

Recurrent Neural Networks (RNN) for Modeling of Nonlinear Systems

Research internship report
of Sebastian Hirt, December 14, 2022

1. Supervisor: M. Schumann, M.Sc.
2. Supervisor: J. Kißkalt, M.Sc.
Auditor: Prof. Dr.-Ing. K. Graichen

Differential equations are essential for describing the behavior of nonlinear dynamic systems over time. But ordinary methods cannot calculate the solutions if the analytical form is unknown. This work shows the ability of long-short-term-memory networks (LSTMs) to predict nonlinear dynamic systems over time. Time series datasets are generated from differential equations of a force pushing an object with drag and a pendulum with an external force and friction. The models are then trained with multiple hyperparameters and training difficulties are highlighted. The best models are subsequently evaluated and shown to perform well. The methods show potential to be used with experimentally obtained data from differential equations with unknown analytical forms.

1 Introduction

Ordinary differential equations (ODEs) describe how the states of systems develop over time. Given the initial conditions, algorithms can calculate the future states of the systems numerically [1][2]. However, there might be systems, for which there is no analytical description, but real-world data of the inputs and outputs. These data can then be used to train a neural network to replace the algorithms and predict future states that are not in the training data.

Due to this data being time series, this work uses a type of recurrent neural network, called LSTM, to predict the future states from one initial condition given at the first time step. LSTMs have been shown to perform well on time series prediction tasks, see subsection 1.1. Additionally, it is equipped with nonlinearities, which should enable it to perform well on the task of this work.

This work tries to show that LSTMs are capable of predicting simple ODEs with just the initial condition and the external force given as the input. This serves as a foundation for future work to try similar methods with more complex ODEs or real-world data.

1.1 Similar work

LSTMs have been shown to perform well in translation [3], video captioning [4] and generating text sequences [5][6]. This demonstrates that LSTMs are capable of learning complex short

and long-term dependencies, which can be important in predicting the future states of dynamic systems.

On the regression task, LSTM architectures are effective at predicting stock price values [7]. The authors show that LSTMs outperform traditional machine learning methods for stock price forecasting. Stock prices have similar properties to unknown dynamic systems, as most inputs (news sentiment, financial, general market indicators) and outputs (stock price) can be measured. But there is no known analytical formulation to calculate future stock prices.

Other research uses LSTMs and compares their ability to predict the states of the Lorenz system with traditional RNNs [8]. The paper indicates that the LSTM models outperform basic RNNs thanks to their ability to learn long-term dependencies.

In the next section, the methods that are used in this report will be introduced. After that, the creation of the dataset is explained. Thereafter, those datasets are used to search for the best hyperparameter for the model. Finally, the best model is evaluated on the test set.

2 Methods

This section explains the relevant methods that are used in this work. First, the LSTM-cell and the fully connected layer are presented, which are then connected to form the used architectures. Afterward, the training and metrics are explained.

2.1 Model

The architecture used in this work is a multilayer LSTM with one fully connected layer at the output. The depth (amount of layers) and nodes in each layer will be determined later through a hyperparameter search.

2.1.1 LSTM

The following section explains the LSTM with a forget gate as proposed in [10][11]. An LSTM cell, see Figure 1, is an advanced node in a neural network, which remembers two internal states c_{t-1} and h_{t-1} from the last timestep of a sequence. Those states are then manipulated when the input of the current timestep flows through the network. The equations

$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \quad (1)$$

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \quad (2)$$

$$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \quad (3)$$

$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \quad (4)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \quad (5)$$

$$h_t = o_t \odot \tanh(c_t) \quad (6)$$

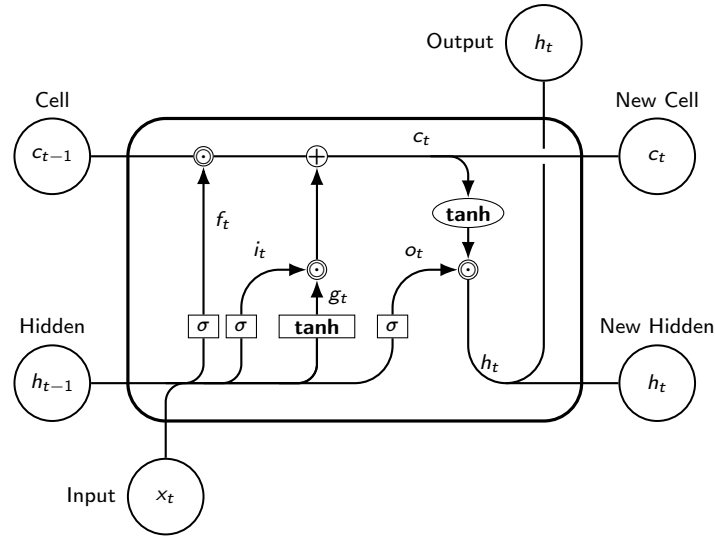


Figure 1: Diagram of LSTM cell with the different gates highlighted. \odot is an elementwise matrix multiplication. Adapted from [9].

explain different steps of the computations inside a cell, also visible in Figure 1. The \odot operator is an elementwise matrix multiplication.

The cell state c can be understood as the long-term memory and h as the short-term memory. The cell state can carry information from several timesteps in the past, while the hidden state mostly carries information from the previous timestep. The input x_t and the previous hidden state h_{t-1} are combined with individual weights and biases for the first four equations. Thus, each LSTM cell has eight weights and biases that can be trained. Those combinations are used to create new states.

The sigmoid function $\sigma(x)$ is used to project one of those combinations onto values between 0 and 1. Therefore, after doing an element-wise matrix multiplication with the previous cell state c_{t-1} , the forget gate f_t determines how much of c_{t-1} should be forgotten, see Equation (1).

The same applies to the input gate i_t , see Equation (2). However, it does not help to forget the cell state but determines how much of the cell gate g_t gets added to the cell state after forgetting. The cell gate combines input and hidden state and then applies a tanh activation function, see Equation (3).

In addition, the new cell state c_t is created by first forgetting a certain amount of old information and then adding a certain amount of new information, see Equation (5). Finally, the output gate o_t again follows the same principle of the forget gate, see Equation (4). But now, it controls how much of that new cell state, after again applying a tanh activation function, will be the new hidden and output state h_t , see Equation (6). The resulting hidden state is the model's output at that time step.

2.1.2 Fully Connected Layer

This section briefly describes the fully connected layer [12], also called the linear layer, and why it is used as the output layer in this work.

A fully connected layer applies a linear transformation to the input. The output of the LSTM is the result of a multiplication of $\sigma(x)$ and $\tanh x$. This leads to an output between -1 and 1. The target values are normalized to a range between -1 and 1 as well. However, some target values of the test set can lie outside of this range because the minimum and maximum values for the normalization are gathered from the training set, see subsection 3.3. Thus, the fully connected layer is needed to access the full range of possible values.

2.1.3 Full Model

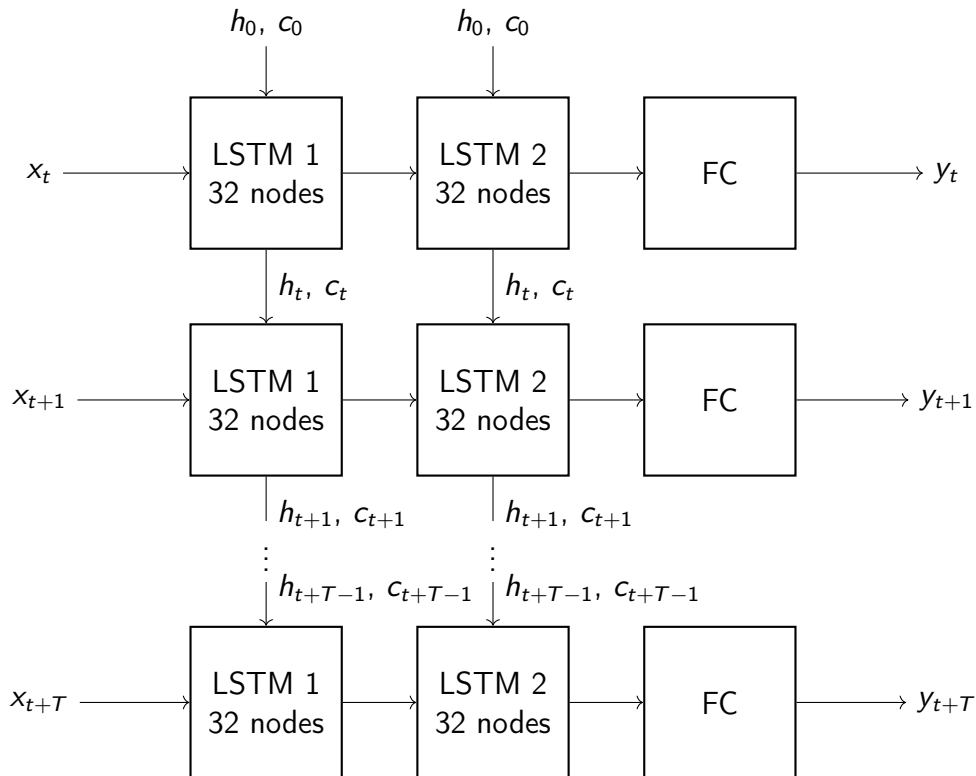


Figure 2: Diagram of the architecture of a model with 2 LSTM layers, 32 nodes (or hidden states) and T timesteps.

