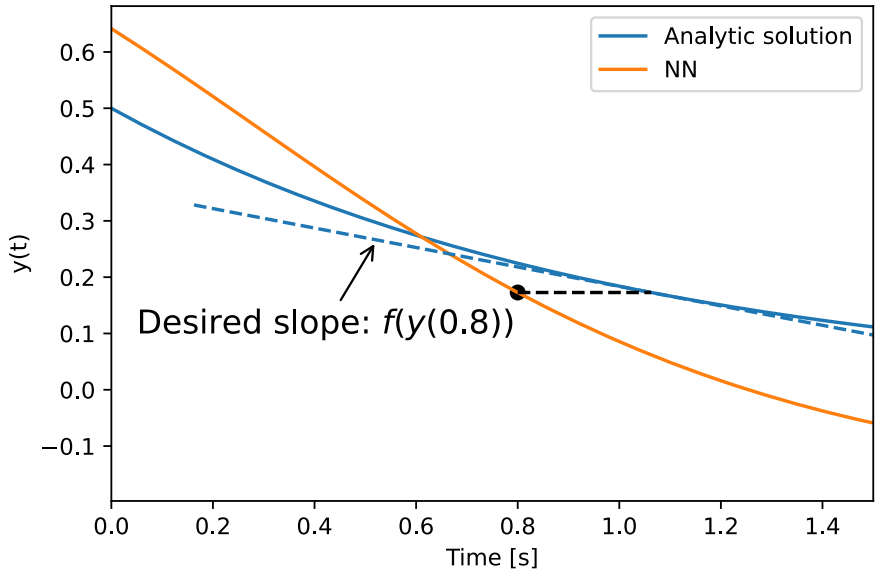


Figure 2.5: The black dot is the NN output $y(0.8) = 0.17$. The dotted blue line illustrates the ODE value of this output, $f(0.17) = -0.17$, which is the slope of the analytic solution where $y = 0.17$. This is the desired slop of our NN (when the output is $y = 0.17$). This is the second term of the residual function in 2.9.

The residual of this collocation point is the difference between the current NN slope and the desired slope:

$$F = y_t(0.8) - f(y(0.8)) = -0.48 - (-0.17) = -0.31 \tag{2.11}$$

Have a look at figure 2.6 for the complete picture. The desire slope is now moved to the collocation point (black dot) for comparison with the NN slope. The difference between these are the residual F. Our goal is to make the residual as close to zero as possible at every collocation point, then the two slopes will match and the NN will make a good approximation of the system dynamics. If the initial conditions also match the desired value, the NN will make a good approximation of the analytic solution.

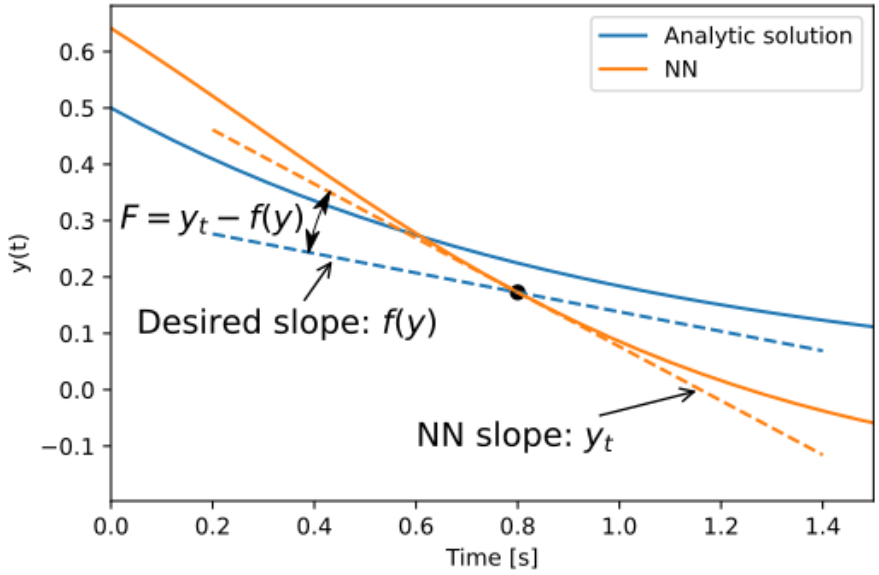


Figure 2.6: Comparing the current NN slope (dotted orange) and the desired slope calculated from the ODE (dotted blue) at a collocation point $t^k = 0.8$. The residual is the difference between these two slopes. As the residual approaches zero, the ability of the NN to reproduce the system dynamics increases.

The total loss of the collocation points is calculated as the MSE of the residuals at every collocation points

$$MSE_F = \frac{1}{N_f}\sum_{k=1}^{N_f}\frac{1}{N_y}\sum_{i=1}^{N_y}\left|F_i(\mathbf{y}(t^k))\right|^2 \tag{2.12}$$

Again, N_y is the dimension of $\mathbf{y}$, and N_f is the number of collocation points. Note that there has been no mention of the desired output for the collocation points, this is because it is not needed. This is maybe the main advantage of PINNs, we can train the NN with minimal amounts of data.

Bringing it together

Now that we have seen the two components used to train a PINN it is time to bring them together into a single loss function:

$$MSE = MSE_y + \lambda_F\, MSE_F \tag{2.13}$$

λ_F is a constant that can be used to scale the two terms according to each other. This is useful both to bring the two terms to the same approximate order of mag-

nitude, and to emphasize which term should be prioritized more by the optimization algorithm.

We can now train our PINN network by minimizing the loss function (2.13) using an optimizer such as Adam or L-BFGS. In my experience Adam is efficient for very simple systems, such as the first order example system (2.6), for more complex systems it progresses rapidly in the beginning but usually does not find a good solution. For second order systems and higher, my experience is that L-BFGS algorithms are much more efficient. Most research papers on PINNs seem to use either L-BFGS, or start with Adam and then finish with L-BFGS [2] [3] [1] [4] [11] [12].

2.4 Physics-informed neural networks extension for control purposes

The PINNs considered in the last section were trained for a single initial condition, which is not very useful for control purposes, as we would like to make a prediction based on the last measurement (or estimate) of the state. The control input $\mathbf{u}$ also needs to be considered if the system is to be controlled. This section is based on [1] which proposes to add the initial condition and a constant control input $\mathbf{u}$ as inputs to the NN, as shown in figure 2.7.

Figure 2.7: Physics-informed neural network with time t, initial condition $\mathbf{y}(0)$ and control input $\mathbf{u}$ as inputs. This NN can make predictions of the state at time t given any initial condition and control input.

Here the time interval $[0, T]$ that the NN is trained for should be chosen small enough so that the control input $\mathbf{u}$ can be considered constant (note that $\mathbf{u}$ is a vector containing all the input to the system, not a sequence of a single input, so for a single input system u is scalar). When using the NN with MPC, a good choice of T is the length of each step of the MPC, because $\mathbf{u}$ will be constant in this interval. Note that both the prediction horizon of the NN and the length of a step in the MPC is referred to as T, these will have the same value for an MPC and the associated NN and will be used interchangeably, more on this in the next section.

We need to know the ranges of the new inputs, $\mathbf{y}(0)$ and $\mathbf{u}$, as we need to sample them for the training data. The range of the initial condition $\mathbf{y}(0)$ should span all the possible values of the state $\mathbf{y}$ during operation, so that the NN is able to make a good prediction from any current state. The range of $\mathbf{u}$ should include all possible values of the input. The new formulation of the NN is:

$$\mathbf{y}(t) = \boldsymbol{N}(t, \mathbf{y}(0), \mathbf{u}), \quad \begin{cases} t \in [0, T] \\ \mathbf{y}(0) \in [\mathbf{y}^{min}, \mathbf{y}^{max}] \\ \mathbf{u} \in [\mathbf{u}^{min}, \mathbf{u}^{max}] \end{cases} \tag{2.14}$$

As the NN now has more inputs, some changes has to be made both to the training data for the initial condition and the collocation points. The new inputs need to be included in the training points.

