The solution of time-dependent ordinary differential equations using neural networks: collocation polynomial neural forms and adaptive neural domain refinement

Von der Fakultät 1 - MINT - Mathematik, Informatik, Physik, Elektro- und Informationstechnik der Brandenburgischen Technischen Universität Cottbus-Senftenberg genehmigte Dissertation zur Erlangung des akademischen Grades eines

Dr. rer. nat.

vorgelegt von

Toni Schneidereit

geboren am 25.08.1991 in Cottbus

Vorsitzender:	Prof. Dr. rer. nat. habil. Gerd Wachsmuth
Gutachter:	Prof. Dr. rer. nat. habil. Michael Breuß
Gutachter:	Prof. Dr. rer. nat. Carsten Hartmann
Gutachter:	Prof. Em. Dr. Isaac Lagaris

Tag der mündlichen Prüfung: 16.12.2022

DOI: https://doi.org/10.26127/BTUOpen-6389

Summary

Solving differential equations is still a topic of major interest, due to their appearance in many fields of science and engineering. Differential equations may arise from modelling processes of, e.g., physical phenomena. Numerical methods for the approximation of their solution are often considered when there is no analytical solution. A classic approach for solving differential equations with neural networks builds upon trial solutions, the so-called neural forms. The latter are incorporated in a cost function that is subject to minimisation, to train the involved neural networks. The cost function can be constructed using the differential equation with a discretisation of the solution domain. Neural forms represent general and flexible tools for solving ordinary differential equations, partial differential equations as well as systems of each. However, the computational approach is in general highly dependent on a variety of computational parameters and the choice of the optimisation methods.

Studying the solution of a simple but fundamental stiff ordinary differential equations with small feedforward neural networks and first order optimisation shows, that it is possible to identify preferable choices for parameters and methods. It also reveals the importance of a careful choice of the computational setup. That is, the neural network weight initialisation appears to be a sensitive topic, while having a major impact on the solution accuracy. Especially the use of non-random (deterministic) weights partially shows poor performance, but removes a stochastic component. Further research reveals, that a new polynomial representation of the neural forms can significantly increase the reliability of a deterministic initialisation (all weights have initially the same values assigned). In general, the neural forms approaches either incorporate given initial values directly in their construction or add them within additional terms to the cost function. In order to maintain smaller neural network architectures and solve the differential equation, even on fairly large domains, a new technique called domain segmentation (for initial value problems) is introduced. The solution domain splits into equidistant subdomains and the above-mentioned collocation polynomial neural forms are solved separately in each domain fragment. At the boundary of any subdomain, a new initial value is provided by the neural forms solution and directly incorporated in the adjacent one. Results for more difficult differential equations show that this approach can even further improve the weight initialisation topic.

In classic adaptive numerical methods for solving differential equations, the mesh as well as the domain may be refined or decomposed, respectively, in order to improve numerical accuracy. The subdomain distribution can also be connected with an adaptive refinement. That is, the neural network training status is combined with an adaptive subdomain size reduction in the new adaptive neural domain refinement algorithm. That is, each subdomain is reduced in size until the optimisation is resolved up to a predefined training accuracy. In addition, while the neural networks are by default small, the number of neurons may also be adjusted in an adaptive way. Conditions are introduced to automatically confirm the solution reliability and optimise computational parameters whenever it is necessary.

After an introduction with the motivation and a literature review,the mathematical methods are described in detail, together with discussing corresponding results. In the end, a discussion brings the main results into context.

Zusammenfassung

Die Lösung von Differentialgleichungen ist auch heutzutage ein wichtiges Thema, da diese in vielen Bereichen der Wissenschaft auftauchen und zum Beispiel bei der mathematischen Modellierung physikalischer Systeme Anwendung finden. Oftmals besitzen diese Gleichungen keine analytische Lösung, weshalb numerische Methoden zur Approximation verwendet werden. Ein klassischer Ansatz aus dem Bereich der Neuronalen Netze setzt auf sogenannte „trial solutions“ oder „neural forms“. Diese werden in eine Kostenfunktion eingebunden, welche zum Zweck des Trainings der Neuronalen Netze minimiert werden soll. Mit Hilfe der Struktur der gegebenen Differentialgleichung und einer Diskretisierung des Lösungsgebiets kann eine solche Kostenfunktion aufgestellt werden. Generell bilden die neural forms flexible Werkzeuge zur Lösung von gewöhnlichen, partiellen und auch Systemen von Differentialgleichungen. Dennoch sind diese rechnergestützten Ansätze stark abhängig von einer Vielzahl an Parametern.

Eine rechenbasierte Studie zur Lösung eines einfachen, steifen Anfangswertproblems mit einfachen, sogenannten „feedforward neural networks“ und Optimierung erster Ordnung zeigt, dass die sorgfältige Auswahl durchaus vorteilhafte Parameterkombinationen ergeben kann. Die Initialisierung der Anfangsgewichte der Neuronalen Netze ist ein sensibles Thema. Während konstant (nicht zufällig) gewählte Gewichte einen stochastischen Effekt eliminieren, so liefern sie stellenweise keine verlässlichen Approximationen.

Weiterführend ist es jedoch möglich, mit einer Polynomdarstellung der neural forms, die Aussagekraft der Lösungen mit konstanten Gewichten deutlich zu verbessern. Die oben genannte rechenbasierte Studie zeigt zudem auf, dass zu groß gewählte Lösungsgebiete zu keinen verlässlichen Ergebnissen führen können. Eine Charakteristik der neural forms bezieht sich auf die Einbindung gegebener Anfangswerte in die neural forms selbst oder direkt in die Kostenfunktion. Um die Architekturen der Neuronalen Netze weiterhin klein zu halten und die Differentialgleichungen trotzdem auch auf größeren Gebieten lösen zu können, wird die Methode der „domain segmentation“ eingeführt. Diese teilt das gesamte Gebiet in kleinere Untergebiete auf und löst die „collocation polynomial neural forms“ einzeln in jedem der gleichverteilten Untergebiete. Die Lösung in einem dieser Untergebiete liefert dann den neuen Anfangswert für das benachbarte Untergebiet. Das Testen dieser Methode an Differentialgleichungen mit speziellen Charakteristiken zeigt, dass dadurch die Verlässlichkeit konstanter Anfangsgewichte weiter vorangetrieben werden kann.

Die Gebietsaufteilung durch domain segmentation ist ein Ansatz speziell für Anfangswertprobleme. In der klassischen Numerik werden oft adaptive Ansätze zur Erhöhung der Genauigkeit mit in die Lösung von Differentialgleichungen eingebaut. Dies können zum Beispiel lokale Gitterverfeinerungen oder Gebietsaufteilungen sein. Der domain segmentation Ansatz kann mit einer zusätzlichen adaptiven Verfeinerung kombiniert werden, in welchem die Kostenfunktion ausgewertet und als Status des Trainings der Neuronalen Netze definiert wird. Zusammen mit einem adaptiven Algorithmus für eine mögliche Verkleinerung der oben genannten Untergebiete ist der „adaptive neural domain refinement“ Ansatz entstanden. Jedes Untergebiet kann in seiner Größe verringer werden, bis die Minimierung der Kostenfunktion bis zu einer vordefinierten Genauigkeit erreicht ist. Eine Komponente zur automatischen Bestätigung des Ergebnisses ist in den Algorithmus aufgenommen worden. Zusätzlich zu den normalerweise

kleinen Neuronalen Netzwerken wird eine automatische Anpassung der Parameter ermöglich, sollten einzelne Untergebiete in ihrer Größe zu klein werden.

Nach einer allgemeinen Einführung mit Beschreibung der Motivation und einer Literaturübersicht zu den bereits existierenden wissenschaftlichen Arbeiten auf diesem Gebiet, werden die mathematischen Grundlagen näher beleuchtet und zusammen mit den entsprechenden Resultaten ausgewertet. Am Ende werden die Hauptresultate noch einmal im gegenseitigen Zusammenhang diskutiert.

Acknowledgement

This work was funded by the Graduate Research School (GRS) from the Brandenburg University of Technology Cottbus-Senftenberg as a part of the Research Cluster Cognitive Dependable Cyber Physical Systems. I am very grateful for the granted opportunity from this scholarship. Many thanks to all the unknown reviewers from the Springer journal "Neural Computing and Applications", Algoritmy Conference and ICAISC 2021 for their excellent work on remarks and suggestions when reviewing the submitted papers. I want to align special thanks to Prof. Dr. Michael Breuß for his supervision and support during the research and creation process of the produced papers. He was always available and motivated to discuss ideas and possible research directions. To my dearest family Slavomíra, Karin, Roland, Diana and Anna.

Contents

1 Introduction

The very first section of this thesis is dedicated to provide the motivation of solving differential equations with numerical and neural network based approaches. Especially initial value problems lay in the focus of the motivation and the formulation of the general problem. Followed up by a brief illustration of the contribution and a detailed literature review.

1.1 Motivation and contribution

Differential equations (DEs) in general are important models in many fields of science and engineering, as they often represent real-world behaviour [1, 2]. They are usually formulated as initial or boundary value problems, where conditions at the beginning of a process or at boundary points are given to obtain one specific solution. The equations incorporate, depending on their type, the solution function, one or more independent variables, additional constant terms and at least the first derivative of the solution function. They may appear as individual equations or as systems of equations, where it is also possible that there are coupling terms involved. For ordinary differential equations (ODEs) [3, 4], the solution function incorporates only one independent variable. Differential equations with more than one independent variable are called partial differential equations (PDEs) [5, 6]. Well known described physical phenomena by ODEs are, e.g., the radioactive decay, the harmonic oscillation and the predator–prey model (Lotka-Volterra equations) as a system of coupled ODEs. Most physical phenomena, however, are modelled as PDEs, since their behaviour depends on several quantities. Important examples are, e.g., the heat equation, the Schrödinger equation and the Navier-Stokes equations. The analytical solution of a differential equation is a function which satisfies (together with its derivative(s)) the DE itself and the given initial and/or boundary conditions. A special class of (first order) ODEs are initial value problems (IVPs), describing the time evolution of a system with a given initial state.

Those equations have in many cases a very difficult to find analytical solution. Therefore, the consideration of numerical methods for an approximation of the solution is often unavoidable. While Runge-Kutta 4 (fourth order Runge Kutta, RK4) [3, 4] is widely used for time integration of initial value problems, the finite element method (FEM) [7] can be used to solve PDEs on arbitrarily shaped domains. In addition, the finite differences approach is also commonly used [1]. Such approaches use a discretisation of the computational domain for finding an approximation of the solution. In order to obtain higher accuracy and robustness, many numerical schemes also feature adaptive mechanisms regarding, e.g., step size control [2, 3] for RK4 or adaptive mesh refinement [8, 9, 10] for FEM. That is, certain areas of the solution domain may require more elements or grid points, in other words a refined mesh, to improve to reliability and accuracy. Such adaptive mesh refinement techniques enable the mesh to be locally refined based on a suitable error estimate. Since numerical methods provide an approximation of the DE solution, the underlying algorithms need to be analysed and somehow classified. Numerical algorithms and the investigation of their characteristics are part of numerical analysis [11, 12]. There are a lot of important aspects, which are briefly described in the following.

As there are various definitions and appearances of numerical stability, this charac-

teristic is connected to the error handling in the beginning or throughout the execution of an algorithm. A method can be considered stable, whenever the error does not stack up in a way that leads to large errors in the final results. Such errors can be related and/or caused by, e.g., the discretisation, rounding errors and machine accuracy. Connected to this interpretation of stability is the condition number, which measures how much the output can be affected by changes in the input values (whereas changes can refer to errors). A small condition number is desirable. Also connected to stability is the convergence and the rate of convergence. As mentioned above, numerical schemes like RK4 use a discretisation of the solution domain. For a certain number of discrete grid points, the numerical solution will be somehow close to the analytical solution. Now, the underlying numerical method converges, if the approximate solution gets closer and closer to the real solution by increasing the number of discrete grid points. In other words, the algorithm approaches a fixed value by refining the discrete domain. The rate of convergence quantifies how fast the algorithms approaches this fixed value. Another important characteristic is consistency, which means, the truncation error approaches zero by refining the grid and the approximate solution advances towards the analytical solution. In general, having stability and consistency implies that the numerical method converges. Related to stability, there is a certain phenomenon, namely stiffness, which is highly interesting. This phenomenon can cause a numerical method to be unstable, although the solution to the corresponding differential equation does not visually imply such a behaviour. In order to stabilise the numerical solution for stiff differential equations, the step size has to be chosen extremely small.

While there already exist numerical solution methods that provide highly reliable solutions, the research in that direction never stops trying to find improvements or even news ways to approach differential equations. Although the approximation accuracy plays a key role, other characteristics such as, e.g., computational time, flexibility or the above mentioned stability, also have a high impact. Since neural networks and deep learning have gained tremendous attention over the last decades, scientists have applied various frameworks from this field of science to find an approximate solution of differential equations. The upcoming section (literature review) will provide a detailed overview of the current literature. The main idea behind modelling a neural networks approach is to construct a cost function from the DE structure and training data from the solution domain. Neural networks feature adjustable weights which are, during the training process, updated until the cost function is minimised. Once a network has been trained on a certain interval, the learned weights can be used for compute the DE solution ab arbitrary grid points in this interval. Changing the distribution of the discrete grid points for (most) numerical methods results in a required restart of the method. Therefore, neural network approaches come with a certain amount of flexibility and also have a generalising characteristic attached. Neural network based frameworks have already been designed to represent several numerical methods. However, several numerical analysis related questions still remain open. Scientific works show the approximation capability and therefore, that the approaches converge towards suitable solutions. However, questions about the quality of convergence, consistency and stability, but also about phenomena like stiffness, are not fully resolved yet and (at least partially) remain open. The presented thesis attempts to approach several open questions in that field on a computational level. The quintessence is a detailed investigation and an extension of the neural forms (NF) methods of Lagaris et al. [13, 14] and Piscopo

et al. [15], which are referred to as trial solution method (short TSM) and modified trial solution method (short mTSM), respectively.

Turning to a brief overview of the content and the contribution. While the TSM approach employs a constructed neural form, satisfying given initial or boundary conditions, mTSM defines the NF to be the neural network output and directly incorporates the given conditions into the cost functions. The general task for approaching differential equations with feedforward neural networks (FFNN), is to construct and to minimise a cost function arising from the NF, which connects the differential equation with a small FFNN (from now on simply referred to as neural network). After initialising the neural network weights, the optimisation, with training points from the solution domain, aims to find a useful minimum in the weight space. Then the incorporated neural network, either directly (mTSM) or as a part of the constructed neural form (TSM), represents the solution function and is capable to find the approximation at arbitrary grid points of the domain.

A detailed computational study on those methods with first order optimisation examines the solution of a simple, yet stiff initial value problem (IVP) with the focus on domain size, the stiffness parameter, the number of training points, the weight initialisation, the first order optimisation methods as well as the neural architecture related number of hidden layers and hidden layer neurons. It turns out that not only all the examined parameters depend on each other, but especially the weight initialisation appears to be a sensitive topic. The reason for this behaviour may connect to the complexity of the weight space (or energy landscape), arising from the cost function, which may inherent multiple local minima. In this context, TSM tends to struggle with deterministic initial weights (zeros) and first order optimisation. Also discussed in this context are convergence, consistency and stability. Here the goal is to try to find similarities and relations to numerical methods and numerical analysis on a qualitative level.

The results, especially focusing on stability, motivate to take directions for improving the deterministic weight initialisation for IVPs. The possible NF extension to higher order polynomials for both TSM and mTSM, including more than one (small) neural network, offers a lot more flexibility than changing the architecture of a single neural network. It turns out that the extension significantly increases the usefulness of TSM with deterministic weight initialisation in context of first order optimisation. This extension is referred to as collocation polynomial neural forms (CNFs). Results have additionally shown, that the solution, especially for TSM, works quite well on smaller domains. That motivates the introduction of domain segmentation approach. The idea behind this method is to split the entire solution domain into smaller subdomains and to solve the problem successively in each subdomain. Together with the CNFs, this method enables TSM to provide suitable approximations for IVPs, even on fairly large domains with deterministic weight initialisation. The corresponding neural forms in the domain segmentation approach are called subdomain collocation polynomial neural forms (SCNFs).

While domain segmentation employs equidistant subdomains, one may desire the subdomain distribution according to the IVP difficulty. Neural network approaches offer with their cost function a tool, which can be used to investigate the network error. Here a training error is defined at the training points of the domain, as well as a verification error at intermediate grid points. Those are introduced to quantify

the training status and to investigate possible overfitting. Small errors are desired and small neural networks reduce to amount of possible minima in the weight space as well as overfitting. Hence, this tool together with an iterative subdomain size reduction can be used to determine, whether a subdomain has reached an appropriate size where the TSM SCNF was able to find a useful solution.

The computational characteristics and method extensions are perhaps, and to anticipate some of the later results, not state-of-the-art when compared to highly efficient, standard numerical solvers. However, the findings are highly interesting and contribute to a better understanding of the method related properties. Therefore, we conjecture that the results have a high potential of being further developed and improved. Nonetheless, this is a step forward to fully understand the relation between standard numerics and neural networks approaches.

Published works related to this thesis:

T. Schneidereit, M. Breuß: *Solving ordinary differential equations using artificial neural networks - a study on the solution variance*, Proceedings of the Conference Algoritmy, pp. 21–30 (2020).

T. Schneidereit, M. Breuß: *Computational characteristics of feedforward neural networks for solving a stiff differential equation*, Neural Computing and Applications, 34, pp. 7975–7989 (2022). doi:10.1007/s00521-022-06901-6

T. Schneidereit, M. Breuß: *Polynomial neural forms using feedforward neural networks for solving differential equations*, Artificial Intelligence and Soft Computing, ICAISC 2021. Lecture Notes in Computer Science, 12854, pp. 236-245 (2021). doi:10.1007/978-3-030-87986-0_21

T. Schneidereit, M. Breuß: *Collocation polynomial neural forms and domain fragmentation for initial value problems*, Neural Computing and Applications, 34, pp. 7141–7156 (2022). doi:10.1007/s00521-021-06860-4

T. Schneidereit, M. Breuß: *Adaptive neural domain refinement for solving time-dependent differential equations*, arXiv:2112.12517, (2021).

1.2 Literature review

Artificial neural networks (ANNs) are computational representations of biological neurons and their connections, in order simulate the human learning behaviour. They are capable of recognising possible underlying pattern in all kinds of tasks and are part of artificial intelligence (AI). The world without AI technology is barely imaginable nowadays and already has a tremendous impact on the daily life [16, 17]. Assistant software from major technology or e-commerce companies are used by customers many times a day, potentially without noticing that they run complex AI behind the scenes [18]. Those assistants may be embedded in the software for plenty devices, e.g., smartphones, televisions and even kitchen gadgets [19]. In general, they are able to receive and process user requests, which may incorporate answering questions, turning devices on and off, text-to-speech activities and many more [20]. The interaction with humans is continuously used for training and updating the artificial assistants. While this contributes to the further development and improvement od such assistants, there are also risks connected to AI [21], like, e.g., data privacy, youth protection or its application to warfare.

1.2.1 A brief historical overview

The idea behind creating a human like learning model dates back almost seventy years [22, 23, 24, 25]. In 1943, Warren McCulloch and Walter Pitts suggested an artificial model of a biological neuron in *A logical calculus of the ideas immanent in nervous activity* [26], known as the McChulloch-Pitts-Neuron. Based on the neurological knowledge of nervous activity to this end, they found that neurons and their behaviour can be interpreted by propositional logic. The McCulloch-Pitts-Neuron is considered to be the first mathematical representation of a real-world neuron. In the next two decades, scientists and engineers further developed the idea and proposed works on the learning ability of neural networks, e.g. Donald Hebb in *The organization of behaviour: a neuropsychological theory* [23, 27, 28] or Bernard Widrow and Marcian Hoff in *Adaptive switching ciruits* [29]. In between, the perceptron was introduced by Rosenblatt et al. in *The perceptron: a probabilistic model for information storage and organization in the brain* [30], a first neural network model featuring adjustable connecting parameters, which was capable of solving simple classification tasks after training. A drawback came in the late 1960s, when Marvin Minsky and Seymour Papert published the book *Perceptrons: an introduction to computational geometry* [31] and have shown how a simple linear perceptron is limited in its function. It took more than ten years for the next major steps. In the first half of the 1980s, John Hopfield published several works, including the Hopfield neural network architecture in *Neurons with graded response have collective computational properties like those of two-state neurons* [32]. One year later, together with David Tank, Hopfield successfully applied the novel neural network to the Traveling Salesman Problem [33] in *"Neural" computation of decisions in optimization problems* [34]. With the works of Hopfield, the area of neural networks gained interest again. The famous backpropagation learning algorithm came up one year ahead by David Rumelhart et al. in *Learning representations by back-propagating errors* [35]. The algorithm is based on the gradient descent method. In the end of the 1980s, and in the early 1990s, the universal approximation theorem was proven for sigmoidal neurons

in *Approximation by superpositions of a sigmoidal function* [36] by George Cybenko. It basically states, that a neural network with only one processing layer and a finite number of sigmoidal neurons is able to approximate every continuous function. Kurt Hornik showed in *Approximation capabilities of multilayer feedforward networks* [37] that the universal approximation characteristic rather arises from the network architecture then from the sigmoidal neurons. In the following years and decades, the interest in artificial neural networks was continuously rising. Then, in 2006, with the work of Geoffrey Hinton et al., proposed in *A fast learning algorithm for deep belief nets* [38], artificial neural networks and artificial intelligence became a tremendous boost with the introduction of deep learning. The term arises from the deep neural network structure, where several hidden processing layers with many neurons are involved.

1.2.2 Artificial neural network architectures

Over the time, many ANN architectures have been developed with their own characteristics [22, 39, 40, 41]. An ANN uses artificial neurons to apply a so-called activation function to incoming information. Often used are the sigmoid activation function, hyperbolic tangent or the rectified linear unit [42]. Depending on the architecture, the neurons are connected to other neurons through adjustable parameters, the so-called neural networks weights. The results of the information processing are returned by the ANN. This output contributes to adjusting the weights, which is referred to as neural network training. The task of training the ANN, is to minimise a cost function in order to update the connecting weights, until the ANN provides the desired output. In the simplest case, the ANN output compares to a value which is known to be the true outcome to a corresponding input. A cost function may arise from the discrepancy between both the ANN and the correct output. Popular optimisation techniques are, e.g., backpropagation [43], Adam optimisation [44] or BFGS quasi-newton methods [46].

A common ANN architecture is the feedforward neural network (FFNN) [40]. It features several layers with processing neurons, which may be linear or non-linear regarding the activation function. The input layer passes data into the network using at least one, usually linear, neuron. In the hidden layer, the neurons are assigned with an activation function, which actually processes the data and passes it either to another hidden layer or to the output layer. The latter produces the ANN outcome, where the amount of output neurons dictate the number of returned values. The term feedforward implies that the data is truly processed forward through the layers, so that there are no circles or back leading connections involved. Important to mention are bias neurons. These neurons contribute to the neurons in the next layer as constant values or offsets. They do not receive information from previous layers, but have a weight assigned as well.

A modification of the FFNN is the recurrent neural network (RNN) [22]. Besides processing information forward through the network, internal connections between neurons may appear as well as connections between output and input neurons. Therefore the output may be directly incorporated in the input again. In this way, the RNN is said to have a memory, because the internal state depends on all previous inputs. For learning a RNN, both input and output data are processed through the network, until the output is stationary. Language processing is a common field of application for RNNs [47].

A specific variant of the RNN comes with the Hopfield neural network [32], where all the neurons are connected to each other. Excluded are connections within one neuron. The neurons themselves are binary with values of positive 1 or negative 1 in case of the binary Hopfield network or have real values between negative one and positive one assigned in case of the continuous Hopfield network architecture.

The radial basis function neural network (RBF) [22] has an input layer, an output layer and a hidden layer with radial basis functions representing the activation functions. That is, depending on the choice of the basis functions, the input data may be expanded into, e.g., polynomials. The weights from the hidden to the output layer are the function coefficients. The RBF may be found in classification tasks [48].

Deep neural networks (DNNs) [38] are often extensions of common architectures, with multiple hidden layers and usually many processing neurons in each layer. This may tremendously increase the number of adjustable network weights, but can contribute to the solution of difficult or complex tasks.

An important DNN architecture is represented by deep convolutional neural networks (CNNs) [49]. Those are common, e.g., in image or audio recognition [50]. The neurons in a CNN represent convolutional kernels, which can extract certain features from, e.g., an image. The more complex or abstract a feature is, the more convolutional kernels may be necessary. So-called pooling layers are additionally incorporated in CNNs. These layers can get rid of unnecessary information by, e.g., averaging the outputs of each convolutional layer in order to speed up the training, as less variables are present. After a series of convolutional and pooling layers, the CNN usually has a deep FFNN connected, where the output neurons provide a probability related to the possible classifications. With supervised learning, where the desired output to a corresponding input in a training set in known, the comparison between the CNN output and the desired output leads to a training error, which may be used together with backpropagation in order to train the network weights.

1.2.3 First approaches using ANNs

In 1990, Lee et al. published their work *Neural algorithm for solving differential equations* [51]. They show how the method of finite differences can be used to construct a cost function incorporating the Hopfield model [52] for IVPs as well as higher order differential equations. The learning rule is derived from the time evolution of the constructed cost function. The authors find that the neural network approach requires, compared to standard numerical methods at this time, less grid points and rather increasing the number of neurons will contribute to the solution. Results on ODEs show that increasing the total number of neurons actually decreases the necessary time for solving the differential equation due to the parallel interactions between all neurons in each iteration.

Maede et al. published in 1994 *Numerical solution of linear ordinary differential equations by feedforward neural networks* [53] and *Solution of nonlinear ordinary differential equations by feedforward feural fetworks* [54]. They constructed a FFNN for both linear and non-linear ODEs directly without the need for training the network. That is realised by approximating the solution with basis functions. The latter represent the activation functions in the network, while the input layer and bias weights arise from the domain data and the hidden layer weights from the expansion coefficients. The

basis expansion may be realised by hat functions, splines or, e.g., sigmoid functions. Since there are various parameters in neural networks, the authors state that this approach imposes constraints on the involved parameters and to assign them specific roles for a better understanding. Additionally, they avoid cost function minimisation find a fixed minimum in the weight space, rather than only a local one. An important result of this approach is the quadratic decrease of the approximation error when increasing the number of hidden layer neurons, which represent the basis functions.

1.2.4 ANN approaches based on neural forms

In 1998, Lagaris et al. proposed a novel approach for solving DEs including a feed-forward neural network in *Artificial neural networks for solving ordinary and partial differential equations* [13]. In order to solve ODEs, PDEs as well as systems of differential equations, a constructed trial solution or neural form is introduced. The solution then comes in a differentiable and closed-form, which is a contrast to numerical methods like Runge-Kutta [55]. However, the approach still requires a discretised domain with grid points acting as the training points. The authors in [13] use a second order optimisation method (quasi-Newton BFGS [46]) to train their fairly small FFNN over the domain. The results are partially compared to the FEM and perform very well. In the end, the authors state that finding an optimal amount of hidden layer neurons may further improve the results and varying the domain discretisation, e.g. with more grid points in regions of a higher error, may also contribute to the accuracy. Regarding the neural form construction, various shapes may be suitable for a single differential equations with given initial or boundary conditions. In *Systematic construction of neural forms for solving partial differential equations inside rectangular domains, subject to initial, boundary and interface conditions* [56], Lagari et al. presented and discussed a systematic construction approach for neural forms.

Lagaris et al. expanded their work in *Neural-network methods for boundary value problems with irregular boundaries* [14] with the focus on boundary value problems and irregular boundaries. They discuss that one may consider a neural form consisting of a RBF added to a FFNN. The latter takes on the structure of the differential equation, while the RBF satisfies given boundary conditions. The authors also suggest to consider a penalty approach, by simply setting the neural form to be a FFNN output and adding the boundary conditions as ℓ_2-norms with a penalty parameter to the cost function.

The neural forms approach has been adopted many times in various applications. In *Chebychev neural network based model for solving Lane-Emden type equations* [57], Mall et al. employed the neural forms approach together with a RBF that expands the input data to Chebychev polynomials in the hidden layer. The hidden layer weights represent the polynomial coefficients. The number of hidden layer neurons determines the order of the Chebychev polynomial. With the Chebychev neural network, Mall et al. construct a neural form and a cost function. The application to astrophysics and related second order non-linear ODEs together with polynomial orders up to five shows useful results with a considerably small numerical error. The authors also proposed *Application of Legendre neural network for solving ordinary differential equations* [58] to show the network construction principle with Legendre polynomials. Experiments on initial and boundary value problems as well as system of coupled ordinary differential equations are a part of the presentation and discussion.

Even two decades after neural forms [13] have been introduced by Lagaris et al., this approach still receives attention. A computational study by Famelis et al. in *Parameterized neural network training for the solution of a class of stiff initial value systems* [59] provides detailed results on the solution of systems of stiff IVPs. Their FFNN features one hidden layer with a small amount of neurons involved. However, the FFNN has a special characteristic in the input layer. Besides the input data from the computational domain, the stiffness parameter is also passed into the neural network. The numerical accuracy between the neural forms approach and an embedded MATLAB numerical solver is discussed for different activation functions, various amounts of hidden layer neurons and stiffness parameter values.

A related study by Schneidereit et al. in *Solving ordinary differential equations using artificial neural networks - a study on the solution variance* [60] lays the focus on a simple stiff IVP, solved by a single hidden layer FFNN with a small amount of sigmoid neurons. The paper investigates the impact of various computational parameters on the results. Especially the experiments on random weight initialisation with different optimisation methods revealed a random stochastic characteristic regarding the results with random weight initialisation. Uncarefully chosen, the optimisation methods may not find a useful (local) minimum in the weight space or even the suitable minima may immensely differ.

Another neural forms related work on initial value problems was proposed by Flamant et al. in *Solving differential equations using neural network solution bundles* [61]. Their method is capable of training a neural network for various initial conditions, instead of necessarily retraining for a new initial condition. In order to achieve this goal, a fairly large neural network is employed and trained with Adam optimisation to find useful results.

In *Polynomial neural forms using feedforward neural networks for solving differential equations* [62], Schneidereit et al. extended the classic neural forms approach for IVPs [13] into polynomials with several FFNNs, taking the role of polynomial coefficients. This extension improves the use of deterministic weight initialisation and removes the stochastic characteristic on the results, which comes with random initialisation [60]. Therefore it is possible to exactly reproduce certain results, important for numerical analysis.

1.2.5 ANN approaches based on suitable cost functions

Alongside methods employing a neural form-based cost function construction, there are several approaches incorporating given initial or boundary conditions directly in a cost function as additional, supervised terms. In *Solving differential equations with neural networks: applications to the calculation of cosmological phase transitions* [15], Piscopo et al. reference to the construction difficulty of the neural forms approach from Lagaris et al. [13] and take on the approach, similar to [14], for setting the neural form to be equivalent to a FFNN output. The given initial or boundary conditions are directly added to the cost function in a supervised way and additionally learned by the neural network. Although this construction principle does not fulfill the above-mentioned conditions exactly, the network itself is trained to represent the solutions without a specifically constructed neural form. In order to prove the effectiveness of this approach, results on examples of both ODEs, PDEs and coupled DEs are presented. Finally, the

authors apply the method to an even more complex topic, namely the computation of phase transitions in the early stage of the universe.

A similar approach regarding the cost function construction was proposed by Shirvany et al. in *Multilayer perceptron neural networks with novel unsupervised training method for numerical solution of the partial differential equations* [63]. The non-linear Schrödinger equation is treated as an example. An algorithm is proposed by the authors to solve the arising eigenvalue problem in an iterative way. The incorporated network is allowed to grow under predefined conditions.

1.2.6 DNN approaches for differential equations

The physics-informed neural networks are introduced by Raissi et al. in *Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations* [64]. This DNN framework has already been applied and modified for a variety of physics related problems. The approach comprehends unsupervised terms in the cost function connected to the DE structure and supervised terms for the given conditions. The DE-related terms are trained by collocation points from the domain, whereas the mentioned method inherent both a continuous time and a discrete time model. The application to various physics-related DEs shows the usefulness of the approach. Jagtap et al. proposed an extension in *Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems* [65] to combine the previous described approach with domain decomposition, where the domain is separated into subdomains using individual neural networks to find a solution in each subdomain. That enables parallel computation and the solution is later stitched together.

Chen et al. introduced a new type of DNN approach in *Neural ordinary differential equations* [66]. They employ deep residual and recurrent neural networks and find the hidden state update to be similar to an Euler discretisation of an ODE. The hidden state is then, in the limit of more layers and smaller steps, parameterised by an ordinary differential equation, incorporating the neural network parameters. The neural network can then be solved with a black box ODE solver.

Continuing with deep neural networks and partial differential equations, an important work was published by Ruthotto et al. in *Deep neural networks motivated by partial differential equations* [67]. The authors take on deep convolutional neural neural and interpret these as a discretisation of a PDE in space and time. In order to address the network type related challenges like designing suitable architectures for learning tasks and analysing the network stability, three new architectures are presented: parabolic, hyperbolic and second-order hyperbolic convolutional neural networks. Each network type comes with specific and useful characteristics. The networks are tested for image recognition and provide competitive results.

Another composition of deep learning and differential equations are PDE-Nets, published by Long et al. in *PDE-Net: learning PDEs from data* [68]. The authors aim to find PDE models of complex systems with a supervised learning approach, incorporating multiple convolutional neural network layers combined in a so-called Network-In-Network architecture [69]. A function is constructed incorporating convolutional operations on the input data, approximating the differential operators. The PDE-Net output is then used to obtain a ℓ_2-norm cost function. The authors also state that with

some knowledge of the differential equations structure, the function of convolutional networks may be simpler to train and this can improve the results. In the experimental section, the method is applied to several PDEs and the results prove the usefulness of the proposed approach.

1.2.7 ANNs combined with numerical methods

Solving differential equation in a non-analytic way may always bring numerical methods onto the scene [1, 2]. Therefore it arises naturally to consider combining numerical methods with neural network approaches. In 1998, the same year as Lagaris et al. proposed the neural forms approach [13], Wang et al. published their work on *Runge-Kutta neural network for identification of dynamical systems in high accuracy* [70]. They proposed a neural network architecture which is based on Runge-Kutta methods and applied their approach to non-linear IVPs. That is, the Runge-Kutta neural network replaces the right-hand side function while the relation between input and output still reads like a classic Runge-Kutta method. Incorporated numerical coefficients are represented by subnetworks in this approach. It is derived how backpropagation and non-linear recursive least-squares learning can be used to update the network weights. The latter has been studied in theory regarding its convergence property.

The fact that the neural network output computation resembles a linear combination of basis (activation) functions leads to a network architecture as presented by Rudd et al. in *A constrained integration (CINT) approach to solving partial differential equations using artificial neural networks* [71]. In this work, the incorporated ANN architecture is combined with Galerkin FEM [72]. The presented ANN features one hidden layer with two sets of activation functions, one of which is supposed to satisfy the PDE and the second dealing with boundary conditions. The basis function coefficients are set to be the connecting weights from the hidden layer to the output neuron, and the sum over all basis functions and coefficients makes up the neural form. The authors find the method to show a high convergence rate, accuracy and the ability to be applied to non-rectangular domains, which is verified by examples on PDEs.

A novel and presumably important work with the title *Connections between numerical algorithms for PDEs and neural networks* [73] has recently been published. Alt et al. investigate, e.g., implicit and explicit discretisation schemes and multi grid methods in order to connect those numerical approaches with different neural network architectures. With the resulting networks, the authors have performed computational experiments on image processing tasks like denoising and inpainting [74]. They are also able to show the effectiveness of incorporating non-monotone activation functions in their constructed neural architectures. U-net architectures [75] have recently become popular as a deep learning application in, e.g., biomedical detection. They perform down sampling and up sampling several times on the input data in combination with a processing through incorporated convolutional neural networks. Alt et al. show how to convert a multi grid method into a U-net architecture and find, that convolutional neural networks and numerical methods for diffusion evolution have structural characteristics in common.

1.2.8 Mesh refinement strategies using ANNs

Several works offer neural network based strategies and approaches to generate optimal meshes or mesh refinements for use with the finite element method [76, 77]. However these approaches do not combine neural solution of DEs with mesh adaptivity, and also they stick to a traditional mesh and do not explore the approximation capability of neural networks. Predicting areas which are of interest in the sense of a required mesh refinement using neural networks is the objective of Manevitz et al. in *Neural network time series forecasting of finite-element mesh adaptation* [78]. Their time-series analysis is employed to predict element-wise solution gradient values. The participating neural network yields an indicator based on local gradient values in space and time. This indicator is then used to predict whether a mesh refinement or a coarsening may be suitable. While in this method the mesh refinement indicator is realised by a neural network, the FEM is used for solving the PDE examples. Complementary to the latter approach, in *Fuzzy numerical schemes for hyperbolic differential equations* [79] Breuß et al. developed a learning strategy is which keeps the mesh fixed but selects the numerical scheme that gives locally high accuracy based on local gradients.