When taking multiple LSTM cells or nodes, see Figure 1, next to each other, the architecture seen in Figure 2 can be constructed. This is one example of the full model with 32 nodes next to each other and two layers after each other. The 32 nodes all have different weights and biases and manage their cell c_t and hidden states h_t individually. At the next timestep, $t+1$ in the figure, the nodes have the same weights but get their new cell and hidden states from the last timestep. The cell c_0 and hidden states h_0 are initialized with zeros.

This means that after training the model, variable length inputs can be used to generate new outputs. So at a single timestep with an initial condition and a force input the model can be used to generate the first output y . Afterward, the cell and hidden states of all the nodes in the model can be remembered and used at any time to give the model a second input and generate the appropriate output. The more interesting conclusion is that future states can be indefinitely predicted regardless of how long the training sequences are.

2.2 Training

This section describes the training process of neural networks, which is depicted in an article by Günther and Fritsch [13].

2.2.1 Forward Pass

Before starting the training process the weights of the model get initialized from the uniform distribution $\mathcal{U} = (-\sqrt{k}, \sqrt{k})$ with $k = \frac{1}{h_{size}}$, with h_{size} as the number of nodes in a hidden layer. As the first step in training, the training samples X get passed through the network $f(X, W)$ to calculate the outputs with the current weights W . Those outputs \hat{y}_t^n are compared to the ground truth y_t^n with the loss function

$$L^n = \sum_{t=0}^T (\hat{y}_t^n - y_t^n)^2 \quad (7)$$

with

$$\hat{y} = f(X, w) \quad (8)$$

to calculate the loss L^n for batch n . The next step is to minimize the loss function

$$\min_W L^n(X, W, y) \quad (9)$$

with regards to W . This is achieved with the help of optimization algorithms.

2.2.2 Optimization

The loss is used to calculate the gradient of the loss function with respect to each weight in the network. Furthermore, backpropagation through time [14] uses the chain rule to efficiently calculate those gradients through all nodes, layers and through time.

The following paragraphs are based on the lecture about convolutional neural networks by Fei-Fei Li et al. [15]. Depending on how much each weight influences the loss of the model, the weight is updated to hopefully decrease the loss in the next forward pass.

The simplest rule to update the weights is to take the current weight and subtract the negative gradient multiplied by the learning rate. So the learning rate determines how strongly all the weights are adjusted.

This whole process of the forward pass, backpropagation and weight update is done for all batches of the training set. After iterating over the whole training set, the model is evaluated on the validation set to figure out whether the model is improving for new data. This is called an epoch. Then the training continues for multiple epochs until the validation loss is at a certain value. The ultimate goal is to reduce the loss as much as possible, which means finding a global minimum. However, this is nearly impossible. Most of the time, the training ends at a local minimum. A high learning rate can lead to fast training but the model might skip over potentially good local minima. This can be an advantage as well since jumping over early local minima is possible and therefore it can be avoided getting stuck in them. A

low learning rate takes longer to converge to a better model and can get stuck in early local minima with high loss. It can however reach into certain minima a higher learning rate cannot and thus achieve a lower loss in the end.

In this work, the Adam optimizer [16] with a constant learning rate is used. Instead of just using the gradient to update the weights, it uses estimates of the first moment (the mean) and the raw second moment (the uncentered variance) of the gradient. The first and second moments are estimated by using an exponentially decaying moving average of the gradient and the squared gradients respectively. These get updated at every step, before using them to update the weights. This results in an adaptive learning rate that can vary between 0 and the given constant learning rate. This is done for every weight in the model individually. A learning rate decay is not guaranteed to help with training because the learning rate is already adaptive. It can still help by setting an upper limit for the learning rate in later epochs. It performs better than most alternatives and requires little tuning.

2.3 Metrics

This work uses the Mean squared error (MSE) [17] as the metric to evaluate the results. Additionally, it is used as a basis for the loss function of the optimization process. It can be calculated by the equation

$$\sum_{t=0}^T (y_t - \hat{y}_t)^2 \quad (10)$$

with the predicted value \hat{y} and the ground truth y over T timesteps t . The MSE is used to make the loss function more sensitive to large errors. Further, those large errors are more important to be avoided than small errors if the predictions were to be used in real applications.

In addition to the methods, training data is needed to train the model, which is created in the next section.

3 Dataset creation

This section describes how the datasets for training the models are created. First, two differential equations are derived, which are used to create time series. These time series are scaled and transformed to have the correct dimensions for training.

3.1 Problem description

To generate data with an ODE solver, a system of first-order differential equations is necessary. Two of these are created in the following subsections.

3.1.1 Drag

The first problem to analyze is the pushing of an object with a Force F_p [18] and air resistance (or drag) [19].

The drag force on the object can be formulated as

$$F_d = -\frac{1}{2}\rho v^2 C_D A \quad (11)$$

with the density of the surrounding fluid ρ , the speed of the object relative to the fluid v , the drag coefficient C_D and the cross-sectional area (seen from the direction of motion) A . The equation can be simplified by summarizing the constants

$$b = \frac{1}{2}\rho C_D A. \quad (12)$$

To get the actual force acting on the object one can add the drag force to the pushing force

$$F = F_p(t) + F_d. \quad (13)$$

The corresponding differential equation is then represented by

$$m \frac{d^2 x}{dt^2} = F_p - b \left(\frac{dx}{dt} \right)^2 \quad (14)$$

with $\frac{d^2 x}{dt^2}$ as a and $\frac{dx}{dt}$ as v . This is a second-order differential equation. To transform it into a system of first-order differential equations $\frac{dx}{dt}$ can be replaced by v to get

$$\frac{dx}{dt} = v \quad (15)$$

$$\frac{dv}{dt} = F_p(t) - \frac{b}{m} v^2. \quad (16)$$

3.1.2 Pendulum

The second problem to analyze is the pushing of a pendulum [20] with a driving force and its interaction with friction. The force of gravity on the object is described by

$$F_g = -mg \sin(\theta) \quad (17)$$

with the mass of the object m , the acceleration due to gravity g and the angle from the vertical equilibrium position θ .

The friction force on the object can be formulated as

$$F_f = -c l \frac{d\theta}{dt} \quad (18)$$

with the length of the pendulum arm l and the friction constant c . The pushing force is modeled as

$$F_p = F(t) \cos(ft) \quad (19)$$

with a constant frequency f , time t and the variable force $F(t)$.

To get the actual force acting on the object, one can add the drag force and the force of gravity to the pushing force

$$F = F_p + F_f + F_g. \quad (20)$$

The corresponding differential equation is then represented by

$$mI \frac{d^2\theta}{dt^2} = F(t) \cos(ft) - mg \sin(\theta) - cI \frac{d\theta}{dt} \quad (21)$$

with $\frac{d^2\theta}{dt^2}$ as $\ddot{\theta}$ and $\frac{d\theta}{dt}$ as $\dot{\theta}$. This is a second-order differential equation. To transform it into a system of first-order differential equations $\frac{d\theta}{dt}$ can be replaced by $\dot{\theta}$ to get

$$\frac{d\theta}{dt} = \dot{\theta} \quad (22)$$

$$\frac{d\dot{\theta}}{dt} = F(t) \cos(ft) - q\dot{\theta} - \frac{g}{l} \sin(\theta) \quad (23)$$

with $q = \frac{c}{m}$.