Initial condition

In the last section, we only needed one point to train the initial conditions. However, now we want to reproduce all possible initial conditions within the domain of $\mathbf{y}$, and these should be reproduced regardless of the value of $\mathbf{u}$. We will create N_t training data points to enforce the initial condition. These data points are $\mathbf{v}^t$, $t = 1, \ldots, N_t$, where each data point consists of the three inputs: $\mathbf{v}^t = (t^t, \mathbf{y}(0)^t, \mathbf{u}^t)$. Each training data point will have an associated desired output $\hat{\mathbf{y}}^t$. As these points are used to train the initial conditions all $t^t = 0$ and all $\hat{\mathbf{y}}^t = \mathbf{y}(0)^t$. As an example, for a system with one state and one control input, a training data point could be $\mathbf{v}^t = (0, 1.4, 0.3)$ with desired output $\hat{\mathbf{y}}^t = 1.4$. This means that at $t = 0$ the initial condition is 1.4 and the control input is 0.3, then the output of the NN should be 1.4. The points for $\mathbf{y}(0)$ and $\mathbf{u}$ should be sampled randomly from their respective ranges to cover the entire input space well.

The loss function needs to be updated to calculate the MSE of all the training points:

$$MSE_y = \frac{1}{N_t} \sum_{j=1}^{N_t} \frac{1}{N_y} \sum_{i=1}^{N_y} \left(y_i(\mathbf{v}^j) - \hat{y}_i^j \right)^2 \tag{2.15}$$

Collocation points

The collocation points also need to include the new inputs. They are now $\mathbf{v}^k = (t^k, \mathbf{y}(0)^k, \mathbf{u}^k)$, where all three are sampled randomly from their respective domains. The residual function must be changed to include the input $\mathbf{u}$:

$$\mathbf{F}(\mathbf{y}) := \frac{\partial \mathbf{y}}{\partial t} - f(\mathbf{y}, \mathbf{u}) \tag{2.16}$$

Again, this residual function is applied to every collocation point. For a single collocation point $\mathbf{v}^k$, a forward pass of the NN calculates the output $\mathbf{y}(\mathbf{v}^k) = \boldsymbol{N}(\mathbf{v}^k)$. This output is then differentiated with respect to t, the first element of the collocation point, resulting in the first term of the residual. The second term is calculated by inserting the output $\mathbf{y}(\mathbf{v}^k)$ and the control input $\mathbf{u}^k$ into the ODE, this is the desired value for $\frac{\partial \mathbf{y}}{\partial t}$ at the given $\mathbf{y}$ and $\mathbf{u}$.

Long range simulation by looping the NN

If we wish to simulate the system for a longer range than the prediction horizon T, we can loop the NN by applying the output of the first NN as the initial condition of the second, and so on. The first iteration will simulate for $t \in [0, T]$, the second iteration for $t \in [T, 2T]$, and so on. This also allows for a different control input to be applied at every time step. Figure 2.8 shows an illustration of how the output of the NN is feedback into the initial condition input of the NN creating a self-loop mode. Note that the time t is replaced by T to utilize the full prediction horizon of the NN.

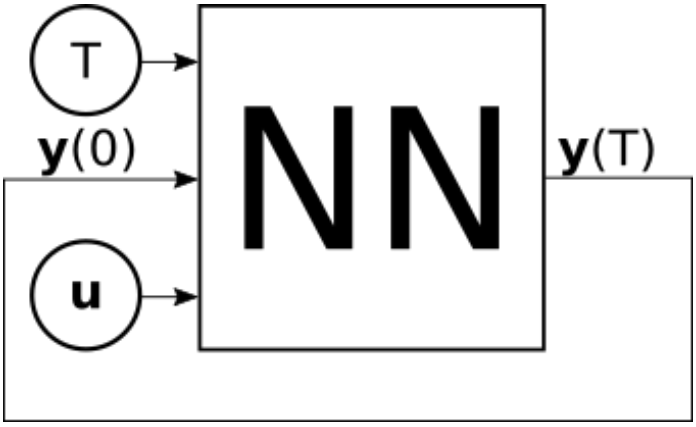


Figure 2.8: Illustration of how the output sate prediction can be feed back as the initial condition input to the NN to create a self-loop. For the first iteration of the self-loop mode, a known initial condition is feed into the NN. The output state prediction of the first iteration is then used as the initial condition input of the second iteration of the self-loop. This looping can be continued for as long as necessary to obtain the desired prediction length.

Say we want to simulate a system from a known initial condition $\mathbf{y}_0$ for $t \in [0, 10T]$ with a known input sequence $\mathbf{u}[k], k = 0, \ldots, 9$. The prediction at $t = T$, the end of the first iteration is given by:

$$\mathbf{y}[1] = \mathbf{y}(T) = \boldsymbol{N}(T, \mathbf{y}_0, \mathbf{u}[0]) \tag{2.17}$$

which can then be used as the initial condition to obtain the state prediction at $t = 2T$. For the following time steps, the prediction at the end of the step is given

by:

$$\mathbf{y}[k+1] = \boldsymbol{N}(T, \mathbf{y}[k], \mathbf{u}[k]), \quad i = 1, \ldots, 9 \tag{2.18}$$

This results in the output sequence $\mathbf{y}[k], k = 1, \ldots, 10$. If we wish to obtain a higher resolution in the simulation, we can sample points within the prediction horizon $[0, T]$ of the NN by replacing T in equation 2.17 and 2.18 with $t \in [0, T]$. For instance, sampling at $t = 2.5T$ is obtained by $\mathbf{y}(2.5T) = \boldsymbol{N}(0.5T, \mathbf{y}[2], \mathbf{u}[2])$.

2.5 Using PINNs with MPC

It is time to put it all together, using the NN with MPC. As stated earlier, the goal has been to implement the mapping $\mathbf{y}[k+1] = \mathbf{F}(\mathbf{y}[k], \mathbf{u}[k])$ in equation 2.1b by a NN. This is done by setting $t = T$ in equation 2.14, as done for the example in equation 2.18. The mapping is then implemented as:

$$\mathbf{y}[k+1] = \mathbf{F}(\mathbf{y}[k], \mathbf{u}[k]) = \boldsymbol{N}(T, \mathbf{y}[k], \mathbf{u}[k]) \tag{2.19}$$

Again, note that T refers both to the prediction horizon of the NN and for the step length of the MPC, both will have the same numeric value, there will therefor not be made any distinction between the two T's.

Figure 2.9 shows a schematic illustration of how the future states $\mathbf{y}[k+j], j = 1, \ldots, N$ are calculated from the current state $\mathbf{y}[k]$ and the inputs $\mathbf{u}[k+j], j = 0, \ldots, N-1$. This is an alternate way of illustrating the self-loop mode with N iterations.

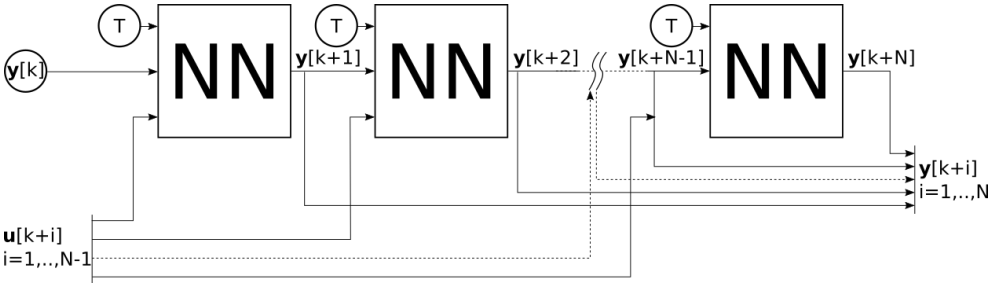


Figure 2.9: Predicting the future states $\mathbf{y}[k+j], j = 1, \ldots, N$ using the current state $\mathbf{y}[k]$ and the control inputs $\mathbf{u}[k+j], j = 0, \ldots, N-1$. This scheme implements the system dynamics constraint 2.1b in MPC. Each of the NN boxes makes a prediction T seconds forward in time, making the total prediction horizon NT seconds. This is a different way of illustrating the self-loop mode shown in figure 2.8, it has the same function.