An adaptive neural approach has been proposed by Anitescu et al. in *Artificial neural network methods for the solution of second order boundary value problems* [80]. It features a FFNN in a framework combining both supervised and unsupervised terms, similar to [15, 65]. The training process includes several evolution steps, each consisting of the optimisation over the training points combined with an evaluation of results at a finer grid. The latter is realised with the same set of neural network parameters obtained from the training step. It is proposed to start with a coarse grid and to perform local grid refinement whenever the resulting network errors differ. The method is developed for boundary value problems arising with stationary PDEs, like e.g. the Poisson equation. Results indicate that more complex neural network architectures (w.r.t. number of layers and neurons) or more training points may increase the accuracy.

2 Computational characteristics of neural forms approaches for solving initial value problems

The sections starts with an introduction to the basic concepts of feedforward neural networks, the neural forms based trial solution method (TSM) and the modified trial solution method (mTSM). In addition, it is shown how the training and optimisation with backpropagation, Adam and BFGS works. Furthermore, important and detailed information about the weight initialisation of neural networks, as well as the measurement metrics for the later discussed computational results are given.

2.1 The feedforward neural network

In order to visualise neural networks, the neurons are usually depicted as circles and the connecting weights as lines. The now described architecture is not fixed by any means and only serves as an explicit example, since similar structures are used in this work. In the corresponding experimental sections, the involved configurations are always carefully addressed. Fig. 1 shows a basic neural network architecture with a

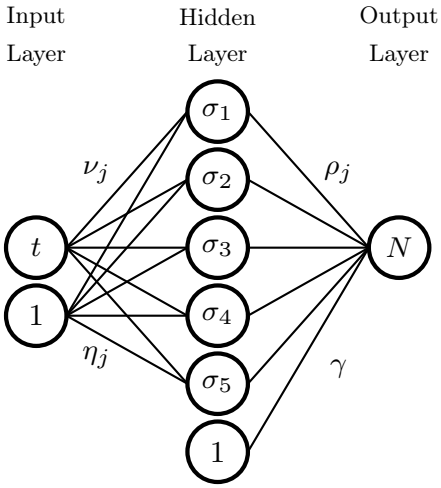


Figure 1: Feedforward neural network with five hidden layer neurons.

total of three layers. There are two kinds of neurons incorporated: processing ones and bias neurons. The latter are assigned with the value one, which is always constant. On the other side, the processing neurons apply an activation function to the incoming information. That is, the processing neuron in the input layer receives (later discretised) time data

$$t \in D \subset \mathbb{R} \tag{1}$$

where D denotes the solution domain. Here, a linear function is applied so that the data gets passed right through the neuron. There is an additional bias neuron incorporated in the input layer, in order to increase the flexibility. Now, both neurons are connected with every single neuron in the hidden layer by weights (adjustable parameters). Therefore, the input layer can pass the domain data t, weighted by ν_j and η_j,

$$z_j = \nu_j t + \eta_j, \quad j = 1, \ldots, 5 \tag{2}$$

to the hidden layer for processing. Each hidden layer neuron applies a non-linear activation function to the incoming data. The activation function used in this work is the sigmoid function

$$\sigma_j = \sigma(z_j) = \frac{1}{1 + e^{-z_j}} \tag{3}$$

which is a continuous and arbitrarily often differentiable function with values between zero and one, cf. Fig. 2. In addition to the five processing neurons in the hidden layer,

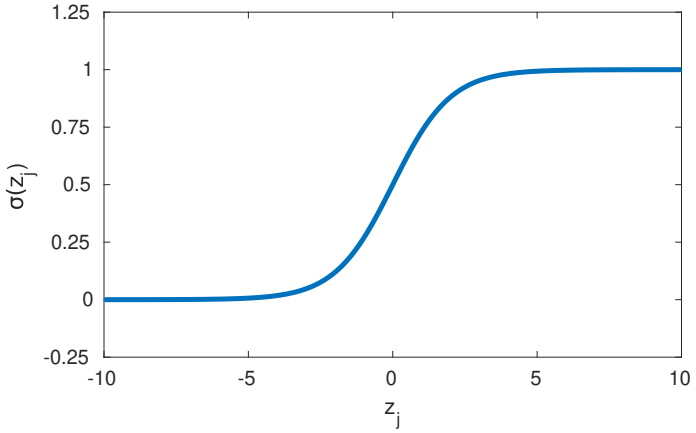


Figure 2: Graph of the sigmoid activation function with values between zero and one.

one bias is added to serve the same purpose as in the input layer. After the data has been processed through all hidden layer neurons, the single output layer neuron receives itself a sum, weighted by $\rho_j \sigma_j$ together with γ, as well. Since this neuron is linear, the neural network output assembles as

$$N(t, \mathbf{p}) = \sum_{j=1}^{5} \rho_j \sigma(z_j) + \gamma \tag{4}$$

where all weights are stored in the weight vector

$$\mathbf{p} = \left(\nu_1, \ldots, \nu_5, \eta_1, \ldots, \eta_5, \rho_1, \ldots, \rho_5, \gamma\right)^T \tag{5}$$

The weights are usually assigned with random values [84]. In case of additional hidden layers, Eq. (4) would act as the weighted sum for the first neuron in the second hidden layer and a non-linear activation function may be applied again. Additional weights would be incorporated as well in case of more hidden layers. Eq. (4) can now be used to construct the cost function $E[\mathbf{p}]$, which is then used in to train the neural network. Training in this context means learning the underlying structure in general, from given training points. Connected to this work, the training points are given by grid points from the discrete time domain arising from Eq. (1) by discretising it into $n + 1$ (equidistant) grid points $t_i, (t_0 < t_1 < \ldots < t_n)$. With the training data, an optimisation method is used to minimise the cost function $E[\mathbf{p}]$ and repeatedly update the weights $\mathbf{p}$ in order to find a suitable minimum. Each of these iterative steps is called an epoch of learning. For supervised learning, where both input data t_i and corresponding correct output data d_i, $i = 1, \ldots, n$, are given, the cost function may be

chosen as the squared ℓ_2-norm

$$E[\mathbf{p}] = \frac{1}{2(n+1)} \Big\| N(t_i, \mathbf{p}) - d_i \Big\|_2^2 := \frac{1}{2(n+1)} \sum_{i=0}^{n} \Big\{ N(t_i, \mathbf{p}) - d_i \Big\}^2 \tag{6}$$

The ℓ_2-norm applies to vectors and therefore the corresponding (middle) expression in Eq. (6) may identify as the vector of corresponding entries (grid points t_i). However, the notation on the right-hand side is followed in this work. Now in the case of unsupervised learning, where no correct output data is known, the cost function is part of the modelling process.

2.2 The neural forms approach

After a formulation of the general problem, the TSM [13] and mTSM [14, 15] solution approaches, and how to obtain their neural forms, are described in detail in this section. The construction for ODEs of order two and above follows the same principle, which also holds for PDEs. However, the latter requires the incorporated neural network to feature additional input neurons, one for each dimension. It is also possible to solve higher order ordinary or partial differential equations, as well as systems of ODEs or PDEs, cf. [13, 15].

A general first order IVP may be given in form of

$$G\big(t, u(t), \dot{u}(t)\big) = 0, \quad t \in D \subset \mathbb{R}, \tag{7}$$

together with the corresponding initial condition

$$u(t_0) = u_0 \tag{8}$$

In Eq. (7), $u(t)$ denotes the analytical solution function with t as the independent variable, whereas $\dot{u}(t)$ identifies the time derivative.

2.2.1 Trial solution method (TSM)

The main idea behind this approach is the construction of a continuous differentiable function (trial solution/neural form) in order to approximate the original solution function $u(t)$ in Eq. (7). That is, the neural form (NF) must satisfy the given initial condition by construction. In order to embed this constraint, the NF in general appears as a sum of two terms

$$\tilde{u}(t, \mathbf{p}) = A(t) + F(t, \mathbf{p}). \tag{9}$$

In Eq. (9), the term $A(t)$ is supposed to satisfy the initial condition $u(t_0)$ at the initial point t_0. The simplest choice here is to set $A(t) = u(t_0)$. Meanwhile, the neural network-related term $F(t, \mathbf{p})$ is constructed to eliminate the impact of $N(t, \mathbf{p})$ at the initial point t_0. The choice of $F(t, \mathbf{p})$ determines the influence of $N(t, \mathbf{p})$ over the domain. For example, the classic NF for IVPs from [13] reads

$$\tilde{u}(t, \mathbf{p}) = u(t_0) + N(t, \mathbf{p})(t - t_0). \tag{10}$$

Please note that the NF can be defined in many possible ways for the same IVP, since the only constraint per definition is to satisfy the given initial condition [56], besides having the neural network-related term.

The NF in Eq. (9) transforms the IVP from Eq. (7) into

$$G\big(t, \tilde{u}(t, \mathbf{p}), \dot{\tilde{u}}(t, \mathbf{p})\big) = 0 \tag{11}$$

Now G incorporates the NF and its time derivative. The latter can be found analytically for the neural form in Eq. (10):

$$\dot{\tilde{u}}(t, \mathbf{p}) = \dot{N}(t, \mathbf{p})(t - t_0) + N(t, \mathbf{p}) \tag{12}$$

The neural network time derivative of $N(t, \mathbf{p})$ is derived from Eq. (4) and reads

$$\dot{N}(t, \mathbf{p}) = \sum_{j=1}^{5} \rho_j \nu_j \sigma'(z_j)$$

The derivative of the sigmoid function w.r.t. its argument can be expressed using the sigmoid function itself:

$$\sigma'(z_j) = \sigma(z_j)(1 - \sigma(z_j)) \tag{13}$$

As mentioned in Section 2.1, the input data for the neural network is obtained by discretising the time domain in Eq. (1) into $n+1$ grid points $t_i, i = 0, \ldots, n$. With the discrete domain and the given initial condition incorporated, Eq. (11) is now solved by an unconstrained optimisation problem using the cost function

$$E[\mathbf{p}] = \frac{1}{2(n+1)} \sum_{i=0}^{n} \left\{ G\big(t_i, \tilde{u}(t_i, \mathbf{p}), \dot{\tilde{u}}(t_i, \mathbf{p})\big) \right\}^2 \tag{14}$$

Solving differential equations of order d with the neural forms approach, one may consider to choose an activation function, which is at least $(d+1)$ times continuously differentiable, since the solution approaches require the d-th activation function derivative and the later introduced gradient-based optimisation methods require another differentiation.

Turning to systems of IVPs, where G (cf. Eq. (7)) represents not a single IVP, but a system of o IVPs:

$$G\big(t, \mathbf{U}(t), \dot{\mathbf{U}}(t)\big) = \mathbf{0}, \quad t \in D \subset \mathbb{R}, \tag{15}$$

In Eq. (15),

$$\mathbf{U}(t) = \big(u^1(t), \ldots, u^o(t)\big)^T \tag{16}$$

represents the involved equations with. Superscript indices are used here, since the subscript index will later be used as well. The given initial values are denoted as

$$\mathbf{U}(t_0) = \big(u^1(t_0), \ldots, u^o(t_0)\big)^T \tag{17}$$

TSM now employs to set up one NF per equation with individual neural networks:

$$\begin{aligned} \tilde{u}^1(t, \mathbf{p}_1) &= u^1(t_0) + N^1(t, \mathbf{p}^1)(t - t_0) \\ &\vdots \\ \tilde{u}^o(t, \mathbf{p}_o) &= u^o(t_0) + N^o(t, \mathbf{p}^o)(t - t_0) \end{aligned} \tag{18}$$

Whereas the cost function results in the sum over all l_2-norm terms

$$E[\mathbf{p}] = \frac{1}{2(n+1)} \sum_{i=0}^{n} \left[\sum_{r=1}^{o} \left\{ G\big(t_i, \tilde{u}^r(t_i, \mathbf{p}^r), \dot{\tilde{u}}^r(t_i, \mathbf{p}^r)\big) \right\}^2 \right] \tag{19}$$

Possible similarities to numerical methods

This paragraph will briefly describe Runge-Kutta 4 and the Euler method [3, 4] without going too much into detail. The main purpose is to show a possible link between the neural forms related approaches and the numerical methods. Both numerical methods use discrete grid points $t_i, i = 0, \ldots, n$ from the solution domain in order to approximate an IVP in form of

$$\dot{u}(t) = f(u(t), t), \quad u(t_0) = u_0 \tag{20}$$

The analytical solution $u(t)$ can be approximated at each grid point by $u_i \approx u(t_i)$. Using the discrete time derivative

$$\dot{u}(t) \approx \frac{u_{i+1} - u_i}{h}, \quad h = t_{i+1} - t_i \tag{21}$$

a simple iterative scheme for Eq. (20) reads

$$u_{i+1} = u_i + hf(u_i, t_i), \tag{22}$$

which is the explicit Euler method. The initial value $u(t_0) = u_0$ is the starting point for iterating towards the approximate solution by evaluating $f(u(t), t)$ at one grid point in each step. The Euler method can also be referred to as a first order Runge-Kutta method. Evaluating $f(u(t), t)$ at several orders leads to Runge-Kutta methods of higher order. The most popular version is the fourth order, often named Runge-Kutta 4 (RK4). This method follows the iterative scheme

$$u_{i+1} = u_i + \frac{h}{6}\big(k_1 + 2k_2 + 2k_3 + k_4\big) \tag{23}$$

$$k_1 = f\big(t_i, u_i\big) \tag{24}$$

$$k_2 = f\left(t_i + \frac{h}{2}, u_i + \frac{k_1}{2}\right) \tag{25}$$

$$k_3 = f\left(t_i + \frac{h}{2}, u_i + \frac{k_2}{2}\right) \tag{26}$$

$$k_4 = f\big(t_i + h, u_i + k_3\big) \tag{27}$$

A quantitative comparison between the neural forms approaches and the numerical methods will be discussed later.

In general, comparing RK4 and the TSM neural forms approach on a qualitative or construction level is not an easy task. In order to find similarities between both methods, the usage of grid points and epochs is worth comparing. A common setup for TSM in this thesis features 10 grid points and 1e5 epochs. That is, in each epoch, the TSM cost function is evaluated at these 10 grid points. On the other hand, RK4 iterates over the grid points and evaluates the right-hand side of Eq. (20) at three different positions, namely $t_i, t_i + h/2, t_i + h$. Therefore, and although a comparison appears to be difficult, possible similarities in the structure can perhaps be found between TSM epochs/RK4 grid points and TSM grid points/RK4 evaluation steps.

Other numerical aspects, e.g., related to convergence or stability are not simple to derive. Perhaps, the cost function can play a significant role in finding equivalent measures such as the Courant–Friedrichs–Lewy condition for convergence of certain PDEs, the von Neumann stability analysis or the quality of approximation [6, 3]. However, these topics are not part of this thesis. Also not covered here are questions related to the solution of inverse problems. In contrast to forward problems, where, e.g., a simulation results from a mathematical model, inverse problems have a system outcome or behaviour given and aim to find the underlying model.

2.2.2 Modified trial solution method (mTSM)

An alternative approach completely replaces the construction by introducing the universal NF

$$\tilde{u}(t, \mathbf{p}) = N(t, \mathbf{p}) \tag{28}$$

The requirement of satisfying given initial conditions is dispensed. Although the construction of the TSM neural form for IVPs appears to be straight forward, this procedure can get very complicated for, e.g., higher order boundary value problems.

One of the main properties of a cost function approach in general, is the ability to add additional terms. This may change the energy landscape completely but enables the approach to come with great flexibility. Therefore, while $\tilde{u}(t, \mathbf{p})$ does not satisfy initial or boundary conditions per definition, they rather appear in the cost function as an additional term, regarding IVPs, namely

$$E[\mathbf{p}] = \frac{1}{2(n+1)} \left\{ G\big(t_i, \tilde{u}(t_i, \mathbf{p}), \dot{\tilde{u}}(t_i, \mathbf{p})\big) \right\}^2 + \frac{1}{2} \left\{ \tilde{u}(t_0, \mathbf{p}) - u(t_0) \right\}^2 \tag{29}$$

whereas for higher order IVPs or boundary value problems, the additional initial or boundary conditions are added as extra terms to the cost function w.r.t. the corresponding domain points. Therefore, the mTSM approach represents a joint framework with an unsupervised term, regarding the IVP structure G in Eq. (29), and a supervised term related to directly learning the initial condition.

2.3 Optimisation, initialisation and evaluation

The optimisation task is to find a (local) minimum in the weight space. This holds, when the squared ℓ_2-terms in the cost function become sufficiently small, and therefore it becomes sufficient to suggest that, cf. Eq. (6),

$$N(t_i, \mathbf{p}) \approx d_i, \quad i = 0, \ldots, n \tag{30}$$

Then the neural network $N(t_i, \mathbf{p})$ has learned the given data structure.

2.3.1 Backpropagation, Adam and BFGS

In order to achieve a final training state, several techniques focus on the cost function gradient $\nabla E[\mathbf{p}]$ with respect to the neural network weights. The gradient points out the direction of the greatest slope. This characteristic can be used to determine the impact of each weight on the output $N(t_i, \mathbf{p})$ to update them accordingly. The procedure is an

adaption of the gradient descent method and is called backpropagation [43]. In general, this method uses the gradient of a differentiable function to make iterative steps in the steepest direction and is therefore able to find a (local) minimum. Backpropagation adopts this principle for neural networks. After forward processing the input data, the difference between the neural network output and the desired results, cf. Eq. (6), defines as the so-called training error. This error is then propagated back through the neural network to adjust the weights in order to improve the results in the next processing. That is where the name backpropagation arises from.

The process of updating the neural network weights is called training. After one complete iteration over all input information, one epoch of training has finished. Effective training usually takes several epochs and results in finding a local minimum in the weight space. The backpropagation algorithm may be incorporated as an optimiser in the above-mentioned task. For epoch k, the backpropagation update rule [85] reads

$$\mathbf{p}(k+1) = \mathbf{p}(k) + \Delta\mathbf{p}(k) \tag{31}$$

where the adjusted weights $\mathbf{p}(k+1)$ are on the left-hand side and compute as the sum of the old weights $\mathbf{p}(k)$ and the current update $\Delta\mathbf{p}(k)$. Prior to specifying the latter term, the differential operator identifying the gradient w.r.t. the neural network weights in $\nabla E[\mathbf{p}]$ follows the notation

$$\nabla := \frac{\partial}{\partial \mathbf{p}} := \left(\frac{\partial}{\partial \rho_1}, \ldots, \frac{\partial}{\partial \rho_5}, \frac{\partial}{\partial \nu_1}, \ldots, \frac{\partial}{\partial \nu_5}, \frac{\partial}{\partial \eta_1}, \ldots, \frac{\partial}{\partial \eta_5}, \frac{\partial}{\partial \gamma} \right)^T \tag{32}$$

based on Eq. (5). For larger neural networks as in Fig. 1, the gradient features additional entries. The weight update term in Eq. (31) mainly depends on the negative cost function gradient. However, there is an additionally factor α involved:

$$\Delta\mathbf{p}(k) = -\alpha \frac{\partial}{\partial \mathbf{p}} E[\mathbf{p}(k)] \tag{33}$$

The parameter α is called learning rate or step size and scales the gradient. Prior to a more detailed description of the gradient computation, the backpropagation updating term can be improved by adding a momentum term [85]:

$$\Delta\mathbf{p}(k) = -\alpha \frac{\partial}{\partial \mathbf{p}} E[\mathbf{p}(k)] + \beta \Delta\mathbf{p}(k-1) \tag{34}$$

with the corresponding momentum parameter β. It uses impact from last epoch to reduce the chance of getting stuck in a shallow local minimum or saddle point. Furthermore, it may also speed up the training in certain declining areas. The complete backpropagation (BP) update rule now reads

$$\mathbf{p}(k+1) = \mathbf{p}(k) - \alpha \frac{\partial}{\partial \mathbf{p}} E[\mathbf{p}(k)] + \beta \Delta\mathbf{p}(k-1) \tag{35}$$

The learning rate α is either constant (cBP) or variable (vBP), which may prevent the optimiser from oscillating around a minimum. Selecting a suitable value can be a challenging task. A not carefully chosen learning rate can distract the optimisation method from landing in a minimum, or can get pushed out of a local minimum, if its

value is too high. On the other hand, with a too small learning rate incorporated, the training process may take disproportional long or does not even finish. There are different approaches for learning rate control [86], where

$$\alpha \;\rightarrow\; \alpha(k) \tag{36}$$

now depends on the epoch k. One approach employs a linear decreasing model

$$\alpha(k) = \begin{cases} \alpha_0 - \frac{\alpha_0 - \alpha_e}{k_c} k, & k \leq k_c \\ \alpha_e, & k > k_c \end{cases} \tag{37}$$

with an initial learning rate α_0, a final learning rate α_e and an epoch cap k_c. When the latter is reached, α switches to the constant learning rate α_e. The employed parameters in this work are $\alpha =$ 1e-3 with $\beta =$ 9e-1 for cBP and $\alpha_0 =$ 1e-2, $\alpha_e =$ 1e-3, $k_c =$ 1e4 with $\beta =$ 9e-1 for vBP.

Adam (adaptive moment estimation) [44] is an adaptive optimisation method based on cost function gradient as well. The explained procedure in the following follows the algorithm in [44]. Preliminary initialised are the moving averages of the gradient $\mathbf{m}(0) = 0$ and $\mathbf{v}(0) = 0$, as well as an initial learning rate α, the exponential decay rates for the moving averages β_1, β_2 and the constant ε. The optimiser uses the cost function gradient in the k-th epoch

$$\mathbf{g}(k+1) = \frac{\partial}{\partial \mathbf{p}} E[\mathbf{p}(k)] \tag{38}$$

to perform updates on

$$\mathbf{m}(k+1) = \beta_1 \mathbf{m}(k) + (1-\beta_1)\mathbf{g}(k+1) \tag{39}$$

$$\mathbf{v}(k+1) = \beta_2 \mathbf{v}(k) + (1-\beta_2)\mathbf{g}^2(k+1) \tag{40}$$

The squared gradient in Eq. (40) represents the vector of the squared gradient elements. Every operation in the algorithm in [44] is performed element by element of the vectors. As mentioned earlier, $\mathbf{m}(k+1), \mathbf{v}(k+1)$ represent the moving averages of the gradient. More specific, $\mathbf{m}(k+1)$ is the estimate of the 1$^{\text{st}}$ moment (mean) and $\mathbf{v}(k+1)$ the 2$^{\text{nd}}$ raw moment (uncentered variance) of the gradient. The corresponding parameters β_1 and β_2 are the exponential decay rates. While $\mathbf{m}(0)$ and $\mathbf{v}(0)$ are initialised with zeros, the moment estimates are biased towards the origin. That leads to a necessary bias-correction with

$$\hat{\mathbf{m}}(k+1) = \frac{\mathbf{m}(k+1)}{1-\beta_1^{k+1}} \tag{41}$$

$$\hat{\mathbf{v}}(k+1) = \frac{\mathbf{v}(k+1)}{1-\beta_2^{k+1}} \tag{42}$$

The Adam update rule then reads

$$\mathbf{p}(k+1) = \mathbf{p}(k) - \alpha \frac{\hat{\mathbf{m}}(k+1)}{\sqrt{\hat{\mathbf{v}}(k+1)} + \varepsilon} \tag{43}$$

where ε is a small positive and additive constant, to prevent the denominator from becoming zero. The standard parameter for Adam [44] are obtained using α = 1e-3, β_1 = 9e-1, β_2 = 9.99e-1 and ϵ = 1e-8.

Both backpropagation and Adam make use of the cost function gradient and are therefore called first order optimisation methods. Another approach uses the curvature (second derivatives/Hessian matrix) of the cost function. The BFGS (Broyden-Fletcher-Goldfarb-Shanno) approach, a Quasi-Newton method [45], is a popular second order optimisation method, approximating the inverse of the Hessian matrix. [87] The general idea of the Newton method is to approximate the cost function $E[\mathbf{p}]$ with a Taylor-series of order two around a current iterate $\mathbf{p}(k)$. The derivative of this series has to be zero as a necessary condition for finding a minimum. In addition, the resulting Hessian matrix must be positive definite. The iterative scheme then reads

$$\mathbf{p}(k+1) = \mathbf{p}(k) - H^{-1}(\mathbf{p}(k))\nabla E[\mathbf{p}(k)] \tag{44}$$

While the Newton method now requires the computation of the inverse Hessian $H^{-1}(\mathbf{p}(k))$, the Quasi-Newton method replaces it with the approximation $H^{-1}(\mathbf{p}(k)) = \alpha(k)B(\mathbf{p}(k))$. Here, $\alpha(k)$ is the step size which can be determined using, e.g., backtracking or Wolfe-Powell line search. The update of $B(\mathbf{p}(k))$, according to the BFGS method reads

$$B(\mathbf{p}(k+1)) = \left(I - \frac{\mathbf{s}\mathbf{y}^T}{\mathbf{y}^T\mathbf{s}}\right)B(\mathbf{p}(k))\left(I - \frac{\mathbf{y}\mathbf{s}^T}{\mathbf{y}^T\mathbf{s}} + \frac{\mathbf{s}\mathbf{s}^T}{\mathbf{y}^T\mathbf{s}}\right) \tag{45}$$

where I denotes the identity matrix and

$$\mathbf{s} = \mathbf{p}(k+1) - \mathbf{p}(k), \quad \mathbf{y} = \nabla E[\mathbf{p}(k+1)] - \nabla E[\mathbf{p}(k)] \tag{46}$$

together with

$$\mathbf{p}(k+1) = \mathbf{p}(k) - \alpha(k)\mathbf{g}(k), \quad \mathbf{g}(\mathbf{p}(k)) = -B(\mathbf{p}(k))\nabla E[\mathbf{p}(k)] \tag{47}$$

The equation for $\mathbf{p}(k+1)$ in Eq. (47) also denotes the neural network weight update in this approach. The Hessian approximate $B(\mathbf{p})$ in Eq. (45) can be initially set to equal the identity matrix or a multiple of the identity matrix [87].

As stated above, the step size $\alpha(k)$ can be found using Wolfe-Powell line search. The idea is to use the search direction $\mathbf{g}(k)$ (with has to be a descent direction, wherefore $\mathbf{g}^T(k)\nabla E[\mathbf{p}k] < 0$ has to hold) in Eq. (47) in order to find $\alpha(k)$, so that

1. sufficient decrease condition:

$$E[\mathbf{p}(k) + \alpha\mathbf{g}(k)] \leq E[\mathbf{p}(k)] + \tau\alpha\nabla E[\mathbf{p}(k)]^T\mathbf{g} \tag{48}$$

2. curvature condition:

$$\nabla E[\mathbf{p}(k) + \alpha\mathbf{g}(k)]^T\mathbf{g}(k) \geq \sigma\nabla E[\mathbf{p}(k)]^T\mathbf{g}(k) \tag{49}$$

are satisfied. In contrast to Adam and backpropagation, which are straight forward to implement, the line search requires a separate algorithm with numerical difficulties [87]. The following algorithm (one possible realisation) requires the weight vector $\mathbf{p} \in \mathbb{R}^n$, the search direction $\mathbf{g} \in \mathbb{R}^n$ with $\nabla E[\mathbf{p}]^T\mathbf{g} < 0$, an initial step size $\alpha = 1$ and parameters $\tau \in (0, 0.5), \sigma \in [\tau, 1)$. Prior, the weight vector $\mathbf{p}(0)$ and the inverse Hessian approximate $B(0) = I$ are initialised. A possible realisation of the algorithm [87, 88, 89] to find the step size $\alpha(k) = \alpha_{WP}$ and to update inverse Hessian approximate reads (initially set $k = -1$):

1. Set $k = k + 1$ and find search direction $\mathbf{g}(\mathbf{p}(k)) = -B(\mathbf{p}(k))\nabla E[\mathbf{p}(k)]$
2. Perform Wolfe-Powell line search to obtain a suitable step size α_{WP}:
   - Set $\alpha(k) = 1$
     - **if** Eq. (48) holds, find the smallest number $b \in \{2^1, 2^2, 2^3, \ldots\}$ for which (with $\alpha(k) = b$) Eq. (48) is not satisfied and set $a = 0.5b$.
     - **else** find the largest number $a \in \{2^{-1}, 2^{-2}, 2^{-3}, \ldots\}$ for which (with $\alpha(k) = a$) Eq. (48) is satisfied and set $b = 2a$.
   - **repeat** with $\alpha(k) = a$
     - **if** Eq. (49) holds, **return** $\alpha_{WP} = \alpha(k)$ and **stop**.
     - **else** set $\alpha(k) = 0.5(a + b)$
       - **if** Eq. (48) holds for $\alpha(k) = 0.5(a + b)$, set $a = \alpha(k)$
       - **else** $b = \alpha(k)$
3. Set $\mathbf{s}(k) = -\alpha_{WP}\mathbf{g}(k)$ and update $\mathbf{p}(k + 1) = \mathbf{p}(k) + \mathbf{s}(k)$
4. Set $\mathbf{y} = \nabla E[\mathbf{p}(k + 1)] - \nabla E[\mathbf{p}(k)]$
5. Update
$$B(\mathbf{p}(k+1)) = \left(I - \frac{\mathbf{s}\mathbf{y}^T}{\mathbf{y}^T\mathbf{s}}\right) B(\mathbf{p}(k)) \left(I - \frac{\mathbf{y}\mathbf{s}^T}{\mathbf{y}^T\mathbf{s}} + \frac{\mathbf{s}\mathbf{s}^T}{\mathbf{y}^T\mathbf{s}}\right) \tag{50}$$
6. Go to 1. until a stopping condition is satisfied.

Stopping conditions may include the gradient or the improvement to the cost function to be close to zero.

At this point it is important to introduce a training method called batch training, related to processing the training points. In one epoch, obtaining the gradient individually after the computation of the cost function for each training point may result in $n + 1$ weight updates in that epoch. From now on, this training approach will be referred to as single batch training. An alternative procedure employs to perform the weight update after a complete iteration over all grid points, averaging the cost function gradient and training error and is called full batch training in this work. One may also consider mini batch training, which employs training with pairs of two or more training samples at ones. The expressions used for each weight update for single and full batch training read:

- single batch training:
$$\text{training error:} \quad \frac{1}{2}E_i[\mathbf{p}(k)] \tag{51}$$
$$\text{gradient:} \quad \frac{1}{2}\frac{\partial}{\partial \mathbf{p}}E_i[\mathbf{p}(k)] \tag{52}$$
- full batch training:
$$\text{training error:} \quad \frac{1}{2(n+1)}\sum_{i=0}^{n} E_i[\mathbf{p}(k)] \tag{53}$$
$$\text{gradient:} \quad \frac{1}{2(n+1)}\sum_{i=0}^{n} \frac{\partial}{\partial \mathbf{p}}E_i[\mathbf{p}(k)] \tag{54}$$

With this notation, $E_i[\mathbf{p}(k)]$ represents the cost function term, regarding the i-th grid point in the k-th epoch. However, to not affect the readability in an unpleasant way, the k-th epoch is not explicitly addressed from now on.

The gradient w.r.t. the neural network weights in Eq. (35) effectively operates on the neural network itself. Therefore it is possible to retrieve the analytical expressions, with the weighted sum $z_j = \nu_j t_i + \eta_j, j = 1, \ldots, 5, i = 0, \ldots, n$, as follows:

$$N(t_i, \mathbf{p}) = \sum_{j=1}^{5} \rho_j \sigma(z_j) + \gamma, \qquad \dot{N}(t_i, \mathbf{p}) = \sum_{j=1}^{5} \rho_j \nu_j \sigma'(z_j) \tag{55}$$

$$\frac{\partial}{\partial \rho_j} N(t_i, \mathbf{p}) = \sigma(z_j), \qquad \frac{\partial}{\partial \rho_j} \dot{N}(t_i, \mathbf{p}) = \nu_j \sigma'(z_j) \tag{56}$$

$$\frac{\partial}{\partial \nu_j} N(t_i, \mathbf{p}) = \rho_j t_i \sigma'(z_j), \qquad \frac{\partial}{\partial \nu_j} \dot{N}(t_i, \mathbf{p}) = \rho_j \sigma'(z_j) + \rho_j \nu_j t_i \sigma''(z_j) \tag{57}$$

$$\frac{\partial}{\partial \eta_j} N(t_i, \mathbf{p}) = \rho_j \sigma'(z_j), \qquad \frac{\partial}{\partial \eta_j} \dot{N}(t_i, \mathbf{p}) = \rho_j \nu_j \sigma''(z_j) \tag{58}$$

$$\frac{\partial}{\partial \gamma} N(t_i, \mathbf{p}) = 1, \qquad \frac{\partial}{\partial \gamma} \dot{N}(t_i, \mathbf{p}) = 0 \tag{59}$$

Here, the neural network time derivative is also considered, since it appears in the NF approaches, described in Section 2.2. The first and second derivative of the sigmoid function are given as

$$\sigma'(z_j) = \sigma(z_j)(1 - \sigma(z_j)) \tag{60}$$

$$\sigma''(z_j) = 2\sigma^3(z_j) - 3\sigma^2(z_j) + \sigma(z_j) \tag{61}$$

2.3.2 Weight initialisation

Poorly chosen training or neural network parameters may lead the (first order) optimisation to find a local minimum that does not sufficiently solve the underlying problem. In addition, a very important task is to assign the neural network weights ($\nu_j, \eta_j, \rho_j, \gamma$ in Fig. 1) with carefully chosen initial values. However, this task may be challenging, since they determine the starting point in the weight space and may be close to a suitable minimum, or far away. In general, it is common to initialise the weights with random values [84, 90, 91]. However, also an initialisation with equal values for each weight, e.g., zeros, is possible. This variant will be named deterministic initialisation, while the random starting weights will additionally have the name non-deterministic initialisation attached. Both descriptions are more meaningful and the reason for choosing such specific names is related to the general outcome of one complete optimisation. In this thesis, the deterministic initialisation will often be referred to as $\mathbf{p}^{init}_{deter}$ and the non-deterministic or random initialisation as $\mathbf{p}^{init}_{rnd}$. Regarding the impact of the non-deterministic (or random) initial weights on the neural network optimisation, the very first computation of $N(t_i, \mathbf{p})$ (see Eq. (4)) in epoch one will return an arbitrary value. This value depends on the initially chosen random weights and therefore $N(t_i, \mathbf{p})$ differs each time another set of initial weights is chosen at the start of an optimisation cycle. Therefore, also the cost function differs, depending on the initial set of weights which causes the gradient to return an individual descent direction each time and the resulting weight updates (e.g., with backpropagation or Adam) as well.

In epoch two, the computation of the neural network output, the cost function and the weight updates start again with the previously updated weights. Therefore, all these values again depend on the initially chosen set of weights (in the epoch before). This hold through all epochs so that it is very unlikely for two individual sets of initial weights (considering the other computational parameters to remain unchanged) to end up in the exact same location of a local minimum. Resulting in different (but maybe very similar) solutions to the optimisation problem and the differential equation. This characteristic can possibly be handled using multi-start stochastic global optimisation [92, 93, 94] for finding the best set of weights. However, neural networks approaches, especially in deep learning with large numbers of hidden layers and neurons, may experience increasing computational effort using such approaches.

Another option is to choose a deterministic initialisation. Setting each neural network weight to, e.g., zero, will result in identical computations regarding the cost function and weight updates for several optimisation cycles (several times restarting the optimisation). This approach will always find the exact same local minimum which can be beneficial for an in depth computational numerical analysis or detailed comparison to other methods. The solution stability is a major advantage of the deterministic initialisation with equal weights. However, the neurons in the hidden layer will act as there is only one hidden layer neuron incorporated. This is expressed by the fact, that all ν_j in Fig. 1 have the exact same values in each epoch during one optimisation cycle. The same holds for η_j and ρ_j in Fig. 1. In other words, the weight vector $\mathbf{p}$ is most likely individual in different epochs, but all ν_j have the same value, all η_j have the same value and all ρ_j have the same value in each epoch.