In the next section, those equations are used to generate the datasets.

3.2 Datasets Overview

The equations from the last section are now used to generate time series. First, there are multiple different force input series generated. Those, with the equations and initial conditions, are then given to the `odeint()` python function [21][22] to generate the states for a specified amount of time steps. The pendulum and the drag dataset are 1000 and 3000 timesteps long, respectively. This is done multiple times with different force inputs to generate different series. In the following, the variations in the different datasets are explained. The details of the generation can be found in the implementation².

3.2.1 Drag-Simple

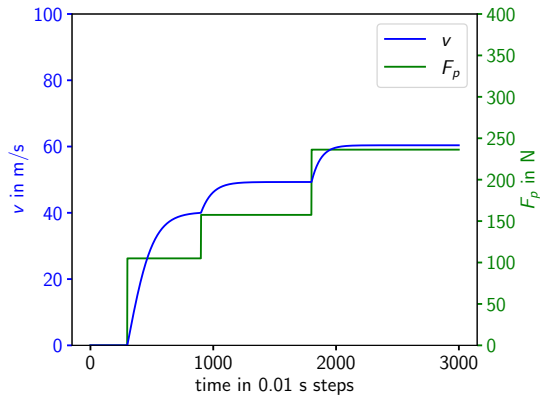
This dataset was created by stepping up the input force to different magnitudes at fixed points in time, see Figure 3. In total this dataset contains 20 different series.

3.2.2 Drag-Complex

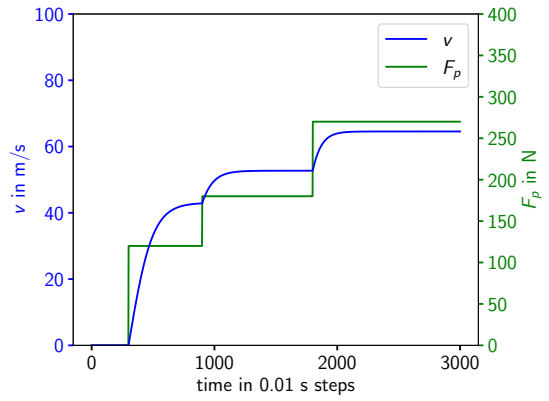
In this more complex dataset in Figure 4, the height of the steps can decrease as well and the positions are randomized. The same is done with a slope instead of a step. In total this dataset contains 40 different series.

¹The force F_0 should be divided by mI and should have the units $\frac{N}{kg \cdot m} = \frac{kg \cdot m}{kg \cdot m \cdot s^2} = s^{-2}$. However, in the rest of this work, the unit N is used. For the issues discussed in this report it is not important, because only proportions might be changed and the values for m and I are 1, so nothing changes. For future applications, however, it will be important, so it should be noted.

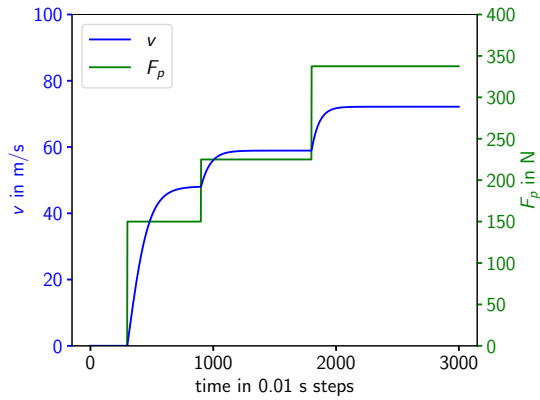
²Code available at <https://github.com/Mokronos/nonlinearLSTM>



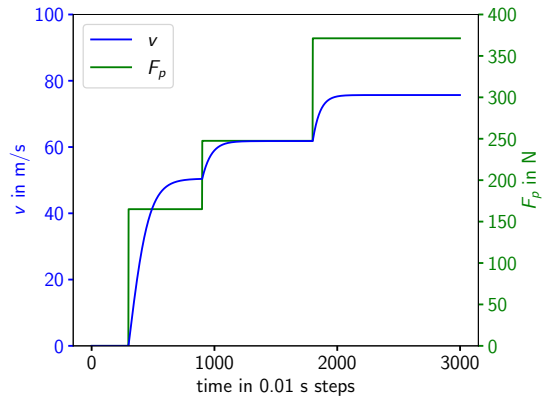
(a) Sample of the Drag-Simple dataset with steps of different height.



(b) Sample of the Drag-Simple dataset with steps of different height.

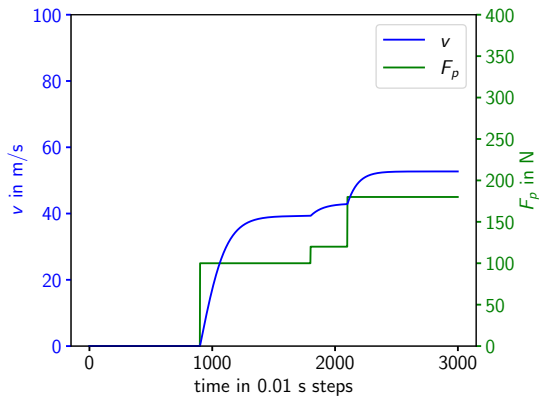


(c) Sample of the Drag-Simple dataset with steps of different height.

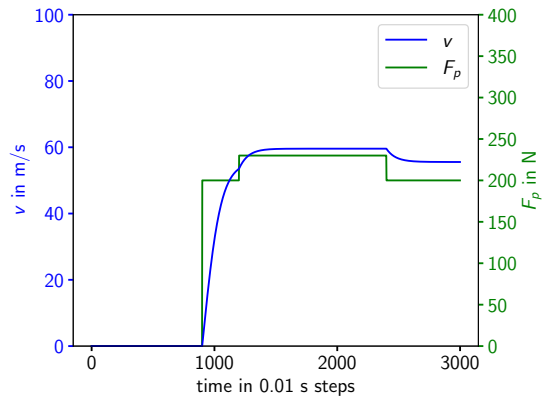


(d) Sample of the Drag-Simple dataset with steps of different height.

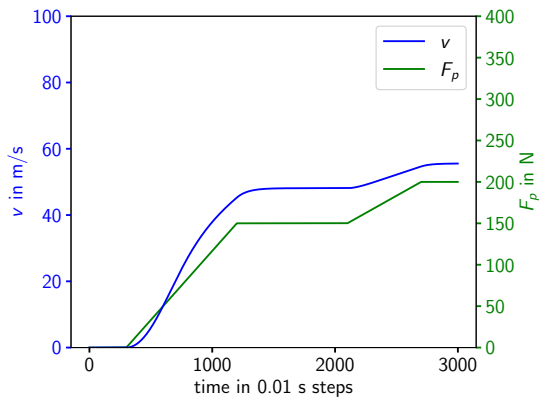
Figure 3: Samples of the Drag-Simple dataset with steps of different height of the force F_p .



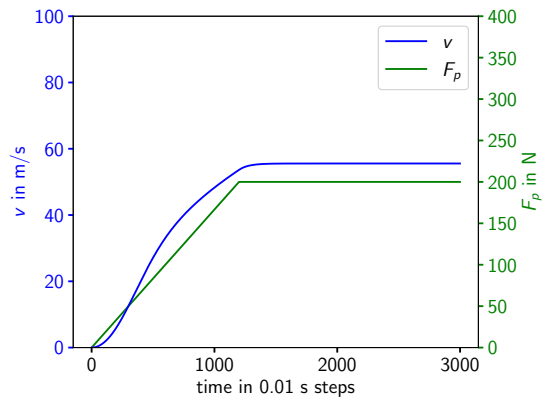
(a) Sample of the Drag-Complex dataset with just increasing steps with variable height.



(b) Sample of the Drag-Complex dataset with increasing and decreasing steps.

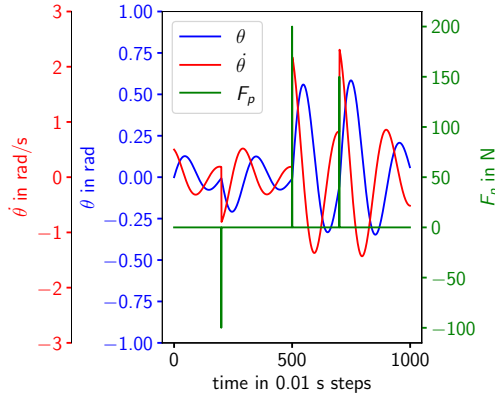


(c) Sample of the Drag-Complex dataset with two medium height slopes.