Chapter 3

Implementation

This chapter will start by introducing the ODE of the Van der Pol oscillator, which will be used for the experiments in this report. Then the structure for approximating this ODE using the traditional PINN framework is shown in section 3.2, and the PINN extended for control in section 3.3, showing some code of the TensorFlow implementation. The datasets used to measure the performance of the models is shown in section 3.4. Section 3.5 shows how the NNs can be transferred to the CasADi framework for use with MPC. Finally, the metrics used to measure the performance of the NNs predictions and the MPC controllers are presented in section 3.6.

Most parameters used for the NNs are taken from [1]. Comparing different NN structures (like number of layers and neurons per layer) requires training many NNs, which is very time-consuming (especially without proper hardware). I therefore chose to use little time on experimenting with the NN structure in this project. When I will apply these techniques to a new system in my mater's thesis, this will be given more attention.

3.1 Van der Pol oscillator

$$\begin{aligned} \dot{y}_1 &= y_2 \\ \dot{y}_2 &= \mu(1-y_1^2)y_2 - y_1 + u \end{aligned} \tag{3.1}$$

Equation 3.1 shows the ODE of the Van der Pol oscillator, which will be used for the experiments in this report. For all experiments $\mu = 1$. The system state is $\mathbf{y} = [y_1, y_2]^T$, and u is the control input. The control input will be limited within the range $u \in [-1, 1]$.

By long range simulation of the system with randomized control inputs (within the input range), the upper and lower extreme values of the states were found.

By adding some margins to these values, the following ranges for the states were obtained:

$$\begin{aligned} y_1 &\in [-2.5, 2.5] \\ y_2 &\in [-3.7, 3.7] \end{aligned} \tag{3.2}$$

These ranges will be used when training the PINN extended for control.

3.2 Classic PINN setup

In the framework of classical PINNs we want to solve the ODE 3.1 for $t \in [0, T]$ given an initial condition $\mathbf{y}(0)$ and a constant input u, as discussed in section 2.3. The input of the PINN is the time t and the output is the state $\mathbf{y}(t)$.

Two NN structures will be considered in the experiments to follow:

- PINN1 - 4 hidden layers with 20 neurons each.
- PINN2 - 10 hidden layers with 200 neurons each.

PINN1 has the structure from [1], while the structure of PINN2 was selected to be a lot larger. Each of the hidden layers has the hyperbolic tangent function as activation, while the output layer has no activation function. Both PINN1 and PINN2 will be trained for five different simulation times: 0.5, 2, 5, 10 and 20 seconds. A new model is trained for each simulation time, for a total of five PINN1 models and five PINN2 models. For all simulations, the initial condition is $\mathbf{y}(0) = [0.3, -1.5]^T$ and the constant control input is $u = 0.5$.

Create NN model

The PINN1 model can be constructed in Tensorflow using the following example code:

Code listing 3.1: Example of creating a NN sequential model in tensorflow.

```
1  import tensorflow as tf
2  from tensorflow.keras.layers import Dense
3
4  model = tf.keras.Sequential()              #Create a new NN model
5  model.add(Dense(20, activation='tanh'))    #Create the hidden layers
6  model.add(Dense(20, activation='tanh'))
7  model.add(Dense(20, activation='tanh'))
8  model.add(Dense(20, activation='tanh'))
9  model.add(Dense(2))                        #output layer
10 model.build(input_shape=(1,1))             #set the input shape
```

`Sequential` constructs a NN model where the layers are stacked sequentially, such that the output of one layer is the input of the next. Lines 5-9 adds dense (fully connected) layers to the model, first four hidden layers then the output layer

which has no activation and size 2 (the dimension of $\mathbf{y}$). The final line sets the input shape and builds the model, which initializes the weights.

Initial condition training data

The following two lines of code generates the single training data point needed to enforce the initial condition $\mathbf{y}(0)$ on the NN. A short repetition: when we pass $t = 0$ to the NN, the desired output of the NN is $\mathbf{y}(0)$. So, the first line of code is the input to the NN, and the second is the desired output.

Code listing 3.2: Create initial condition training data for the PINN.

```
t_train = tf.zeros((1,1))          #t=0
y_train = tf.reshape(y0, (1,2))    #y0 - initial condition
```

During each training iteration, we need to pass the training data point `t_train` through the NN to obtain the output `y`, then calculate the loss function from equation 2.8:

Code listing 3.3: A forward pass of the NN followed by calculating the loss function 2.8.

```
y = model(t_train)
MSE_y = tf.reduce_mean((y - y_train) ** 2)
```

Collocation points

$N_f = 1000 \cdot T$ collocation points are sampled from a uniform distribution within the time horizon $t \in [0, T]$.

Code listing 3.4: Create randomly sampled collocation points.

```
N_f = tf.math.floor(1000 * T)                    #ensure integer value by flooring
t_collocation = tf.random.uniform((N_f,1), 0, T)
```

At every iteration of the training, we have to calculate the collocation loss function(2.12) along with the initial condition loss in code listing 3.3.

Code listing 3.5: Calculate the loss of the collocation points

```
1  with tf.GradientTape(persistent=True) as tape:
2    tape.watch(t_collocation)     #ensures that we can differentiate wrt. t
3    y = PINN(t_collocation)
4
5    y1 = tf.reshape(y[:,0], (N_f,1))
6    y2 = tf.reshape(y[:,1], (N_f,1))
7
8  y1_t = tape.gradient(y1, t_collocation)
9  y2_t = tape.gradient(y2, t_collocation)
10 del tape
11
12 y_t = tf.concat((y1_t, y2_t), 1)
13
14 F = y_t - f(t_collocation, y1, y2)
15 MSE_F = tf.reduce_mean(F ** 2)
```

For TensorFlow to track the operations so that we can utilize automatic differentiation, we need an `GradientTape` object, here called `tape`. Anything that happens within the scope of this tape (the indent in lines 2-6) will be recorded by the tape, and derivatives can be obtained. The `persistent` is set so that it is possible to call the `tape.gradient()` function several times, if not the tape will be deleted at the first call in line 8 and return an error in line 9. The tape must now be deleted manually (line 10), if not we will continue storing tapes at every training iteration and leak memory.

Notice that the state `y` is decomposed into `y1` and `y2`, and that this happens within the recording scope of the tape. If we simply call `tape.gradient(y, t_collocation)`, the result would not be a vector of the gradients $[\frac{\partial y1}{\partial t}, \frac{\partial y2}{\partial t}]$ as we might expect, but the gradient of the sum of the two outputs $\frac{\partial (y1+y2)}{\partial t}$. That is the reason for the decomposition in lines 5-6, it has to be done within the recording scope of the tape (indent) to allow for the gradient calculations in lines 8-9. Line 14 calculates the residuals from equation 2.9, here `f()` is the ODE function shown in code listing 3.6 below. Finally, the loss is calculated as the MSE of the residuals `F`.

Code listing 3.6: Van der Pol ODE function with constant predefined input.

```
def f(t, y1, y2):
  f1 = y2
  f2 = (1 - y1**2) * y2 - y1 + u                #u is a constant predefined input
  return tf.concat((f1,f2),1)
```

Training loop

How the training loop is constructed depends on which training algorithm is used, but the main idea is the same. The following code shows how the Adam optimizer is applied:

Code listing 3.7: Training loop using the Adam optimization algorithm.

```
1 for i in range(epochs):
2     with tf.GradientTape() as tape:
3         loss = calculate_loss(t_train, y_train, t_collocation)
4
5     gradients = tape.gradient(loss, model.trainable_variables)
6     optimizer_Adam.apply_gradients(zip(gradients, model.trainable_variables))
```

`calculate_loss()` calculates the loss function 2.13 utilizing the code in listings 3.3 (MSE_y) and 3.5 (MSE_F). The gradient of the loss with respect to the models' trainable variables (weights and biases) is calculated in line 5, and passed to the Adam algorithm, which updates the weights and biases. This is repeated in the loop for the given number of epochs.