For further details on the optimisation, the following IVP is considered:

$$\dot{u}(t) = t\sin(10t) - u(t), \quad u(0) = -1 \tag{62}$$

for which G in Eq. (14), involved in the cost function, is found as

$$G = \dot{\tilde{u}}(t_i, \mathbf{p}) + \tilde{u}(t_i, \mathbf{p}) - t_i \sin(10t_i) \approx 0 \tag{63}$$

The minimisation of Eq. (14) aims to get G in Eq. (63) as close to zero as possible. That is, the expression of interest actually reads, cf. Eq. (30),

$$\dot{\tilde{u}}(t_i, \mathbf{p}) + \tilde{u}(t_i, \mathbf{p}) \approx t_i \sin(10t_i) \tag{64}$$

Here, the values of $t_i \sin(10t_i)$ are predetermined by the domain, whereas the NF and its time derivative additionally depend on the neural network weights and their optimisation. Hence, Eq. (64) can be considered as satisfied for various combinations of $\dot{\tilde{u}}(t_i, \mathbf{p}) + \tilde{u}(t_i, \mathbf{p})$. Therefore, the results may highly depend on the final location in the weight space.

One reason for this circumstance may relate to the complexity of the energy landscape which can inherent multiple local minima that can lead to several combinations of the left-hand side in Eq. (64). Not all of these combinations must be real or useful solutions. This issue may occur, e.g., when the initial weights are far away from a suitable minimum for a helpful approximation. When there is a minimum nearby the initialisation with unfavourable optimisation parameters, such that the optimiser can get stuck inside. However, fine tuning all the incorporated computational parameters is an ungrateful task since some of these are not independent of each other [81, 60].

2.3.3 Evaluation metrics and overfitting

In order to classify the solution of the optimisation problem and the approximation of the differential equation, this paragraph introduces the metrics for evaluation and verification. On the numerical side, the solution can be evaluated regarding its accuracy with a comparison between the analytical and the neural forms solution. Therefore, the averaged l_1-norm (computed in each epoch individually)

$$\Delta u_i = \big|u(t_i) - \tilde{u}(t_i, \mathbf{p})\big|, \quad \Delta u = \frac{1}{n+1} \sum_{i=0}^{n} \Delta u_i \tag{65}$$

is a fundamental evaluation metric in this thesis, as it averages the absolute values of the differences between the analytical solution $u(t_i)$ to the neural forms solution $\tilde{u}(t_i, \mathbf{p})$ at each of the $n+1$ grid points t_i. Another approach may use the expected value over the initial weights here. In addition, it is often useful to consider the l_∞-norm, which returns the maximum value

$$\Delta u_\infty = \max_i(\Delta u_i) = \max(\Delta u_0, \ldots, \Delta u_n) \tag{66}$$

The metrics in Eqs. (65) and (66) contribute to the numerical evaluation and are also referred to as l_1-error and l_∞-error. On the neural network and optimisation side, the grid points heavily support the training process as the cost function is build upon them. Therefore, it appears natural to evaluate the training process at the training points so that an error metric can be defined as

$$E[\mathbf{p}] = \frac{1}{2(n+1)} \sum_{i=0}^{n} \left\{ G\big(t_i, \tilde{u}(t_i, \mathbf{p}), \dot{\tilde{u}}(t_i, \mathbf{p})\big) \right\}^2 \tag{67}$$

Since Eq. (67) is not the only way to express and compute a cost function related metric, the training error is later often specified by $E[\mathbf{p}] \equiv E^{TP}[\mathbf{p}]$

$$E^{TP}[\mathbf{p}] = \frac{1}{2(n_{TP}+1)} \sum_{i=0}^{n_{TP}} \left\{ G\big(t_i, \tilde{u}(t_i, \mathbf{p}), \dot{\tilde{u}}(t_i, \mathbf{p})\big) \right\}^2 \tag{68}$$

In other words, whenever $E[\mathbf{p}]$ appears, it is related to the computation and evaluation at the grid or training points $n = n_{TP}$ (TP for training points) and therefore equivalent to $E^{TP}[\mathbf{p}]$ (assuming that the amount of grid points for training and evaluation is the same). On the other hand, one of the main features which comes with neural network approaches is the ability to generalise functions or solutions. Once (successfully) trained on a set of grid points, the neural forms are able to find the solution of the differential equations in between the training points. These intermediate grid points can be arbitrarily distributed over the solution domain. The evaluation (or verification) with intermediate grid points $n = n_{VP}$ (VP for verification points)

$$E^{VP}[\mathbf{p}] = \frac{1}{2(n_{VP}+1)} \sum_{i=0}^{n_{VP}} \left\{ G\big(t_i, \tilde{u}(t_i, \mathbf{p}), \dot{\tilde{u}}(t_i, \mathbf{p})\big) \right\}^2 \tag{69}$$

is important to detect a phenomenon named overfitting [95, 96, 97]. A trained neural network shows overfitting, when the training points (or the training data in general)

are too well approximated. This is the case when the training error E^{TP} is sufficiently small, but the verification error E^{VP} returns a much larger value. This resembles a major problem because overfitting highly impacts the generalisation performance in a bad way. Therefore it is important to investigate and avoid overfitting.

2.4 Computational results for TSM and mTSM

The reliability and accuracy of the approximation still represent not fully resolved issues in the current literature. Computational approaches are in general highly dependent on a variety of computational parameters and the amount of parameters related to the differential equation, neural network, optimisation and cost function structure is numerous. The contribution in this section is a study with the variation of (i) weight initialisation methods, (ii) number of hidden layer neurons, (iii) number of hidden layers, (iv) number of training epochs, (v) stiffness parameter and domain size, (vi) optimisation methods, (vii) cost function construction approaches, and especially their mutual dependence. Let us note that it has turned out to be a nontrivial task to set up a meaningful proceeding that gives an account of the latter aspect. We consider the evaluation presented here as a number of carefully chosen experiments that are in many aspects related to each other. The intention of this section is to make a step towards resolving the previously mentioned open issues. To this end we study here the solution of a simple but fundamental stiff ordinary differential equation modelling a damped system. We consider two computational approaches for solving differential equations by neural forms with first order optimisation. These are the classic but still present method of trial solutions (or neural forms) defining the cost function, and a direct construction of the cost function related to the trial solution method. By a detailed computational study we show that it is somehow possible to identify preferable choices to be made for parameters and methods. We also illuminate some interesting effects that are observable in the neural network simulations. By doing this we illustrate the importance of a careful choice of the computational setup.

2.4.1 TSM construction example

The computational characteristics in this section are analysed using the test equation

$$G = \dot{u}(t) + 5u(t) = 0, \quad u(0) = 1, \quad t \in [0, 2] \tag{70}$$

which resembles a homogeneous first order ordinary differential equation, also named initial value problem (IVP) [98]. With the IVP in Eq. (70), we will demonstrate how to construct the classic neural form employed by Lagaris et al. [13], the cost function and the corresponding gradient used for backpropagation, Adam and BFGS. Here we also have the opportunity to compare all three optimisation methods, and on top, with numerical solution methods. The differential equation in Eq. (70) has the analytical solution $u(t) = e^{-5t}$ and represents a simple model for stiff phenomena involving a damping mechanism. This test equation is similar to the one later used one for the extensive computational experiments, where the constant coefficient in Eq. (70) is replaced by a more general parameter $\lambda \in \mathbb{R}$, $\lambda < 0$. This will be mentioned again at the corresponding text passage. The equation was chosen since it incorporates stiff

behaviour, which results for some numerical methods in a limitation of the chosen step size between two grid points in terms of stability. In other words, stiff behaviour may force a numerical methods to tremendously decrease the step size in regions where such a behaviour is not expected.

Turning to the construction principle as introduced in Section 2.2.1, the classic TSM neural form (NF) for the IVP in Eq. (70) can be considered as

$$\tilde{u}(t_i, \mathbf{p}) = 1 + N(t_i, \mathbf{p})t_i \tag{71}$$

It satisfies the initial condition $u(0) = 1$ and already incorporates the discretised domain with grid points $t_i, i = 0, \ldots, n$. The classic NF in Eq. (71) is set to replace $u(t)$ in Eq. (70), resulting in

$$G = \dot{\tilde{u}}(t_i, \mathbf{p}) + 5\tilde{u}(t_i, \mathbf{p}) \approx 0 \tag{72}$$

where the NF time derivative reads

$$\dot{\tilde{u}}(t_i, \mathbf{p}) = N(t_i, \mathbf{p}) + \dot{N}(t_i, \mathbf{p})t_i \tag{73}$$

Eq. (73) features the neural network $N(t_i, \mathbf{p})$ and its derivative $\dot{N}(t_i, \mathbf{p})$ w.r.t. the time domain data input. In this example, both read

$$N(t_i, \mathbf{p}) = \sum_{j=1}^{5} \rho_j \sigma(\nu_j t_i + \eta_j) \tag{74}$$

$$\dot{N}(t_i, \mathbf{p}) = \sum_{j=1}^{5} \nu_j \rho_j \sigma'(\nu_j t_i + \eta_j) \tag{75}$$

with the derivative of the sigmoid function

$$\sigma'(\nu_j t_i + \eta_j) = \sigma(\nu_j t_i + \eta_j)(1 - \sigma(\nu_j t_i + \eta_j)) \tag{76}$$

The neural network incorporates one hidden layer with five sigmoid neurons and an input layer for t_i and a unit reference (bias neuron), as displayed in Fig. 3. For training,

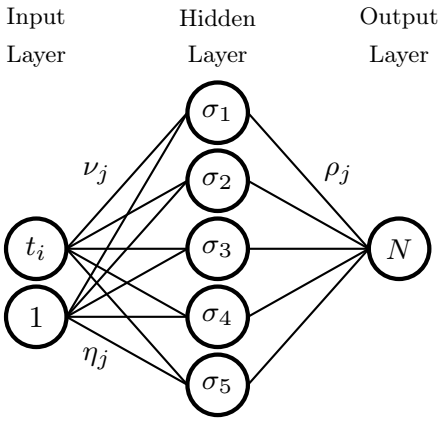


Figure 3: The neural network architecture as used in this section.

the discretised time domain with 10 equidistant grid points t_i is used: The resulting

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
0	0.22	0.44	0.66	0.88	1.11	1.33	1.55	1.77	2.00

Table 1: Training points.

cost function used for full batch training of the neural network follows Eq. (14):

$$E[\mathbf{p}] = \frac{1}{2(n+1)} \sum_{i=0}^{n} \left\{ \dot{\tilde{u}}(t_i, \mathbf{p}) + 5\tilde{u}(t_i, \mathbf{p}) \right\}^2 \tag{77}$$

$$E[\mathbf{p}] = \frac{1}{20} \sum_{i=0}^{9} \left\{ \dot{\tilde{u}}(t_i, \mathbf{p}) + 5\tilde{u}(t_i, \mathbf{p}) \right\}^2 \tag{78}$$

$$= \frac{1}{20} \sum_{i=0}^{9} \left\{ N(t_i, \mathbf{p}) + \dot{N}(t_i, \mathbf{p})t_i + 5(1 + N(t_i, \mathbf{p})t_i) \right\}^2 \tag{79}$$

$$= \frac{1}{20} \sum_{i=0}^{9} \left\{ \dot{N}(t_i, \mathbf{p})t_i + N(t_i, \mathbf{p})(1 + 5t_i) + 5 \right\}^2 \tag{80}$$

Let us note at this point, that the cost function in Eq. (77) is one of two general evaluation metrics in this paragraph. One evaluates the neural forms solution at the training points (TP) and is denoted by $E[\mathbf{p}] = E^{TP}[\mathbf{p}]$, while the over one is evaluated at so called verification points (VP) $E[\mathbf{p}] = E^{VP}[\mathbf{p}]$ (cf. Sec. 2.3.3). The other metric is related to the numerical error, which is defined as

$$\Delta u_1 = \frac{1}{n+1} \sum_{i=0}^{n} \left| u(t_i) - \tilde{u}(t_i, \mathbf{p}) \right| \tag{81}$$

the l_1-error at the corresponding grid points $\Delta u_1 = \Delta u_1^{TP}$ and $\Delta u_1 = \Delta u_1^{VP}$ (cf. Sec. 2.3.3). The next step is to minimise Eq. (77) with an optimisation method (see 2.3.1). For full batch training, cf. Eq. (53), one averages the gradients over all training points

$$\frac{\partial}{\partial \mathbf{p}} E[\mathbf{p}] = \frac{1}{2(n+1)} \sum_{i=0}^{n} \frac{\partial}{\partial \mathbf{p}} \left\{ \dot{N}(t_i, \mathbf{p})t_i + N(t_i, \mathbf{p})(1 + 5t_i) + 5 \right\}^2 \tag{82}$$

$$= \frac{1}{n+1} \sum_{i=0}^{n} e_i(\mathbf{p}) \frac{\partial}{\partial \mathbf{p}} \left\{ \dot{N}(t_i, \mathbf{p})t_i + N(t_i, \mathbf{p})(1 + 5t_i) + 5 \right\} \tag{83}$$

where

$$e_i(\mathbf{p}) = \dot{N}(t_i, \mathbf{p})t_i + N(t_i, \mathbf{p})(1 + 5t_i) + 5 \tag{84}$$

results from the quadratic function derivative. The corresponding neural network derivatives w.r.t. the network weights are given in Eq. (55)-(59). In Eq. (82), the differential operator-related partial derivatives w.r.t. $\mathbf{p}$ are applied to the neural net-

work and its derivative in Eqs. (74),(75). In this case, the cost function gradient reads

$$\frac{\partial E_i[\mathbf{p}]}{\partial \rho_j} = e_i(\mathbf{p})\big[\nu_j \sigma'(\nu_j t_i + \eta_j) t_i + \sigma(\nu_j t_i + \eta_j)(1 + 5t_i)\big] \tag{85}$$

$$\frac{\partial E_i[\mathbf{p}]}{\partial \nu_j} = e_i(\mathbf{p})\big[\nu_j \rho_j t_i^3 \sigma''(\nu_j t_i + \eta_j) + \rho_j t_i \sigma'(\nu_j t_i + \eta_j)(1 + 6t_i)\big] \tag{86}$$

$$\frac{\partial E_i[\mathbf{p}]}{\partial \eta_j} = e_i(\mathbf{p})\big[\nu_j \rho_j \sigma''(\nu_j t_i + \eta_j) t_i + \rho_j \sigma'(\nu_j t_i + \eta_j)(1 + 5t_i)\big] \tag{87}$$

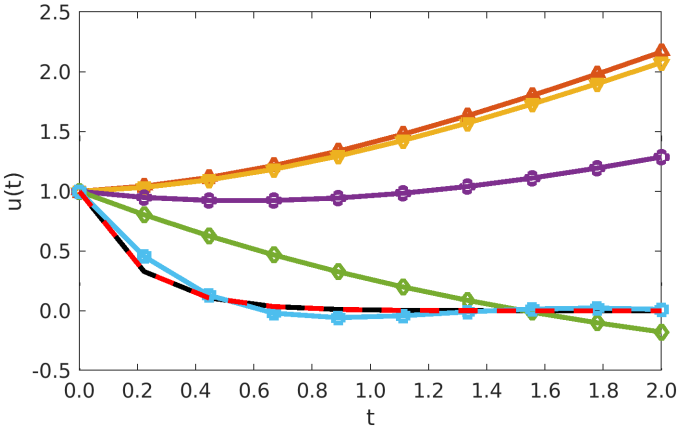
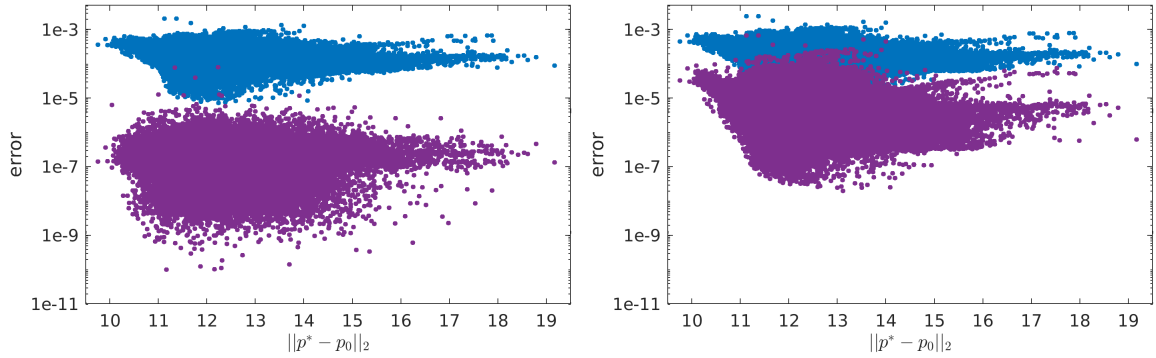


Figure 4: Evolution of approximating the IVP in Eq. (70) with a small neural networks, (black/solid) analytical solution, (orange/triangle (up)) 1e0 epoch, (yellow/triangle (down) 1e1 epochs, (purple/circle) 1e2 epochs, (green/diamond) 1e3 epochs, (blue/square) 1e4 epochs, (red/dash) 1e5 epochs.



(a) Errors at training points $n_{TP} = 10$ (b) Errors at verification points $n_{VP} = 100$

Figure 5: Resulting error values of non-deterministic (random) weight initialisation depending on the Euclidean distance between initial $\mathbf{p}_0$ and learned $\mathbf{p}^*$ weights, (blue) numerical error, (purple) cost function error.

As mentioned above, the training process for updating the neural network weights $\mathbf{p}$ is performed with full batch training, where the optimiser receives an averaged gradient over the entire training points. In order to obtain the NF solution to Eq. (70), the cost function in Eq. (77) is minimised using different optimisation methods. The optimisation parameters are given in Sec. 2.3.1 and the training is performed with 1e5 epochs

and 10 equidistant training points. The initial neural network weights are randomly chosen from $\mathbf{p}_{rnd}^{init} \in [-1, 1]$.

The diagram in Fig. 4 shows different NF solution stages during the optimisation process. Taking at least 1e3 epochs (green/diamond) to start recognising the shape of the analytical solution (black/solid) is remarkable. Then the transition from 1e3 epochs to 1e4 (blue/square) epochs is significant, while the values at some training points still lay below or above the desired solution function. Finally, after 1e5 epochs, the NF solution (red/dash) lays on top of the analytical solution without any visual differences. The numerical error Δu measured at the training points and averaged over the entire domain is $\Delta u = 9.0668\text{e-}5$ for 1e5 epochs (red/dash).

The Figure 5 and Table 2 serve the purpose to deliver a brief insight into the general investigations and characteristics later shown in this section. The results in Fig. 5 show 1e5 complete optimisation cycles with $\mathbf{p}_{rnd}^{init} \in [-1, 1]$, where all other computational parameters remain unchanged in each cycle. Displayed are both the cost function related training/verification error and the numerical error, depending on the euclidean distance between the initial $\mathbf{p}_0$ and learned $\mathbf{p}^*$ neural network weights. Please note that there is no information given about the direction, in which the optimiser moved in the weight space. In Fig. 5(a), the errors are evaluated at the 10 training points n_{TP} right after the training process has finished and in Fig. 5(b), the errors are evaluated at 100 intermediate verification points n_{VP} with the corresponding learned weights. The results show how the error evaluations are distributed and connected to a variety of final states. Depending on the starting point in the weight space, determined by the initial weights, the end results differ a lot. While some numerical errors (blue) are in range of $\Delta u_1 \approx$ 1e-3, others approximate the IVP way better with an accuracy around $\Delta u_1 \approx$ 1e-5 to $\Delta u_1 \approx$ 1e-6, even referring to similar $\|\mathbf{p}^* - \mathbf{p}_0\|_2$. One of the most remarkable characteristics shown in Fig. 5 however, both the numerical error at the training points and the numerical error at the verification points (blue) are very similar. On the other hand, the training and verification error (purple) show significant differences. This is an indication of the complexity of the energy landscape, spanned by the cost function, and it shows that the numerical error is somehow way more robust to the evaluation than the cost function related verification error. The difference between the best and the least good approximation in Fig. 5 lets the question arise, if this is an Adam-related characteristic or a common behaviour. Table 2 shows results for different optimisation methods in effort to a possible answer of the previously asked question. Ten different weight initialisation lead to ten different results for each optimiser. Clearly the best results, as expected, are provided by the second order optimiser BFGS with Wolfe-Powell line search. On the other hand, the first order optimiser Adam also shows useful approximations and backpropagation with constant step size (cBP) is still decent. However, BFGS is one order of accuracy ahead of Adam, while Adam is up to two orders ahead of cBP. Why backpropagation is performing in this way is mainly because both BFGS and Adam come with a variable step size which is important for convergence. Although backpropagation converges here, its ability to find a better minimum is limited. However, due to an adaptive computation of step sizes, Adam and especially BFGS take more computational time.

In context of this thesis, a comparison of both the deterministic weight initialisation (with zeros) and numerical methods are highly interesting. Table 3 clearly shows how good the approximation of the IVP in Eq. (70) with Runge-Kutta 4 (RK4) is. However,

No.	BFGS		Adam		cBP	
	Δu_1^{TP}	$E^{TP}[\mathbf{p}]$	Δu_1^{TP}	$E^{TP}[\mathbf{p}]$	Δu_1^{TP}	$E^{TP}[\mathbf{p}]$
1	1.4738e-6	6.7649e-12	4.2002e-4	4.0616e-6	1.7612e-3	8.5323e-5
2	1.0605e-6	2.9857e-12	5.7301e-5	1.5680e-7	2.2369e-3	1.4033e-4
3	1.6127e-4	1.0884e-8	6.1655e-4	1.0822e-5	2.3431e-3	1.7050e-4
4	5.5344e-5	3.2874e-8	9.2765e-5	2.5113e-7	1.8863e-3	1.5189e-4
5	2.5290e-6	2.1285e-11	3.4758e-4	8.3530e-7	1.8524e-3	9.2519e-5
6	5.5974e-5	1.0898e-10	1.0348e-4	1.5602e-7	3.6199e-3	5.4148e-4
7	4.1287e-5	6.5687e-10	1.5558e-3	4.4211e-5	2.6456e-3	2.8715e-4
8	3.8454e-6	1.9548e-11	6.9340e-4	5.4726e-7	2.3210e-3	1.5105e-4
9	7.5611e-7	2.2515e-14	8.0274e-5	1.4119e-7	2.5934e-3	3.1952e-4
10	7.6278e-6	2.0684e-11	4.2030e-5	1.9385e-8	1.9461e-3	1.1138e-4

Table 2: Results for ten complete optimisation cycles with different random weight initialisation $\mathbf{p}_{rnd}^{init}$.

	BFGS	Adam	cBP	Euler	RK4
Δu_1	6.3848e-7	8.5091e-1	8.3705e-2	4.9975e-6	1.6659e-10

Table 3: $\mathbf{p}_{deter}^{init}$=0 and 1e5 grid points for Euler and RK4.

also the Euler method shows reliable results, under the fact, that the number of grid points is chosen to be really high. That is, to match the number of training epochs for TSM with the number of grid points for the numerical methods, see Sec. 2.2.1. The numerical error for both Adam and cBP are not useful by any means. Since in combination with the deterministic initialisation, all optimisation methods return a constant but individual result. Only BFGS shows a very good approximation which is even better compared to the random initialisation with the particular parameter setup.

Concluding this introducing experiment, we will focus in the following on first order optimisation methods (Adam and backpropagation) in order to investigate their characteristics. This is because first order optimisation methods are more often used in machine learning since the computational time they require is lower compared to second order methods. Additionally, we want to find out how to improve the deterministic weight initialisation for the first order methods. Up to the current results, BFGS fulfills the expectations to provide very good results so that in the context of this thesis the question arises, how to improve the use of first order optimisation methods, especially in the context of a deterministic weight initialisation.

2.4.2 Details on the experiments

For experiments on both solution approaches (TSM [13] and mTSM [14, 15]) with different parameter variations, as well as optimisation with Adam and backpropagation, we make use of the model problem

$$\dot{u}(t) = \lambda u(t), \quad u(0) = 1 \tag{88}$$

a homogeneous first order ordinary differential equation with $\lambda \in \mathbb{R}$, $\lambda < 0$. The IVP in Eq. (88) has the analytical solution $u(t) = e^{\lambda t}$ and represents a simple model for stiff

Weights initialised with equal values	$\mathbf{p}_{deter}^{init}$
Weights initialised with random values	$\mathbf{p}_{rnd}^{init}$
Numeric error for $\mathbf{p}_{deter}^{init}$	Δu_{deter}
Mean value of numeric error for $\mathbf{p}_{rnd}^{init}$	$\overline{\Delta u}_{rnd}$
Trial solution method	TSM
Modified trial solution method	mTSM
Backpropagation with constant learning rate	cBP
Backpropagation with variable learning rate	vBP
Adam optimisation	Adam
Number of training points	n_{TP}
Number of maximal epochs	k_{max}
Stiffness parameter	λ
Left domain boundary	t_{end}

Table 4: Abbreviations in the experiment section.

phenomena involving a damping mechanism. With $\tilde{u}(t_i, \mathbf{p}) = 1 + t_i N(t_i, \mathbf{p})$ we take the form of the classic neural form for TSM proposed by Lagaris et al. in [13] to construct the cost function

$$E[\mathbf{p}] = \frac{1}{2(n+1)} \sum_{i=0}^{n} \left\{ \dot{N}(t_i, \mathbf{p}) t_i + N(t_i, \mathbf{p}) - \lambda \left(1 + N(t_i, \mathbf{p}) t_i\right) \right\}^2 \tag{89}$$

For mTSM the neural form $\tilde{u}(t_i, \mathbf{p}) = N(t_i, \mathbf{p})$ results in the cost function [14, 15]

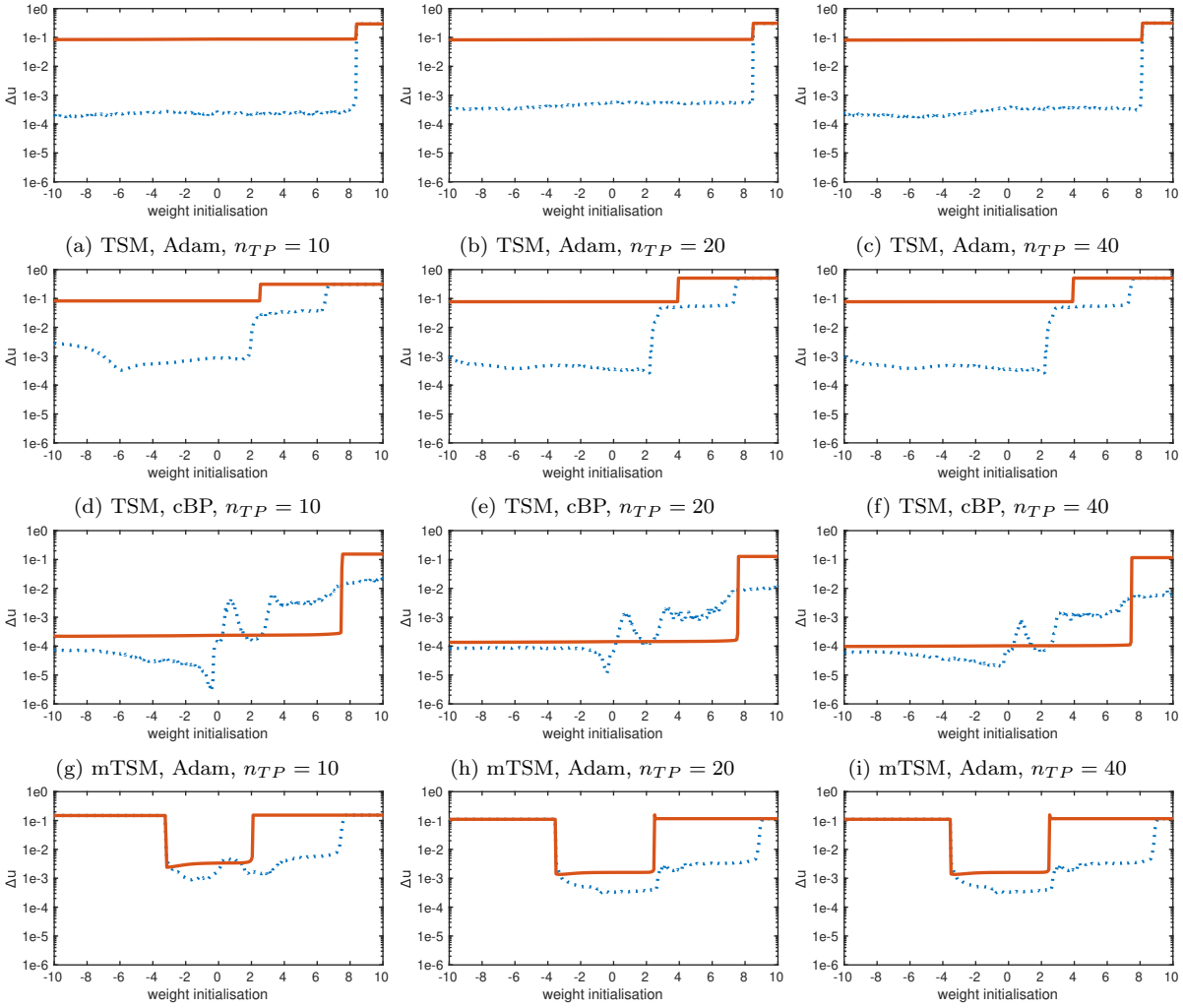
$$E[\mathbf{p}] = \frac{1}{2(n+1)} \sum_{i=0}^{n} \left\{ \dot{N}(t_i, \mathbf{p}) - \lambda N(t_i, \mathbf{p}) \right\}^2 + \frac{1}{2} \left\{ N(t_0, \mathbf{p}) - 1 \right\}^2 \tag{90}$$

In subsequent experiments we study Δu with respect to several, meaningful variations of computational parameters.

The main parameters and abbreviations of the computational settings are defined as in Table 4. In most subsequent experiments we used cBP instead of vBP, to reduce the amount of parameters. The numerical error Δu is exclusively evaluated at the training points in this section. However, it is important to note at this point, that usually a step size control or line search is required in order to make α adaptive. A constant step size can be chosen too small which may result in a very slow convergence. On the other hand, a larger step size can result in an oscillation around a minimum with an impact on the accuracy. Nonetheless, the later presented results using backpropagation with a constant step size are still meaningful, as this contributes to the speed of computation and makes backpropagation faster compared to Adam. Specific parameters for each optimisation method are documented in Sec. 2.3.1. The optimisation was realised with single batch training (cf. Eq. (51)). In addition, some experiments show averaged graphs to see the general trend with a reduced influence of fluctuations. If we do

not say otherwise in the subsequent experiments, the computational parameters are fixed with one hidden layer, five hidden layer neurons, number of maximal epochs $k_{max} = 1e5$, domain data $t \in [0, 2]$ and stiffness parameter $\lambda = -5$. Concerning the following experiments, let us stress again that these are not considered to be separate or independent of each other. We consequently follow a line of argumentation that enables us *(i)* to reduce step by step the degrees of freedom in the choice of computational settings, and *(ii)* to clarify the influence of individual computational parameters. In doing this we also demonstrate how to achieve tractable results. We consider this as an important part of our work since this makes the whole approach more meaningful.

2.4.3 Experiment: weight initialisation



(a) TSM, Adam, $n_{TP} = 10$ (b) TSM, Adam, $n_{TP} = 20$ (c) TSM, Adam, $n_{TP} = 40$

(d) TSM, cBP, $n_{TP} = 10$ (e) TSM, cBP, $n_{TP} = 20$ (f) TSM, cBP, $n_{TP} = 40$

(g) mTSM, Adam, $n_{TP} = 10$ (h) mTSM, Adam, $n_{TP} = 20$ (i) mTSM, Adam, $n_{TP} = 40$

(j) mTSM, cBP, $n_{TP} = 10$ (k) mTSM, cBP, $n_{TP} = 20$ (l) mTSM, cBP, $n_{TP} = 40$

Figure 6: **Experiment 2.4.3.** Weight initialisation variation, (orange/solid) Δu_{deter}, (blue/dotted) $\overline{\Delta u}_{rnd}$.

This experiment illustrates differences between the two weight initialisation methods, employing either $\mathbf{p}^{init}_{deter}$ or $\mathbf{p}^{init}_{rnd}$. We averaged 1e2 complete optimisation cycles for displaying each point in the graph depicting $\overline{\Delta u}_{rnd}$, implying that for the values given

at the lower axis we perform computations with 1e2 overlaid random perturbations, with random numbers in range of 1e-2, as initialisation around each point. The averaging is important to mention, because every iteration with $\mathbf{p}_{rnd}^{init}$ and exactly the same computational parameter setup, is expected to return different results.

Let us first comment on our choice of the deterministic initial weights $\mathbf{p}_{deter}^{init}$. Evidently, one has to choose here some fixed value, and by further experiments, the value zero appears to be a suitable generic choice for mTSM. Considering the experiments documented in Fig. 6, TSM with both cBP and Adam (see illustrations (a)–(f)) does not return helpful results for the deterministic initialisation $\mathbf{p}_{deter}^{init}$ with the current parameter setup. All experiments for TSM with $\mathbf{p}_{deter}^{init}$ give here uniformly a very high error (depicted by orange/solid lines), even when increasing the number of training points. Besides returning a stable solution (no variations caused by different initial weights), the TSM approach combined with a deterministic initialisation clearly lacks flexibility. This is mainly because initialising each weight with the same value effectively reduces the amount of hidden layer neurons to one. Considering five hidden layer neurons with $\mathbf{p}_{deter}^{init}$, the five weights from the input neuron to the hidden layer each have the same impact on the neural network outcome. The same holds for the five weights from the bias neuron to the hidden layer and the weights from the hidden layer to the output layer. In other words, each hidden layer neuron receives the exact same weighted sum and therefore the output neuron also receives a weighted sum consisting of five equal terms. In total, this characteristic lets the hidden layer neurons effectively act as one neuron. However, there is still a difference regarding the initial weights since employing either, e.g., small or large initial values can still have an impact on the results.

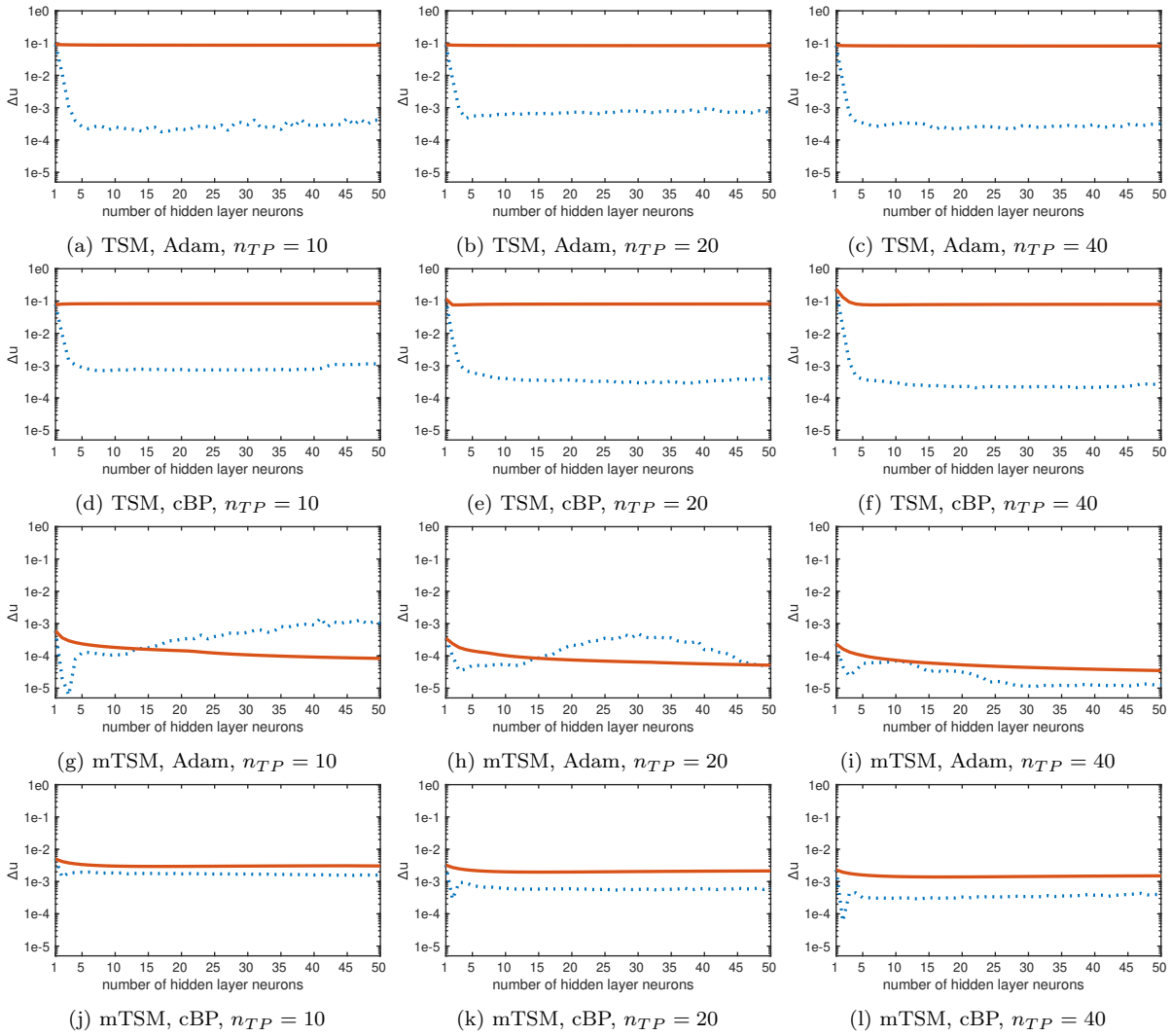
The overall clearly best results for $\mathbf{p}_{deter}^{init}$ are provided by mTSM in combination with Adam and $n_{TP} = 40$. Not only is the Δu_{deter} (orange/solid) slightly decreasing with higher numbers of training data (n_{TP}), but also the region of stable results appears to be slightly larger over $n_{TP} = 10$ and $n_{TP} = 20$. However, the results demonstrate that the overall visual dependency of $\mathbf{p}_{deter}^{init}$ on the number of training points seems to be insignificant while Adam provides a desirable proceeding.