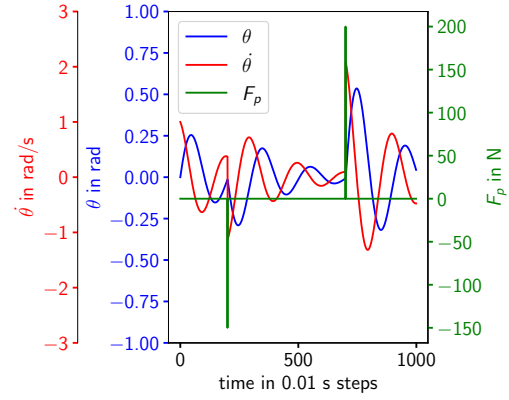


(d) Sample of the Drag-Complex dataset with a long slope.

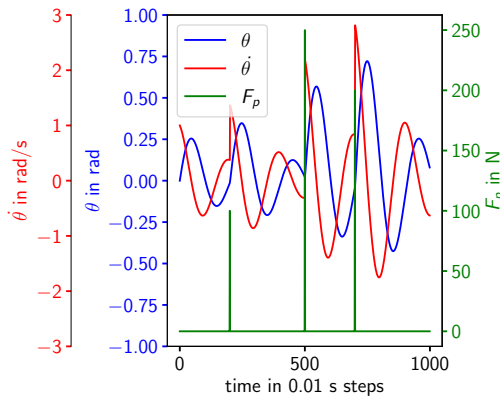
Figure 4: Samples of the Drag-Complex dataset with steps and slopes of different height of the force F_p .



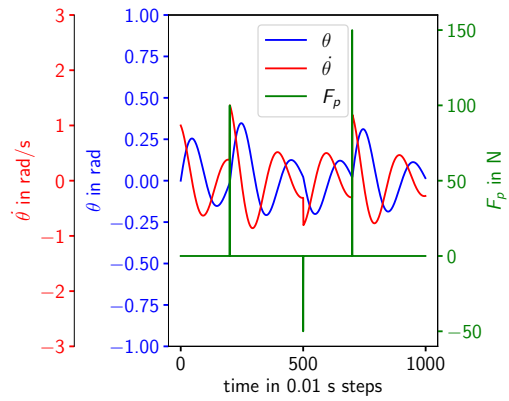
(a) Sample of the Pendulum-Simple dataset with negative and positive spikes.



(b) Sample of the Pendulum-Simple dataset only two spikes (third one is zero).



(c) Sample of the Pendulum-Simple dataset with only positive spikes.



(d) Sample of the Pendulum-Simple dataset with negative and positive spikes.

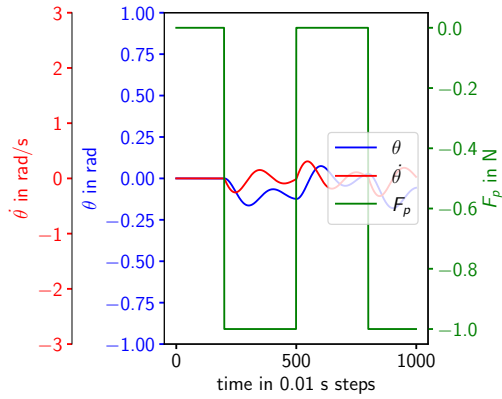
Figure 5: Samples of the Pendulum-Simple dataset with spikes of different height of the force F_p .

3.2.3 Pendulum-Simple

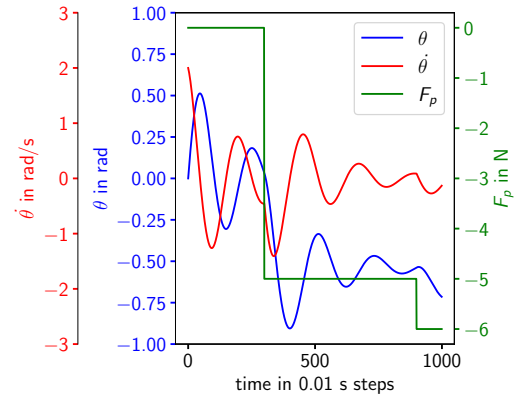
This dataset was created by combining different starting conditions for θ with spikes at fixed positions with randomized heights, see Figure 5. In total this dataset contains 45 different series.

3.2.4 Pendulum-Complex

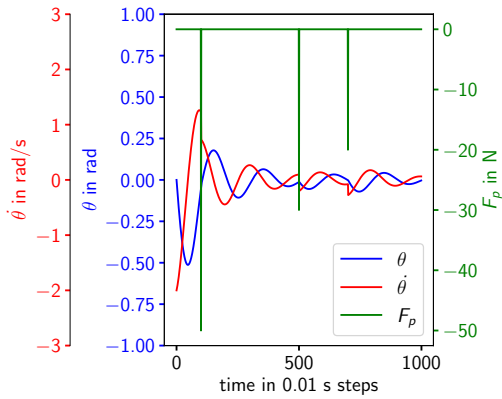
The more complex pendulum dataset additionally varies the position of the spikes and adds steps with random positions and heights, see Figure 6. In total this dataset contains 200 different series.



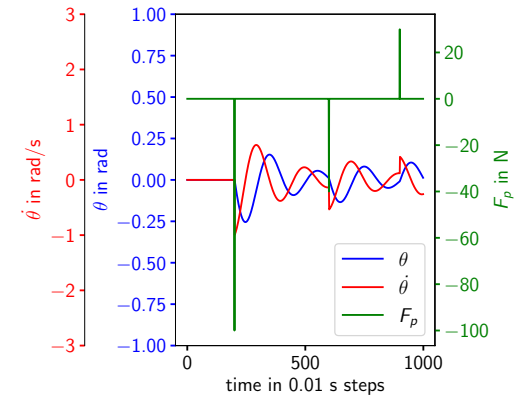
(a) Sample of the Pendulum-Simple dataset with negative and positive steps



(b) Sample of the Pendulum-complex dataset with negative and positive steps.



(c) Sample of the Pendulum-complex dataset with only negative spikes.



(d) Sample of the Pendulum-complex dataset with negative and positive spikes.

Figure 6: Samples of the Pendulum-complex dataset spikes and steps of varying position and magnitude.

3.3 Preparation for Training

The following paragraphs are based on a chapter of the lecture by Fei-Fei Li et al. [23]. These datasets are randomly split into training, validation (or development) and test set³. The training set is used to train the model. The validation set is used to evaluate different models on data the model has not seen before. This is mainly done to compare different hyperparameters and to find the model that generalizes best on new data. Finally, the test data is used to evaluate the best model after the hyperparameter search on new data.

The test set is only used after all the training and hyperparameter selection is done, to minimize the possibility of influencing the novelty of the test set. In other words, the test set is used to simulate new data as well as possible. The dataset is then normalized to a range between -1 and 1. This is achieved by first fitting a MinMaxScaler [24] on the training set. This scaler is then applied to all 3 sets. It is crucial to only use the information from the training set to scale the data. Otherwise, information about the validation and test set could leak into the training set. Furthermore, at inference time, the input samples might be given to the model one by one, without the possibility of scaling them with their statistics. In other words, any information about the validation and test datasets is assumed to only be available for evaluating the model to best imitate real-world data.

The following steps illustrate the generation of the training samples and the corresponding ground truth, but they are the same for the three sets. With a series of length L , the first state of the system is used as input to the system, this represents the initial condition of the system. After the first timestep, those inputs are set to zero. The other $L - 1$ states of the series are used as ground truth. The force inputs of the system are used as additional inputs next to the initial condition. So if a differential equation has 2 state variables and 1 force input variable, the model will have 3 inputs and 2 outputs. Hence a pair of input X and target matrices y of one of the pendulum datasets can be expressed as

$$X = \begin{bmatrix} \theta_t & \dot{\theta}_t & F_t \\ 0 & 0 & F_{t+1} \\ 0 & 0 & F_{t+2} \\ \vdots & \vdots & \vdots \\ 0 & 0 & F_{L-1} \end{bmatrix} \quad y = \begin{bmatrix} \theta_{t+1} & \dot{\theta}_{t+1} \\ \theta_{t+2} & \dot{\theta}_{t+2} \\ \theta_{t+3} & \dot{\theta}_{t+3} \\ \vdots & \vdots \\ \theta_L & \dot{\theta}_L \end{bmatrix}. \quad (24)$$

Those are then split into batches, which determine how many samples are looked at by the optimizer before updating the weights once, and used for training the different models. In the following section, multiple models will be trained and compared.