As TensorFlow does not have an implementation of L-BFGS ready for NNs this requires some more code to apply a general implementation of L-BFGS from SciPy

(or TensorFlow Probability). This algorithm requires the variables (model parameters in this case) to be presented in a column vector. We need to provide a function that returns a scalar loss to be minimized along with a column vector of the gradients. So, we need to transform the model parameters into a column vector, they are store as one weight matrix and one bias vector for each layer within the model. The following code shows the idea, lacking some implementation details:

Code listing 3.8: Training loop using L-BFGS.

```
1  def f_val_and_grad(x):                  #x is a 1D vector of the model parameters
2      update_model_parameters(model, x)   #update the model with the new parameters
3
4      with tf.GradientTape() as tape:
5        loss = calculate_loss(t_train, y_train, t_collocation)
6
7      gradients = tape.gradient(loss, model.trainable_variables)
8      gradients_vec = parameters_to_1D_vector(gradients)
9
10     return loss, gradients_vec
11
12 initial_parameters = parameters_to_1D_vector(model.trainable_variables)
13 scipy.optimize.fmin_l_bfgs_b(f_val_and_grad, initial_parameters, maxiter=epochs)
```

`f_val_and_grad` is the function to be minimized. `x` is the models' weights and biases extracted into a single column vector, the model is updated with these new values in line 2. Line 4-6 calculates the loss and gradients, as for the Adam case, both are returned to the optimization algorithm. The initial model parameters are extracted in line 12, these are used as the initial position of the optimizer, so `x` will be equal to this initial position in the very first call to `f_val_and_grad`. The last line runs the optimization algorithm, the loop happens within this function. A more comprehensive guide to using this optimizer is given in [10].

3.3 PINN for control - PINC

This PINN model with the initial condition and control input as additional inputs will have 4 hidden layers of 20 neurons each, as the PINN1 model. The model will be referred to as PINC (physics-informed neural network for control), a name introduced by [1]. The PINC model will be trained to simulate the system for $T = 0.5$ seconds, to obtain longer simulations it is put in self-loop mode as discussed in section 2.4.

Initial condition training data

1 000 training data points are created to train the initial conditions. The time is $t = 0$ for all points, while $\mathbf{y}$ and u are sampled from a uniform distribution within their ranges.

Code listing 3.9: Create initial condition training data for the PINC.

```
N_t = 1000
t_train = tf.zeros((N_t,1))
y0_train = tf.random.uniform((N_t,2), minval=y_min, maxval=y_max)
u_train = tf.random.uniform((N_t,1), minval=u_min, maxval=u_max)

y_train = y0_train;
```

Collocation points

100 000 collocation points are generated, spanning the ranges of all three inputs.

Code listing 3.10: Create randomly sampled collocation points.

```
N_f = 100000
t_collocation = tf.random.uniform((N_f,1), minval=0, maxval=T)
y0_collocation = tf.random.uniform((N_f,2), minval=y_min, maxval=y_max)
u_collocation = tf.random.uniform((N_f,1), minval=u_min, maxval=u_max)
```

The ODE function is modified to include the input u, it is now passed as an input to the `f()` function instead of being a predefined constant.

Code listing 3.11: Van der Pol ODE function with input u.

```
def f(t, y1, y2, u):
  f1 = y2
  f2 = (1 - y1**2) * y2 - y1 + u
  return tf.concat((f1,f2),1)
```

Other than this, the implementation is very similar to that of the last section.

3.4 Validation and test data

To measure the performance of the models we need something to compare against, this will be simulations of the system generated using a 4th order Runge-Kutta method. The validation data is used to measure the performance of the PINC model during training and will be used to choose the number of training iterations. Test data is not used during training, only for measuring the performance of the final models.

PINN

The PINN1 and PINN2 models will be trained for a fixed number of iterations, no validation data will be used during training. The test data for evaluation of the final models is created using initial condition $\mathbf{y}(0) = [0.3, -1.5]^T$ and constant control input $u = 0.5$. Test data for $T = 10$ seconds is show in figure 3.1.

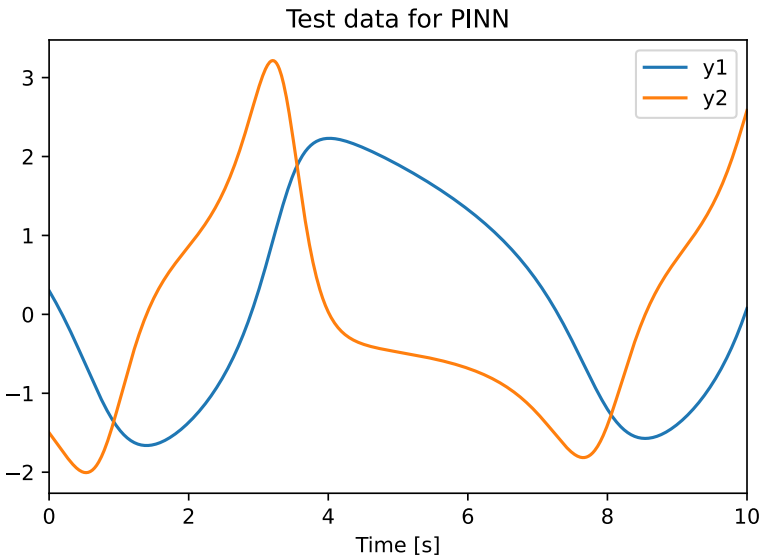


Figure 3.1: Runge-Kutta simulation of the Van der Pol oscillator for 10 seconds with initial condition $\mathbf{y}(0) = [0.3, -1.5]^T$ and constant control input $u = 0.5$. This test data will be used to evaluate the performance of the PINN1, PINN2 and PINC models.

PINC

For validation and test data for the PINC model, an input sequence of 20 steps is sampled from a uniform distribution. The validation data used during training along with the random input is shown in figure 3.2. The dots represent the values of the states at the discrete points separated by $T = 0.5$ seconds, only these points will be used to calculate the validation error during training. At every iteration of training, the validation performance is computed by running the PINC model in self-loop mode applying the input sequence from the validation data, comparing the PINC outputs with the validation data outputs. Test data used for performance measurements of the final model is generated in the same manner, but with a different input sequence.

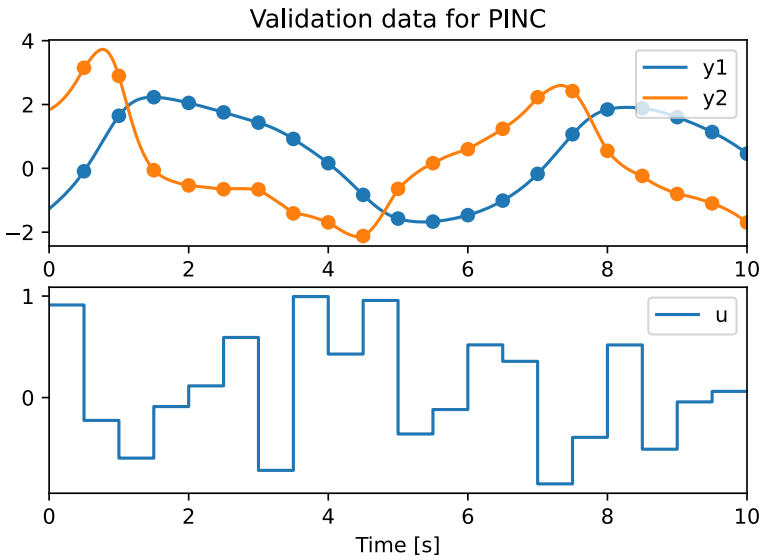


Figure 3.2: Validation data used for training the PINC model. u is sampled randomly from a uniform distribution. The outputs are simulated using a 4th order Runge-Kutta method. The dots indicate the states at the discrete time steps of $T = 0.5$ seconds, representing on self-loop iteration of the PINC model. During training, the PINC model will be run in self-loop mode with this input sequence, and the output will be compared with the output of this Runge-Kutta simulation to compute the validation loss.