When considering the non-deterministic weight initialisation $\mathbf{p}_{rnd}^{init}$ in Fig. 6, the Adam solver gives also for TSM reasonable results in terms of the numerical error with a large stable region. This behaviour has similarities to the characteristics shown for mTSM, Adam and $\mathbf{p}_{deter}^{init}$ in Fig. 6(g)–(i). Furthermore, that is maybe an indication for a relation between both solution methods. However, the energy landscape is spanned by the cost function in the weight space and both TSM and mTSM are more likely to show different behaviours with different minima. Therefore, the visible similarities in the behaviour are perhaps only random occurrences. In Fig. 6(g)–(i), the random initialisation shows a complete different trend, compared to Fig. 6(a)–(c), with several highs and lows. For these diagrams, there seems to be no clear relation to any other result in this experiment. As a general trend in all experiments with $\mathbf{p}_{rnd}^{init}$, we observe that the weight initialisation in a small range around zero seems to work best. Let us also comment on the illustrations in Fig. 6(j)–(l) in this context, since we observe here the behaviour that both $\mathbf{p}_{deter}^{init}$ and $\mathbf{p}_{rnd}^{init}$ around zero seem to work reasonably with cBP. One may conjecture for other example ODEs, that there could be some deterministic initialisation and a range of random fluctuations around it that may work well.

Concluding the deterministic initialisation, the results in Fig. 6 show the behaviour of both solution approaches and optimisation methods for a specific set of computational

parameters, where only the initial weights are subject to variation. Since a suitable choice of $\mathbf{p}_{deter}^{init}$ and $\mathbf{p}_{rnd}^{init}$ is important in all subsequent experiments, we decided as a consequence of the experiments discussed here to initialise $\mathbf{p}_{deter}^{init}$ with zeros and $\mathbf{p}_{rnd}^{init}$ with random values in range of 0 to 1e-2 from now on.

2.4.4 Experiment: number of hidden layer neurons



(a) TSM, Adam, $n_{TP} = 10$ (b) TSM, Adam, $n_{TP} = 20$ (c) TSM, Adam, $n_{TP} = 40$

(d) TSM, cBP, $n_{TP} = 10$ (e) TSM, cBP, $n_{TP} = 20$ (f) TSM, cBP, $n_{TP} = 40$

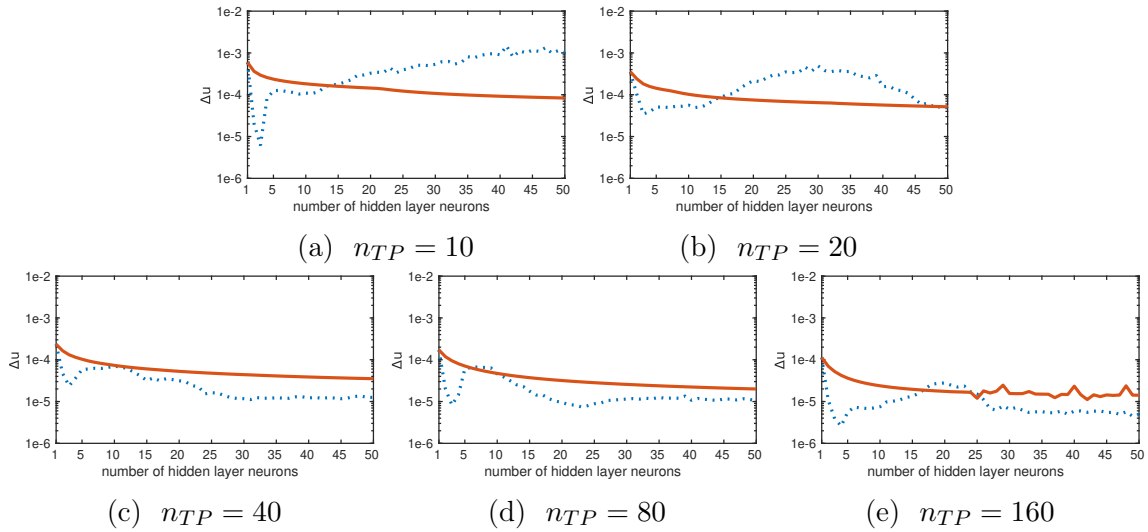
(g) mTSM, Adam, $n_{TP} = 10$ (h) mTSM, Adam, $n_{TP} = 20$ (i) mTSM, Adam, $n_{TP} = 40$

(j) mTSM, cBP, $n_{TP} = 10$ (k) mTSM, cBP, $n_{TP} = 20$ (l) mTSM, cBP, $n_{TP} = 40$

Figure 7: **Experiment 2.4.4.** Number of hidden layer neurons variation, (orange/solid) Δu_{deter}, (blue/dotted) $\overline{\Delta u}_{rnd}$.

The behaviour of Δu_{deter} and $\overline{\Delta u}_{rnd}$ when increasing the number of hidden layer neurons is subject to this experiment, where $\overline{\Delta u}_{rnd}$ is averaged over 1e2 optimisation cycles for every tested number of hidden layer neurons.

There is almost no difference between the experiments for TSM in Fig. 7(a)–(f), they all show a similar saturating behaviour. As discussed in the previous experiment, it is clear that we have to focus here on the random initialisation, and for this setup we observe here desirable results for about five or more neurons.



(a) $n_{TP} = 10$ (b) $n_{TP} = 20$

(c) $n_{TP} = 40$ (d) $n_{TP} = 80$ (e) $n_{TP} = 160$

Figure 8: **Experiment 2.4.4.** Number of hidden layer neurons variation with the focus on mTSM and Adam, extending Fig. 7, (orange/solid) Δu_{deter}, (blue/dotted) $\overline{\Delta u}_{rnd}$.

Turning to mTSM, a higher number of hidden layer neurons leads to an increase in accuracy for Adam for larger numbers of training points explored here ($n_{TP} = 40$). For smaller numbers of training points ($n_{TP} = 10$ and $n_{TP} = 20$) we observe here indications that the number of hidden layer neurons and thus the degrees of freedom introduced by the neural network should be in a relatively small range, e.g., about half the amount of training points. In this context, the results shown in Fig. 7(g)–(i) are further investigated in Fig. 8. Here, the vertical axis is slightly different scaled in order to have a better view on the general trend, since results for $n_{TP} = 80$ and $n_{TP} = 160$ are added. In general, duplicating the number of training points in each diagram allows us to comment the computational convergence behaviour. For results with $\mathbf{p}_{deter}^{init}$ we see that there is a saturation towards higher numbers of hidden layer neurons, while too many training points may cause an unstable behaviour. Returning to $\overline{\Delta u}_{rnd}$, a clear relation between the number of training points and the number of hidden layer neurons is visible. Namely, increasing n_{TP} leads to a smaller $\overline{\Delta u}_{rnd}$ for a higher number of hidden layer neurons. While for $n_{TP} = 10$ in Fig. 8(a) the numerical error related to $\mathbf{p}_{rnd}^{init}$ is continuously increasing, does $n_{TP} = 20$ stabilise the results towards the end of diagram 8(b). Further increasing the number of training points dampens the numerical error in general. However, the most remarkable characteristic in Fig. 8 relates to smaller amounts of hidden layer neurons. Throughout all diagrams, the numerical error $\overline{\Delta u}_{rnd}$ rapidly decreases in the beginning (for around three to four neurons) and starts increasing again. This local minimum has most of the time the best results attached. This phenomenon appears to be related to double descent [99]. The local minimum is most likely a combination of an initially decreasing trend and a diverging trend. Once the error starts increasing we would except it to totally diverge. However, the model manages to be able to decrease the numerical error again. The initial trend of a decreasing numerical error is something we would also expect since the flexibility should theoretically increase using more hidden layer neurons. This assumption is related to the universal approximation theorem [36]. The theorem basically states that a feedfor-

ward neural networks with a finite number of hidden layer neurons can approximate every continuous function on a subset of $\mathbb{R}$ with arbitrary accuracy. Therefore, one may assume that an increasing number of hidden layer neurons may continuously lower the numerical error. However, the theorem does not relate to the optimisation problem and it also does not state how many neurons are required. With the rest of the computational parameters remaining unchanged, the general trend of the initially decreasing numerical error differs to expected values in the investigated range. Let us note at this point, that in context of double descent we do not see the training loss depicted in the diagrams, but the numerical error. Nonetheless, this phenomenon seems to apply here as well.

Also for cBP the saturation value of the numerical error in Fig. 7(j)–(l) for mTSM is affected by increasing the number of hidden layer neurons. The general trend for $\overline{\Delta u}_{rnd}$ provides a slightly higher accuracy in this way, and that the saturation level is visible already when using a small number of neurons. Here the double descent phenomenon is also visible.

As a consequence of these investigations, we employ five hidden layer neurons in the other experiments (note that this setting has also been used in the previous experiment) as this appears to be justified by the stable solutions and the amount of computational time.

2.4.5 Experiment: number of hidden layers

In order to focus on the impact of the number of hidden layers, we decided here to keep the number of neurons in the hidden layers constant, employing five neurons in each layer. As in previous experiments, $\overline{\Delta u}_{rnd}$ is averaged over 1e2 optimisation cycles.

The results in Tab. 5 show that one hidden layer is not always enough to provide useful results. In the previous experimental sections we found TSM in combination with $\mathbf{p}^{init}_{deter}$ to not be flexible enough to provide reliable results. Here we find employing one or two additional hidden layers to be beneficial. As expected, also the number of training points (n_{TP}) effects the accuracy and usefulness of adding hidden layers. The more hidden layers a neural network features, the more neurons and degrees of freedom (weights) are incorporated. In numerical approximation methods, increasing the number of grid points often results in a better accuracy. In contrast, and as we have shown in the previous experimental sections, maintaining or improving the accuracy is not always related to the number of training points. However, universal conclusions are difficult to make. Let us now provide a brief summary of findings related to Tab. 5:

- cBP and Δu_{deter}:
  This combination together with TSM highly benefits from a second and third layer by gaining one order of accuracy for each layer. This holds for $n_{TP} = 10$ and $n_{TP} = 20$. Further increasing n_{TP} is not beneficial anymore. Increasing the layers has only minor impact on the combination together with mTSM.

- cBP and $\overline{\Delta u}_{rnd}$:
  The impact of adding layers is close to irrelevant for both TSM and mTSM. A slightly higher impact does the number of training points have. Overall, using a second hidden layer is a minor improvement.

$n_{TP} = 10$

method	cBP				Adam			
	Δu_{deter}		$\overline{\Delta u}_{rnd}$		Δu_{deter}		$\overline{\Delta u}_{rnd}$	
hidden Layer	TSM	mTSM	TSM	mTSM	TSM	mTSM	TSM	mTSM
1	8.25e-2	3.25e-3	7.84e-4	1.98e-3	8.74e-2	2.20e-4	2.67e-4	1.26e-4
2	2.72e-3	2.11e-3	6.40e-4	1.15e-3	3.73e-3	5.27e-4	5.34e-4	1.84e-4
3	6.88e-4	1.54e-1	8.15e-4	1.54e-1	2.10e-3	3.45e-4	8.69e-4	4.78e-4
4	2.94e-1	1.54e-1	2.94e-1	1.54e-1	2.94e-1	2.93e-4	2.94e-1	1.21e-2
5	2.94e-1	1.54e-1	2.94e-1	1.54e-1	2.94e-1	1.28e-1	2.94e-1	4.12e-2

$n_{TP} = 20$

method	cBP				Adam			
	Δu_{deter}		$\overline{\Delta u}_{rnd}$		Δu_{deter}		$\overline{\Delta u}_{rnd}$	
hidden Layer	TSM	mTSM	TSM	mTSM	TSM	mTSM	TSM	mTSM
1	7.85e-2	2.14e-3	5.43e-4	7.00e-4	8.49e-2	1.33e-4	5.51e-4	5.04e-5
2	1.82e-3	2.24e-3	2.74e-4	5.16e-4	2.54e-3	2.06e-4	5.05e-4	2.12e-4
3	7.76e-4	1.28e-1	3.40e-4	1.28e-1	9.91e-4	3.37e-4	9.18e-4	2.92e-4
4	3.08e-1	1.28e-1	3.09e-1	1.28e-1	3.09e-1	1.08e-4	3.09e-1	2.26e-2
5	3.08e-1	1.28e-1	3.09e-1	1.28e-1	3.09e-1	1.04e-1	3.09e-1	1.33e-2

$n_{TP} = 40$

method	cBP				Adam			
	Δu_{deter}		$\overline{\Delta u}_{rnd}$		Δu_{deter}		$\overline{\Delta u}_{rnd}$	
hidden Layer	TSM	mTSM	TSM	mTSM	TSM	mTSM	TSM	mTSM
1	7.62e-2	1.54e-3	4.04e-4	3.16e-4	8.24e-2	9.35e-5	2.79e-4	6.01e-5
2	2.33e-2	2.17e-3	2.37e-4	3.01e-4	1.55e-3	5.84e-5	4.68e-4	5.23e-5
3	2.01e-3	1.16e-1	3.28e-4	1.16e-1	2.25e-3	1.80e-4	4.91e-4	1.19e-4
4	3.15e-1	1.16e-1	3.15e-1	1.16e-1	3.14e-1	2.29e-4	3.13e-1	2.94e-2
5	3.15e-1	1.16e-1	3.15e-1	1.16e-1	3.14e-1	9.46e-2	3.13e-1	1.03e-2

Table 5: **Experiment 2.4.5.** Number of hidden layer variation.

- Adam and Δu_{deter}:
  The overall trend for TSM is very similar to cBP and Δu_{deter}. and the combination together with mTSM does not really benefit from adding layers.

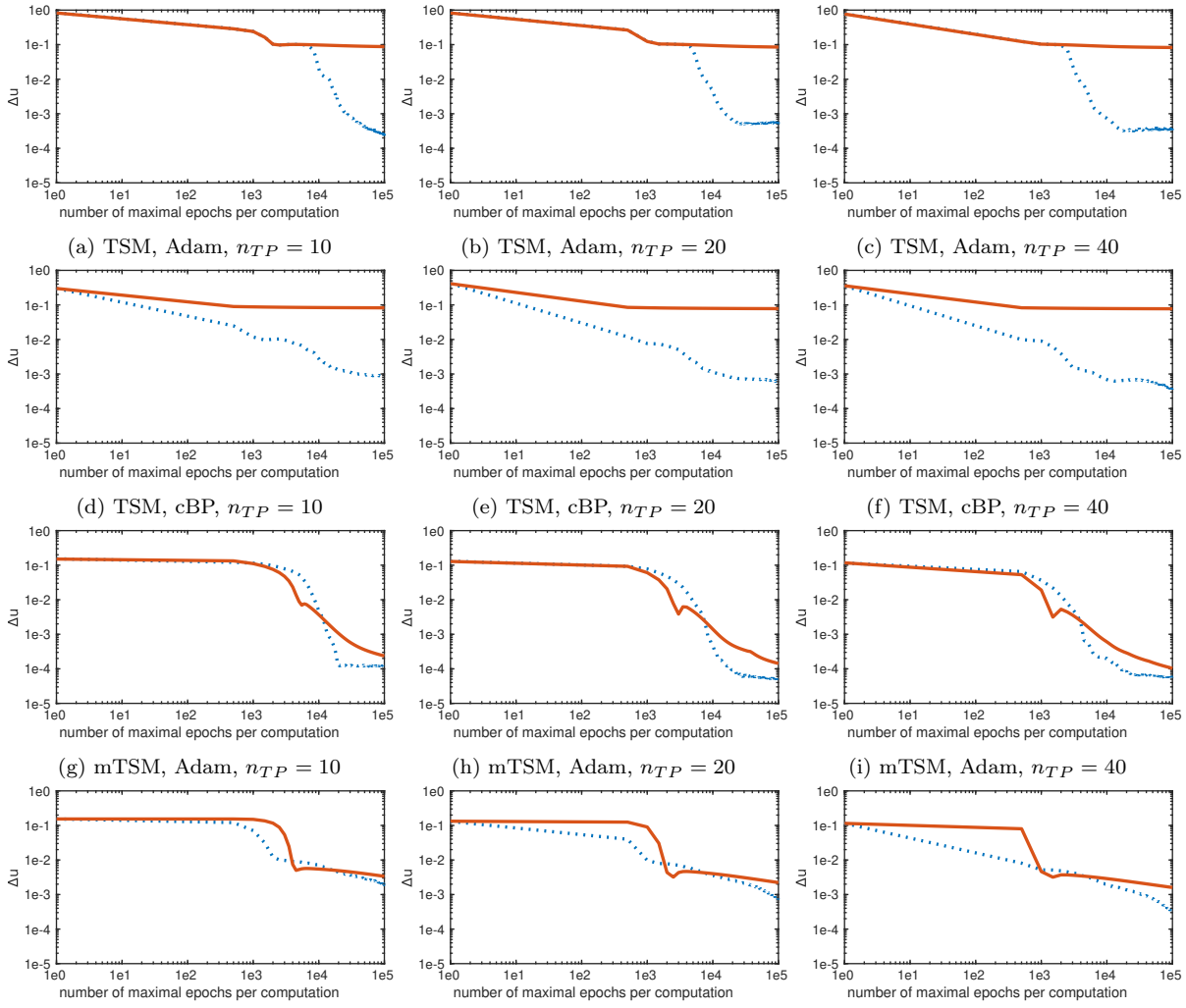
- Adam and $\overline{\Delta u}_{rnd}$:
  Here it is neither beneficial for TSM nor mTSM to add more layers.

For cBP we find TSM in general to be accurate for up to three layers, while mTSM is accurate up to two layers. With Adam on the other hand, the deterministic initialisation $\mathbf{p}^{init}_{deter}$ for mTSM is accurate up to four layers while TSM with $\mathbf{p}^{init}_{deter}$ and both TSM and mTSM with $\mathbf{p}^{init}_{rnd}$ show useful approximations for up to three hidden layers. However, please note that this conclusion may highly depend on the employed computational parameters and the fact that only first order optimisation methods are used. The result in general appears to be to some degree surprising, as the universal approximation theorem should imply that one hidden layer could be enough to give

here experimentally an accurate approximation of our solution function. Let us recall in this context Experiment 2.4.4, where we have seen that increasing of the number of neurons in one hidden layer leads to a saturation in the accuracy for $\mathbf{p}_{rnd}^{init}$, while we observe here a clear improvement. Increasing the number of neurons and using $\mathbf{p}_{deter}^{init}$ did not lead to reasonable results there, while $\mathbf{p}_{deter}^{init}$ here in combination with more hidden layers gives good results plus a significant improvement.

As a consequence of this investigation, we decided to use one hidden layer for all computations in the other experiments, having in mind that TSM may allow an accuracy gain for more hidden layers. However, a possible reason for the general observation of decreasing accuracy can be an indication for the approximation and optimisation error to highly increase with more hidden layers, under the employed parameter setup with first order optimisation.

2.4.6 Experiment: number of epochs



(a) TSM, Adam, $n_{TP} = 10$ (b) TSM, Adam, $n_{TP} = 20$ (c) TSM, Adam, $n_{TP} = 40$

(d) TSM, cBP, $n_{TP} = 10$ (e) TSM, cBP, $n_{TP} = 20$ (f) TSM, cBP, $n_{TP} = 40$

(g) mTSM, Adam, $n_{TP} = 10$ (h) mTSM, Adam, $n_{TP} = 20$ (i) mTSM, Adam, $n_{TP} = 40$

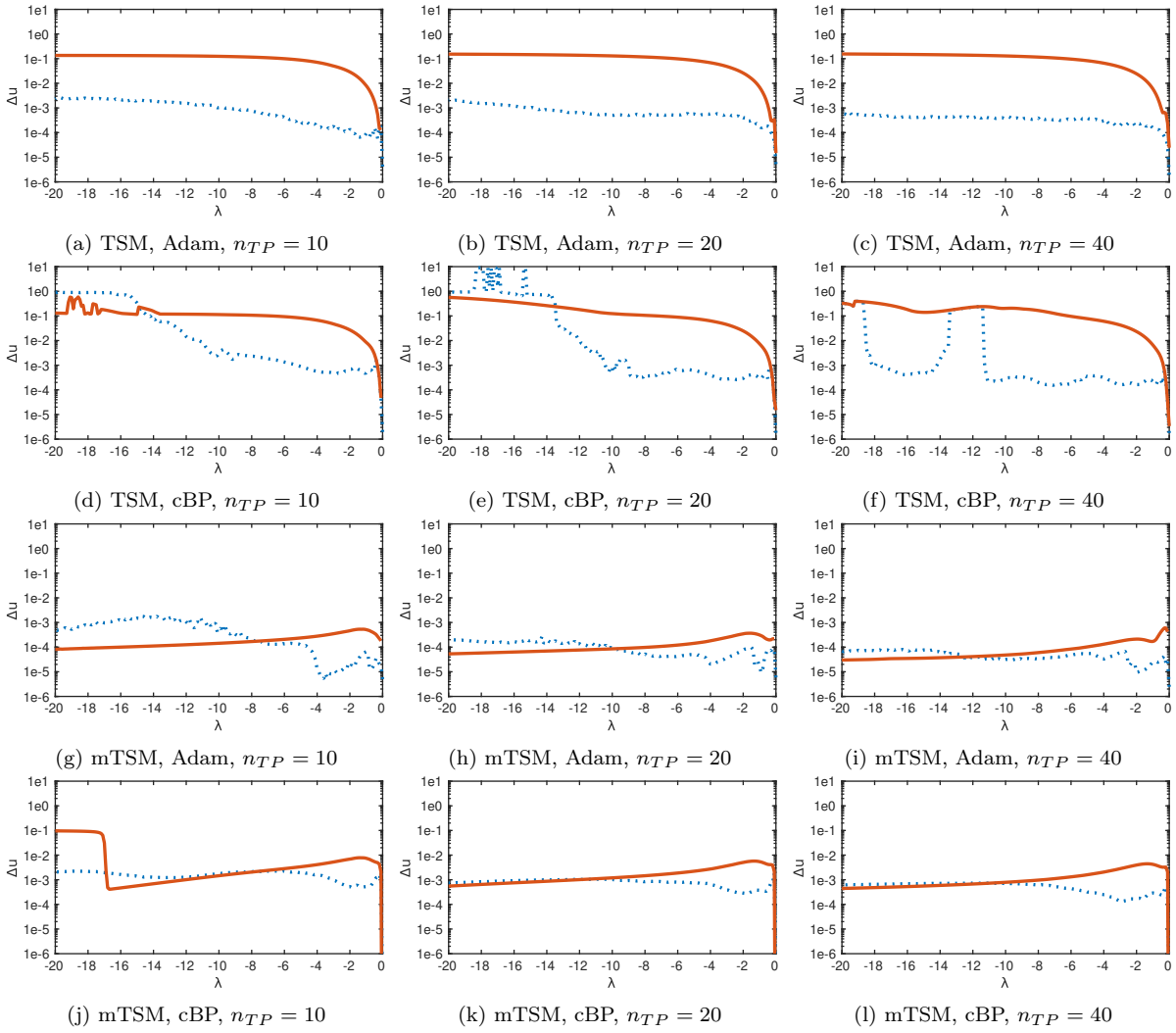
(j) mTSM, cBP, $n_{TP} = 10$ (k) mTSM, cBP, $n_{TP} = 20$ (l) mTSM, cBP, $n_{TP} = 40$

Figure 9: **Experiment 2.4.6.** Number of maximal epochs variation, (orange/solid) Δu_{deter}, (blue/dotted) $\overline{\Delta u}_{rnd}$.

In this experiment we aim to investigate and fix the maximal number of training epochs per optimisation cycle to a convenient value. This relates to the question if one could bound the computational load by employing in general a small number of iterations. To this end, we consider the convergence of the training as a function of an increasing maximal number of epochs k_{max}. In addition we discuss the influence of the number of training points again. More precisely, we increased k_{max} from 1 to 1e5 and averaged 1e2 optimisation cycles for one and the same k_{max}. In other words, and to make clear the meaning of the lower axis in Fig. 9, one entry of the number k_{max} relates to 1e2 corresponding complete optimisations of the neural network. Let us note again, that in the case of $\mathbf{p}_{rnd}^{init}$, the convergence behaviour can only be evaluated by average values, and that each computation in this context was realised with a new set of random initial values. As can be seen in Fig. 9, best results are returned by mTSM with Adam for both $\mathbf{p}_{rnd}^{init}$ (especially $n_{TP} = 20$) and $\mathbf{p}_{deter}^{init}$ (especially $n_{TP} = 40$). Except for TSM and $n_{TP} = 10$, the Adam optimiser clearly reaches a saturation regime showing convergence for TSM and mTSM with the non-deterministic initialisation $\mathbf{p}_{rnd}^{init}$. For cBP, Δu_{deter} and $\overline{\Delta u}_{rnd}$, still may decrease for even higher k_{max} as evaluated here. However, let us note here that we employed in cBP a constant learning rate, for decreasing learning rates as often used for improved convergence behaviour, we may expect that a saturation regime may be observed. However, with Adam, $\overline{\Delta u}_{rnd}$ shows a small fluctuating behaviour in the convergence regime, so that results for non-averaged computations with $\mathbf{p}_{rnd}^{init}$ may be not satisfying. The cBP optimiser together with both TSM and mTSM shows very minor fluctuations, but also provides less good approximations. However, these tend to get better with higher n_{TP}. Nonetheless, since we see averaged values for $\overline{\Delta u}_{rnd}$, individual optimisation cycles may converge earlier or fluctuate around on their own. Speaking of the latter in other words, some realisations are likely to stop (after, e.g., 1e5 epochs) at a useful position around a local minimum while others perhaps end at a slightly less useful position around the same local minimum. Besides selecting a fixed number of epochs for training, another strategy to use is early stopping [100, 101]. The main idea is to stop the training process, before the training error and a validation error (evaluation on test data during training) differ too much, e.g., due to overfitting. A possible way to employ this technique is to stop training when the training error itself does not significantly change over the last epochs. In this case a minimal number of training epochs should be considered since we observe in Fig. 9 that in some cases the training error first takes some time before it starts decreasing. In case of a possible oscillation around a local minimum in the end of an optimisation cycle, rapid and repeatedly changing in the training error can be eliminated by stopping at a suitable position. All diagrams related to mTSM again show a certain phenomenon, namely double descent, which we have discussed in Experiment 7. In the context of our results, let us note that in [15] the authors employed 5e4 epochs. Our investigation shows that the corresponding results are supposed to be in the convergence regime. In conclusion, we find that k_{max}=1e5 as used for all other experiments is suitable to obtain useful approximations.

2.4.7 Experiment: stiffness parameter (part 1) and domain size (part 2)

Let us now investigate the solution behaviour with respect to interesting choices of the stiffness parameter λ (Fig. 10), and it turns out that it makes sense to do this together



(a) TSM, Adam, $n_{TP} = 10$ (b) TSM, Adam, $n_{TP} = 20$ (c) TSM, Adam, $n_{TP} = 40$

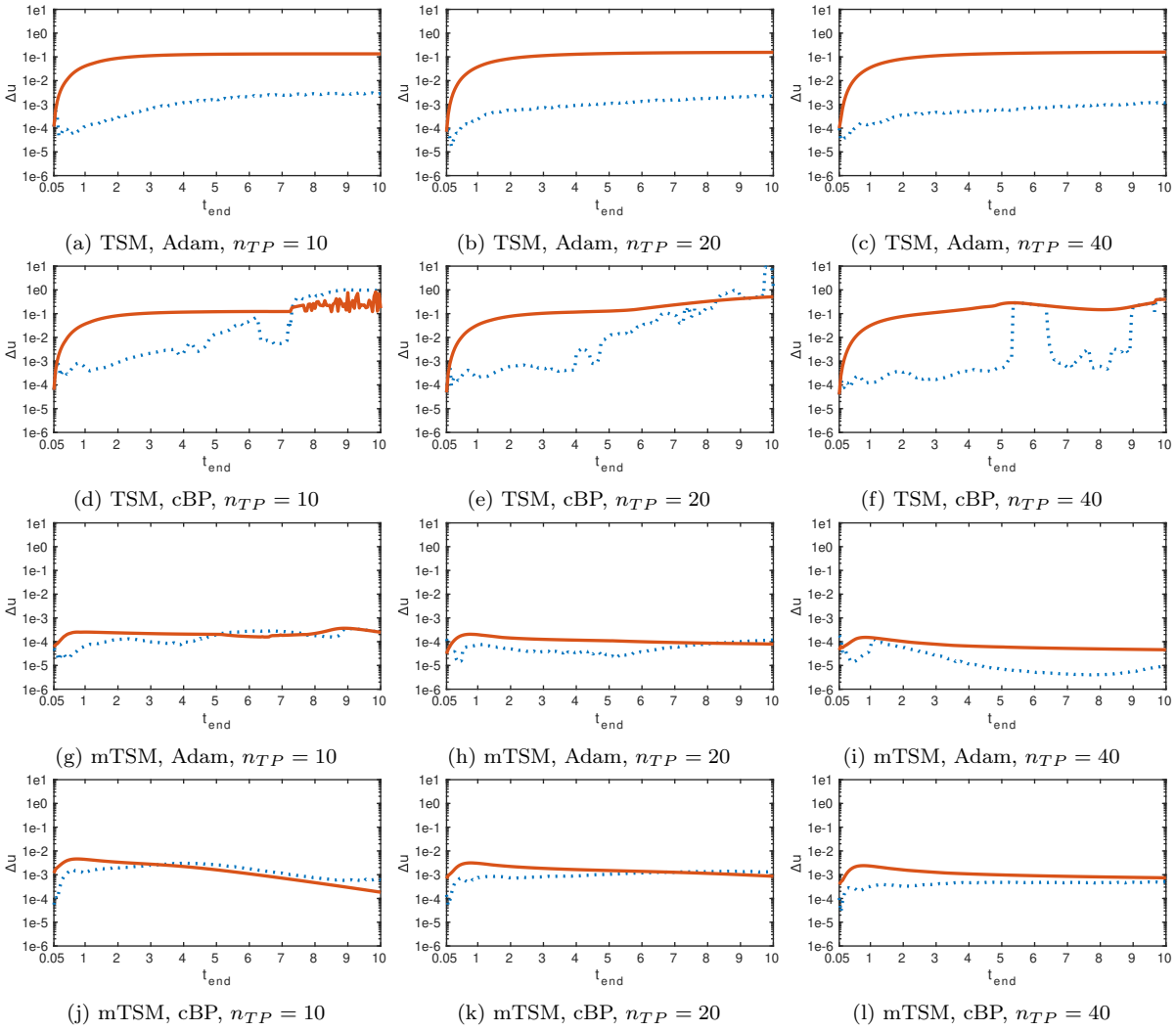
(d) TSM, cBP, $n_{TP} = 10$ (e) TSM, cBP, $n_{TP} = 20$ (f) TSM, cBP, $n_{TP} = 40$

(g) mTSM, Adam, $n_{TP} = 10$ (h) mTSM, Adam, $n_{TP} = 20$ (i) mTSM, Adam, $n_{TP} = 40$

(j) mTSM, cBP, $n_{TP} = 10$ (k) mTSM, cBP, $n_{TP} = 20$ (l) mTSM, cBP, $n_{TP} = 40$

Figure 10: **Experiment 2.4.7 (part 1).** Stiffness parameter λ variation, (orange/solid) Δu_{deter}, (blue/dotted) $\overline{\Delta u}_{rnd}$.

with an investigation of the solution domain size (Fig. 11). Informally speaking, these parameters also impact the general trend of the analytical solution in a similar way so that it appears also from this point of view natural to evaluate them together in one experiment. As shown in Fig. 11, the influence of different domains with increasing n_{TP} is the objective of this experiment. Intervals used for computations are given in terms of $t \in [0, t_{end}]$, with the smallest interval being $t \in [0, 5\text{e-}2]$ and then increasing in steps of 5e-2. As also in the first experimental part here, $\overline{\Delta u}_{rnd}$ is averaged by 1e2 optimisation cycles for each domain. Turning to the results, first we want to point out that for TSM, cBP and $n_{TP} = 20$ there are values displayed as $\overline{\Delta u}_{rnd} = 9\text{e}0$, to visualise them. In reality, these values were Not a Number (NaN), which means, that at this point at least one of the 1e2 averaged optimisation cycles diverged for the corresponding of λ in Fig. 10(c), or large domains. Furthermore, the solution accuracy for TSM and cBP in Fig. 10 is strictly decreasing for larger negative values of λ and larger domains until it saturates in unstable regions. While increasing the number of training points from $n_{TP} = 10$ to $n_{TP} = 20$ some iterations diverged, another increase to $n_{TP} = 40$ enlarges



(a) TSM, Adam, $n_{TP} = 10$ (b) TSM, Adam, $n_{TP} = 20$ (c) TSM, Adam, $n_{TP} = 40$

(d) TSM, cBP, $n_{TP} = 10$ (e) TSM, cBP, $n_{TP} = 20$ (f) TSM, cBP, $n_{TP} = 40$

(g) mTSM, Adam, $n_{TP} = 10$ (h) mTSM, Adam, $n_{TP} = 20$ (i) mTSM, Adam, $n_{TP} = 40$

(j) mTSM, cBP, $n_{TP} = 10$ (k) mTSM, cBP, $n_{TP} = 20$ (l) mTSM, cBP, $n_{TP} = 40$

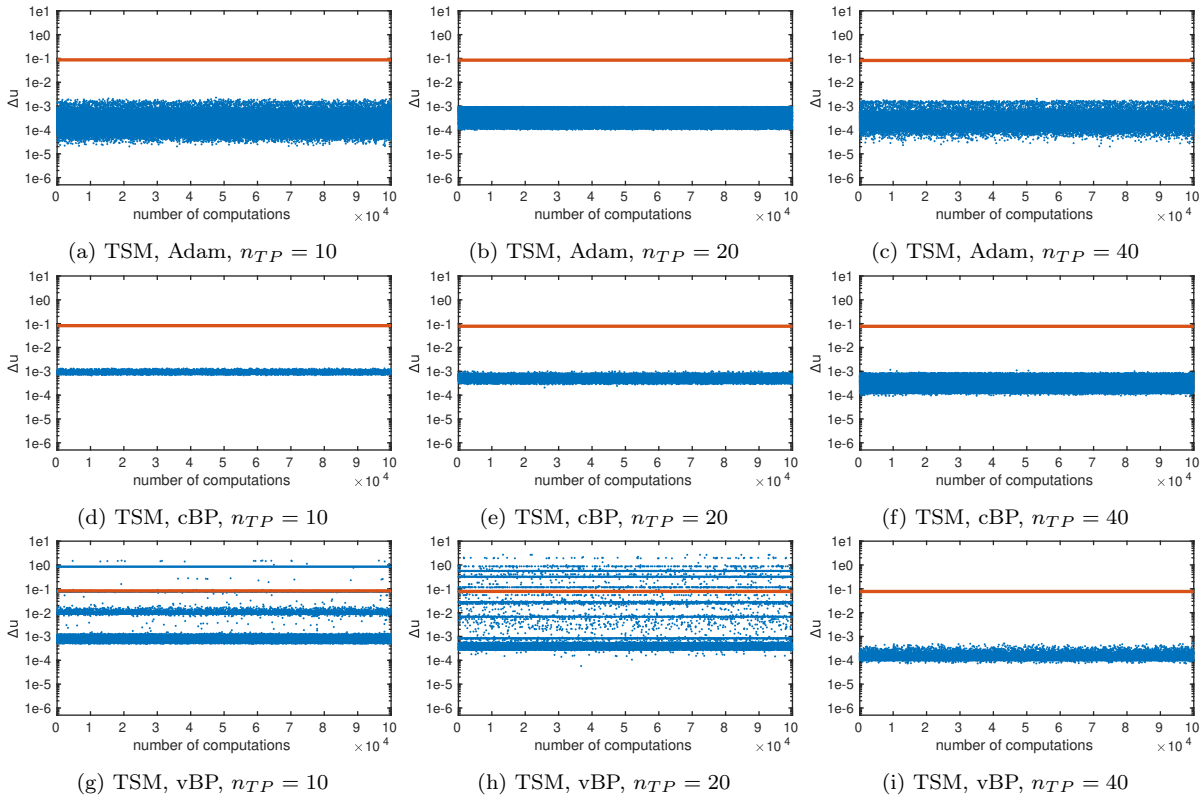
Figure 11: **Experiment 2.4.7 (part 2).** Domain size variation, (orange/solid) Δu_{deter}, (blue/dotted) $\overline{\Delta u}_{rnd}$.

the unstable region with a stabilisation in between. In the total, we observe that there seems to be a relation between the experiments that one may roughly formulate as a relation between λ and domain size given by t_{end} as a factor of -2. We also conjecture, that the higher the values of $-\lambda$ and t_{end}, the more neurons or layers are required for a convenient solution. As a consequence of these experiments, we decided to fix $\lambda = -5$ and $t \in [0, 2]$ for all computations in the other experiments.