4 Hyperparameter Search

In this section, the training process is expanded by searching for hyperparameters that improve the model performance.

³The split for train/val/test sets is 0.6/0.2/0.2. Which variations of each dataset end up in which set is randomly determined. In the future, it might be helpful to handcraft the test set to evaluate the model on specific samples.

4.1 Validation Set

Without supervision, models can easily overfit on the training set and underfit on the validation set (and any other new data). To combat this, the validation loss is compared at every epoch to the lowest validation loss until now. If the validation loss is lower than the lowest loss beforehand, the model state is saved. This is easier than saving all model states, which can require a considerable amount of storage ⁴.

4.2 Manual Search

When searching manually through hyperparameters it is important to do so in the correct order. Research [25] determined that the learning rate has by far the most effect on the performance of the model. It is recommended to first search for the best learning rate with a small model and to determine the number of hidden layers and the number of nodes in those layers afterward. However, it is to note, that the study used the LSTM for classification tasks, not the regression task, like in this work.

Thus the approach in this work is the following.

1. Train small models with different learning rates.
2. Train models with different numbers of LSTM layers.
3. Train models with different numbers of nodes in all layers.

Each model is trained for 500 epochs. The hyperparameter with the best validation loss at every step is used for the next step. After finding the best hyperparameters, the model gets trained for a larger amount of epochs with those hyperparameters to further improve performance. It would be preferable to train every model longer while searching for hyperparameters, but the training times get quite long, which makes it impractical.

Another approach would be to use random search [26] to search over the hyperparameter grid.

4.3 Search

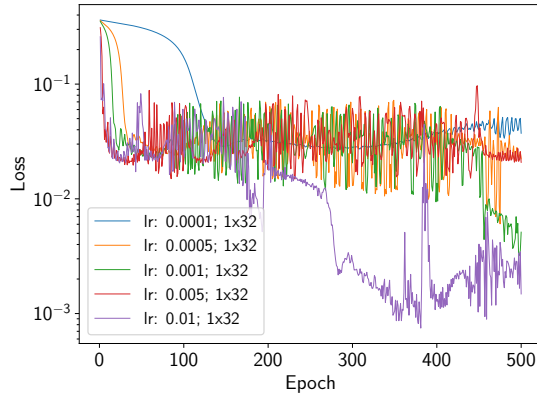
In this section, the hyperparameter search as defined above is applied to the four different datasets.

4.3.1 Drag-Simple

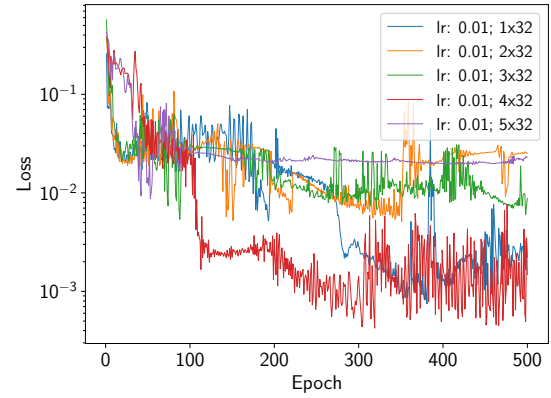
First, the simple drag dataset is examined.

In Figure 7a different learning rates are compared. Initially, the higher the learning rate, the faster the drop. After plateauing for a while, the learning rates 0.01 and 0.0001 fall even lower, however, there is no clear pattern recognizable. The best loss is achieved with a learning rate of 0.01.

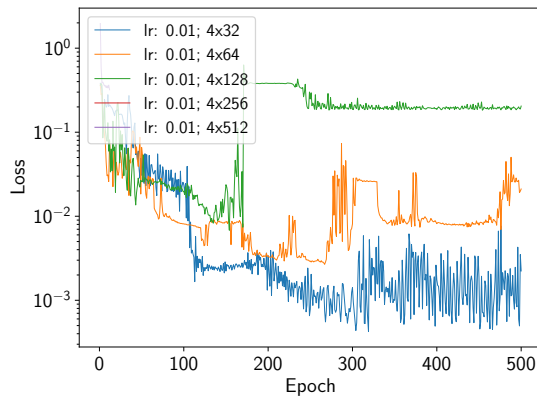
⁴A model with two layers and 256 nodes requires around 3 MB. If the training took 500 epochs, saving the model after every epoch would take 1.5 GB to save all the models of one configuration.



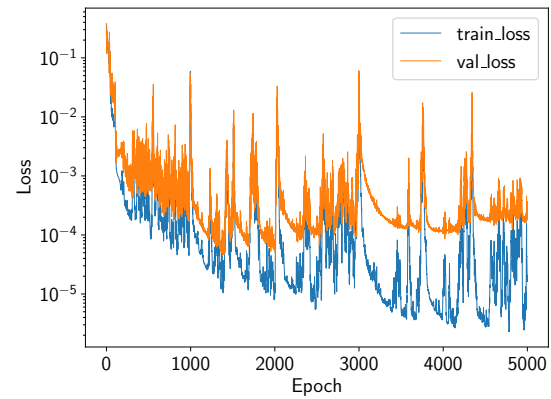
(a) Comparison of validation loss for different learning rates.



(b) Comparison of validation loss for amounts of layers.



(c) Comparison of validation loss for amounts of nodes in each layer.



(d) Comparison of training and validation loss of the best model.

Figure 7: Hyperparameter search for the Drag-Simple dataset; the y-axis shows the model loss (lower is better), the x-axis shows training epoch. The models are identified by " $\#$ layers \times $\#$ nodes", e.g. the model "1x32" has one hidden layer and 32 nodes in that layer.

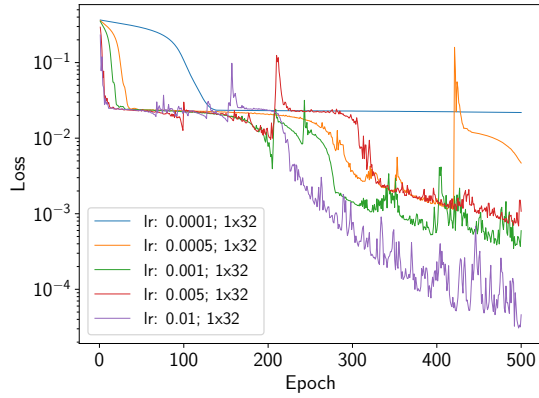
In Figure 7b the models with two and four layers perform best.

Figure 7c shows that the smaller the model the better it trains. Additionally, it is to note, that the models with 256 and 512 stop early in training because the losses are NaN values. This seems to imply that the training is unstable due to the high learning rate and the larger models. The model with 32 nodes achieves the lowest loss.

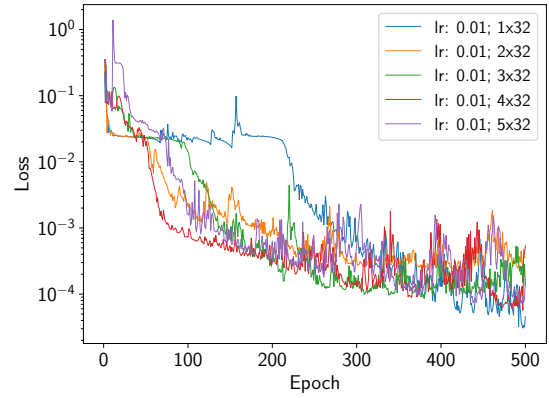
When training this model over a longer period of time, the model is overfitting on the training set, which leads to a further decrease in the training loss and an increase in the validation loss after 1500 epochs, see Figure 7d.

4.3.2 Drag-Complex

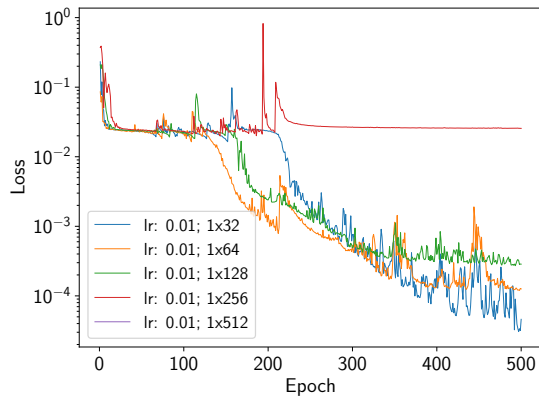
Figure 7a shows how the higher learning rates converge faster. The model with one layer initially learns slower in Figure 7b, but later achieves the lowest loss.