3.5 MPC setup

The MPC will be implemented in CasADi, an open-source framework for numerical optimization. The prediction horizon of the MPC is set to $N = 5$ steps of $T = 0.5$ seconds each, totaling to 2.5 seconds of prediction. The control input is allowed to change at every step, so $N_u = N = 5$. The control input is constrained to $u \in [-1, 1]$. The weights used in the cost function (2.1a) are:

$$\mathbf{Q}\begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix}, \quad R = 1 \tag{3.3}$$

The mapping $\mathbf{x}[k+1] = \mathbf{F}(\mathbf{x}[k], \mathbf{u}[k])$ from equation 2.1b will be created in two different ways: one with the PINC model, and one with Runge-Kutta numerical integration for comparison. The code for creating this MPC setup in CasADi will not be given here, an example code can be found in the CasADi docs [13]. Only the implementation of the two mappings will be shown.

PINC mapping

We start by creating the mapping with the PINC model. The weights and biases are copied from the TensorFlow model and pasted into `NN_weights` and `NN_biases`.

The code for creating this mapping is shown below:

Code listing 3.12: Implementing a NN as a CasADi function.

```
1  T = 0.5
2  y0 = SX.sym('y0', 2)
3  u = SX.sym('u', 1)
4  v = vertcat(T, y0, u)
5
6  a = v
7  for i in range(0, NN_layers-1):
8    z = transpose(NN_weights[i]) @ a + NN_biases[i]
9    a = tanh(z)
10 y = transpose(NN_weights[-1]) @ a + NN_biases[-1]
11 F_NN = Function('F_NN', [y0, u], [y])
```

Line 1-4 sets up the inputs to the NN. $T = 0.5$ because we want the mapping to predict one time step of the MPC. Line 7-9 calculates the activation of the hidden layers using equation 2.3, and line 10 calculates the output layer which does not have an activation function. Finally, line 11 creates a CasADi function of the mapping.

Runge-Kutta mapping

The following code creates the mapping using a Runge-Kutta integrator

Code listing 3.13: Creating a Runge-Kutta mapping as a CasADi function.

```
1  x = MX.sym('x', 2)
2  u = MX.sym('u')
3  f = Function('f', [x, u], [vertcat(x[1], (1-x[0]**2) * x[1] - x[0] + u)])
4
5  intg_opt = dict(tf=T, number_of_finite_elements=1)
6  dae = dict(x=x, p=u, ode=f(x,u))
7  intg = integrator('intg', 'rk', dae, intg_opt)
8
9  F_RK = Function('F_RK', [x,u], [intg(x0=x, p=u)["xf"]])
```

Line 1-3 sets up the ODE. A Runge-Kutta 4th order integrator with a single step is set up in line 5-7. Line 9 creates the mapping in the form of a CasADi function.

3.6 Metrics

For the PINC model, MSE is used to calculate the validation error:

$$MSE_{val} = \frac{1}{N_y}\sum_{i=1}^{N_y}\frac{1}{N}\sum_{k=1}^{N}(y_i[k]-\hat{y}_i[k])^2 \tag{3.4}$$

where $\mathbf{y}[k]$ is the k'th iteration of the PINC in self-loop and $\hat{\mathbf{y}}[k]$ is the corresponding value of the validation data indicated by the dots in figure 3.2. N is the number of data points in the validation set.

For calculating the performance on test data sets for both PINNs and PINC model, the Root Mean Square Error (RMSE) is used:

$$RMSE = \frac{1}{N_y}\sum_{i=1}^{N_y}\sqrt{\frac{1}{N}\sum_{k=1}^{N}(y_i[k]-\hat{y}_i[k])^2} \tag{3.5}$$

here, the location of the data points $\mathbf{y}[k]$ and $\hat{\mathbf{y}}[k]$ may be anywhere within the simulated time interval, not only at the discrete points with spacing 0.5s.

RMSE will also be used to measure the deviation between the sates and their reference trajectory in the MPC control application. The RMSE is modified as shown in equation 3.6, where $y_i[k]$ is state i (1 or 2) at time step k, $y_i^{ref}[k]$ is the respective reference, and N is the number of time steps in the simulation.

$$RMSE = \frac{1}{N_y}\sum_{i=1}^{N_y}\sqrt{\frac{1}{N}\sum_{k=1}^{N}\left(y_i[k]-y_i^{ref}[k]\right)^2} \tag{3.6}$$

Chapter 4

Results

This chapter will show the results of training the PINC model and how this compares to validation and test data in section 4.1. Then how it compares to the classic PINN1 and PINN2 models for different simulation lengths in section 4.2. And lastly, a comparison between using the PINC and Runge-Kutta mapping for MPC in section 4.3.

The three models structures that will be compared in this chapter are show in the following two tables. Table 4.1 shows the NN structure and number of training data points of the three models. Table 4.2 shows the number of iterations of L-BFGS used to train the networks, after the initial 500 epochs of Adam. For each simulation time T a new PINN1 and PINN2 model is trained, but the PINC model is only trained for $T = 0.5$ seconds, it is used in self-loop mode when doing longer simulations.

		PINN1	PINN2	PINC
Layers		4	10	4
Neurons/layer		20	200	20
N_t		1	1	1 000
N_f	$T = 0.5s$	500	500	100 000
	$T = 2s$	2 000	2 000	
	$T = 5s$	5 000	5 000	
	$T = 10s$	10 000	10 000	
	$T = 20s$	20 000	20 000	

Table 4.1: Structure and the number of training data points for the three different neural networks. N_t is the number of points used to train the initial condition, and N_f is the number of collocation points used to enforce the underlying physics.

Iterations of L-BFGS			
T	PINN1	PINN2	PINC
0.5s	200	200	26 000
2.0s	800	800	
5.0s	2 000	2 000	
10.0s	10 000	10 000	
20.0s	20 000	20 000	

Table 4.2: Number of iterations of L-BFGS used to train the different networks after the initial training of 500 epochs with Adam. The PINC network was trained until the L-BFGS algorithm converged at approximately 26 000 iterations.

4.1 PINC model

The PINC model was trained initially with 500 epochs of Adam, then until convergence with L-BFGS, a plot of the training and validation loss is shown in figure 4.1. The model obtaining the lowest error on the validation set, indicated by the red arrow in the figure, will be used for the remainder of the experiments.

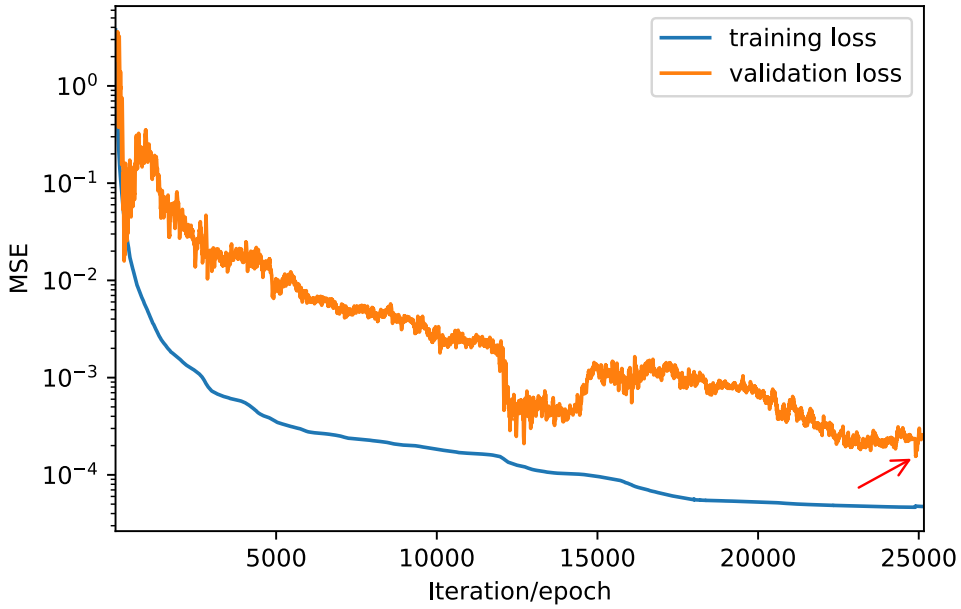


Figure 4.1: Training progression of the PINC model. Training error in blue, and error on the validation set in orange. The red arrow indicates the lowest validation loss obtained during training, the model checkpoint from this iteration is used for the remainder of the experiments. The validation loss is $1.63 \cdot 10^{-4}$.