2.4.8 Experiment: optimisation methods

The final experiment in this section compares Adam, cBP and vBP optimisation for TSM and mTSM, depending on $n_{TP} = 10, 20, 40$ with the other computational parameters fixed to one hidden layer, five hidden layer neurons, $k_{max} = 1e5$, $\lambda = -5$ and $t \in [0, 2]$. Fig. 12 and 13 show 1e5 (non-averaged) computed results for each parameter setup and weight initialisation.

Previous experiments led to the conclusion, that TSM in combination with $\mathbf{p}^{init}_{deter}$



(a) TSM, Adam, $n_{TP} = 10$ (b) TSM, Adam, $n_{TP} = 20$ (c) TSM, Adam, $n_{TP} = 40$

(d) TSM, cBP, $n_{TP} = 10$ (e) TSM, cBP, $n_{TP} = 20$ (f) TSM, cBP, $n_{TP} = 40$

(g) TSM, vBP, $n_{TP} = 10$ (h) TSM, vBP, $n_{TP} = 20$ (i) TSM, vBP, $n_{TP} = 40$

Figure 12: **Experiment 2.4.8.** Optimiser comparison (part 1), (orange/solid) Δu_{deter}, (blue/dotted) Δu_{rnd}.

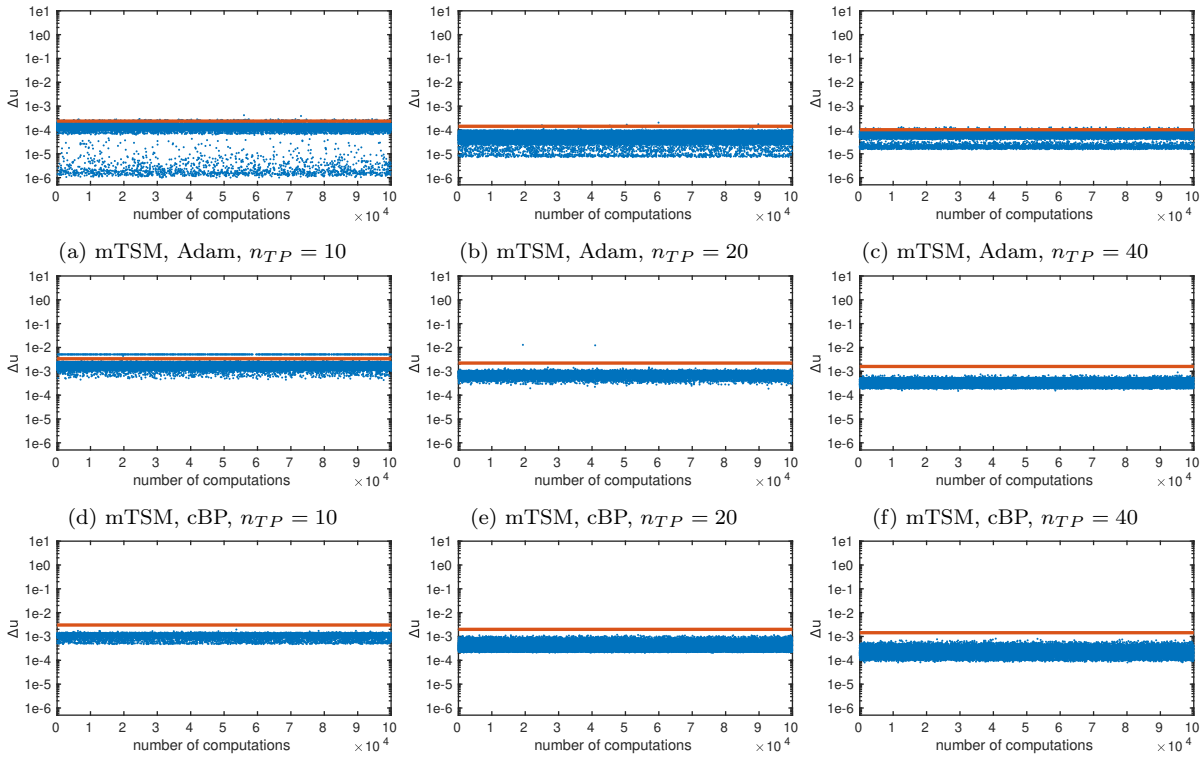
only provides unstable solutions for the chosen parameter setup. Therefore, when evaluating TSM, we only refer to the non-averaged numeric error Δu_{rnd} for $\mathbf{p}_{rnd}^{init}$.

To start the evaluation with TSM and Adam, there are almost no visible differences between $n_{TP} = 10$ and $n_{TP} = 40$, with a large difference between the best and the least good approximation, see first row in Fig. 12. Only for $n_{TP} = 20$ the solutions tend to be more similar.

In contrast, the difference between the best and the least good approximation for TSM and cBP grows by one order of magnitude with a higher number of training points while simultaneously the accuracy for the best approximations increases, cf. second row in the figure.

The reason we show results on vBP only in this final experiment (see third row in the figure) is, that the efficiency of an adaptive step size method may be in general highly dependent on the used step size model and parameters. However, the results turn out to be interesting. In combination with $n_{TP} = 10$, vBP and TSM reveal several minima far away from the best approximation. Even more minima appear for a training points increase to $n_{TP} = 20$. However, another increase to $n_{TP} = 40$ stabilises the solutions. In addition, $n_{TP} = 40$ provides the best approximations for TSM and vBP. One may conjecture here, that either one has here to reach a critical number of training points, or that the weight initialisation here is not adequate together with lower n_{TP}.

Now we turn to mTSM and Adam, see first row in Fig. 13. We find $\mathbf{p}_{deter}^{init}$ to show useful results (Δu_{deter}) and a small gain in accuracy for higher n_{TP}. For $\mathbf{p}_{rnd}^{init}$, we find the best approximations throughout the whole experiment to be provided by



(a) mTSM, Adam, $n_{TP} = 10$ (b) mTSM, Adam, $n_{TP} = 20$ (c) mTSM, Adam, $n_{TP} = 40$

(d) mTSM, cBP, $n_{TP} = 10$ (e) mTSM, cBP, $n_{TP} = 20$ (f) mTSM, cBP, $n_{TP} = 40$

(g) mTSM, vBP, $n_{TP} = 10$ (h) mTSM, vBP, $n_{TP} = 20$ (i) mTSM, vBP, $n_{TP} = 40$

Figure 13: **Experiment 2.4.8.** Optimiser comparison (part 2), (orange/solid) Δu_{deter}, (blue/dotted) Δu_{rnd}.

$n_{TP} = 10$. However, most of the 1e5 computed results appear around a less good (but still reasonable) accuracy with only a few results peaking further in accuracy. Increasing the number of training points to $n_{TP} = 20$ and $n_{TP} = 40$ results in a drop of accuracy from the former best solutions, while overall the results become more similar.

For mTSM and cBP, see second row in the figure, we find a likewise behaviour of Δu_{deter}, similar to the case mTSM and Adam. The solutions become slightly more accurate and similar with higher n_{TP}. However both weight initialisation methods can not compete with the combination mTSM and Adam.

Now for mTSM and vBP as displayed by the last row in the figure, we find stable results for all n_{TP}, which is in sharp contrast to TSM and vBP. Again, Δu_{deter} behaves like the other computations for mTSM, and we find similarities in the overall behaviour of Δu_{rnd} compared to TSM and cBP. Increasing n_{TP} leads to slightly better approximations, while the difference between the best and the least good approximation grows.

Turning to Table 6, we now discuss the stochastic quantities for $\mathbf{p}_{rnd}^{init}$, related to the results shown in Fig. 12 and 13. We focus on the results for random weight initialisation as the diagrams have shown the deterministic weight initialisation to always return the same numerical error. The 1e5 complete computations (optimisations) should sufficiently support the meaning of the analysed data.

Regarding the mean value, Adam has the overall smallest value and seems to be the best choice. However, for TSM and $n_{TP} = 40$, cBP almost equals Adam with vBP outperforming Adam in this specific setting. That result is particular interesting, since

method	training points	optimiser	mean value	standard deviation	10%-quantile	20%-quantile	30%-quantile
TSM	$n_{TP} = 10$	Adam	2.70e-4	1.89e-4	9.54e-5	1.31e-4	1.66e-4
		cBP	8.79e-4	5.56e-5	8.16e-4	8.40e-4	8.54e-4
		vBP	8.89e-2	2.32e-1	6.32e-4	7.09e-4	8.05e-4
	$n_{TP} = 20$	Adam	5.49e-4	2.38e-4	1.88e-4	2.80e-4	3.91e-4
		cBP	6.07e-4	1.11e-4	4.48e-4	5.08e-4	5.55e-4
		vBP	4.78e-2	1.44e-1	4.04e-4	4.64e-4	4.70e-4
	$n_{TP} = 40$	Adam	3.55e-4	1.86e-4	1.40e-4	1.96e-4	2.49e-4
		cBP	3.60e-4	1.69e-4	1.64e-4	2.06e-4	2.46e-4
		vBP	1.54e-4	3.31e-5	1.10e-4	1.22e-4	1.34e-4
mTSM	$n_{TP} = 10$	Adam	1.24e-4	2.51e-5	1.06e-4	1.10e-4	1.13e-4
		cBP	2.01e-3	4.45e-4	1.64e-3	1.77e-3	1.89e-3
		vBP	1.24e-3	1.04e-4	1.18e-3	1.23e-3	1.25e-3
	$n_{TP} = 20$	Adam	5.01e-5	1.51e-5	3.35e-5	3.75e-5	4.11e-5
		cBP	7.46e-4	1.67e-4	5.73e-4	6.02e-4	6.24e-4
		vBP	4.02e-4	1.11e-4	2.72e-4	3.10e-4	3.40e-4
	$n_{TP} = 40$	Adam	5.67e-5	1.30e-5	4.76e-5	5.03e-5	5.23e-5
		cBP	3.22e-4	7.26e-5	2.53e-4	2.70e-4	2.79e-4
		vBP	2.29e-4	6.28e-5	1.70e-4	1.83e-4	1.93e-4

Table 6: **Experiment 2.4.8.** Optimiser comparison (part 3), quantitative data for Δu_{rnd}.

vBP shows for TSM and both $n_{TP} = 10$ and $n_{TP} = 20$ very limited approximations. In contrast to TSM, Adam dominates for mTSM the lowest mean value without any exception.

The former statement however does only hold partially when it comes to the standard deviation. Here, TSM seems to favour cBP over Adam, again with vBP for $n_{TP} = 40$ to pass downwards. Excluding vBP for $n_{TP} = 10$ and $n_{TP} = 20$, the standard deviation in the other cases are in an acceptable range. That is, the mean value and standard deviation could be suitable when evaluating stability and reliability. However, it can be difficult to specify the term reliability. The lower the numerical error, the better the approximation. Nonetheless, defining a threshold needs justification and discussion on how the neural network methods behave compared to standard numerical algorithms like Runge-Kutta 4.

We also take different quantiles (10%,20%,30%) into account. The percentage specifies the relative amount of data points which appear below the quantile value itself. Although several minima for TSM and vBP in Fig. 10(g),(h) appear to be less useful than the lowest one, all quantiles for these cases are better than for the same settings with cBP. The situation for mTSM is the same, while Adam outperforms both cBP and vBP in this context. Therefore one may find that further adjusting the optimisation parameters for vBP can in general lead to perform better than cBP. However, it is questionable if this would also perform better than Adam. We find all quantiles values to be good in case of Adam optimisation. In this sense, we consider Adam here as the most reliable optimiser.

Concluding, the overall best performance related to the numeric error shows mTSM and Adam for both $\mathbf{p}_{deter}^{init}$ and $\mathbf{p}_{rnd}^{init}$. Although TSM and vBP appear to have some stability flaws for smaller n_{TP}, it stabilises for $n_{TP} = 40$. Overall, both vBP and cBP can not compete with Adam and mTSM. However, this conclusion is made based on the results in this section. As we have seen in the very beginning, BFGS was performing even better than Adam so that basically more experiments are necessary to make a final conclusion.

2.4.9 Conclusion

When solving the stiff model IVP with feedforward neural networks, the solution reliability depends on a variety of parameters. We find the weight initialisation to have a major influence. While the initialisation with zeros does not provide reasonable approximations for TSM with one hidden layer, it is capable to work reasonably well for mTSM. First setting the weights to small random values shows the best results with Adam and mTSM, although the use of more training points may yield less suitable results. This may indicate an overfitting and could be resolved by employing more neurons or other adjustments. This is a possible subject for a future study.

However, our work also indicates that all the investigated issues may have to be considered together as a complete package, e.g, the investigated aspects may not be evaluated completely independent of each other. Even after a detailed investigation as provided here it seems not to be possible to single out an individual aspect that dominates the overall accuracy and reliability.

We tend to favour the combination of Adam and mTSM in further computationally oriented research, since it provides the best approximations for both weight initialisation methods. Future research may also include theoretical work, e.g., on sensitivity and different neural forms for TSM. One main goal in this context is to decrease the variation of possible solutions together with an increase of the solution accuracy.

Moreover, our third experiment has shown that it may make sense to investigate deep networks, since these could result in a significant accuracy gain, reminding of higher order effects in classic numerical analysis.

Furthermore, our future work will include more difficult differential equations with a focus on initial value problems and the improvement of deterministic weight initialisation.

3 (Subdomain) Collocation polynomial neural forms for solving initial value problems

Motivated by the construction principle of collocation methods in numerical analysis, we propose here a novel extension of the classic neural forms approach. Our extension is based on the observation, that the neural form using one feedforward neural network as employed by Lagaris et al. [13] may be interpreted as a first order collocation polynomial. The novel collocation-type construction includes several feedforward neural networks, one for each order. Compared to a collocation method from standard numerics, the networks take on the role of coefficients in the collocation polynomial expansion. Furthermore, we aim to approximate initial value problems on fairly large domains. Therefore, and based on the NF structures, we also propose the domain segmentation extension, which splits the computational domain into subdomains. In each subdomain, we solve the initial value problem with a collocation polynomial neural form. This is done proceeding in time from one domain segment to the adjacent subdomain. The interfacing grid points in any subdomain provide the initial value for the next subdomain. The neural forms are solved on each subdomain, whereas the interfacing grid points overlap in order to provide initial values over the whole segmentation. We also illustrate in experiments that the combination of collocation neural forms of higher order and the domain segmentation allow to solve initial value problems over large domains with high accuracy and reliability.

3.1 The collocation neural forms approach

The classic NF in the TSM approach for an IVP $G = 0$, cf. Eq. (7), with the initial value $u(0) = u_0$ reads

$$\tilde{u}(t_i, \mathbf{p}) = A(t_i) + F(t_i, \mathbf{p}) \tag{91}$$

$$= u_0 + N(t_i, \mathbf{p})t_i \tag{92}$$

Compared to a first order polynomial

$$q_1(t_i) = a_0 + a_1 t_i \tag{93}$$

the structure of Eq. (92) and Eq. (93) inherent similarities. Motivated by the expansion of an m-th order collocation function polynomial [2]

$$q_m(t_i) = a_0 + \sum_{\kappa=1}^{m} a_\kappa t_i^\kappa \tag{94}$$

it appears natural to also expand the polynomial order of the classic NF. Considering a more general formulation of the given initial value, it is now represented by $u(t_0) = u_0$. Whereas setting $A(t_i)$ in Eq. (92) to equal the given initial value $u(t_0)$ still is a suitable choice, the neural network term transforms into

$$F(t_i, \mathbf{p}) \to F(t_i, \mathbf{P}_m) = \sum_{\kappa=1}^{m} N_\kappa(t_i, \mathbf{p}_\kappa)(t_i - t_0)^\kappa \tag{95}$$

This leads to the collocation polynomial neural form (CNF) for the TSM approach:

$$\tilde{u}_C(t_i, \mathbf{P}_m) = u(t_0) + \sum_{\kappa=1}^{m} N_\kappa(t_i, \mathbf{p}_\kappa)(t_i - t_0)^\kappa \tag{96}$$

This polynomial extension adds more flexibility to the approach. However, this is achieved in a different way than just increasing the number of hidden layer neurons in a single neural network, since additional networks arise that are multiplied by the factors $(t_i - t_0)^\kappa$ [82]. The weight vector in Eq. (96) is denoted by $\mathbf{p}_\kappa$ and the matrix $\mathbf{P}_m$ is defined over the m weight vectors as $\mathbf{P}_m = (\mathbf{p}_1, \ldots, \mathbf{p}_m)$.

The use of higher order powers of $(t_i - t_0)^\kappa$ as in Eq. (96) not only generalises previous methods, but may also enable better stability and accuracy properties. From the structural similarities to the polynomial in Eq. (94), the neural networks take on the roles of coefficient functions for the values of $(t_i - t_0)^\kappa$. It is important to mention that the new CNF construction in Eq. (96) still fulfills the initial condition.

The proposed appearance in Eq. (96) includes m neural networks, where $N_\kappa(t_i, \mathbf{p}_\kappa)$ represents the κ-th neural network

$$N_\kappa(t_i, \mathbf{p}_\kappa) = \sum_{j=1}^{H} \rho_{j,\kappa} \sigma(\nu_{j,\kappa} t_i + \eta_{j,\kappa}) + \gamma_k \tag{97}$$

The corresponding cost function is then given as in Eq. (14):

$$E[\mathbf{P}_{m,l}] = \frac{1}{2(n+1)} \sum_{i=0}^{n} \left\{ G\left(t_i, \tilde{u}_C(t_i, \mathbf{P}_m), \dot{\tilde{u}}_C(t_i, \mathbf{P}_m)\right) \right\}^2 \tag{98}$$

Turning to mTSM, the extension can be obtained in a similar way as found for TSM in Eq. (96):

$$\tilde{u}_C(t_i, \mathbf{P}_m) = N_1(t_i, \mathbf{p}_1) + \sum_{\kappa=2}^{m} N_\kappa(t_i, \mathbf{p}_\kappa)(t_i - t_0)^{\kappa-1} \tag{99}$$

Thereby the first neural network $N_1(t_i, \mathbf{p}_1)$ is set to learn the initial condition in the same way as stated in Eq. (29):

$$\begin{aligned} E[\mathbf{P}_{m,l}] = \frac{1}{2(n+1)} \sum_{i=0}^{n} \left\{ G\big(t_i, \tilde{u}_C(t_i, \mathbf{P}_m), \dot{\tilde{u}}_C(t_i, \mathbf{P}_m)\big) \right\}^2 \\ + \frac{1}{2} \left\{ N_1(t_0, \mathbf{p}_1) - u(t_0) \right\}^2 \end{aligned} \tag{100}$$

Now G in Eq. (98) and Eq. (100) shares the same structure as the general problem in Eq. (7). However, the original solution function $u(t)$ has been replaced by the CNF $\tilde{u}_C(t_i, \mathbf{P}_m)$. Therefore, G involved in the cost function now relies on one or more neural networks, depending on the neural forms order. From now on, the number of neural networks in the neural form will be referred to as the collocation polynomial neural form order m.

Considering other polynomials

A more general notation of the (TSM) neural network-related term may be given by

$$F(t_i, \mathbf{P}_m) = \sum_{\kappa=1}^{m} N_\kappa(t_i, \mathbf{p}_\kappa) T_\kappa(t_i - t_0) \tag{101}$$

which enables the usage of different polynomial expansions, different from $T_\kappa(t_i - t_0) = (t_i - t_0)^\kappa$. For example, Chebyshev polynomials of first kind are (recurrence definition):

$$T_1(t_i - t_0) = 1 \tag{102}$$
$$T_2(t_i - t_0) = t_i - t_0 \tag{103}$$
$$T_{\kappa+1}(t_i - t_0) = 2(t_i - t_0) T_\kappa(t_i - t_0) - T_{\kappa-1}(t_i - t_0) \tag{104}$$

One may also consider Legendre polynomials (recurrence definition):

$$T_1(t_i - t_0) = 1 \tag{105}$$
$$T_2(t_i - t_0) = t_i - t_0 \tag{106}$$
$$(\kappa + 1) T_{\kappa+1}(t_i - t_0) = (2\kappa + 1)(t_i - t_0) T_\kappa(t_i - t_0) - \kappa T_{\kappa-1}(t_i - t_0) \tag{107}$$

Please note, that those polynomials come with the drawback of incorporating terms independent of $(t_i - t_0)$ and therefore not fulfilling the condition of eliminating the impact of any neural network at the initial point $t_i = t_0$. However, due to the flexibility of the NF approach, it is possible simply add an additional factor in order to handle this issue:

$$F(t_i, \mathbf{P}_m) = (t_i - t_0) \sum_{\kappa=1}^{m} N_\kappa(t_i, \mathbf{p}_k) T_\kappa(t_i - t_0) \tag{108}$$

This ensures $F(t_i, \mathbf{P}_m)$ to become zero at the initial point but also makes the neural form more complex which may result in a complete different behaviour. On the other hand, one may only consider even orders for the above-mentioned polynomials. However, this work does focus on polynomials in form of $T_\kappa(t_i - t_0) = (t_i - t_0)^\kappa$.

3.2 The domain segmentation approach

The (classic) NF approaches use the IVP structure together with the given initial value in order to train the neural networks on a certain domain. An experimental study [60] has figured out, that especially TSM tends to struggle with approximating the solution on larger domains. However, on small domains the numerical error tends to remain small. Now, the previously introduced CNF incorporates the domain variable in $(t_i - t_0)^\kappa$, which effectively acts as a scaling of $N_\kappa(t_i, \mathbf{p}_\kappa)$. A large domain size variation may introduce the need for a significant higher amount of training points or the use of a more complex neural network architecture.

Therefore, this section introduces domain segmentation, a combination of the CNF and a technique that refines the computational domain. This approach may be interpreted as a second stage of discretising the domain. That is, the solution domain D is split into h equidistant subdomains D_l: symbolically

$$D \to D_l, \quad l = 1, \dots, h \tag{109}$$

with $n+1$ equidistant grid points $t_{i,l}$ in each subdomain. The CNF now transforms into the subdomain collocation neural form (SCNF): symbolically

$$\tilde{u}_C(t_i, \mathbf{P}_m) \to \tilde{u}_C(t_{i,l}, \mathbf{P}_{m,l}) \tag{110}$$

and is solved separately in each domain fragment. The computation starts in D_1, the leftmost subdomain, since the initial value $u(t_0) = u_0$ is given there. The interfacing grid points in each subdomain overlap, whereby the computed SCNF value $\tilde{u}_C(t_{n,l-1}, \mathbf{P}_{m,l-1})$ at the last grid point of any subdomain D_{l-1} is set to be the new initial value $\tilde{u}_C(t_{0,l}, \mathbf{P}_{m,l})$ for the next subdomain D_l. Therefore the SCNF also holds the characteristic of satisfying the given/computed initial value by construction, namely

$$\tilde{u}_C(t_{i,l}, \mathbf{P}_{m,l}) = \tilde{u}_C(t_{0,l}, \mathbf{P}_{m,l}) + \sum_{\kappa=1}^{m} N_\kappa(t_{i,l}, \mathbf{p}_{\kappa,l})(t_{i,l} - t_{0,l})^\kappa \tag{111}$$

Since the cost function construction still requires the SCNF time derivative, it can be analytically retrieved as

$$\dot{\tilde{u}}_C(t_{i,l}, \mathbf{P}_{m,l}) = \sum_{\kappa=1}^{m} \left[\dot{N}_\kappa(t_{i,l}, \mathbf{p}_{\kappa,l})(t_{i,l} - t_{0,l})^\kappa + N_\kappa(t_{i,l}, \mathbf{p}_{\kappa,l})\kappa(t_{i,l} - t_{0,l})^{\kappa-1} \right] \tag{112}$$

The general idea of domain segmentation is visualised in Fig. 14, where the black/vertical marks represent the equidistantly distributed subdomain boundaries for the solution of an example IVP.

The neural networks that are now scaled by $(t_{i,l} - t_{0,l})^\kappa$, may in fact avoid higher scaling factors, depending on the subdomain size. The arising cost function, similar to Eq. (98), is

$$E_l[\mathbf{P}_{m,l}] = \frac{1}{2(n+1)} \sum_{i=0}^{n} \left\{ G\left(t_{i,l}, \tilde{u}_C(t_{i,l}, \mathbf{P}_{m,l}), \dot{\tilde{u}}_C(t_{i,l}, \mathbf{P}_{m,l})\right) \right\}^2 \tag{113}$$

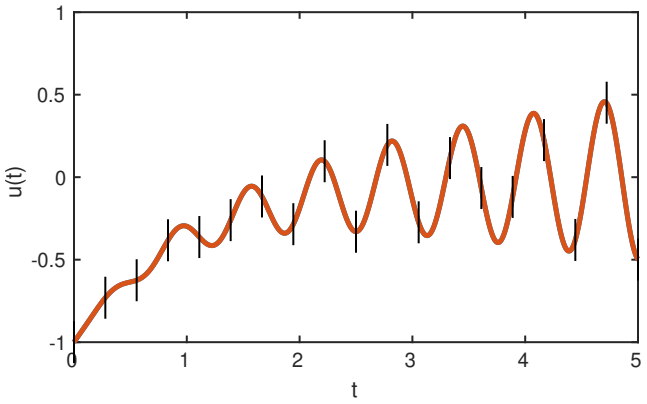


Figure 14: SCNF domain segmentation example for the IVP $\dot{u}(t) - t\sin(10t) + u(t) = 0, \quad u(0) = -1$ (later discussed) with fixed and equidistant subdomains, (orange) analytical IVP solution, (black/marked) subdomain boundaries.

Proceeding to mTSM, it is also possible to adopt the CNF approach and to set the first neural network to learn the new initial value in each subdomain. That is, the SCNF for mTSM now reads

$$\tilde{u}_C(t_{i,l}, \mathbf{P}_{m,l}) = N_1(t_{i,l}, \mathbf{p}_{1,l}) + \sum_{\kappa=2}^{m} N_\kappa(t_{i,l}, \mathbf{p}_{\kappa,l})(t_{i,l} - t_{0,l})^{\kappa-1} \tag{114}$$

with its time derivative

$$\dot{\tilde{u}}_C(t_{i,l}, \mathbf{P}_{m,l}) = \dot{N}_1(t_{i,l}, \mathbf{p}_{1,l}) + \sum_{\kappa=2}^{m} \Big[\dot{N}_\kappa(t_{i,l}, \mathbf{p}_{\kappa,l})(t_{i,l} - t_{0,l})^{\kappa-1} + N_\kappa(t_{i,l}, \mathbf{p}_{\kappa,l})(\kappa - 1)(t_{i,l} - t_{0,l})^{\kappa-2} \Big] \tag{115}$$

The corresponding cost function follows

$$E_l[\mathbf{P}_{m,l}] = \frac{1}{2(n+1)} \sum_{i=0}^{n} \left\{ G\left(t_{i,l}, \tilde{u}_C(t_{i,l}, \mathbf{P}_{m,l}), \dot{\tilde{u}}_C(t_{i,l}, \mathbf{P}_{m,l})\right) \right\}^2 + \frac{1}{2} \left\{ N_1(t_{0,l}, \mathbf{p}_{1,l}) - \tilde{u}_C(t_{0,l}, \mathbf{P}_{m,l}) \right\}^2 \tag{116}$$

Once trained, each subdomain has its unique learned weight matrix $\mathbf{P}_{m,l}$, which can later be used to recreate the solution or evaluate the solution at grid points intermediate to the training points.

In order to keep the overview of all terms and indices, a summary reads: The i-th grid point in the l-th subdomain is denoted by $t_{i,l}$, while $t_{0,l}$ is the initial point in the subdomain D_l with the initial value $\tilde{u}_C(t_{0,l}, \mathbf{P}_{m,l})$. That is, $t_{n,l-1}$ and $t_{0,l}$ are overlapping grid points. In D_1, $\tilde{u}_C(t_{0,1}, \mathbf{P}_{m,1}) = u(t_0)$ holds. The matrix $\mathbf{P}_{m,l}$ contains the set of the m neural network weight vectors in the corresponding subdomain l. Finally, $N_\kappa(t_{i,l}, \mathbf{p}_{\kappa,l})$ denotes the κ-th neural network in D_l.

If G in, e.g., Eq. (113) represents a system of o IVPs, each solution function requires its own NF and the cost function derives from the sum over o separate ℓ_2-norm terms, one for each equation involved, cf. Eq. (19).

3.3 Computational results for CNF and SCNF

This section is divided into experiments on the collocation polynomial neural form (CNF), followed by experiments on the subdomain collocation polynomial neural form (SCNF). Prior to this, we will provide detailed information about how the weight initialisation for the different neural networks are realised. The discussion of the deterministic weight initialisation is also one of the main subjects in the experimental section. As stated before, the specific neural network configurations will be addressed in the subsequent experiments.

No.	$\Delta u(\mathbf{p}_{rnd}^{init})$	$\Delta u(\mathbf{p}_{deter}^{init})$
1	5.7148e-6	2.6653e-6
2	7.5397e-6	2.6653e-6
3	3.7249e-5	2.6653e-6
4	1.1894e-5	2.6653e-6
5	7.7956e-6	2.6653e-6

Table 7: Results for five different realisations during optimisation (mTSM, $m = 2$).

Weight initialisation with $\mathbf{p}_{deter}^{init}$ applies to all corresponding neural networks so that they use the same initial values. Increasing the neural forms order m for the initialisation with $\mathbf{p}_{rnd}^{init}$ works systematically. For $m = 1$, a set of random weights for the neural network is generated. For $m = 2$ (now with two neural networks), the first neural network is again initialised with the generated weights from $m = 1$, while for neural network number two, a new set of weights is generated. This holds for all m for higher orders, subsequently, in all experiments. To achieve comparability, the same random initialised weights are used in all experiments. We use the Adam optimiser here, for details see Sec. 2.3.1. Let us recall the choice of initial weight values in some detail as it is important for the upcoming experiments. The weight initialisation plays an important role and determines the starting point for gradient descent. Poorly chosen, the optimisation method may fail to find a suitable local minimum. The initial neural network weights are commonly chosen as small random values [102]. Let us note that this is sometimes considered as a computational characteristic of the stochastic gradient descent optimisation. Another option is to choose the initialisation to be deterministic. This method is not commonly used for the optimisation of neural networks since random weight initialisation may lead to better results (stochastic aspect). However, initialising the weights with equal values may also return reliable results of reasonable quality if the computational parameters in the network remain unchanged. As previous experiments have documented [60, 13, 15], both TSM and mTSM are able to solve differential equations up to a certain degree of accuracy. However, an example illustrating the accuracy of five computations with random $\mathbf{p}_{rnd}^{init}$ and deterministic weights $\mathbf{p}_{deter}^{init}$ shows that the quality of approximations may vary considerably, see Table 7. As observed in many experiments, even a small discrepancy in the initialisation with several sets of random weights in the same range, may lead to a significant difference in accuracy. On the other hand, the network initialisation with deterministic/equal values very often gives reliable results by the proposed novel approach. This motivates us to study in detail the effects of a deterministic network initialisation under the consideration of a first order optimisation method.

3.3.1 Experiments on the collocation polynomial neural form (CNF)

In this section, we want to test our novel CNF approach (see Sec. 3) with the initial value problem

$$\dot{u}(t) + 5u(t) = 0, \quad u(0) = 1 \tag{117}$$

which has the analytical solution $u(t) = e^{-5t}$ and is solved over the entire domain $D = [0, 2]$ (without domain segmentation). The Eq. (117) involves a damping mechanism, making this a simple model for stiff phenomena [98].

The numerical error Δu shown in subsequent diagrams is again defined as the l_1-norm (l_1-error) of the difference between the analytical solution and the corresponding CNF

$$\Delta u = \frac{1}{n+1} \sum_{i=0}^{n} \left| u(t_i) - \tilde{u}(t_i, \mathbf{p}) \right| \tag{118}$$

If we do not say otherwise, the fixed computational parameters in the subsequent experiments are: 1 input layer bias, 1 hidden layer with 5 sigmoid neurons, 1e5 training epochs, 10 training points, $D = [0, 2]$ and the weight initialisation values which are $\mathbf{p}_{deter}^{init} = -10$ and $\mathbf{p}_{rnd}^{init} \in [-10.5, -9.5]$. These values may seem arbitrarily chosen, but we found them to work well for both TSM and mTSM so that a useful comparison is possible.

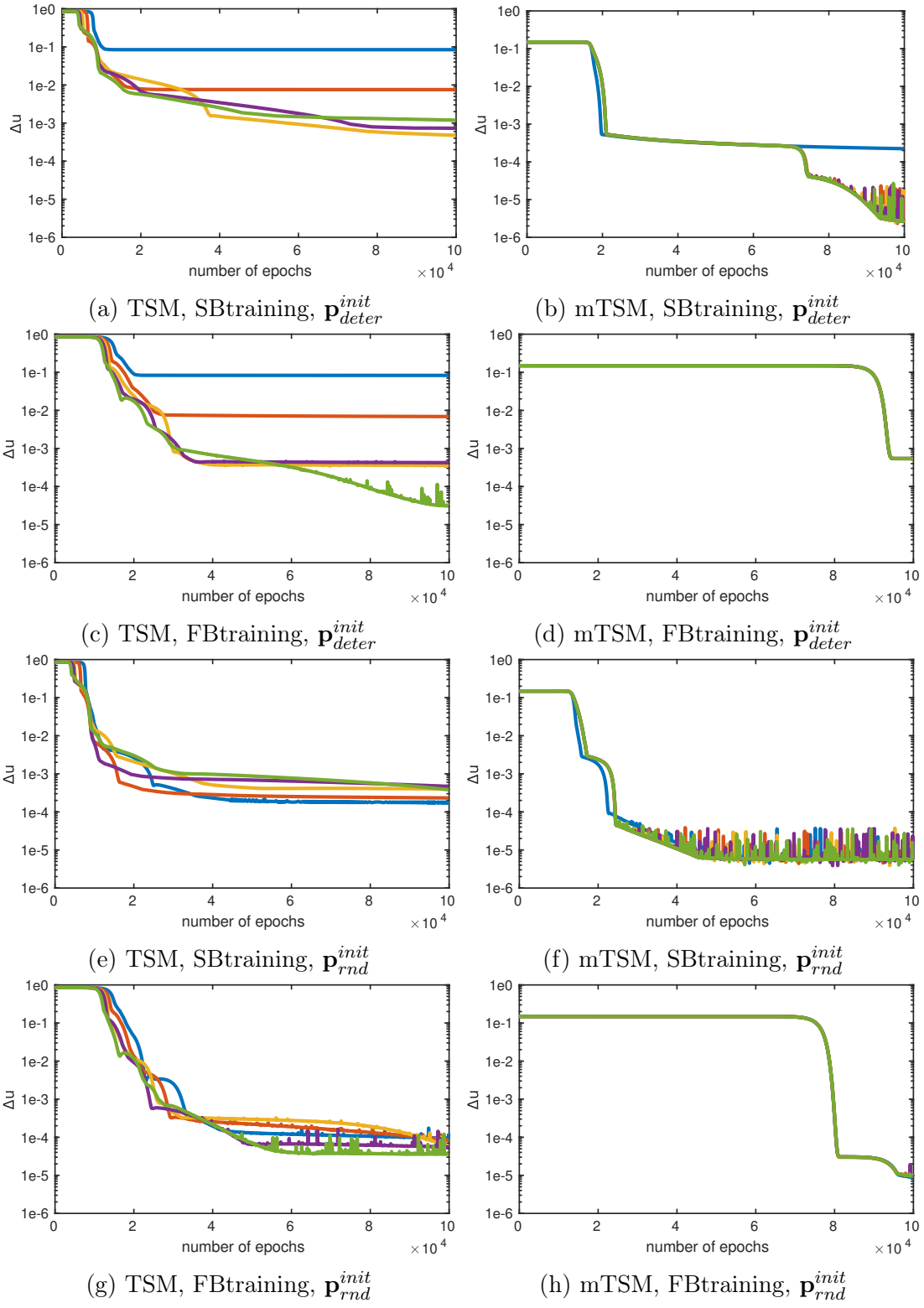
3.3.2 CNF Experiment: number of training epochs

t_i	$\Delta u(t_i)$ (TSM)	$\Delta u(t_i)$ (RK4)
0.00	0.0000e0	0.0000e0
0.22	3.8158e-5	0.1769e0
0.44	3.5101e-5	0.1478e0
0.66	1.4318e-5	9.4013e-2
0.88	1.2001e-5	5.3900e-2
1.11	4.5407e-5	2.9361e-2
1.33	5.2069e-6	1.5546e-2
1.55	6.6105e-5	8.0942e-3
1.77	1.2052e-5	4.1712e-3
2.00	7.9787e-5	2.1357e-3

Table 8: Numerical error comparison at individual grid points with $m = 5$, $\mathbf{p}_{deter}^{init}$ and an equal amount of grid points.