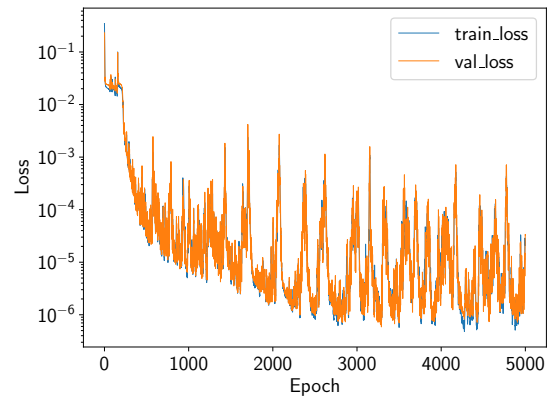
(a) Comparison of validation loss for different learning rates.



(b) Comparison of validation loss for amounts of layers.



(c) Comparison of validation loss for amounts of nodes in each layer.



(d) Comparison of training and validation loss of the best model.

Figure 8: Hyperparameter search for the Drag-Complex dataset; the y-axis shows the model loss (lower is better), the x-axis shows training epoch. The models are identified by "#layers x #nodes", e.g. the model "1x32" has one hidden layer and 32 nodes in that layer.

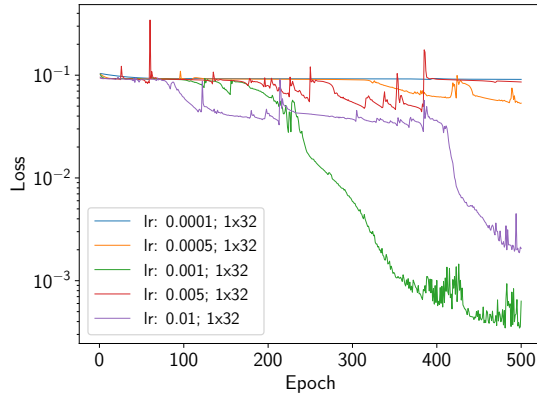
Figure 7c indicates again that the training of the models with a high amount of nodes is unstable, as the model with 512 nodes stops early and the model with 256 nodes does not improve.

The validation loss follows the training loss really closely in Figure 7d. This implies that the training and validation sets might be too similar.

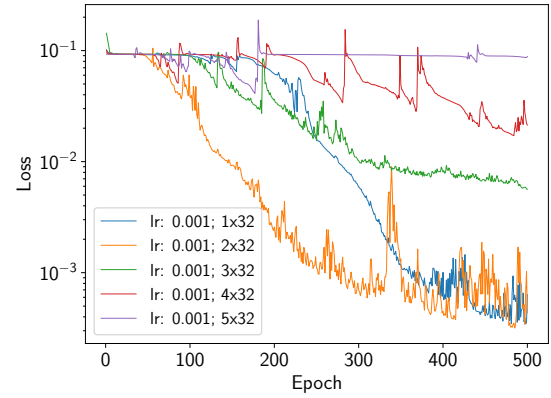
4.3.3 Pendulum-Simple

Figure 7a shows that the best learning rate is 0.001, but there is no pattern recognizable.

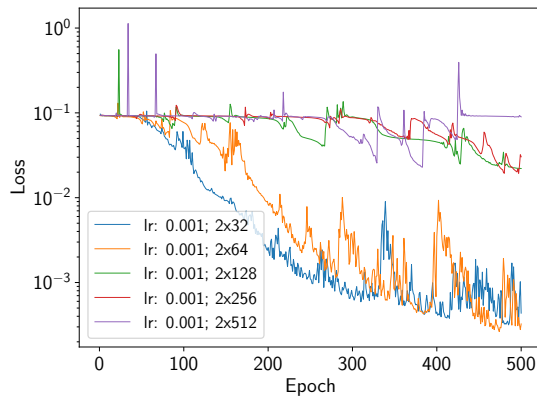
In Figure 7b however, there is a pattern of models with more layers learning worse. The model with two layers performs best.



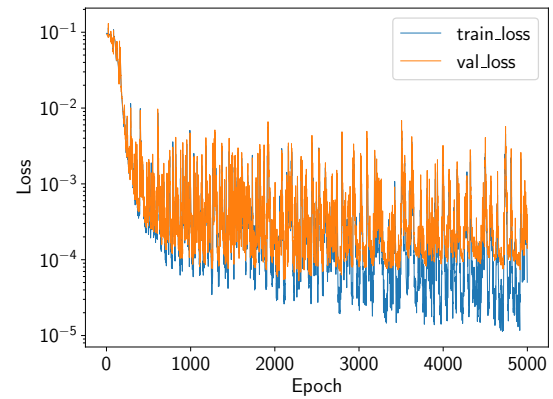
(a) Comparison of validation loss for different learning rates.



(b) Comparison of validation loss for amounts of layers.



(c) Comparison of validation loss for amounts of nodes in each layer.



(d) Comparison of training and validation loss of the best model.

Figure 9: Hyperparameter search for the Pendulum-Simple dataset; the y-axis shows the model loss (lower is better), the x-axis shows training epoch. The models are identified by "#layers x #nodes", e.g. the model "1x32" has one hidden layer and 32 nodes in that layer.

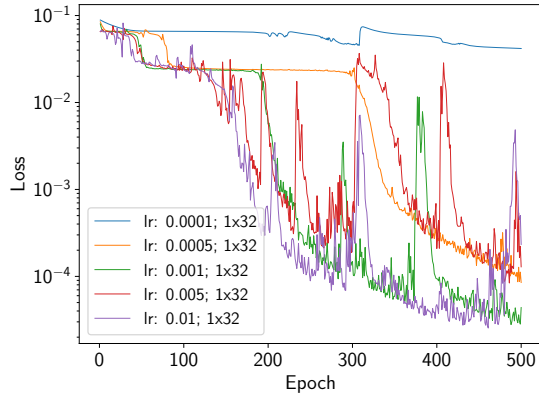
Figure 7c shows that, with the lower learning rate, the models with a high amount of nodes are now at least training, but are still unstable and not converging, while the other models converge well.

With more training epochs the validation loss increases again while the training loss decreases further, see Figure 7d.

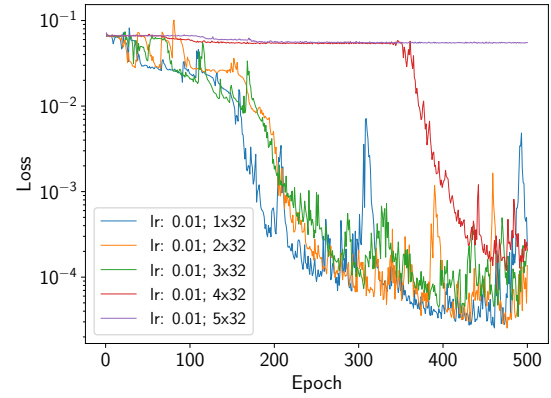
4.3.4 Pendulum-Complex

Of the models in Figure 10a, the bigger learning rates train well, only the lowest learning rate of 0.0001 does not converge.

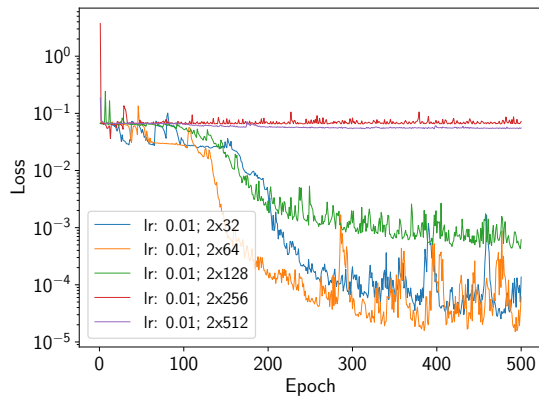
In Figure 10b the smaller the model the faster it trains. The loss of the largest model only falls slightly and stagnates. It seems to be stuck in a local minimum.