Figure 4.2 shows a comparison between the Runge-Kutta simulation of the validation set in solid lines, and the PINC simulation using the validation input sequence

in dotted line. The dots indicate the state estimate of the PINC model at the end of every self-loop iteration (0.5 seconds).

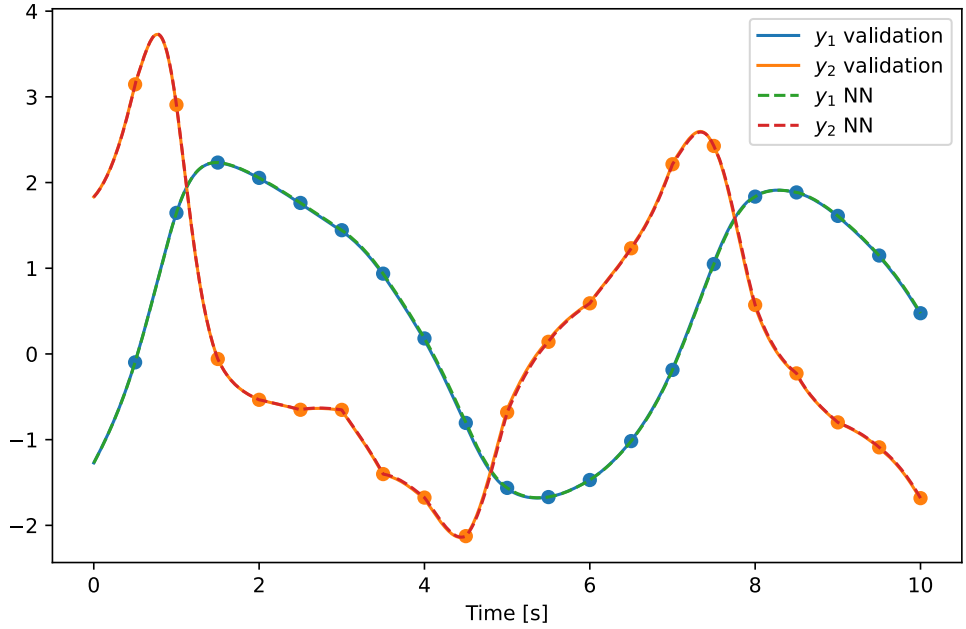


Figure 4.2: Comparing the PINC in self-loop mode (dotted lines) with the validation data (solid lines). The dots indicate the NN output at every iteration of the self-loop (every $T = 0.5$ seconds), when using the model with MPC only these points will be calculated. The MSE between the NN output and validation data is $1.63 \cdot 10^{-4}$.

To assess the performance of the PINC model, it was evaluated on 1 000 test data sets of the same nature as the validation set. All test sets have a sequence of 20 random inputs for a simulation of 10 seconds. The MSE for each of these sets are shown in figure 4.3. For comparison, the MSE on the validation set is shown in the red dotted line.

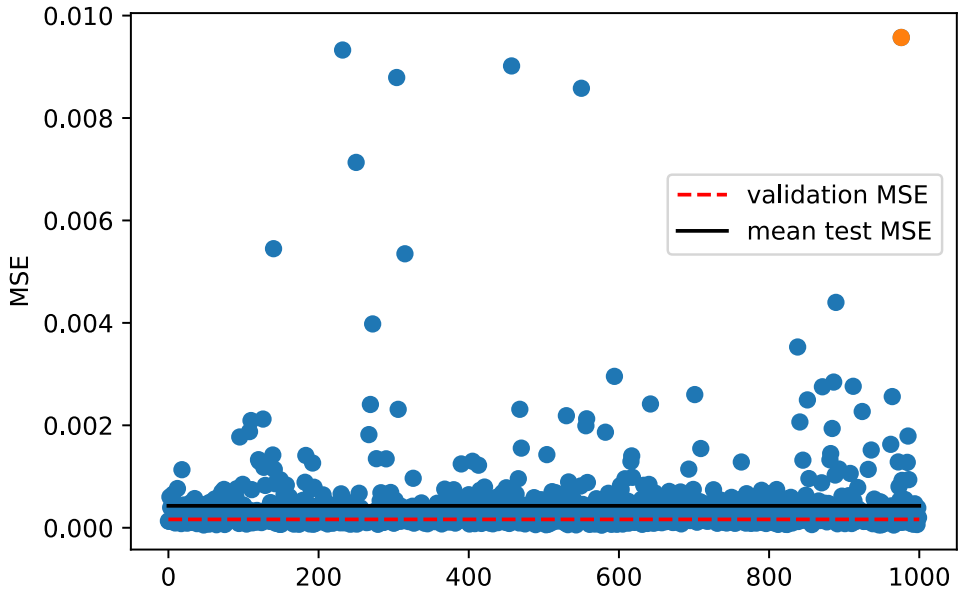


Figure 4.3: Showing the MSE of the NN compared with 1 000 test sets simulated for 10 seconds. The test sets are generated in the same way as the validation set (as illustrated in figure 3.2) with the difference of drawing a new input sequence for every set. The average MSE is $4.28 \cdot 10^{-4}$ (black line) and the standard deviation is $8.18 \cdot 10^{-4}$. The MSE of the validation set of $1.63 \cdot 10^{-4}$ is shown in the dotted red line. The orange dot in the upper right corner indicates the highest MSE obtained from these 1 000 test sets, it is $9.57 \cdot 10^{-3}$.

The average MSE on the test sets is $4.28 \cdot 10^{-4}$, higher than the validation MSE of $1.63 \cdot 10^{-4}$. The training set obtaining the highest MSE is indicated by the orange dot in figure 4.3, the comparison between this test set and the NN is shown in figure 4.4.

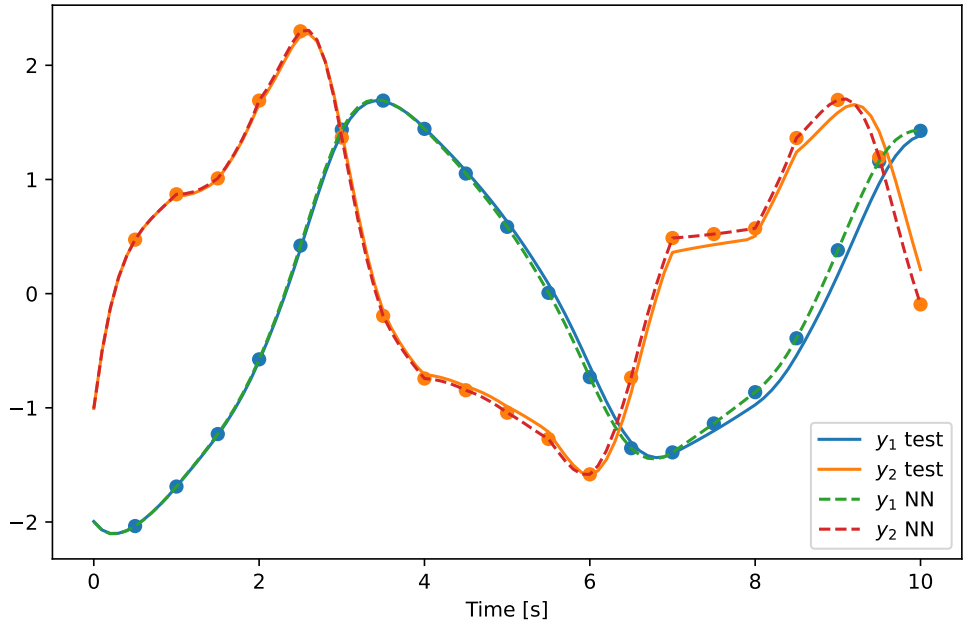


Figure 4.4: Comparing the PINC in self-loop mode (dotted lines) with the test set (solid lines) obtaining the highest MSE, as indicated by the red dot in figure 4.3. The MSE between the NN output and test data is $9.57 \cdot 10^{-3}$.The deviation between the NN and test set can clearly be seen for $t \geq 4$.