The first experiment in Fig. 15 shows for different orders m of the neural form, how the numerical error Δu behaves depending on the number of training epochs. The diagrams only display every hundredth data point and investigate various combinations for TSM and mTSM together with single batch training (SBtraining), full batch training (FBtraining), $\mathbf{p}_{deter}^{init}$ and $\mathbf{p}_{rnd}^{init}$. Let us recall, that for SBtraining, the weights updates are performed individually after each training point processing, while FBtraining averages the gradient and updates the weights only once per epoch.

The visual differences shown in Fig. 15 are remarkable. Results in Fig. 15(d),(h) appear to be independent of the neural forms order m, which seems to be characteristic



(a) TSM, SBtraining, $\mathbf{p}^{init}_{deter}$

(b) mTSM, SBtraining, $\mathbf{p}^{init}_{deter}$

(c) TSM, FBtraining, $\mathbf{p}^{init}_{deter}$

(d) mTSM, FBtraining, $\mathbf{p}^{init}_{deter}$

(e) TSM, SBtraining, $\mathbf{p}^{init}_{rnd}$

(f) mTSM, SBtraining, $\mathbf{p}^{init}_{rnd}$

(g) TSM, FBtraining, $\mathbf{p}^{init}_{rnd}$

(h) mTSM, FBtraining, $\mathbf{p}^{init}_{rnd}$

Figure 15: **Experiment in 3.3.2.** Number of training epochs, (blue) $m = 1$, (orange) $m = 2$, (yellow) $m = 3$, (purple) $m = 4$, (green) $m = 5$.

of the specific combination of mTSM and FBtraining. Both require many epochs of training before they begin to approximate the IVP solution. Here we have an example,

where the random initialisation still outperforms the deterministic initialisation. On the other hand, we find that mTSM and SBtraining do not have such characteristics in common. There is Fig. 15(f) with a relatively fast accuracy improvement during the training process (using $\mathbf{p}_{rnd}^{init}$) but with heavy oscillation. This is most likely an indication of a local minimum which has been found by Adam, since at some point Adam tends to oscillate around a minimum. Nonetheless, the results here are in range of 1e-6 which can be considered as useful and reliable. Turning to the deterministic initialised counterpart in Fig. 15(b), the differences between each order m are quiet insignificant in the beginning. After tens of thousands of epochs, we observe that only the transition from $m = 1$ (blue) to $m = 2$ (orange) affects Δu with increasing accuracy, while heavy oscillations start to occur again. In case of best performance related to mTSM, we find that $\mathbf{p}_{deter}^{init}$ does indeed provide the best results here, although the results between Fig. 15(b) and Fig. 15(f) are minor. Nonetheless, the results have already shown the potential of the collocation polynomial neural forms. However, already $m = 1$ converges to a solution accuracy that can be considered reliable or useful.

Now we want to evaluate the results related to TSM. As mentioned before, this method had the least favourable results in the investigation of the computational characteristics (under the employed parameter setup). In general, dependencies can be observed regarding the different weight initialisation methods. That is, the overall behaviour shows similarities on the one hand related to $\mathbf{p}_{deter}^{init}$ and on the other hand regarding $\mathbf{p}_{rnd}^{init}$. In Fig. 15(a) with TSM and the deterministic weight initialisation $\mathbf{p}_{deter}^{init}$, results for $m = 1$ (blue) do not provide any useful approximation, independent of the batch training method selected. This confirms the results from the previous experimental section on computational characteristics. Let us recall, that there was not a single indication for TSM in combination with $\mathbf{p}_{deter}^{init}$ to provide any useful result. However, with a second polynomial term $m = 2$ in the neural form (orange), Δu approximately lowers by one order of magnitude so that we now obtain a solution which can be considered to rank at the lower end of reliability. The combination of TSM, SBtraining and $\mathbf{p}_{deter}^{init}$ shows best results for $m = 3$, as orders $m = 4, 5$ downgrade the accuracy again.

Switching the training method from single batch to full batch training, the first three orders show almost no remarkable accuracy-related differences. The most interesting result in Fig. 15(c) is $m = 5$ (green) with the best accuracy at the end of the training process. The oscillation once more indicate that Adam has found a local minimum. We find that several neural networks with polynomial orders in the domain variable $(t_i - t_0)^\kappa$ may restore the characteristic of random initialisation which was lost by initialising the neural networks in a deterministic way. In other words, while $\mathbf{p}_{deter}^{init}$ effectively reduces the hidden layer to one neuron, additional small neural networks that are scaled by $(t_i - t_0)^\kappa, \kappa = 1, \ldots, m$ appear to act as individual neural networks with (effectively) one hidden neuron. In total, we assume that the scaled neural networks cooperate in order to improve the solution accuracy. However, these statements tend to last only regarding TSM, as we have seen that mTSM related CNF show completely different characteristics.

In the diagrams showing TSM and $\mathbf{p}_{rnd}^{init}$, m has only minor impact on the accuracy, as 15(e),(g) show the same general trend for both training methods. However, single batch training here does not benefit from any order above $m = 1$ (classic neural form) and this is in fact a unique behaviour compared to the other results.

For full batch training, at the end of the training process, again raising the neural form order has a positive but minor impact on the numerical error.

Table 8 shows the numerical error $\Delta u(t_i)$ at the individual and equidistant grid points t_i. For both CNF and Runge-Kutta 4 (RK4), the results were computed with ten grid points, resulting in the CNF approach with $\mathbf{p}_{deter}^{init}$ performing better over RK4. The amount of training points was chosen to somehow achieve a comparable parameter setup. However, further refining the grid for RK4 will result in significantly lower $\Delta u(t_i)$. We also find this comparison to only be interesting related to the grid points. In total, while RK4 evaluates the right-hand side of an IVP for each grid point at three different steps, the neural forms approach evaluates the IVP at ten grid points in each epoch. Therefore, a fair comparison should perhaps consider as many grid points for RK4 as there are epochs for TSM. In the end, RK4 will most likely outperform TSM.

Concluding this experiment, we were able to achieve equal or sometimes even better results with deterministic weight initialisation $\mathbf{p}_{deter}^{init}$ over $\mathbf{p}_{rnd}^{init}$. Increasing m to at least order five seems to be a good option for TSM and FBtraining, whereas further raising m may provide even better approximations. For mTSM we can not observe benefits for m above order 2. Moreover, we see especially that the increase in the order of the neural form in (96) appears to have a similar impact on solution accuracy as the discretisation order in classical numerical analysis.

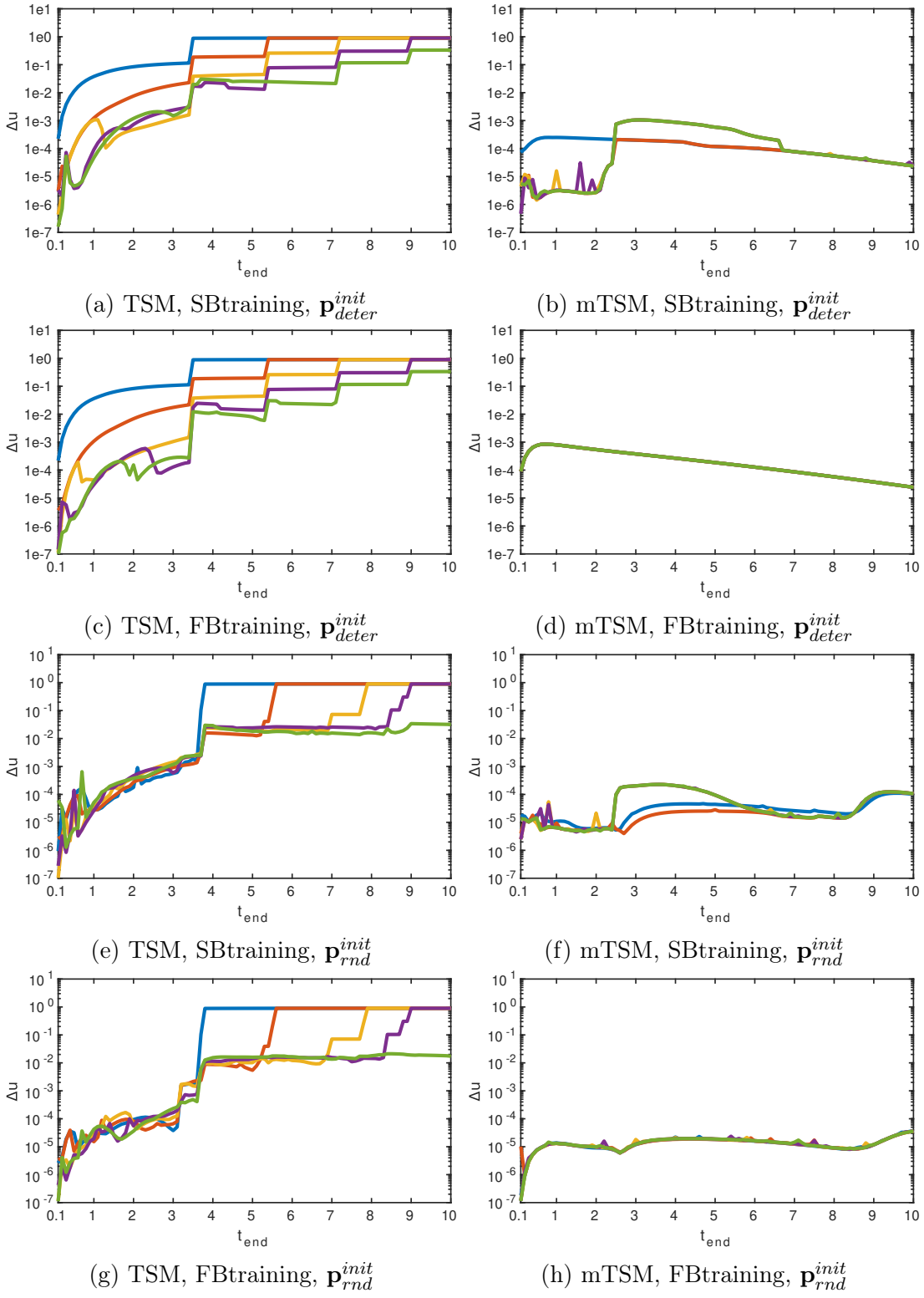
3.3.3 CNF Experiment: domain size variation

Investigating the methods concerning different domain sizes provides information on the reliability of computations on larger domains. The domains in this experiment read as $D = [0, t_{end}]$ and we directly compare in this experiment $\mathbf{p}_{deter}^{init}$ with $\mathbf{p}_{rnd}^{init}$.

In Fig. 16, we observe TSM from around $t_{end} = 3.5$ to incrementally plateau to unreliable approximations. Increasing m improves Δu on small domains and shifts the observable step-like accuracy degeneration towards larger domains. However, even with $m = 5$ (green) the results starting from domain size $t_{end} = 3.5$ towards larger sizes are unreliable. Previous to the first plateau higher m provide significant better Δu for the deterministic initialisation $\mathbf{p}_{deter}^{init}$, while there are only minor changes for $\mathbf{p}_{rnd}^{init}$ for the TSM method. This holds for both SBtraining and FBtraining, and one can say that in this experiment TSM works better with $\mathbf{p}_{rnd}^{init}$, even without increasing m.

Turning to the mTSM extension, we observe in Fig. 16(b) with $\mathbf{p}_{deter}^{init}$ and SBtraining the existence of a certain point from where different m return equal values, whereas FBtraining returns (close to) equal results for all the investigated domain sizes in Fig. 16(h). However, we see some evidence for the use of $m = 2$ (orange) over $m = 1$ (blue) to show an overall good performance. A further increase of m is not necessary with this approach, confirming results from Experiment 3.3.2. The latter proved the characteristic of mTSM and FBtraining to be independent of m and this is (mostly) confirmed here as well. Although the random initialisation for mTSM in Fig. 16(h) shows minor differences, we believe that these are very insignificant and based on, e.g., rounding errors.

Let us finally mention a very remarkable characteristic regarding mTSM, which is related the overall behaviour of $\mathbf{p}_{deter}^{init}$ in Fig. 16(b),(d). For both training methods, the numerical error lowers towards the end and appears to not have saturated, even for domains around $D = [0, 10]$. Especially FBtraining has a highly decreasing numerical



(a) TSM, SBtraining, $\mathbf{p}^{init}_{deter}$

(b) mTSM, SBtraining, $\mathbf{p}^{init}_{deter}$

(c) TSM, FBtraining, $\mathbf{p}^{init}_{deter}$

(d) mTSM, FBtraining, $\mathbf{p}^{init}_{deter}$

(e) TSM, SBtraining, $\mathbf{p}^{init}_{rnd}$

(f) mTSM, SBtraining, $\mathbf{p}^{init}_{rnd}$

(g) TSM, FBtraining, $\mathbf{p}^{init}_{rnd}$

(h) mTSM, FBtraining, $\mathbf{p}^{init}_{rnd}$

Figure 16: **Experiment in 3.3.3.** Domain size variation, (blue) $m = 1$, (orange) $m = 2$, (yellow) $m = 3$, (purple) $m = 4$, (green) $m = 5$.

error attached and a local maximum for smaller domains.

Concluding this experiment, TSM overall struggles with larger domains, as it shows in all configurations undesirable results for domains larger than $D = [0, 3.5]$. On the

other hand, mTSM again provides useful and reliable results for the chosen parameter setup. To mention the latter is very important, since minor changes or even using BFGS for examples, may completely change the behaviour. However, these characteristics are very important to investigate.

3.3.4 CNF Experiment: number of training points variation

The behaviour of numerical methods highly depend on the chosen amount of grid points, so that in this experiment we analogously investigate the influence of the number of training points (n_{TP}). In every computation, the domain D is discretised by equidistant grid points.

As in the previous experiments, m shows a major influence on the results with TSM, and the best approximations are provided by $\mathbf{p}_{deter}^{init}$ with $m = 5$ (green) as seen in Fig. 17(c). An interesting behaviour (observed also in a different context, e.g, in Fig. 15(c)) is the equivalence between $m = 3$ (yellow) and $m = 4$ (purple). Both converge to almost exactly the same Δu, where one may assume a saturation for the m. However, another increase of the order decreases the numerical error again by one order of accuracy. However, this mainly holds for TSM, FBtraining and the deterministic initialisation. Both TSM and mTSM together with FBtraining and $\mathbf{p}_{rnd}^{init}$ show an almost constant behaviour from $n_{TP} = 10$ on.

Turning to mTSM with $\mathbf{p}_{deter}^{init}$ and SBtraining in Fig. 17(a) we again find a major increase in accuracy after a transition from $m = 1$ (blue) to $m = 2$ around $n_{TP} = 10$. However, the behaviour towards higher n_{TP} is interesting, as for $m = 1$ (the classic neural form) the numerical error lowers as expected, but higher orders show continuously less good results. Increasing the order tends to be beneficial for smaller n_{TP}, while its meaning decreases towards higher n_{TP}.

Concluding this experiment, we again find evidence that increasing m in the proposed approach provides an improved accuracy for $\mathbf{p}_{deter}^{init}$. However, increasing n_{TP} seems not to improve the accuracy from a certain point on for every combination, unlike for numerical methods. But one could argue, that the analogy between the number of grid points for numerical methods here is the number of epochs.

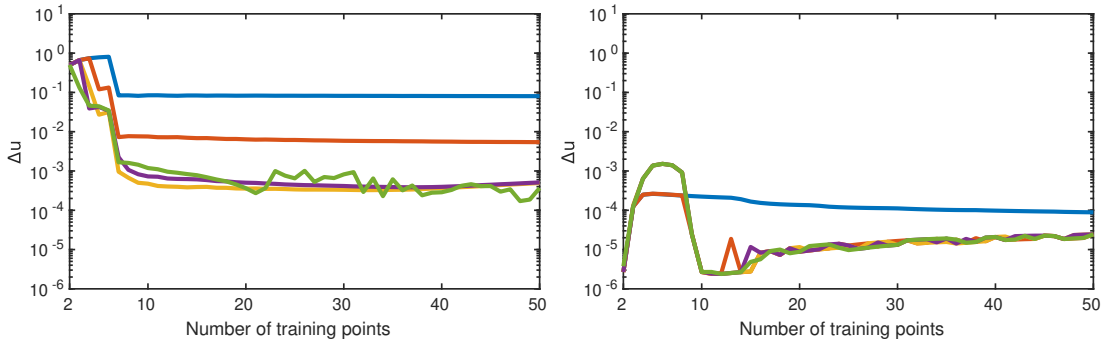
3.4 Experiments on the subdomain polynomial collocation neural form (SCNF)

In Section 3.3.1, while the test equation is stiff, its solution function is at the same time very smooth and the equation is solved on a small domain. However, Fig. 16 in Experiment 3.3.3, shows that TSM does not provide reliable solutions on larger domains. Hence, we want to show that the novel SCNF approach is able to work even on a fairly large domain with a different initial value problem. Therefore we use the following test equation

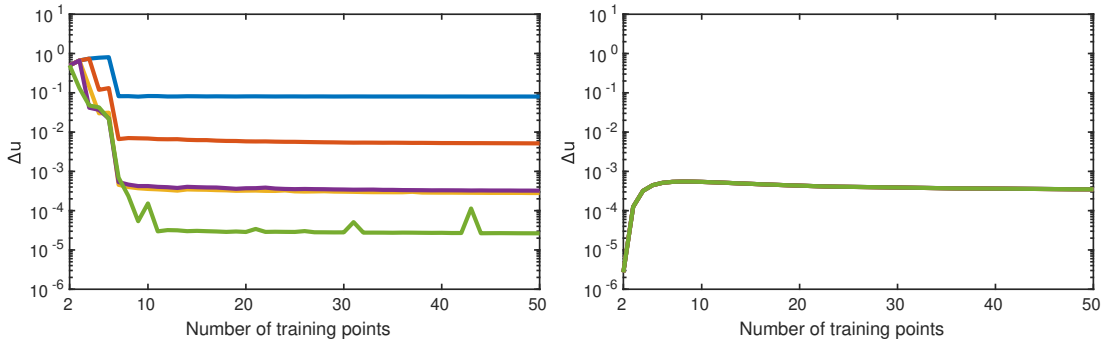
$$\dot{u}(t) - t\sin(10t) + u(t) = 0, \quad u(0) = -1 \tag{119}$$

with the analytical solution

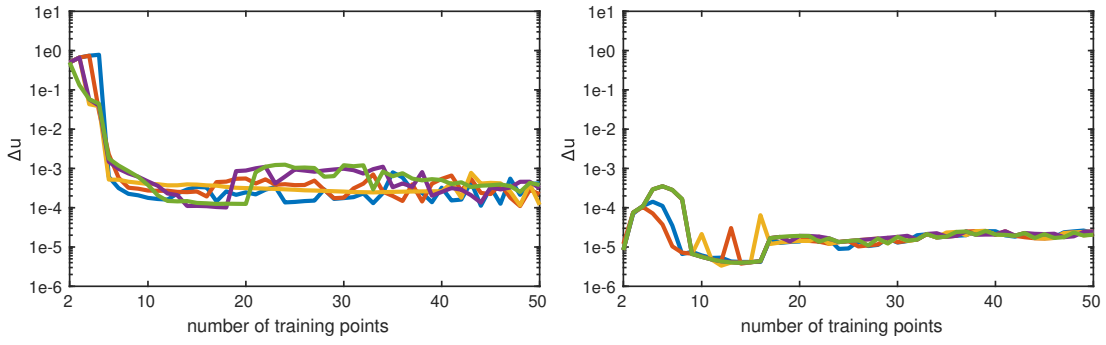
$$u(t) = \sin(10t)\left(\frac{99}{10201} + \frac{t}{101}\right) + \cos(10t)\left(\frac{20}{10201} - \frac{10t}{101}\right) - \frac{10221}{10201}e^{-t} \tag{120}$$



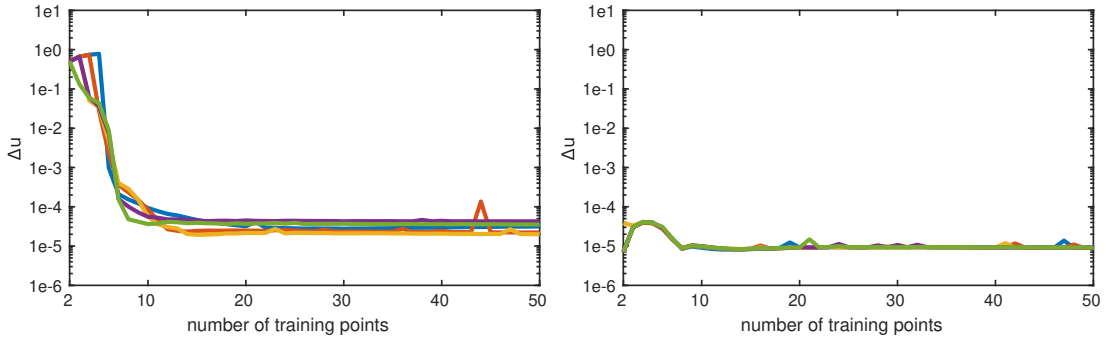
(a) TSM, SBtraining, $\mathbf{p}_{deter}^{init}$ (b) mTSM, SBtraining, $\mathbf{p}_{deter}^{init}$



(c) TSM, FBtraining, $\mathbf{p}_{deter}^{init}$ (d) mTSM, FBtraining, $\mathbf{p}_{deter}^{init}$



(e) TSM, SBtraining, $\mathbf{p}_{rnd}^{init}$ (f) mTSM, SBtraining, $\mathbf{p}_{rnd}^{init}$



(g) TSM, FBtraining, $\mathbf{p}_{rnd}^{init}$ (h) mTSM, FBtraining, $\mathbf{p}_{rnd}^{init}$

Figure 17: **Experiment in 3.3.4.** Number of training points variation, (blue) $m = 1$, (orange) $m = 2$, (yellow) $m = 3$, (purple) $m = 4$, (green) $m = 5$.

The solution is shown in Fig. 18 for $t \in [0, 15]$ and incorporates heavily oscillating and increasing characteristics, similar to instabilities. Although our approach is not limited

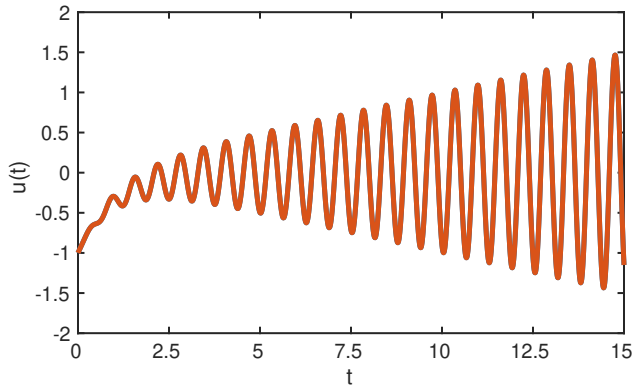


Figure 18: Analytical solution for initial value problem (119).

to certain types of IVPs, we find Eq. (119) to represent possible real-world behaviour and find it suitable to serve as an example IVP.

The weight initialisation works as employed in Section 3.3 and the values are fixed to $\mathbf{p}^{init}_{deter} = 0$ and $\mathbf{p}^{init}_{rnd} \in [-0.5, 0.5]$. In the subsequent experiments, the solution domain is kept constant to $D = [0, 15]$ and the neural networks are training with 1e5 epochs.

In addition we use the method of training the neural networks incrementally which has been employed in [15]. That is, we initially train the neural networks for the first grid point, afterwards for the first two grid points. We continue the procedure up to a FBtraining of all grid points in each subdomain. The initial weight initialisation is the same in each subdomain.

Please note at this point, that we provide an explicit comparison to the Runge-Kutta 4 method only in the last experiment of this section. We find that our approach provides by construction a very high flexibility, to deal with many types of initial value problems that may require specific constructions in classic numerical analysis (e.g. by symplectic integration). However, in simple settings we will usually find the neural network based approach (with first order optimisation) at the moment to be not competitive to numerical state-of-the-art solvers (w.r.t. computational efficiency) which have been developed and refined over decades. We think that because of the much higher flexibility of the network based tool, this comparison would not be entirely adequate, considering the employed optimisation methods.

As an example for the flexibility, a recent neural network approach [61] makes it superficial to restart the computation of numerical solutions when considering multiple, different initial conditions. We also demonstrate the flexibility of the approach in Section 3.4.5 and show how invariants can be simply added to the cost function. Regarding numerical methods for handling those problems, special constructions are often needed.

In our test example in Eq. (119), we find a graphical comparison in the subsequent experiments to not provide further information since the visual differences between analytical solution and solution with Runge-Kutta 4 with adaptive time stepping are minor.

A scaling experiment

The original TSM neural form (cf. Eq. (10)) is theoretically capable of approximating every continuous function, according to the universal approximation theorem [36].

However, Table 9 shows results for a TSM neural form with a single neural network. For different domains we scaled the number of hidden layer neurons linearly and averaged ten computations for each domain with the same computational parameters.

domain D	Δu	neurons	n_{TP}
$[0, 1]$	8.4228e-4	5	10
$[0, 2]$	9.2191e-4	10	20
$[0, 3]$	1.9448e-3	15	30
$[0, 4]$	1.6751e-2	20	40

Table 9: TSM neural form, $\mathbf{p}_{rnd}^{init}$.

The results in Table 9 provide the following message. Increasing the domain size forces the neural network to incorporate more hidden layer neurons and grid points. Indeed, to reach, e.g., (averaged) $\Delta u = 9.5873\text{e-}4$ for $D = [0, 3]$, learning the neural network required 50 hidden layer neurons and 75 grid points. In general, determining a suitable architecture in terms of the number of hidden layer neurons and training points is a challenging task. Please note, that the results and the statements hold under the employed computational parameters and first order optimisation.

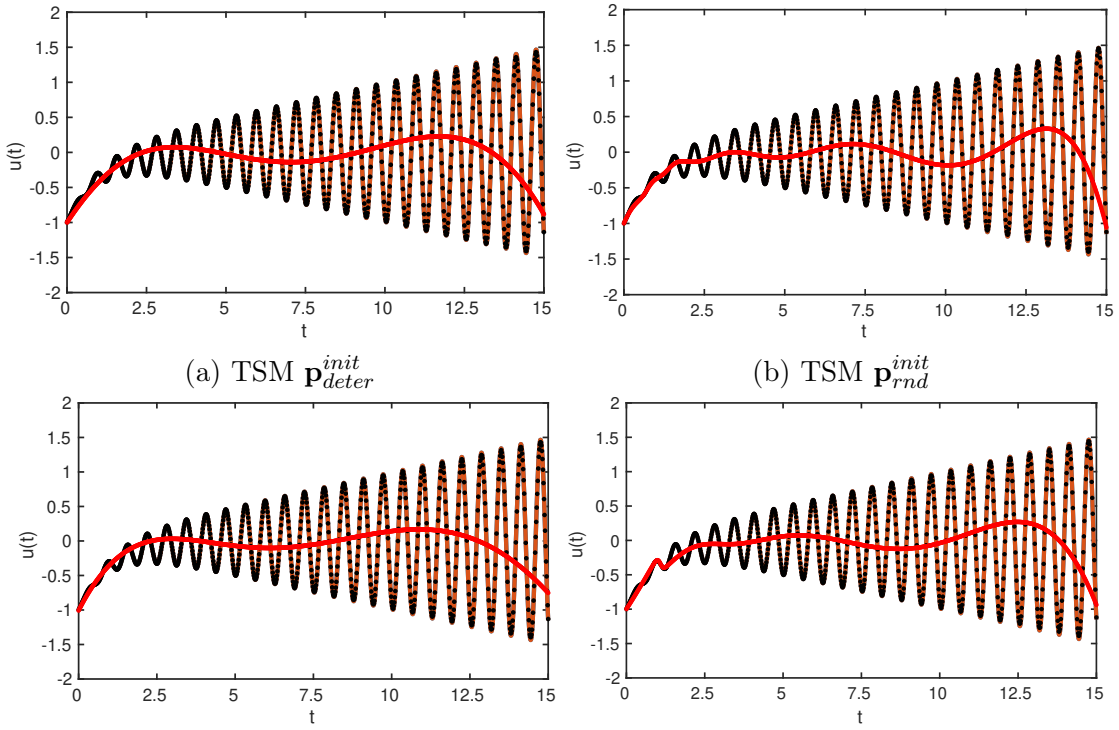
In subsequent experiments, we find the SCNF to be able to solve the initial value problem with neural networks including a small fixed amount of hidden layer neurons and training points in each subdomain. At the same time, this allows to define various important parameters in a simple and straightforward way.

3.4.1 SCNF Experiment: CNF versus SCNF

In the first experiment we compare results of a SCNF with a CNF that is solved over the entire domain. For comparability, the total number of training points is constant, namely $n_{TP} = 1000$ for the red line and $n_{TP} = 10$ with 100 subdomains for the black/dotted line. However, the comparison of two CNFs with the same architecture would not be meaningful because the domain size has a significant influence. Therefore we decided to realise the CNF (red) with a neural network incorporating 1 input layer bias and 100 sigmoid neurons with 1 bias. The SCNF (black/dotted) features neural networks with 1 input layer bias and 5 sigmoid neurons with 1 bias per subdomain. Both CNF and SCNF incorporate $m = 3$. In addition we did not increase the domain size incrementally for this experiment, to reduce the number of parameters that prevent comparability.

The CNF solution (red) shows throughout all experiments in Fig. 19 no useful approximation. In total, the number of hidden layer neurons and training points that would be needed to obtain a useful approximation seems to be much higher. Nonetheless, the SCNF approach (black/dotted) working with the same number of training points was able to solve the initial value problem in a satisfactory way. From a qualitative perspective both TSM and mTSM together with $\mathbf{p}_{deter}^{init}$ and $\mathbf{p}_{rnd}^{init}$ provide similar results. These findings confirm the results from previous experiments.

Concluding this experiment, we see that the SCNF method provides a useful solution to the initial value problem. In addition, the incorporated small number of hidden layer neurons enables a much more effective training of the neural networks.



(a) TSM $\mathbf{p}_{deter}^{init}$ (b) TSM $\mathbf{p}_{rnd}^{init}$

(c) mTSM $\mathbf{p}_{deter}^{init}$ (d) mTSM $\mathbf{p}_{rnd}^{init}$

Figure 19: **Experiment in 3.4.1.** CNF versus SCNF, (orange) analytical solution, (red) CNF solution, (black/dotted) SCNF solution.

3.4.2 SCNF Experiment: CNF order variation

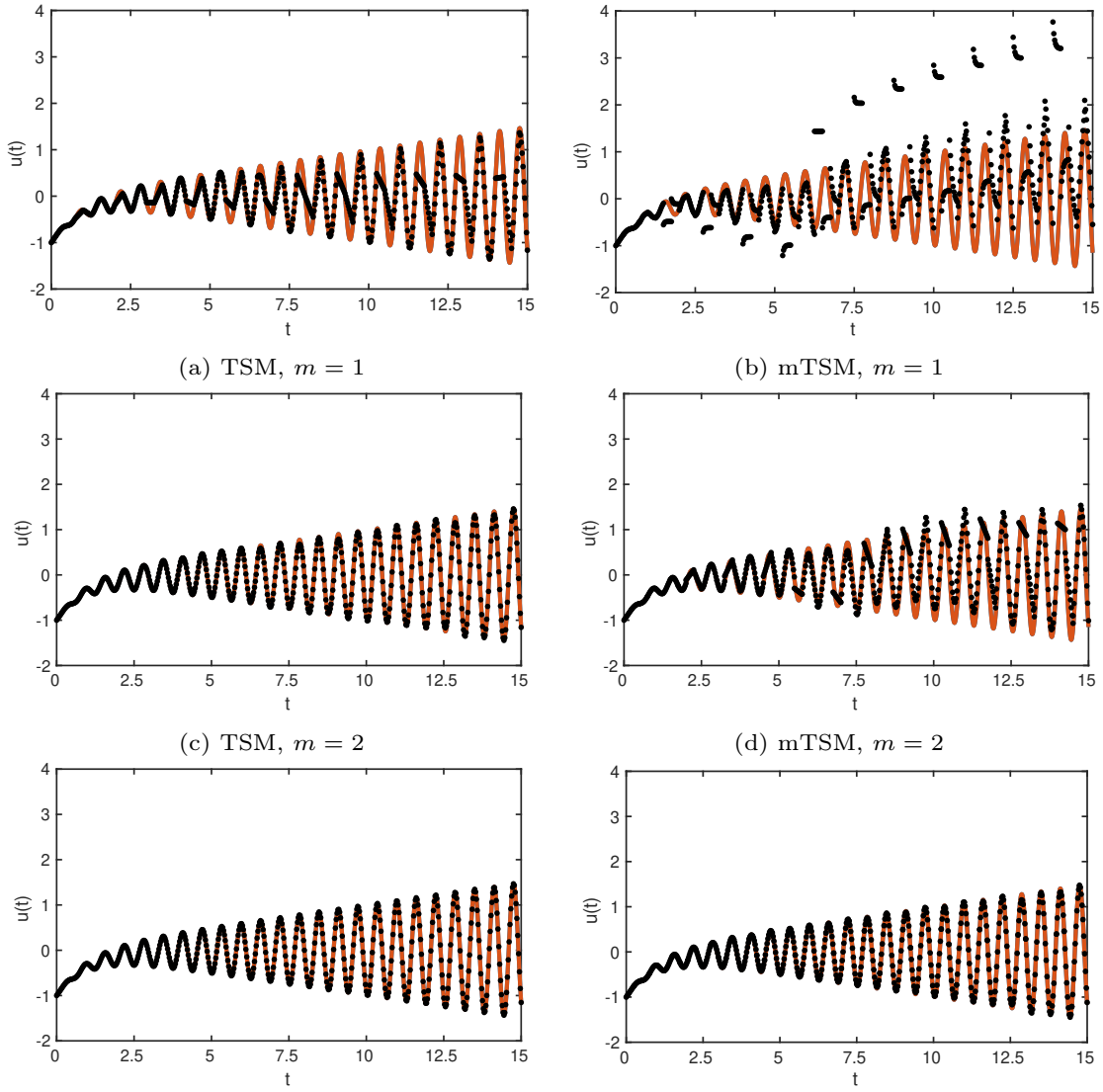
m	$\Delta u(t_{n,l-1})$	$\Delta u(t_{0,l})$
1	2.7834	4.8758
2	0.5763	0.7478
3	0.0846	0.0848

Table 10: Numerical error for mTSM interface grid points, $\mathbf{p}_{deter}^{init}$.

The ability to approximate the initial value problem with SCNF, depending on different m, is subject to this experiment. Here the SCNFs include 1 input layer bias and 5 sigmoid neurons with 1 bias. The solution domain is split into 60 subdomains with 10 grid points in each subdomain. Here, we employ incremental learning in the subdomains.

Results for TSM with $m = 1$ in Fig. 20(a) and mTSM with $m = 1$ in Fig. 20(b) indicate that the original TSM and mTSM methods are not useful over larger domains, even when employing domain segmentation. However, the SCNF of first order is able to get back on the solution trend, although several subdomains do not provide correct approximations. In total, both solutions for $m = 1$ (especially mTSM) cannot be considered to be reliable.