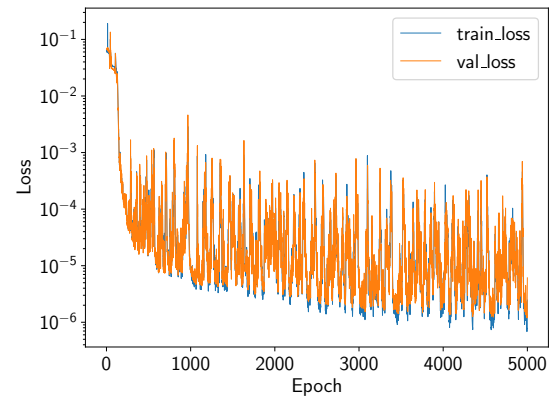
(a) Comparison of validation loss for different learning rates.



(b) Comparison of validation loss for amounts of layers.



(c) Comparison of validation loss for amounts of nodes in each layer.



(d) Comparison of training and validation loss of the best model.

Figure 10: Hyperparameter search for the Pendulum-Complex dataset; the y-axis shows the model loss (lower is better), the x-axis shows training epoch. The models are identified by "#layers x #nodes", e.g. the model "1x32" has one hidden layer and 32 nodes in that layer.

Figure 10c shows again, that the smaller models train well, but the two large models seem both to be stuck in a local minimum.

When comparing the training and validation loss of the best model in Figure 10d, it seems that the validation dataset is very similar to the training set. The training loss falls a bit lower than the validation loss, but they mostly follow each other.

4.3.5 Patterns

Larger models seem to be way harder to train than smaller ones. There is no clear connection between a high learning rate and unstable training. The higher learning rates generally converge faster, which is to be expected. Furthermore, for the more complex datasets, the training loss and the validation loss are very similar.

In the next section, the best model after each hyperparameter search will be evaluated on the test set.

5 Evaluation

In this section, the best models, as determined in the last section, are tested on the test sets. To evaluate the models the squared error is used. The loss function employs the same error. However, this time the error is calculated with the values scaled back to their original range to show the potential real-world error. Additionally, the error is analyzed at certain time steps, to see at what points the model has difficulties to predict the function.

5.1 Drag-Simple

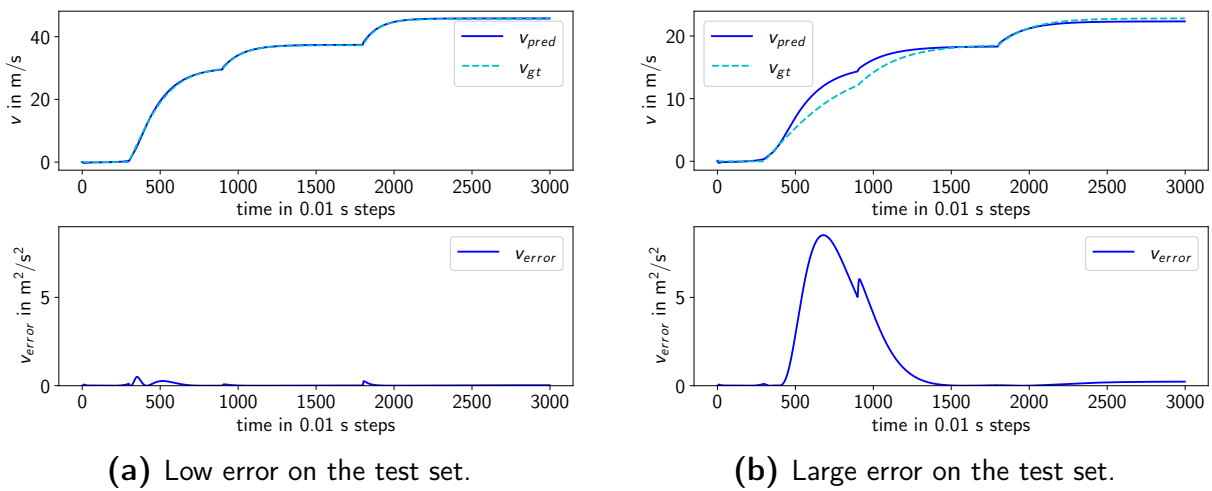


Figure 11: Model prediction on the drag-simple test set with the squared error below.

Figure 11a shows that the model can follow the ground truth well. There are however some small bumps when the input changes and at the beginning of the sequence. Figure 11b shows another sample of the test set. The error is considerably worse.

The overall MSE on the test set is $3828 \times 10^{-4} m^2/s^2$, which is mainly caused by the large error in Figure 11b. The model performs well for all other samples.

5.2 Drag-Complex

It is clear to see in Figure 12a and Figure 12b that the model follows the ground truth well. The only recognizable errors are again at the points when the input changes.

The overall MSE on the test set is $5 \times 10^{-4} m^2/s^2$. This is orders of magnitude lower than the MSE of the drag-simple dataset.

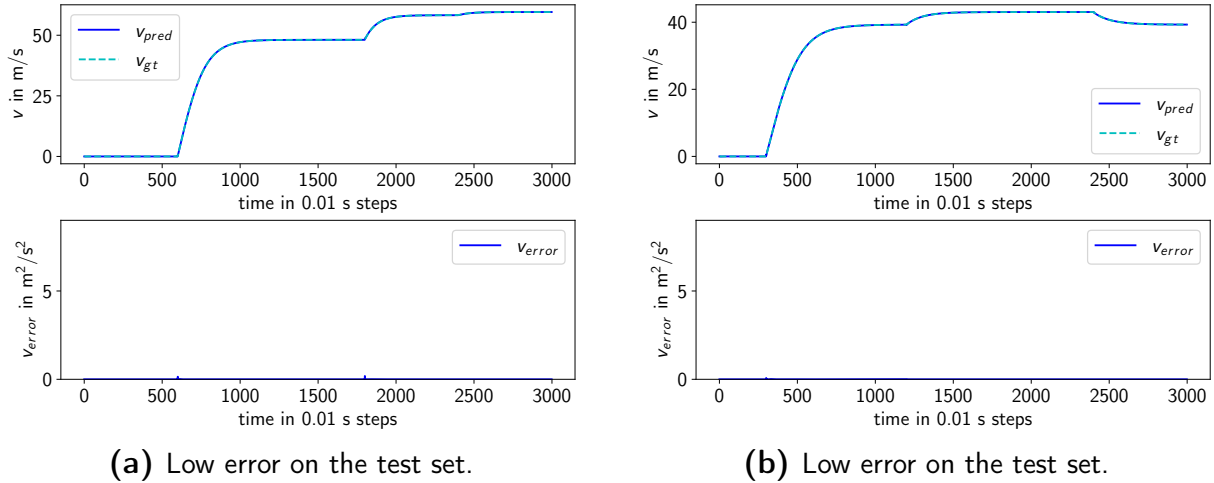


Figure 12: Model prediction on the drag-complex test set with the squared error below.

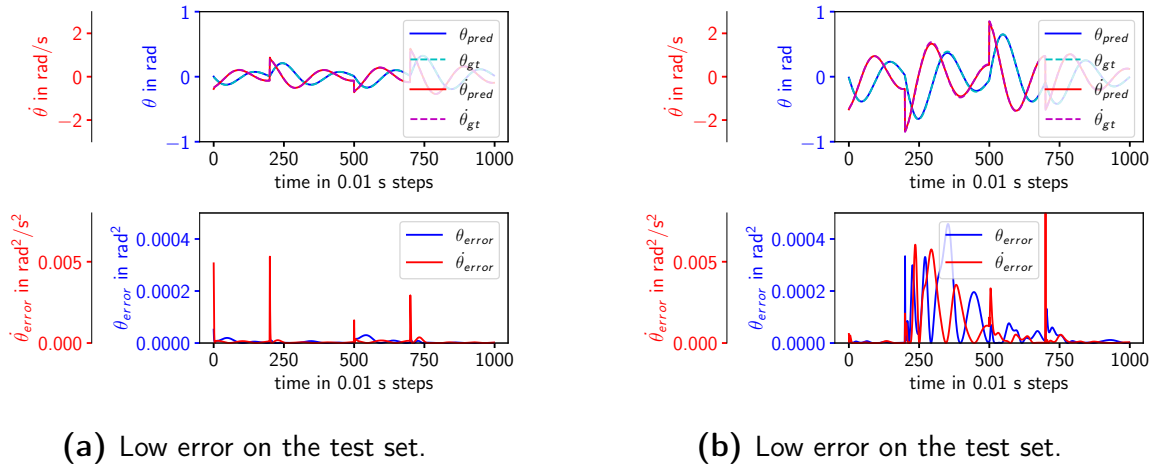


Figure 13: Model prediction on the pendulum-simple test set with the squared error below.