4.2 Comparing PINC with PINN1 and PINN2

Now, the PINC model will be compared against the two traditional PINN models, PINN1 and PINN2. Figure 4.5 shows the comparison for simulation lengths 0.5, 2 and 5 seconds, and figure 4.6 for 10 and 20 seconds. For each of these simulations lengths, a new PINN1 and PINN2 model was trained, but the same PINC is used in self-loop mode for all simulations. For the simulation lengths up to 10 seconds booth PINNs gives results very similar to the Runge-Kutta simulation, but for $T = 20s$ the outputs start to deviate at around 5 seconds for booth PINN1 and PINN2. The PINC model in self-loop mode reproduces the behavior in all five simulation lengths. Only the discrete time steps of $T = 0.5$ seconds are shown for the PINC model, if more points were plotted, they would superimpose the Runge-Kutta simulation, as the PINNs do in the first four simulations.

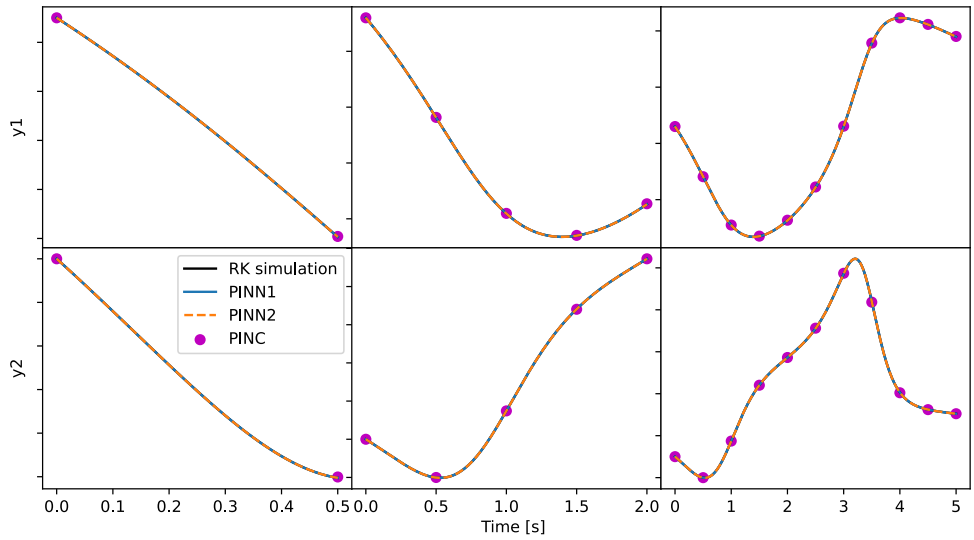


Figure 4.5: Comparison of PINC, PINN1 and PINN2 against the Runge-Kutta simulation. The first row shows y_1 and the second row y_2 for all three models and the Runge-Kutta simulation. For the PINC model, only the outputs at the discrete intervals of 0.5 seconds are shown. For PINN1 and PINN2 a new model is trained for each of the three estimation time horizons (0.5, 2.0 and 5.0 seconds), whereas for the PINC the same model is used in self-loop mode for all three simulation lengths.

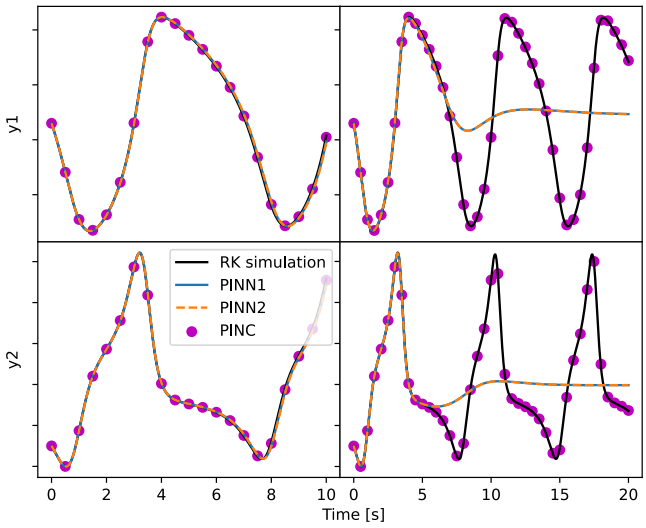


Figure 4.6: Continuation of figure 4.5 with $T = 10s$ and $T = 20s$. Comparing of PINC, PINN1 and PINN2 against the Runge-Kutta simulation. The first row shows y_1 and the second row y_2 for all three models and the Runge-Kutta simulation.

The performance measured in RMSE of PINN1, PINN2 and PINC compared with

these five test sets are shown in figure 4.7. Here the RMSE is calculated at a finite set of points evenly distributed across the time horizon, the same points are used for all three models. The PINNs outperform the PINC model for $T \leq 5s$, but the PINC degrades less with increasing simulation time, and outperform both PINN models at 10 and 20 seconds. PINN1 performs better than PINN2 in all five cases, the added complexity of the PINN2 model clearly does not improve predictions in this case.

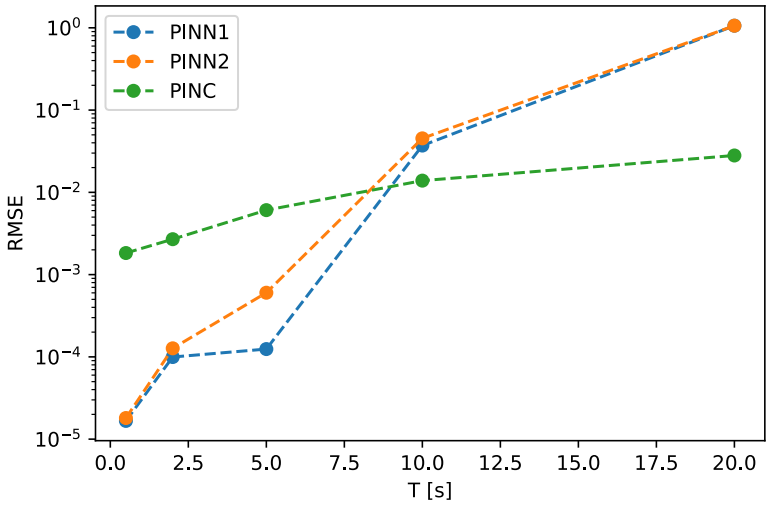


Figure 4.7: RMSE between the models and the test sets (shown in figure 4.5 and 4.6) for different simulation time horizons T.

4.3 MPC using PINC

Figure 4.8 show a control application of the Van der Pol system using MPC. Two controllers are compared, one using the PINC mapping and the other using the Runge-Kutta mapping. The simulation last 60 seconds and has three step changes in reference, in addition to the initial condition deviating from the initial reference. The two controllers show similar results.