That changes for $m = 2$, at least for TSM in Fig. 20(c). Here we find, with the exception of some local extreme points, the SCNF to be a reasonable approximation of



(a) TSM, $m = 1$ (b) mTSM, $m = 1$

(c) TSM, $m = 2$ (d) mTSM, $m = 2$

(e) TSM, $m = 3$ (f) mTSM, $m = 3$

Figure 20: **Experiment 3.4.2.** CNF order (m) variation, $\mathbf{p}_{deter}^{init}$, (orange) analytical solution, (black/dotted) SCNF solution.

the initial value problem. This statement however, does not hold for mTSM. Although the general trend now is much closer to the analytical solution, there are still subdomains which do not approximate the solution well.

Results shown in Table 10 represent the numerical differences between the analytical and the computed solution, for mTSM as displayed Fig. 20, measured at the last grid points in D_{l-1}, namely $t_{n,l-1}$, and the corresponding initial points in D_l, $t_{0,l}$. We propose to consider this measure, since it indicates how well the solution can be met over the subdomains. We find that increasing m has a major influence on the accuracy.

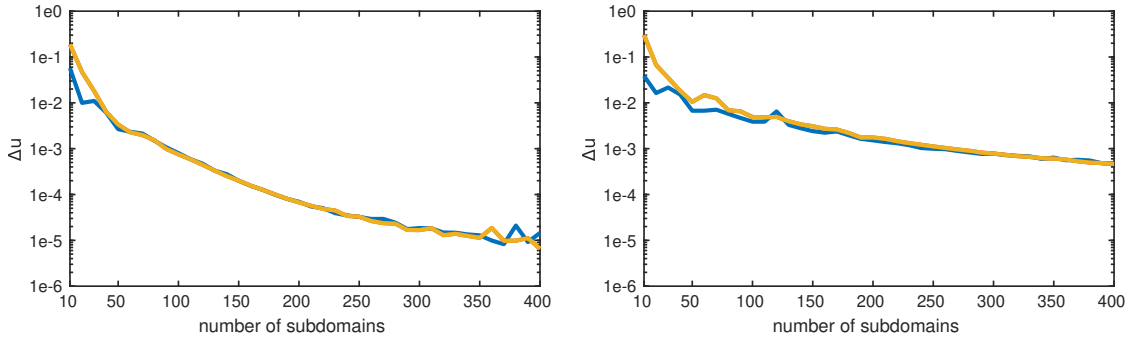
We conjecture that learning the subdomain initial values becomes easier for mTSM, the more neural networks are incorporated. That is mainly because the first neural network can so to say focus on learning the initial values, while the other networks are more engaged with the IVP structure. We think that this conjecture can be confirmed

by the decreasing discrepancy between the overlapping at the interfaces for higher orders of m.

The overall best solutions here are provided by $m = 3$ (Fig. 20(e),20(f)) for both TSM and mTSM in this experiment.

We tend to favor TSM over mTSM, since the initial value in each subdomain is satisfied by the corresponding SCNF (where the learned value at $t_{n,l-1}$ is set to be the initial value for $t_{0,l}$) and does not have to be learned again.

3.4.3 SCNF Experiment: number of subdomain variation



(a) TSM (b) mTSM

Figure 21: **Experiment in 3.4.3.** Number of subdomain variation, (blue) $\mathbf{p}^{init}_{rnd}$, (yellow) $\mathbf{p}^{init}_{deter}$.

In this experiment we investigate the influence of the total number of subdomains on the numeric error Δu. Fig. 21 shows the behaviour for $\mathbf{p}^{init}_{rnd}$ (blue) and $\mathbf{p}^{init}_{deter}$ (yellow). The SCNF incorporate $m = 3$, 1 input layer bias, 5 sigmoid neurons with 1 bias and 10 grid points in each subdomain. We again employ incremental learning in the subdomains.

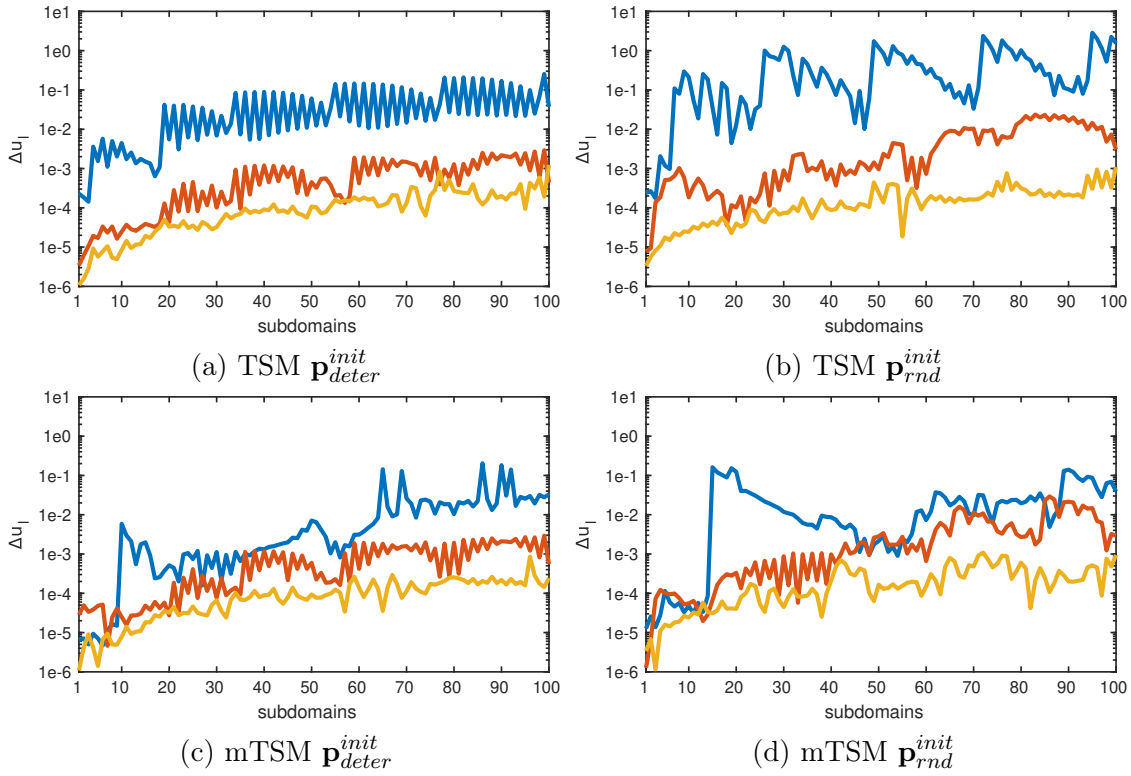
Let us first comment on the SCNF for TSM in Fig. 21(a). Despite minor differences between the solutions corresponding to $\mathbf{p}^{init}_{rnd}$ and $\mathbf{p}^{init}_{deter}$ for smaller numbers of subdomains, both initialisation methods show a very similar trend. A saturation regime seems to appear for around 350 subdomains with $\Delta u \approx$ 1e-5.

Turning to mTSM in Fig. 21(b), we again observe a similar behaviour between the methods with $\mathbf{p}^{init}_{rnd}$ and $\mathbf{p}^{init}_{deter}$. Although the differences disappear not before larger numbers of subdomains. We find that even at 400 subdomains the numerical error Δu can not compete with TSM here.

Let us note again, that the chosen weight initialisation approach for $\mathbf{p}^{init}_{rnd}$ (see Section 3.3) means that the random weights are initialised in the same way in each subdomain. In undocumented tests we observed that the results may show slight to significant variations, when the random weights are generated independently for each network over the subdomains. However, the results we have shown here using $\mathbf{p}^{init}_{rnd}$ represent a rather typical trend observed in the results.

In conclusion, one can obtain very good approximations with the TSM SCNF approach for both weight initialisation methods. That means, choosing $\mathbf{p}^{init}_{deter}$ over $\mathbf{p}^{init}_{rnd}$ has no downsides, which leads us to again support the use of deterministic weight initialisation with the employed first order optimisation method.

3.4.4 SCNF Experiment: numerical error in the subdomains



(a) TSM $\mathbf{p}^{init}_{deter}$ (b) TSM $\mathbf{p}^{init}_{rnd}$

(c) mTSM $\mathbf{p}^{init}_{deter}$ (d) mTSM $\mathbf{p}^{init}_{rnd}$

Figure 22: **Experiment in 3.4.4.** Numerical error in the subdomains, (blue) $m = 1$, (orange) $m = 3$, (yellow) $m = 5$.

The last experiment investigates the numeric error Δu_l in each subdomain D_l, depending on different m. Again, the SCNFs feature 1 input layer bias and 5 sigmoid neurons with 1 bias. The solution is computed with 100 subdomains together with 10 grid points each and incremental learning in the subdomains.

Throughout Fig. 22, $m = 1$ shows the least good results. Although, if we compare mTSM with $\mathbf{p}^{init}_{deter}$ in Fig. 22(b) to results for 60 subdomains in Fig. 20(b), increasing the domain fragments by 40 subdomains seems to prevent the solution from diverging. Random weight initialisation works better for $m = 1$, especially with TSM.

Solutions provided by $m = 3$ and $m = 5$ are much better than for $m = 1$, and increasing the order clearly tends to increase the accuracy. For TSM with both $m = 3$ and $m = 5$, as well as for mTSM with $m = 5$ from a certain subdomain on, the numerical error saturates. Let us note, that for both $\mathbf{p}^{init}_{deter}$ and $\mathbf{p}^{init}_{rnd}$ the differences in the overall numerical error Δu are not significant in these cases.

In this experiment, we again tend to favour TSM with the deterministic weight initialisation $\mathbf{p}^{init}_{deter}$. Although $m = 1$ does not work well (with first order optimisation), the other shown higher orders provide good approximations with saturation regimes. The results confirm our preference of deterministic initialisation, because $\mathbf{p}^{init}_{deter}$ does not depend on a good generation of random weights by chance.

3.4.5 SCNF Experiment: system of initial value problems

In this section we study a non-linear system of initial value problems. The example system we consider reads

$$\dot{u}(t) = a_u v(t) w(t) \tag{121}$$
$$\dot{v}(t) = a_v w(t) u(t) \tag{122}$$
$$\dot{w}(t) = a_w u(t) v(t) \tag{123}$$

with

$$a_u = \frac{I_v - I_w}{I_v I_w}, \quad a_v = \frac{I_w - I_u}{I_w I_u}, \quad a_w = \frac{I_u - I_v}{I_u I_v} \tag{124}$$

where I_u, I_v, I_w are non-zero real numbers, and given initial values for u, v, w. The equations describe the angular momentum of a free rigid body [103, 104] with the centre of mass at the origin. These coupled initial value problems are often denoted as Euler equations and feature time invariant characteristics since the independent variable t (time) does not explicitly appear on the right-hand side. The quadratic invariant expression

$$R^2 = u^2(t) + v^2(t) + w^2(t) \tag{125}$$

conserves the so-called magnitude and describes a sphere, while another quadratic invariant

$$H = \frac{1}{2}\left(\frac{u^2(t)}{I_u} + \frac{v^2(t)}{I_v} + \frac{w^2(t)}{I_w}\right) \tag{126}$$

conserves the kinetic energy and describes an ellipsoid. Both invariant quantities force the solution to stay on the intersection formed by the sphere and the ellipsoid. Since Eqs. (125) and (126) remain unchanged over time along the solutions, we have

$$H = \frac{1}{2}\left(\frac{u^2(0)}{I_u} + \frac{v^2(0)}{I_v} + \frac{w^2(0)}{I_w}\right) \tag{127}$$
$$R^2 = u^2(0) + v^2(0) + w^2(0) \tag{128}$$

The corresponding initial values $u(0), v(0), w(0)$ are given as in Fig. 23 and the fixed principle moments of interior (see [103], Section 14.3) have the values

$$I_u = 2, \quad I_v = 1, \quad I_w = \frac{2}{3} \tag{129}$$

assigned. In general, the neural network approach allows given invariant expressions to be directly added to the cost function, due to its flexibility. For systems of initial value problems, the cost function can be obtained by assigning each solution function its own SCNF and sum up the retrieved l_2-norms, here together with the invariant quantities:

$$\begin{aligned} E[\mathbf{P}_{m,l}] = \frac{1}{2(n+1)} \sum_{i=0}^{n} \Bigg[& \left\{ \dot{\tilde{u}}_C - a_u \tilde{v}_C \tilde{w}_C \right\}^2 + \left\{ \dot{\tilde{v}}_C - a_v \tilde{w}_C \tilde{u}_C \right\}^2 \\ & + \left\{ \dot{\tilde{w}}_C - a_w \tilde{u}_C \tilde{v}_C \right\}^2 + \left\{ \tilde{u}_C^2 + \tilde{v}_C^2 + \tilde{w}_C^2 - R^2 \right\}^2 \\ & + \left\{ \frac{\tilde{u}_C^2}{2I_u} + \frac{\tilde{v}_C^2}{2I_v} + \frac{\tilde{w}_C^2}{2I_w} - H \right\}^2 \Bigg] \end{aligned} \tag{130}$$

We use $\tilde{u}_C = \tilde{u}_C(t_{i,l}, \mathbf{P}_{m,l})$, $\tilde{v}_C = \tilde{v}_C(t_{i,l}, \mathbf{P}_{m,l})$ and $\tilde{w}_C = \tilde{w}_C(t_{i,l}, \mathbf{P}_{m,l})$ as shortcuts.

In order to visualise the invariant behaviour, the computed results in Fig. 23 are obtained for $t \in [0, 30]$, which allows the solution to pass its own initial points more than once. The solution domain is fragmented into 40 subdomains with ten training points in each subdomain. Due to overlapping grid points at the subdomain intersections, the total number of unique training points is $n_{TP} = 361$, the same amount was used to obtain computational result with Runge-Kutta 4. Each SCNF has $m = 3$ and therefore features three neural networks involved. The latter are initialised with zeros and other training parameters and methods remain unchanged (see Section 3.4).

In Fig. 23, we display both the Runge-Kutta 4 (coloured/solid) and the SCNF (black/dots) solution. As mentioned above, the curves lay on intersections formed between a sphere and an ellipsoid. Changing the point of view in Fig. 24 reveals the contours of intersections between the corresponding spheres and ellipsoids for the different initial conditions.

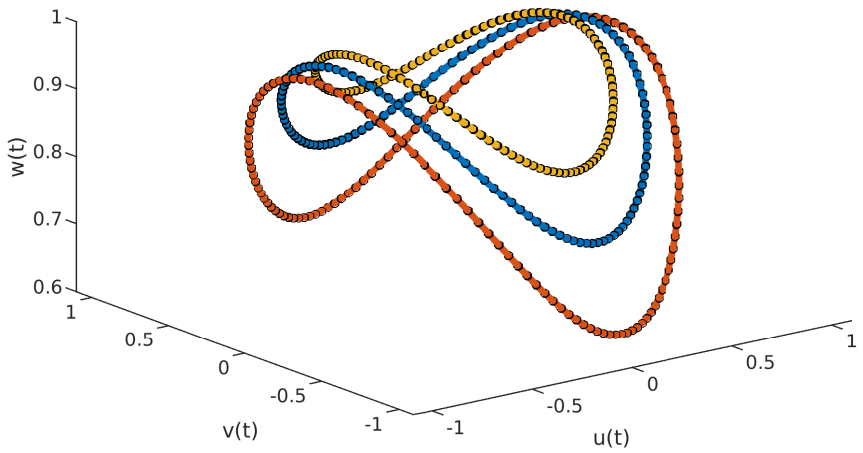


Figure 23: **Experiment in 3.4.5.** Runge-Kutta 4 (coloured/solid) and SCNF (black/dots) solution of the free rigid body problem in Eqs. (121)-(123) for different initial values, (blue) $\{u(0)$=cos(1.1); $v(0)$=0.6; $w(0)$=sin(1.1)$\}$, (orange) $\{u(0)$=cos(1); $v(0)$=0.7; $w(0)$=sin(1)$\}$, (yellow) $\{u(0)$=cos(1.2); $v(0)$=0.5; $w(0)$=sin(1.2)$\}$.

Fig. 25 shows the training error arising from the cost function in Eq. (130) over the incorporated subdomains. Since the SCNF solution in each subdomain is computed independently (except for the provided initial value), the training error can differ significantly, even between two adjacent subdomains. Additionally, the flexibility of those independent computations enables this decreasing behaviour of the training error, although the previous subdomain may have shown large errors.

In conclusion, the SCNF approach can be easily extended to handle systems of initial value problems, even with non-linear and time invariant characteristics. A qualitative comparison between the SCNF solution and the Runge-Kutta 4 solution shows only very minor visual differences. As mentioned in Section 3.4, a quantitative comparison may favour Runge-Kutta 4 in terms of the numerical error in this experiment.

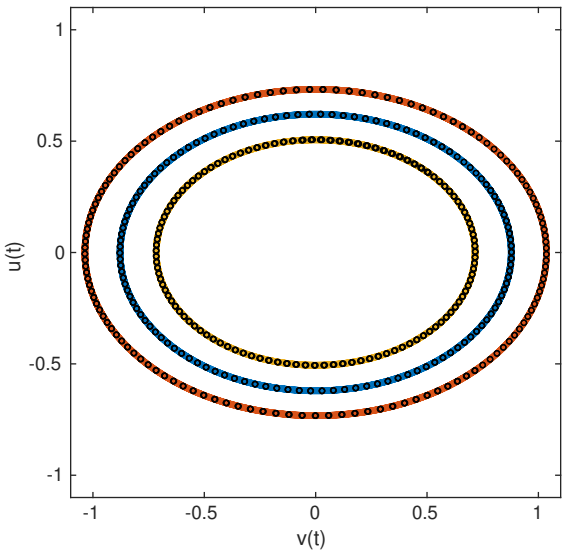


Figure 24: **Experiment in 3.4.5.** Top view of Fig. 23 to visualise the spherical and ellipsoidal intersections, colours are adopted from Fig. 23.

3.4.6 Comparison with numerical methods

Putting the SCNF results in context with the Euler method and Runge-Kutta 4, Fig. 26 shows the numerical (l_1-) error in each subdomain for the IVP in Eq. (119). The numerical methods were computed with 500 grid points per subdomain, whereas the neural forms approaches used 10 grid points. We can clearly see that each configuration of TSM and mTSM with deterministic $\mathbf{p}_{deter}^{init}$ and random $\mathbf{p}_{rnd}^{init}$ weight initialisation have a similar behaviour, where TSM in general performs slightly better than mTSM. Although it is arguable whether 500 grid points here are enough for the Euler method to provide useful results. In order to compare the numerical methods with even conditions, we decided to limit the number of grid points. Therefore, the Euler method shows a worse accuracy over the neural forms approaches, while RK4 has a tremendous accuracy advantage in this comparison. Table 11 confirms the findings where the averaged numerical error over the 100 subdomains for TSM, mTSM is shown, together with results for Euler, RK4 over the entire domain (1 subdomain). RK4 clearly outperforms

method	TSM $\mathbf{p}_{rnd}^{init}$	TSM $\mathbf{p}_{deter}^{init}$	mTSM $\mathbf{p}_{rnd}^{init}$	mTSM $\mathbf{p}_{deter}^{init}$	Euler	RK4
Δu	1.1318e-4	1.4339e-4	2.5184e-4	1.8362e-4	7.2374e-2	3.9164e-9

Table 11: Comparison between TSM, mTSM with deterministic/random weight initialisation (100 subdomains) and both Euler method and RK4 (1 subdomain, 500 grid points).

the other methods here. However, we want to point out that this is not a big disadvantage in context of the major findings related to the SCNF. That is, we were able to highly improve the results, especially for TSM and deterministic initial weights $\mathbf{p}_{deter}^{init}$, over larger domains in combination with first order optimisation. Since second order optimisation is not part of the investigations, but we believe that BFGS works very well, TSM could perform even better. Nonetheless, the comparison is highly useful and

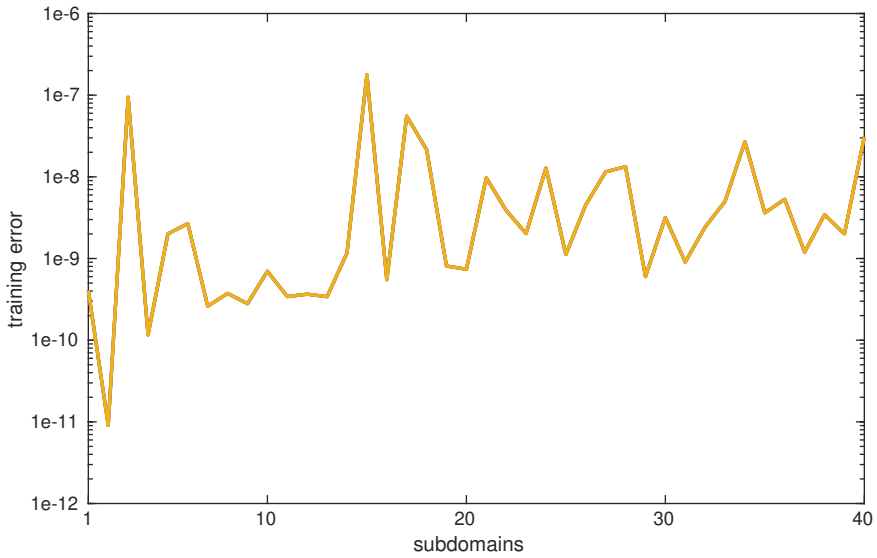


Figure 25: Experiment in 3.4.5. Training error over the subdomains for $\{u(0)=\cos(1.2); v(0)=0.5; w(0)=\sin(1.2)\}$.

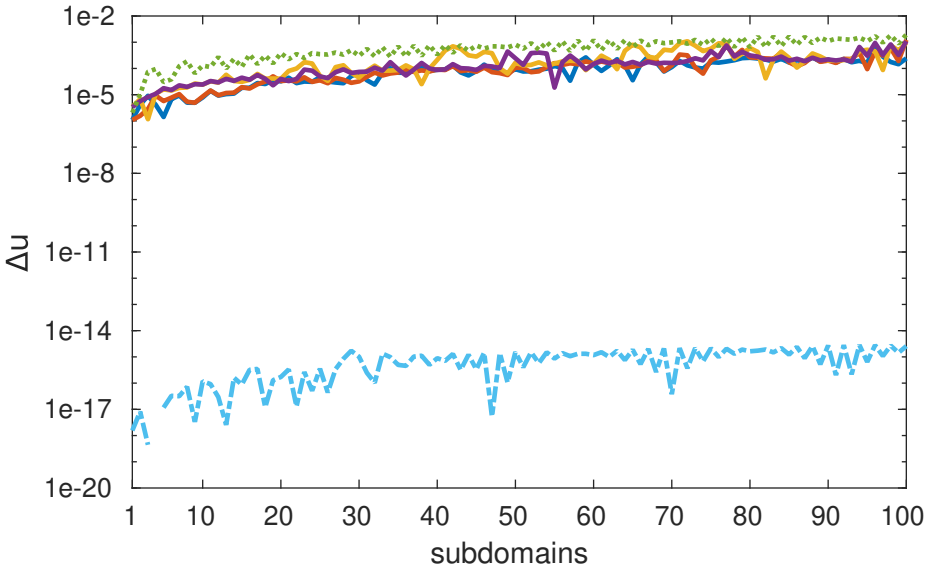


Figure 26: Comparison of neural forms and numerical methods (500 grid points) with an equal amount of subdomains, (dark blue/solid) TSM $\mathbf{p}_{rnd}^{init}$, (orange/solid) TSM $\mathbf{p}_{deter}^{init}$, (yellow/solid) mTSM $\mathbf{p}_{rnd}^{init}$, (purple/solid) mTSM $\mathbf{p}_{deter}^{init}$, (green/dotted) Euler method, (light blue/dashed) RK4.

shows that although we made good progress with the domain segmentation approach, further development and investigations are necessary.

3.4.7 Conclusion

The proposed CNF and SCNF approaches merging collocation polynomial basis functions with neural networks and domain segmentation show clear benefits over the previous neural form constructions for the employed first order optimisation. We have studied in detail the deterministic weight initialisation for our novel CNF approach

with a basic stiff initial value problem. Depending on the batch learning methods, the collocation-based extension seems to have some benefits for both TSM and mTSM. For the TSM CNF, this effect is more significant than observed for the mTSM extension.

Focusing on mTSM and the CNF approach, using two neural networks, one for learning the initial value and one multiplied by $(t_i - t_0)^\kappa$, seems to have some advantages over other possible mTSM settings. Considering approximation quality as most imperative, we find mTSM with $m = 2$ to provide the overall best results for the investigated initial value problem.

We find that the proposed SCNF approach combines many advantages of the new developments. Employing higher order CNF methods, it is possible to solve initial value problems over large domains with very high accuracy, and at the same time with reasonable optimisation effort. Moreover, many computational parameters can be fixed easily for this setting, which is a significant issue with other TSM and mTSM variations.

As another important conclusion, in the experiments we were able to show that we can favour deterministic weight initialisation over random weight initialisation. Nonetheless, the complexity of the IVP has an impact on the architecture and on the values of the initially deterministic/equal weights. As we have pointed out in a computational study [81], DE and network related parameters may not be independent of each other. However, the underlying relation between problem complexity and necessary neural network architecture is yet part of future work.

When focusing on deterministic weight initialisation, we find a further investigation on how to find suitable (equal) initial weights to be of interest. The same holds for the sensitivity of the neural network parameter.

Future research may also include work on other possible collocation functions and on combining the networks with other discretisation methods. In addition to that, let us note that we find optimal control problems to be a possible and relevant potential field of applications for our method, see for instance [106] for recent research in that area.

4 ANDRe: adaptive neural domain refinement for solving initial value problems

In the previous section, we proposed a collocation polynomial extension for the neural forms and a subdomain division approach, which splits the solution domain into equidistant subdomains [82]. Since the neural forms adopted from [13] directly incorporate the initial condition in its construction, each temporal subdomain generates a new initial condition for the subsequent subdomain. As it turns out, both extensions to the original neural forms approach were able to improve the computational results with respect to weight initialisation and larger domain sizes under the consideration of first order optimisation. However, equidistant subdomains may not be the optimal choice in regions where the solution is easy or difficult to learn. In classic adaptive numerical methods, the mesh as well as the domain may be refined or decomposed, respectively, in order to improve accuracy. Also the degree of approximation accuracy may be adapted. As it is desirable to transfer such important and successful strategies to the field of neural network based solutions, this section investigates computational results on the adaptive neural domain refinement algorithm, short ANDRe, which makes use of the subdomain collocation (polynomial) neural forms (SCNF). The adopted subdomain collocation polynomial neural forms from the Section 3.3 are optimised repeatedly over the domain. The domain itself is allowed to split into subdomains which may locally decrease in size, whenever the network error is not sufficiently small. Therefore,the advantageous characteristics from domain decomposition are combined with adaptive mesh refinement. Furthermore, we embed into the described process a means to adapt the number of neurons used for optimisation in each subdomain. This is done with the aim to increase reliability and accuracy of the approximation. Thus, ANDRe combines the adaptive refinement of the domain with adaptivity in the neural sense. We also introduce conditions to automatically confirm the solution reliability and optimise computational parameters whenever it is necessary. In addition to that, the results open an opportunity to discuss the relation between neural network and numerical measurement metrics. The SCNF appearance for an IVP does not change for ANDRe, reading

$$\tilde{u}_C(t_{i,l}, \mathbf{P}_{m,l}) = \tilde{u}_C(t_{0,l}, \mathbf{P}_{m,l}) + \sum_{\kappa=1}^{m} N_\kappa(t_{i,l}, \mathbf{p}_{\kappa,l})(t_{i,l} - t_{0,l})^\kappa \tag{131}$$

The adaptive neural domain refinement approach abbreviates with ANDRe.

4.1 Algorithm summary

In Fig. 27, an artificial example to sketch the principle behind ANDRe is visualised. The basic idea is to optimise the cost function $E_l[\mathbf{P}_{m,l}]$ for a given number of equidistant training points (n_{TP}) in each subdomain and to evaluate the results at equidistant verification points (n_{VP}), intermediate to n_{TP}. To obtain the subdomains, the algorithm starts with the cost function optimisation on the entire domain (Fig. 27(1.)). If the predefined verification error bound $\sigma > 0$ is not fulfilled, the domain is split in half. Now the optimisation task starts again for the left half since only there the initial value for this subdomain is known. In case of the verification error $E_l^{VP}[\mathbf{P}_{m,l}]$ (cost function

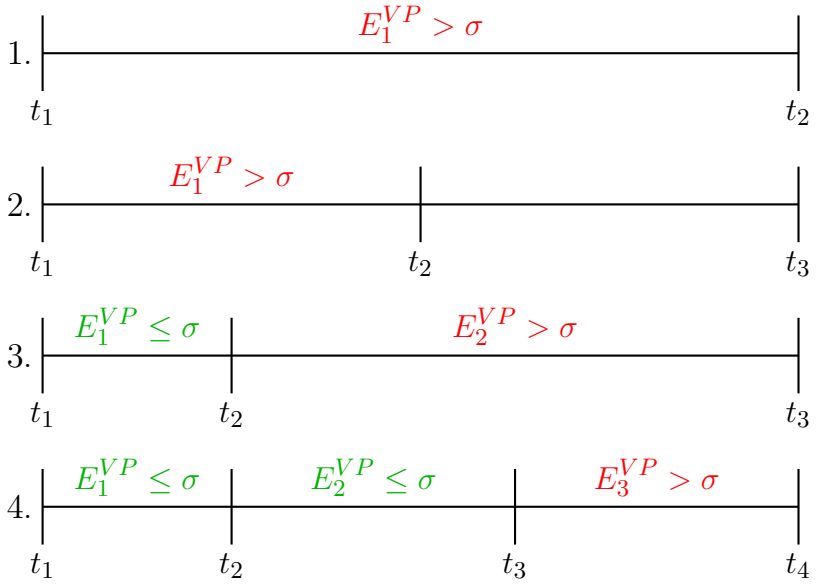


Figure 27: A visualisation of the basic idea behind ANDRe, with the error comparison and (sub-)domain split/reduction, E_l^{VP} denotes the verification metric and the constant σ is the verification error bound.

evaluated for n_{VP}) again fails to go below σ, the current (left) subdomain is reduced in size (see differences in Fig. 27(2.) to (3.)). Whereas a splitting is only performed when the computation takes place in the rightmost subdomain and σ is not satisfied by $E_l^{VP}[\mathbf{P}_{m,l}]$, meaning that the original right domain border is always kept and not shifted during refinement. The process of comparing the verification error to its error bound, reducing the current subdomain and starting the optimisation another time, is repeated until $E_l^{VP}[\mathbf{P}_{m,l}] \leq \sigma$. Therefore, in the artificial example in Fig. 27(3.), the leftmost subdomain is now considered to be learned.

Now, the process starts again for the rightmost subdomain (see Fig. 27(3.) and (4.)) with a new initial condition provided by the learned (left) subdomain. However, the current (new) subdomain starts at the right boundary of the first (learned) subdomain and ends at the right boundary of the entire domain. Therefore, the already learned subdomain is excluded from further computations.

If a subdomain becomes too small or if the verification error increases after a subdomain split/reduction, the computational parameters are adjusted in a predefined, automated way. Details on the parameter adjustment will be provided in a corresponding paragraph later.

Let us now provide detailed information about ANDRe, which is shown as a flowchart in Fig. 28. Starting point is the choice of the SCNF order m and the subdomain resize parameter δ, which acts as a size reduction whenever a decrease is necessary. For the optimisation, equidistant training points n_{TP} are used. An important constant is the verification error bound $\sigma > 0$, used to verify the SCNF solution in the corresponding subdomain. After each complete optimisation, the results are evaluated by the cost function with the previous learned weights at intermediate verification points, resulting in $E_l^{VP}[\mathbf{P}_{m,l}]$. The latter (scalar value) is then compared to σ in order to find out whether the solution can be considered as reliable or not. The subdomain index is de-

fined by l, in which the SCNF is currently solved and h represents the total number of subdomains. The latter is not fixed and will increase throughout the algorithm. Finally, the very first domain is set to be the entire given domain $D_1 = [t_{start}, t_{end}] = [t_1, t_2]$. Please note, while on the computational side, the subdomains are discretised and corresponding grid points denoted by $t_{i,l}$, in this paragraph the subdomain boundaries are represents by t_l, for simplicity.

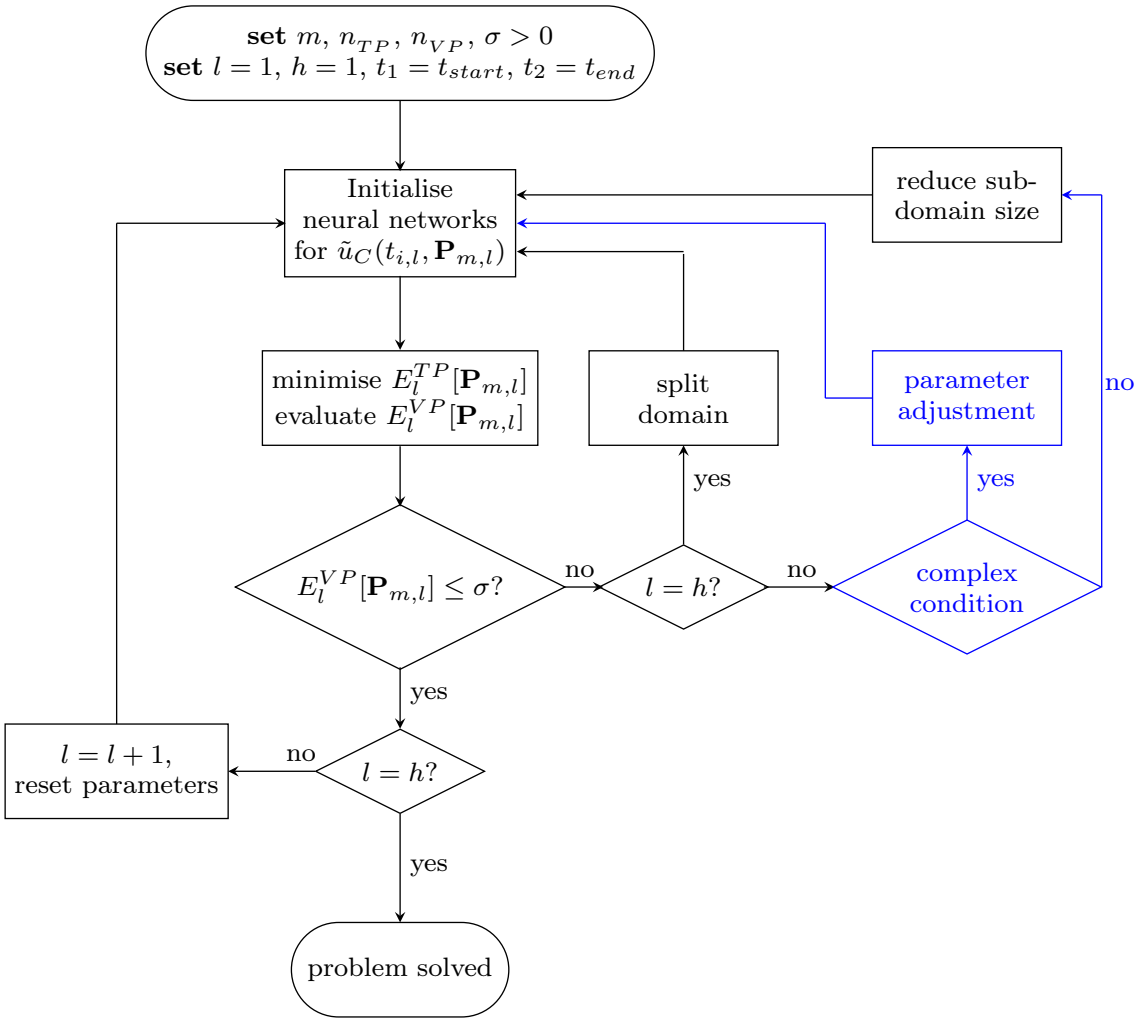


Figure 28: Flowchart for the ANDRe algorithm.

4.2 ANDRe Flowchart explanation

The first processing operation

$$\text{Initialise neural networks for } \tilde{u}_C(t_{i,l}, \mathbf{P}_{m,l}) \tag{132}$$

covers setting the initial architecture parameters such as number of hidden layer neurons, Adam learning rate and initialising the weights for $\mathbf{P}_{m,l}$.