5.3 Pendulum-Simple

Both variables in Figure 13a and Figure 13a follow the function well. Both samples show clear spikes of error in $\dot{\theta}$ at points of input change, and at the beginning of the sequence.

For θ and $\dot{\theta}$ the overall MSE for the test set is $2 \times 10^{-5} \text{ rad}^2$ and $20 \times 10^{-5} \text{ rad}^2/s^2$, respectively. The error is one order of magnitude smaller for θ .

5.4 Pendulum-Complex

In the case of Figure 14a and Figure 14b both variables show almost no error in comparison with the simpler dataset. The only recognizable small bumps are again at the points when the input changes.

The MSE for θ and $\dot{\theta}$ on the whole test set is $4 \times 10^{-6} \text{ rad}^2$ and $29 \times 10^{-6} \text{ rad}^2/s^2$, respectively. As for the pend-simple dataset, the θ error is almost one magnitude smaller. Furthermore, the model performs better for the more complex dataset. This might imply that even though

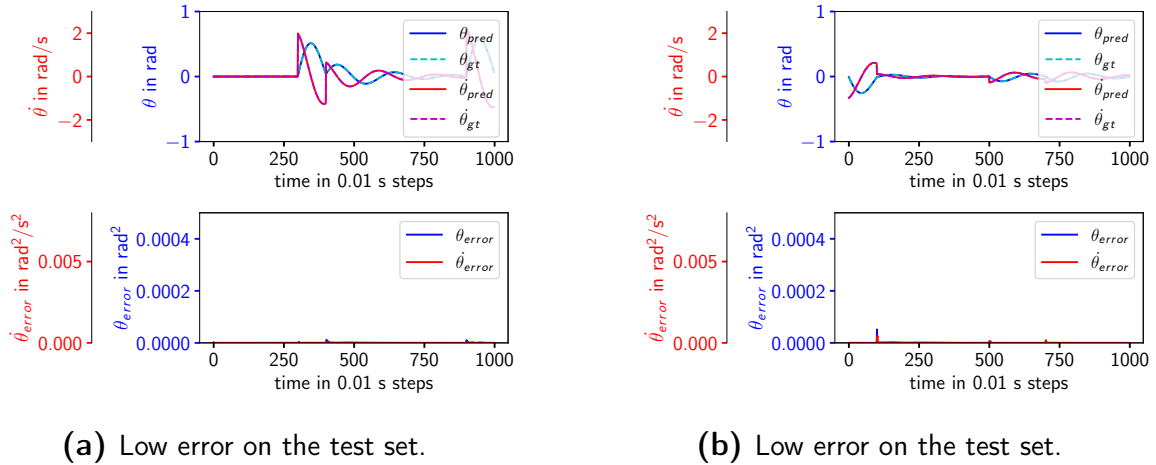


Figure 14: Model prediction on the pendulum-complex test set with the squared error below.

the model needs to learn a wider variety of data, it can overcome this with the larger dataset. However, it is possible that the test set is too similar to the training set and the model doesn't generalize well.

6 Conclusion

The Hyperparameter search clearly shows that training larger models is more difficult.

The evaluation of the best models on the test set shows a clear pattern of relatively large error spikes when the input variable changes. A similar error spike can be noticed at the beginning of a sequence. However, in general, the models can predict the states of the ODEs really well.

In summary, this work shows that LSTMs are clearly able to predict future states of nonlinear differential equations from only the initial condition and the impulse. For similar dynamic systems like the two in this work, it can be recommended to employ a small model with 32 or 64 nodes, two LSTM layers and a learning rate of 0.01. However, when using the methods for more complex systems, it is recommended to perform a hyperparameter search for that task specifically as the model might need to be larger to perform well.

Furthermore, there is much work to be done. This includes creating testing datasets, that are better at testing the generalization of the model. It might be helpful to analyze exactly what training data is necessary to get the model to predict accurately for a certain range of test data. This information can then be used to run a real-world system to efficiently generate the exact training data for the model, which is later needed to predict the states that are in demand at inference time. When the dataset is expanded and/or the ODEs get more complex, there might be a larger model needed to increase its capacity. For those larger models, a deeper look into the learning instabilities is necessary.

References

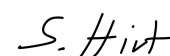
- [1] E. Hairer, S. P. Nørsett, and G. Wanner, *Solving ordinary differential equations. I, Nonstiff problems*. Springer-Vlg, 1993.
- [2] E. Hairer and G. Wanner, *Solving ordinary differential equations. II, Stiff and differential-algebraic problems*. Springer-Vlg, 2002.
- [3] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” *CoRR*, vol. abs/1409.3215, 2014. [Online]. Available: <http://arxiv.org/abs/1409.3215>
- [4] J. Donahue, L. A. Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, “Long-term recurrent convolutional networks for visual recognition and description,” *CoRR*, vol. abs/1411.4389, 2014. [Online]. Available: <http://arxiv.org/abs/1411.4389>
- [5] A. Graves, “Generating sequences with recurrent neural networks,” *CoRR*, vol. abs/1308.0850, 2013. [Online]. Available: <http://arxiv.org/abs/1308.0850>
- [6] A. Karpathy, “The unreasonable effectiveness of recurrent neural networks,” May 2015, accessed on 23/11/2022. [Online]. Available: <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>
- [7] S. Mehtab, J. Sen, and A. Dutta, “Stock price prediction using machine learning and lstm-based deep learning models,” in *Symposium on Machine Learning and Metaheuristics Algorithms, and Applications*. Springer, 2021, pp. 88–106.
- [8] M. Madondo and T. Gibbons, “Learning and modeling chaos using lstm recurrent neural networks,” in *MICS 2018 proceedings*, 2018.
- [9] Y. Yu, X. Si, C. Hu, and J. Zhang, “A review of recurrent neural networks: Lstm cells and network architectures,” *Neural computation*, vol. 31, no. 7, pp. 1235–1270, 2019.
- [10] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
- [11] F. A. Gers, J. Schmidhuber, and F. Cummins, “Learning to forget: Continual prediction with lstm,” *Neural computation*, vol. 12, no. 10, pp. 2451–2471, 2000.
- [12] W. Ma and J. Lu, “An equivalence of fully connected layer and convolutional layer,” *arXiv preprint arXiv:1712.01252*, 2017.
- [13] F. Günther and S. Fritsch, “Neuralnet: training of neural networks.” *R J.*, vol. 2, no. 1, p. 30, 2010.
- [14] P. J. Werbos, “Backpropagation through time: what it does and how to do it,” *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.
- [15] F.-F. Li, J. Wu, and R. Gao, “Learning,” 2022, accessed on 08/12/2022. [Online]. Available: <https://cs231n.github.io/neural-networks-3/>
- [16] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

- [17] Z. Wang and A. C. Bovik, "Mean squared error: Love it or leave it? a new look at signal fidelity measures," *IEEE Signal Processing Magazine*, vol. 26, no. 1, pp. 98–117, 2009.
- [18] R. P. Feynman, R. B. Leighton, and M. Sands, "The feynman lectures on physics; vol. i," *American Journal of Physics*, vol. 33, no. 9, pp. 750–752, 1965.
- [19] G. Parker, "Projectile motion with air resistance quadratic in the speed," *American Journal of Physics*, vol. 45, no. 7, pp. 606–610, 1977.
- [20] G. L. Baker and J. A. Blackburn, *The pendulum: a case study in physics*. OUP Oxford, 2008, pp. 27–52.
- [21] K. Ahnert and M. Mulansky, "Odeint - solving ordinary differential equations in C++," *CoRR*, vol. abs/1110.3397, 2011. [Online]. Available: <http://arxiv.org/abs/1110.3397>
- [22] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, Í. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.
- [23] F.-F. Li, J. Wu, and R. Gao, "Image classification," 2022, accessed on 08/12/2022. [Online]. Available: <https://cs231n.github.io/classification/>
- [24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
- [25] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 10, pp. 2222–2232, 2016.
- [26] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization." *Journal of machine learning research*, vol. 13, no. 2, 2012.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den December 14, 2022



Sebastian Hirt