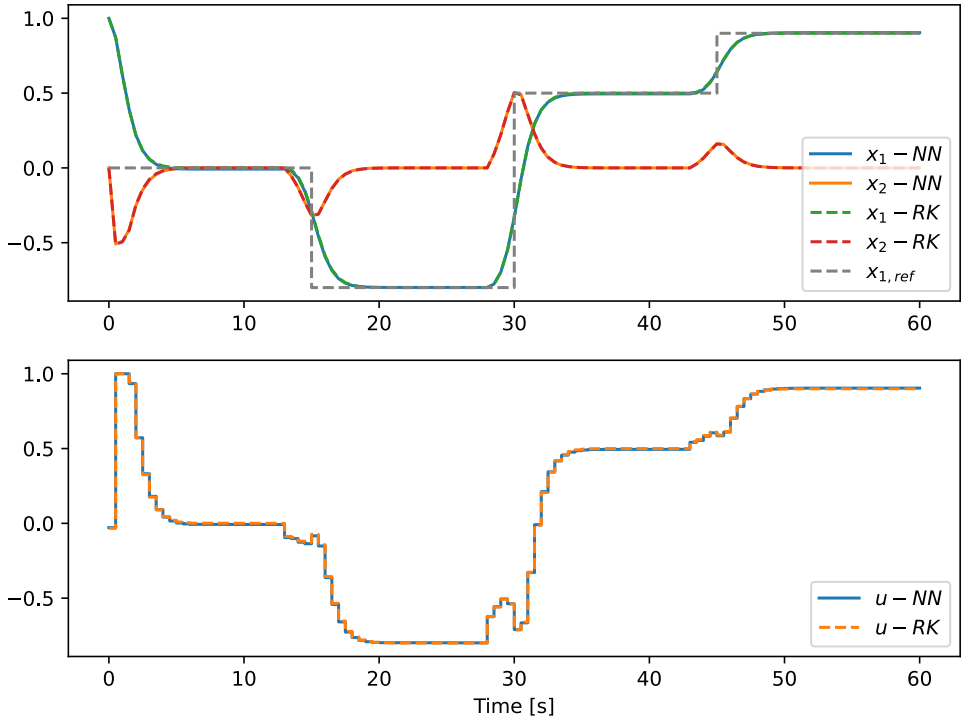


Figure 4.8: Comparison of controlling the system with MPC using the PINC model (NN) in solid lines and the Runge-Kutta integrator in dotted lines. The reference for y_1 is shown in a gray dotted line, and the reference for y_2 is constant at 0. Their responses are so similar they are hard to tell apart in this plot.

The performance of the controllers is shown in table 4.3, a five step Runge-Kutta mapping is included to demonstrate that the sing step Runge-Kutta mapping is good enough in terms of RMSE. There is little difference in the RMSE between the system states and reference trajectory for the three controllers. The table also shows the average computation time of the simulation shown in figure 4.8, this was computed by doing 100 simulations for each controller in the same Google Colaboratory session.

	RMSE	Time (s)
PINC	0.1592	1.158 ± 0.056
RK 1 step	0.1594	0.877 ± 0.045
RK 5 steps	0.1594	1.473 ± 0.088

Table 4.3: Results from the simulations shown in figure 4.8. A five step Runge-Kutta mapping is included for comparison. RMSE indicates how much the system states deviates for their reference trajectory. The time column shows the average computation time for the 60 second simulation.

Chapter 5

Discussion

The main objective of this project has been to gain knowledge that I will need for my master thesis, mainly about PINNs and needed prerequisite knowledge of machine learning and NNs in general. I have put much effort into explaining PINNs in chapter 2, which has given me a greater understanding of the subject. The remainder of this chapter will discuss the results presented in chapter 4, commenting on the implementations and possible improvements.

5.1 PINC training and performance on test sets

The model structure of the PINC was taken from an article [1] which has implemented the same system, when applying this technique to a new system different model structures should be considered. It is important to consider that the computation time will increase with increasing network size, so performance in terms of prediction accuracy and computation time may have to be weighted against each other.

The training of the PINC model converges after approximately 26 000 iterations/epochs, but the performance is quite bad on some of the 1 000 test sets (as seen in figure 4.3). This is something I would like to investigate further. I suspect that the model performs badly on some areas of the input space, if these areas can be identified during training the collocation points can be altered to cover these areas better. This can possibly also be used in general to identify the performance of the model across the input space during training, more weight can then be put on the areas with poor performance, hopefully decreasing time needed for training. I believe this will be even more important for larger systems as the dimension of the input space grows. Then it may be problematic or inefficient to use enough collocation points simultaneously to cover the entire input space well due to memory and computation limitations.

Another idea for improving training of the NN is to include some simulation or

measurement data in the training process. For instance, for a system where numerical estimation is too slow for MPC applications, we could still obtain some simulations that may help in training the NN, hopefully resulting in a NN that is fast enough to be used with MPC.

5.2 Comparing PINC with PINN1 and PINN2

The comparisons demonstrate that the traditional PINNs obtain good performance on short simulations, but degrades quickly when increasing the simulation horizon. At 20 seconds, the PINNs do not converge to the correct solution. Even the added complexity of the PINN2 network (10 layers of 200 neurons) gives no improvement compared with the simpler PINN1 (4 layers of 20 neuron). In fact, PINN1 outperforms PINN2 in all five experiments, indicating that we cannot simply increase network size to obtain better results. The PINC however, degrades less with increasing simulation time, and is able to reproduce the system behavior in the 20 seconds simulation.

All PINN1 models uses less time for training than the PINC network. They have the same network structure, fewer inputs, less training data points and use fewer training iterations. Their accuracy is also better up to 5 seconds. The PINC model need more training because it estimates a much more complex system, it has four inputs compared to a single input of the PINNs. The major advantage of the PINC here is that it is trained for a range of initial conditions and control input, unlike the PINNs where a new model must be trained when changing the initial condition or control input. This renders the classical PINNs useless in control applications where the current state and control input changes.

5.3 MPC

The MPC application case first and foremost show that it is viable to implement the system dynamics constraint 2.1b in MPC by a NN. The difference in RMSE performance comparing with the Runge-Kutta mapping is minimal. In terms of computation time, the PINC is not as fast as the single step Runge-Kutta. Looking at these two factors, the Runge-Kutta is the better choice for this system. In further work, it will be interesting to investigate how the computational time of the PINC evolves with increasing system complexity compared with numerical methods. This is especially interesting for systems that are time-consuming to solve numerically, if a NN could make fast predictions on the same system this could open new opportunities for use of MPC and other optimization techniques.

Chapter 6

Conclusion

This report has show how neural networks can be trained to estimate dynamical systems, using the differential equation and minimal amounts of simulation data. The Van der Pol example has shown that it is viable to use these kinds of neural networks as the system dynamics constraint in model predictive control. For this example, the neural network mapping was not faster than the Runge-Kutta mapping used for comparison, but this may change for more complex systems. For future work, this method will be applied to other, more complex systems.

Bibliography

[1] E. Antonelo, E. Camponogara, L. Seman, E. Souza, J. Jordanou and J. Hübner, 'Physics-informed neural nets-based control,' Apr. 2021.

[2] M. Raissi, P. Perdikaris and G. Karniadakis, 'Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations,' Nov. 2017.

[3] M. Raissi, P. Perdikaris and G. Karniadakis, 'Physics informed deep learning (part ii): Data-driven discovery of nonlinear partial differential equations,' Nov. 2017.

[4] J. Nicodemus, J. Kneifl, J. Fehr and B. Unger, 'Physics-informed neural networks-based model predictive control for multi-link manipulators,' Sep. 2021.

[5] M. Huba, S. Skogestad, M. Fikar, M. Hovd, T. Johansen and B. Rohal-Ilkiv, *Selected Topics on Constrained and Nonlinear Control*. STU/NTNU, 2011.

[6] I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`.

[7] *Neural network svg generator*. [Online]. Available: `http://alexlenail.me/NN-SVG/index.html`.

[8] D. Kingma and J. Ba, 'Adam: A method for stochastic optimization,' *International Conference on Learning Representations*, Dec. 2014.

[9] D. C. Liu and J. Nocedal, 'On the limited memory bfgs method for large scale optimization,' *MATHEMATICAL PROGRAMMING*, vol. 45, pp. 503–528, 1989.

[10] P. Chuang, *Optimize tensorflow & keras models with l-bfgs from tensorflow probability*, Accessed: 2021-11-19, Nov. 2019. [Online]. Available: `https://pychao.com/2019/11/02/optimize-tensorflow-keras-models-with-l-bfgs-from-tensorflow-probability/`.

[11] Z. Mao, A. Jagtap and G. Karniadakis, 'Physics-informed neural networks for high-speed flows,' *Computer Methods in Applied Mechanics and Engineering*, Mar. 2020. DOI: `10.1016/j.cma.2019.112789`.

[12] D. Hurtado, F. Costabal, Y. Yang, P. Perdikaris and E. Kuhl, 'Physics-informed neural networks for cardiac activation mapping,' *Frontiers of Physics*, vol. 8, Feb. 2020. DOI: `10.3389/fphy.2020.00042`.

[13] *Casadi docs - optimal control with casadi*, Accessed: 2021-12-7. [Online]. Available: `https://web.casadi.org/docs/#document-ocp`.