Afterwards the optimisation problem

$$\text{minimise } E_l^{TP}[\mathbf{P}_{m,l}] \tag{133}$$

$$\text{evaluate } E_l^{VP}[\mathbf{P}_{m,l}] \tag{134}$$

is solved by training the SCNF framework for given equidistant n_{TP} over the entire domain $D_1 = [t_1, t_2]$ (cf. Fig. 27(1.)). The evaluation for equidistant and intermediate n_{VP} leads to the verification error $E_l^{VP}[\mathbf{P}_{m,l}]$.

Then the first decision block compares the verification error (after the training process has ended) to the error bound σ:

$$E_l^{VP}[\mathbf{P}_{m,l}] \leq \sigma? \tag{135}$$

- **Eq.** (135) **NO**: In case the verification error did not go below σ, the size of the current subdomain will be reduced. But first, another decision has to be made here. Namely, has $E_l[\mathbf{P}_{m,l}]$ been solved for the first time on the current, rightmost (sub-)domain or in other words, is the current domain index l equal to number of total subdomains h:

$$l = h? \tag{136}$$

  - **Eq.** (136) **YES**: That means the right boundary is t_{end} and the current subdomain l is split in half first, which leads to an increase of the number of total subdomains by 1 ($h = h + 1$). The boundaries now have to be adjusted with the left one t_l to remain unchanged, while the former right boundary is now scaled by $t_{l+1} = t_l + \delta(t_{l+1} - t_l)$, after $t_{l+2} = t_{l+1}$ is set to be the right boundary of domain $l + 1$. For example, if an entire domain $D_1 = [t_1, t_2] = [0, 10]$ has to be split for the first time with $\delta = 0.5$, the resulting subdomains are $D_1 = [t_1, t_2] = [0, 5]$ and $D_2 = [t_2, t_3] = [5, 10]$. Afterwards, the algorithm leads back to Eq. (132).
  - **Eq.** (136) **NO**: In this case the current subdomain has already been split up. Now the right boundary has to be adjusted in order to decrease the current subdomain size. But beforehand ANDRe checks for a complex condition (highlighted in blue) to ensure that a subdomain does not become too small. Additionally ANDRe also checks if the verification error decreased compared to the prior computation on the same subdomain l. That is, the algorithm compares the verification error from the formerly larger subdomain l to the current, size reduced subdomain l. The condition itself may come in different shapes. It was decided to check for one of the

$$\begin{gathered}\text{complex conditions:}\\ t_{l+1} - t_l \leq 0.1?\\ \text{or}\\ E_l^{VP} \text{ from previous } (l \neq h) \text{ subdomain } \leq \text{ current } E_l^{VP}?\end{gathered} \tag{137}$$

    - **Eq.** (137) **YES**: At this point the framework employs a

$$\text{parameter adjustment} \tag{138}$$

which may be realised problem specific and is later addressed in a corresponding paragraph. Afterwards, the algorithm leads back to Eq. (132). Basically speaking, the adjustable parameters may include the number of hidden layer neurons, the learning rate, the number of training points and so on.

- **Eq.** (137) **NO**: In this case, the subdomain is still large enough to be reduced in size while the verification error decays. Therefore ANDRe resizes the right subdomain boundary t_{l+1} to

$$t_{l+1} = t_l + \delta(t_{l+1} - t_l) \tag{139}$$

where δ denotes the domain resize parameter. Continuing the example from above, resizing D_1 of the already split domain leads to $D_1 = [t_1, t_2] = [0, 2.5]$ and $D_2 = [t_2, t_3] = [2.5, 10]$. Afterwards, the algorithm leads back to Eq. (132).

- **Eq.** (135) **YES**: In case of the verification error being smaller or equal compared to σ, the current subdomain l has been successfully learned by means of a sufficiently small verification error. Now it is necessary to determine, whether ANDRe is already in the last subdomain (right boundary is t_{end}) or if there is still one subdomain to solve the optimisation problem on, namely

$$l = h? \tag{140}$$

  - **Eq.** (140) **NO**: There is at least one subdomain left and therefore the current subdomain index is updated to $l = l + 1$ in order to solve the optimisation problem on the adjacent subdomain. Additionally ANDRe resets all the possibly adjusted parameters to the initial ones. This is to make sure to not overuse the variable parameters in regions where the solution computes by using the initial ones. The algorithm then leads back to Eq. (132).
  - **Eq.** (140) **YES**: All subdomains have been successfully learned and the initial value problem is entirely solved.

ANDRe was developed in four steps, making it an adaptive neural algorithm for domain refinement. Excluding the blue part in Fig. 28, the black part represents a fully functional algorithm that can refine the domain in an adaptive way with the focus laying on the verification error. Prior to this final version, the training error was used as the main training status indicator. The evaluation stage (verification error) on the other hand was later added, in addition to the training error. It turned out that small training errors do not necessarily result in a comparable numerical error, presumably due to possible overfitting. Therefore the verification stage was included, to reduce the impact of overfitting on the end result. However, it was later recognised that the verification error has a much stronger relation to the numerical error. Therefore it was possible to reduce the complexity by laying the focus directly on the verification. Furthermore, some examples have shown, that the preset neural network architecture may not be flexible enough to learn certain subdomains. Hence, ANDRe was upgraded to incorporate an automated parameter adjustment mechanism, highlighted in Fig. 28 as blue. Whenever a subdomain becomes too small or the verification error in

a subdomain increases compared the previous optimisation on the same subdomain (e.g., prior to a size reduction), network and optimisation related parameters may be re-balanced in a predefined way.

4.3 Computational results for ANDRe

The computational results and detailed investigations of ANDRe are part of this section. However, and prior to this, the computational parameter setup, the measurement metrics and the studied IVPs are discussed.

4.3.1 Details on parameters and measurement metrics

The neural forms approach comes with plenty parameters. We have already shown in a computational study [81], that they are not independent of each other. Changing one parameter may require another parameter to be changed as well in order to improve or maintain the reliability.

computational parameter	value
hidden layer neurons*	5
initial weight values	0
initial learning rate*	1e-3
number of epochs	1e5
training points (n_{TP})	9
verification points (n_{VP})	11
SCNF order (m)	5
resize parameter (δ)	0.5

Table 12: Initial computational parameters, (*) part of parameter adjustment.

Tab. 12 lists the computational parameters which are initially fixed in our computational setup. Parameters marked with (*) will be separately discussed in the corresponding paragraph. The initial weight values, the SCNF order as well as the number of epochs and training points (n_{TP}) have been previously investigated and are fixed to suitable values, see previous sections and [81, 82] for further details. Nonetheless, each parameter has its impact on the solution. Key in training the neural networks are the training points $t_i,\ i = 0, \ldots, n_{TP}$, schematically depicted in Fig. 29 as green circles. Generally speaking, Fig. 29 shows an arbitrary subdomain and the notation t_i was chosen for simplicity. From now on, the grid points in subdomain l are again referred to as $t_{i,l}$ and follow the structure in Fig. 29.

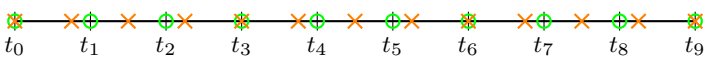


Figure 29: Visualisation of grid point distribution in a subdomain, (green/circle) 10 equidistant training points t_i, (orange/cross) 12 equidistant verification points. Please note that, e.g., $n_{TP} = 9$ refers to ten training points in Tab. 12.

They serve as the input data and are important for the optimisation of the cost function

$$E_l^{TP}[\mathbf{P}_{m,l}] = \frac{1}{2(n_{TP}+1)} \sum_{i=0}^{n_{TP}} \left\{ G\left(t_{i,l}, \tilde{u}_C(t_{i,l}, \mathbf{P}_{m,l}), \dot{\tilde{u}}_C(t_{i,l}, \mathbf{P}_{m,l})\right) \right\}^2 \tag{141}$$

in the l-th subdomain. Let us comment on an optimisation procedure in some detail, referred to as incremental learning, employed in [15]. Here, the computation includes several complete optimisations per (temporarily untouched) subdomain. That is, for example with five increments in Fig. 29, the training points are split into five sets and the first optimisation only takes t_0 and t_1 into account. Then, the second one uses t_0, t_1, t_2, t_3 with the same weights from the first (complete) optimisation. This is continued until the optimisation uses all training points. The incremental learning procedure only applies to the training process and $E_l^{TP}[\mathbf{P}_{m,l}]$, not to the verification.

Speaking of that, the verification is performed with the cost function and the corresponding verification points (n_{VP}), which are differently distributed (cf. Fig. 29) than the training points. With these discrete points and after the training process, the cost function returns a scalar value named verification error

$$E_l^{VP}[\mathbf{P}_{m,l}] = \frac{1}{2(n_{VP}+1)} \sum_{i=0}^{n_{VP}} \left\{ G\left(t_{i,l}, \tilde{u}_C(t_{i,l}, \mathbf{P}_{m,l}), \dot{\tilde{u}}_C(t_{i,l}, \mathbf{P}_{m,l})\right) \right\}^2 \tag{142}$$

As the naming suggests, this verification error is used to evaluate and verify the training results to indicate whether the IVP has been solved sufficiently well or not. For this purpose, the verification error bound σ will compare to Eq. (142).

The domain resize parameter has also been fixed for all computations to $\delta = 0.5$. A larger value, up to $\delta = 0.9$, would find individual subdomains faster due to the bigger size reduction but perhaps result in too many subdomains. On the other hand, a smaller value, down to $\delta = 0.1$ may find the individual subdomains more carefully but would also heavily increase the computation time. We will discuss an experiment regarding the domain resize parameter later.

Tab. 12 will later also be extended by problem specific parameters, which are *(i)* verification error bound σ, *(ii)* computational domain size, *(iii)* initial conditions and *(iv)* learning increments. These parameters will be specified and discussed in a subsequent paragraph.

Turning to the measurement metrics for the results, we will compare ANDRe to the analytical solutions of four different initial value problems. We make use of the absolute value differences between the analytical solution and ANDRe in context of the (averaged) l_1-norm $\Delta u_{l,1}$ and the l_∞-norm $\Delta u_{l,\infty}$

$$\Delta u_{l,1} = \frac{1}{n+1} \sum_{i=0}^{n} \left| u(t_{i,l}) - \tilde{u}_C(t_{i,l}, \mathbf{P}_{m,l}) \right| \tag{143}$$

$$\Delta u_{l,\infty} = \max_i \left| u(t_{i,l}) - \tilde{u}_C(t_{i,l}, \mathbf{P}_{m,l}) \right| \tag{144}$$

whereas Δu_1 and Δu_∞ average the numerical error over all subdomains

$$\Delta u_1 = \frac{1}{h} \sum_{l=1}^{h} \Delta u_{l,1} \tag{145}$$

$$\Delta u_\infty = \frac{1}{h} \sum_{l=1}^{h} \Delta u_{l,\infty} \tag{146}$$

The l_∞-norm basically returns the largest numerical error value. We will later refer to the corresponding norms as l_1-error and l_∞-error.

4.3.2 Details on parameter adjustment

Let us now comment on the parameter adjustment since this part of the algorithm required a lot of fine tuning. After several experiments with different parameter adjustment methods, not documented here, the Adam learning rate and the number of hidden layer neurons were chosen to be a part of the parameter adjustment and may change during the process. Not only determine the hidden layer neurons the amount of adjustable weights, they are also connected to the universal approximation theorem [36]. It basically states, that one hidden layer with a finite number of sigmoidal neurons is able to approximate every continuous function on a subset of $\mathbb{R}$. Since the finite number is not known beforehand, making the number of hidden layer neurons an adjustable parameter in this approach, seems reasonable and so does starting with a small amount (five neurons).

The initial learning rate of Adam optimisation impacts how vast the location in the weight space changes after a weight update. Figuratively speaking, the larger the initial learning rate, the farther the optimiser can travel in the weight space, adding more flexibility and increasing the chance to find a suitable minimum. In this context, such a suitable minimum can be located at different positions, depending on the subdomain. It is not guaranteed by any means to find one near by the starting point. That motivates to start the computation with a fairly small initial learning rate (values taken from [44]) and to enable ANDRe to increase this value outside the optimisation cycle. That is, the initial learning rate can increase several times before the number of hidden layer neurons rearranges by two additional neurons. Adjusting the number of neurons resets the learning rate to its default parameter.

Has a subdomain in this way been successfully learned, both parameters are reset to their initial values. Let us recall, that the parameter adjustment does not take place during an optimisation cycle, it rather appears outside. In other words, we do not perturb the neural network training during the optimisation process.

4.3.3 The evaluation of ANDRe for different initial value problems

In [82] we have shown, that the SCNF with a fixed number of subdomains is capable of solving IVPs on larger domains. Increasing this number resulted in a decreasing numerical error. Now with ANDRe, we show that by demanding the network error to become sufficiently small in each subdomain, the algorithm can automatically determine a suitable number (and distribution) of the subdomains.

The following paragraph will introduce initial value problems (IVPs) for our evaluation. We have chosen these examples because *(i)* each one represents a different IVP type, *(ii)* expect for the last (system of IVPs) example, the analytical solutions are available and *(iii)* each of them incorporates at least one interesting behaviour. However, the difficulty is limited because of *(ii)*, but the focus of this approach does not lay on competitiveness in the first place. We rather show that the neural forms approach [13, 81] benefits from our extension in terms of accuracy on large domains. In addition, this paper serves as an investigation of the relation between both numerical and neural network errors.

Example IVPs and their analytical solutions

As a first example, we take on the following IVP with constant coefficients

$$\begin{cases} \dot{\psi}(t) - t\sin(10t) + \psi(t) = 0, \quad \psi(0) = -1 \\ \psi(t) = \sin(10t)\left(\frac{99}{10201} + \frac{t}{101}\right) + \cos(10t)\left(\frac{20}{10201} - \frac{10t}{101}\right) - \frac{10221}{10201}e^{-t} \end{cases} \tag{147}$$

which incorporates heavily oscillating and increasing characteristics, similar to instabilities. This example is still relatively simple and serves to demonstrate the main properties of our approach. We then proceed to an IVP with non-constant coefficients, that includes trigonometric and exponentially increasing terms:

$$\begin{cases} \dot{\phi}(t) + \frac{1 + \frac{1}{1000}e^t\cos(t)}{1+t^2} + \frac{2t}{1+t^2}\phi(t) = 0, \quad \phi(0) = 5 \\ \phi(t) = \frac{1}{1+t^2}\left(-t - \frac{e^t\cos(t)}{2000} - \frac{e^t\sin(t)}{2000} + \frac{10001}{2000}\right) \end{cases} \tag{148}$$

Furthermore, we choose to investigate the results for the non-linear IVP

$$\begin{cases} \frac{\dot{\omega}(t)}{\cos^2(\omega(t))}\frac{1}{\cos^2(2t)} - 2 = 0, \quad \omega(0) = \frac{\pi}{4} \\ \omega(t) = \arctan\left(\frac{1}{4}\sin(4t) + t + 1\right) \end{cases} \tag{149}$$

which also has non-constant coefficients. Finally, we used ANDRe to solve the following non-linear system of IVPs

$$\begin{cases} \dot{\tau}(t) = A\tau(t) - B\tau(t)\iota(t), \quad \tau(0) = \tau_0 \\ \dot{\kappa}(t) = -C\iota(t) + D\tau(t)\iota(t), \quad \iota(0) = \iota_0 \end{cases} \tag{150}$$

which is also known as the Lotka-Volterra equations [107], with parameters $A = 1.5$, $B = 1$, $C = 3$, $D = 1$. The initial values are $\tau_0 = 3$, $\iota_0 = 1$, $\iota_0 = 3$, $\iota_0 = 5$ depending on the subsequent experiment. The chosen value for ι_0 will be explicitly addressed. Since the Lotka-Volterra equations in Eq. (150) do not have an analytical solution, we will compare the results to a numerical solution method, namely Runge-Kutta 4.

We take the coupled IVPs in Eq. (150) to demonstrate how the cost function for the neural forms approach reads. It is obtained as the sum of ℓ_2-norms of each equation. We use $\tilde{\tau}_C = \tilde{\tau}_C(t_{i,l}, \mathbf{P}_{m,l})$ and $\tilde{\iota}_C = \tilde{\iota}_C(t_{i,l}, \mathbf{P}_{m,l})$ as shortcuts:

$$E_l[\mathbf{P}_{m,l}] = \frac{1}{2(n+1)} \sum_{i=0}^{n} \Bigg[\left\{ \dot{\tilde{\tau}}_C - A\tilde{\tau}_C + B\tilde{\tau}_C\tilde{\iota}_C \right\}^2 + \left\{ \dot{\tilde{\iota}}_C + C\tilde{\iota}_C - D\tilde{\tau}_C\tilde{\iota}_C \right\}^2 \Bigg] \tag{151}$$

This equation is then subject to optimisation/training and verification.

4.3.4 ANDRe and the analytical solutions

In this paragraph we demonstrate the results for applying ANDRe to the previously introduced example IVPs. We discuss the contrast to the analytical solutions and in case of Lotka-Volterra, to the numerical results provided by Runge-Kutta 4. In addition to the already given computational parameters in Tab. 12, the problem specific parameters are listed in Tab. 13. The corresponding initial conditions are given with the examples above.

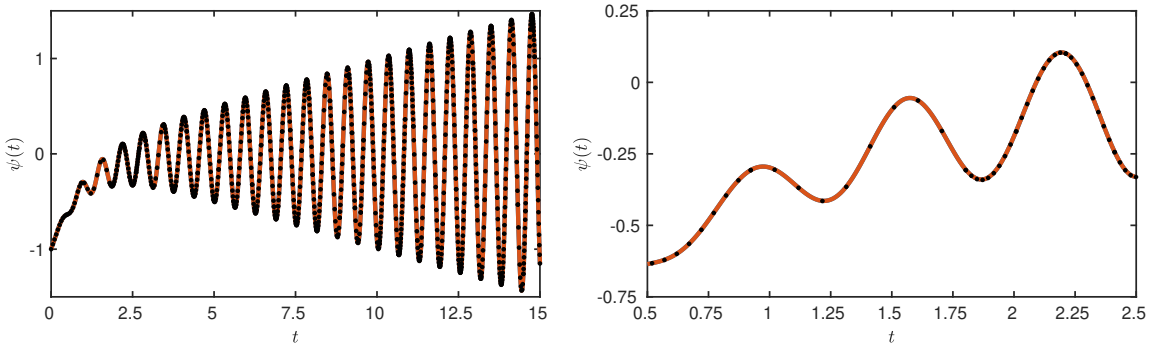
example	domain	σ	incr.
IVP in Eq. (147)	$t \in [0, 15]$	1e-5	5
IVP in Eq. (148)	$t \in [0, 25]$	1e-4	5
IVP in Eq. (149)	$t \in [0, 20]$	1e0	2
IVP in Eq. (150)	$t \in [0, 30]$	1e-3	5

Table 13: Problem specific parameters, (σ) represents the verification error bound, (incr.) is short for increments and refers to the learning procedure discussed in context of Fig. 29.

The domain sizes are chosen in this way, so that interesting parts in the analytical solution are visible and as challenges available for ANDRe. During the experimental testing, we recognised that the neural network errors and especially the verification error $E_l^{VP}[\mathbf{P}_{m,l}]$ were not becoming arbitrarily small. In addition, the experiments revealed the problem specific dependencies of (local) minima locations in the weight space. Therefore we had to find (in an experimental way) the verification error bounds σ for each example IVP.

example	domain	h	l_1-error	l_∞-error
IVP in Eq. (147)	$t \in [0, 15]$	113	1.4499e-4	1.9268e-4
IVP in Eq. (148)	$t \in [0, 25]$	50	6.8152e-4	9.8980e-4
IVP in Eq. (149)	$t \in [0, 20]$	32	4.6545e-3	4.9861e-3
IVP in Eq. (150)	$t \in [0, 30]$	51	-	-

Table 14: Overview of the numerical results for the example IVPs in Eq. (147)–Eq. (150), (h) total number of learned subdomains.



(a) entire domain (b) clipped domain

Figure 30: **IVP in Eq.** (147) Comparison between (orange/solid) analytical solution and (black/dotted) ANDRe solution.

In Fig. 30, both the analytical solution (orange/solid) and ANDRe solution (black/dotted) are shown for the IVP in Eq. (147). Tab. 14 shows that 113 subdomains were necessary in order to satisfy the chosen verification error bound. The corresponding (averaged) l_1-error indicates a decent behaviour, which we consider to represent a reliable solution to the IVP. It also compares to the results from our SCNF experiments [82] (predefined equidistant subdomain distribution). The same IVP, solved with 100 equidistant subdomains, returned an l_1-error of 1.4339e-4. Therefore, ANDRe maintains the solution accuracy and comes with an advanced measurement metric.

The total number of training points for all subdomains is not equidistantly distributed. This circumstance is demonstrated in Fig. 30(b) for a clipped domain of Fig. 30(a). Because of the general trend of the solution, we expect the density to be higher at the peaks and dips, while declining in between. However, the subdomain $D_3 = [0.9229, 1.8027]$ is fairly large and includes two peaks and almost two dips as well. Compared to its adjacent subdomain $D_4 = [1.8027, 2.0089]$, the size of D_3 is unique, but also has a lower numerical error assigned. So the (local) numerical error in one subdomain, as well as the subdomain size itself do not necessarily share the global behaviour, where a higher amount of subdomains leads to a decreasing numerical error. [82]

Fig. 31 shows the subdomain distribution related to Fig. 30 in the beginning for $D = [0, 5]$. We find the domain size adjustment parameter δ to show a significant influence here. It appears to be very important where one subdomain ends, because this may cause the adjacent one to be more difficult to solve. Please note that this statement holds under the consideration of the chosen neural network parameters.

In contrast to the previous example, the IVP in Eq. (148) is solved on an even larger domain with extensively increasing values. The results are shown in Fig. 32 and aim to show that ANDRe is capable of solving time-integration problems on large domain with small neural networks. This is of particular importance as the domain size has been identified as an intricate parameter of the underlying problem, see also the detailed study in [81].

As displayed in Fig. 32, the ANDRe solution fits the analytical solution (orange) again on a qualitative and useful level. In total, the algorithm has finished after splitting the solution domain into 50 subdomains with the averaged l_1-error of $\Delta\phi_1 = 6.8152\text{e-}4$

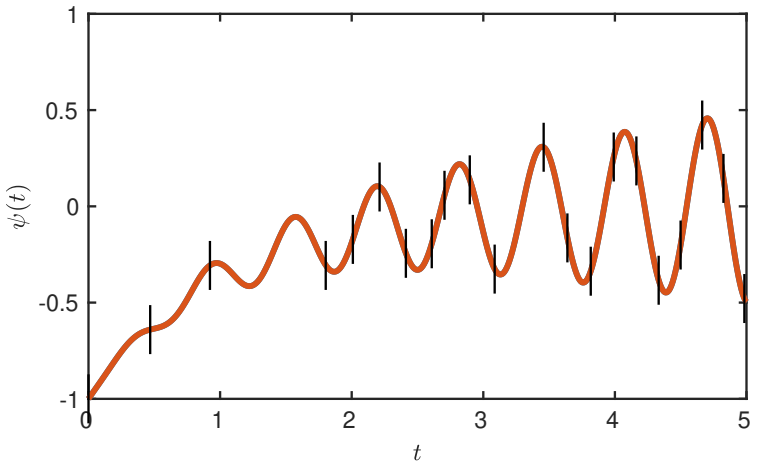
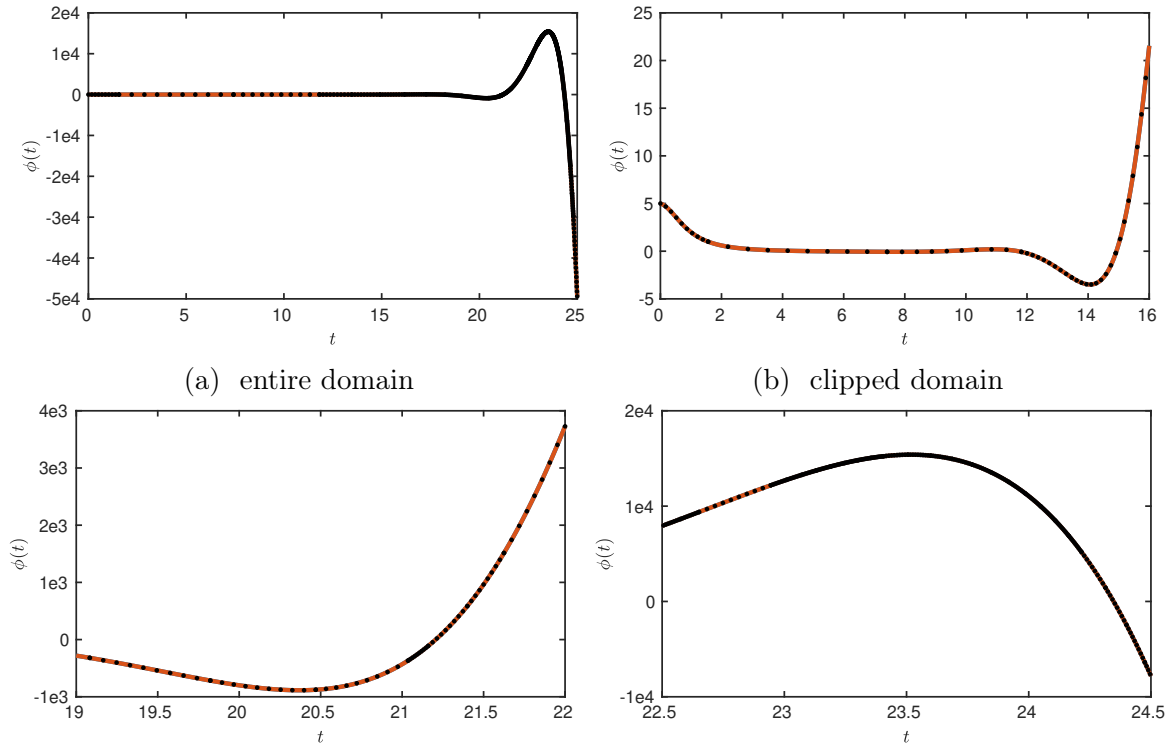


Figure 31: **IVP in Eq.** (147) ANDRe subdomain distribution for a cut-out of Fig. 30a, (orange/solid) analytical solution, (black/marked) subdomain boundaries, cf. Fig. 14.



(a) entire domain

(b) clipped domain

(c) clipped domain

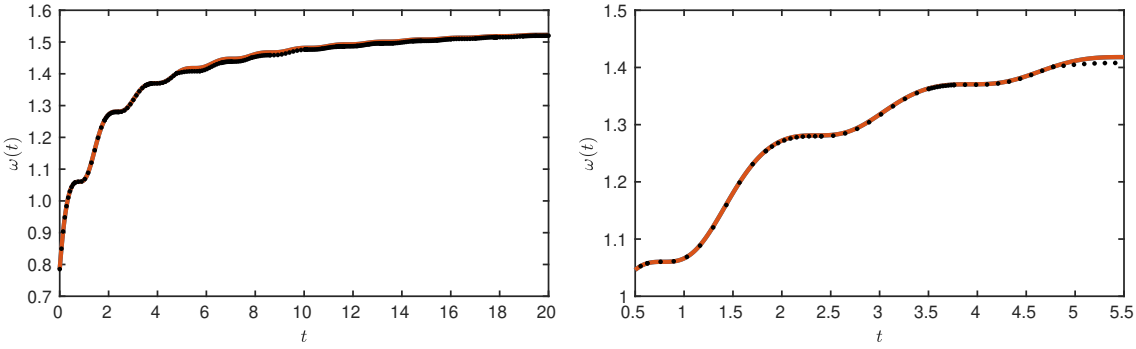
(d) clipped domain

Figure 32: **IVP in Eq.** (148) Comparison between (orange/solid) analytical solution and (black/dotted) ANDRe solution.

(cf. Tab. 14). In comparison, $\Delta\phi_\infty$ differs more from $\Delta\phi_1$ than the counterparts for IVPs in Eqs. (147),(149).

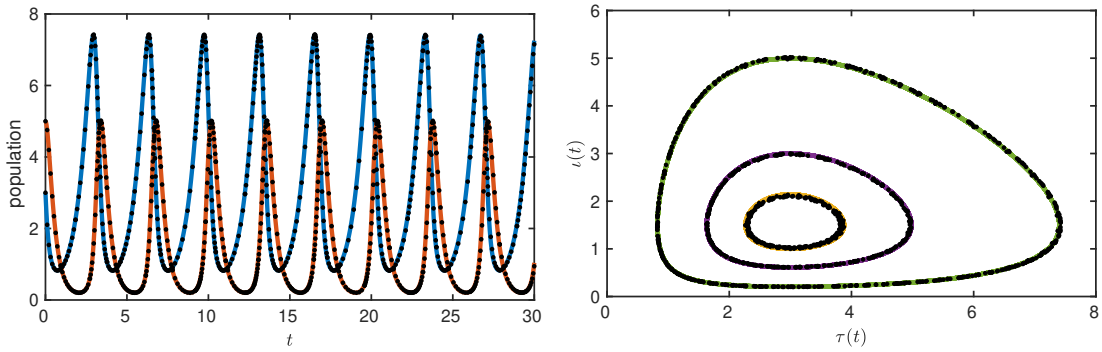
While Fig. 32(a) shows the ANDRe solution for the entire domain, Figs. 32(b),32(c) and 32(d) are zoomed in, to provide a more detailed view on certain areas. In Figs. 32(b) and 32(d), we observe the local extreme points to be covered by more densely

packed subdomains, especially the maximum in range of high function values. This may indicate, that the domain refinement not only depends on the complexity of a certain region, but on finding suitable minima in the weight space in order to get the training error below the verification error bound σ. This however, does not hold for the extreme point in Fig. 32(c). We see the local minimum to be covered by approximately equidistant subdomains (on a qualitative level) up to $t = 21.0362$. The next three subdomains however, are densely packed, only to be stretched again afterwards.



(a) entire domain (b) clipped domain

Figure 33: **IVP in Eq.** (149) Comparison between (orange/solid) analytical solution and (black/dotted) ANDRe solution.



(a) Population over time, $\tau(0) = 3, \iota(0) = 5$ (b) Comparison between $\tau(t)$ and $\iota(t)$ in phase space with $\tau(0) = 3$ and (yellow) $\iota(0) = 1$, (purple) $\iota(0) = 3$, (green) $\iota(0) = 5$.

Figure 34: **IVP in Eq.** (150) Comparison between (coloured/solid) Runge-Kutta 4 solution with 1e3 grid points and (black/dotted) ANDRe solution.

Results for the IVP in Eq. (149) are displayed in Fig. 33. We decided to investigate this example because of the saddle points, which are repeatedly occurring. We observe a reliable solution approximation in the beginning of Fig. (33)(a). However, from subdomain D_7 and $t = 4.7760$ on, we can see that ANDRe starts to differ from the analytical solution. Although it keeps the general trend, and seems to converge against the analytical solution again in the end, the differences in this region are remarkable. This also marks a turning point computational-wise, which we will discuss more in detail in the corresponding experimental paragraph. Nonetheless, we had to decide to

limit the verification error bound to $\sigma = 1e0$, since the computation with a lower error bound always got stuck around this area. This means both the Adam learning rate and the number of hidden layer neurons started to increase heavily. Although one would suggest, based on the universal approximation theorem, that at some point ANDRe would move on, we cancelled the time consuming computation at this point. This circumstance is definitely interesting to further investigate. We do not see a limitation of ANDRe here, since on a theoretical level, there should be an amount of hidden layer neurons, which is able to finish the computation even for a smaller verification error bound σ.

Fig. 33(b) confirms the results from the previous examples, that the appearance of local extreme points (saddle points in this case) not necessarily result in more densely packed subdomains directly at their location. However, the final distribution has some packed subdomains involved, prior to the local extreme points. The reason for this seems to be that a saddle point can not be part of a subdomain that is too large. Therefore the antecedent subdomain results in a smaller size so that the saddle point can be part of an appropriate sized subdomain.

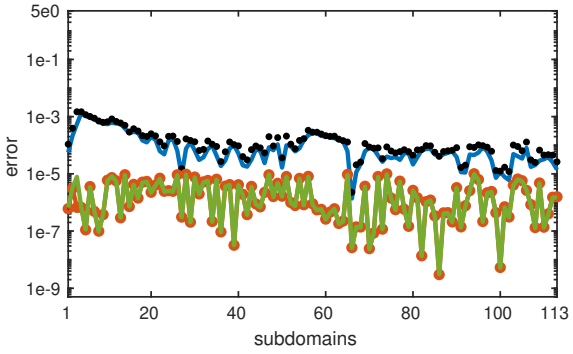
In Fig. 34 both the Runge-Kutta 4 solutions and the ANDRe solutions are shown. For a fair comparison on the quantitative side, both method should use equal amount of training points, which in this case would arise from ANDRe solution. However, we are more interested in a qualitative comparison, since the Runge-Kutta 4 is known to provide very good results. ANDRe found a useful solution for the Lotka-Volterra equations in Fig. 34(a), since there are only minor differences from the qualitative perspective.

Fig. 34(b) shows the solution related to three different initial values for the predators. Let us note, that although Fig. 34(b) only displays the solution at the training points (the same holds for the previous example IVPs), the trained SCNF is capable of evaluating the solution at every arbitrary discrete grid point over the entire domain, which is an advantage over numerical integration methods.

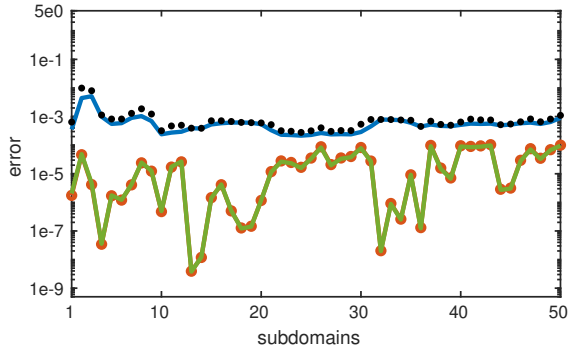
4.3.5 Numerical and neural network errors

The measurement metrics (numerical and verification error) are highly relevant to discuss for ANDRe. In the subsequent diagrams we show the l_1-error (blue/solid), the l_∞-error (black/dotted), as well as the verification error (green/solid) and the training error (orange/marked) over the successfully learned subdomains.

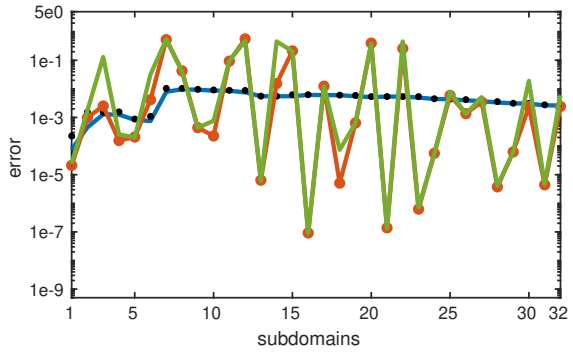
Commenting on the relation between the verification and the training error in Fig. 35a for the IVP in Eq. (147) (cf. Eqs. (141),(142)), we see that both are mostly equal. This implies, that the corresponding subdomains have been effectively learned up to the desired state. Turning to the numerical errors, in regions where $\Delta\psi_1$ shows an approximately constant slope, e.g., D_{58} to D_{64}, $\Delta\psi_\infty$ appears to deviate less from the averaged l_1-error. Since the main goal of ANDRe is to make use of the verification error as a measurement metric for the numerical error, finding a relation between both is desirable. In Fig. 35a, there are some regions that may indicate such a relation. The network errors from around D_{58} to D_{64} show a slightly decreasing behaviour, only to highly differ in D_{65}. In comparison, $\Delta\psi_1$ also slightly decreases for some subdomains and dips, together with the network errors, for D_{65}. However, in other regions almost no consistent relation is visible.



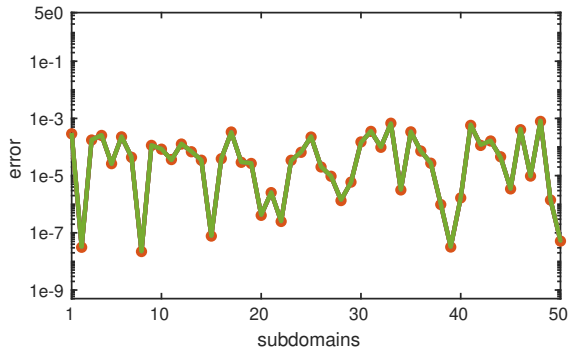
(a) **IVP in Eq.** (147)



(b) **IVP in Eq.** (148)



(c) **IVP in Eq.** (149)



(d) **IVP in Eq.** (150)

Figure 35: Error comparison, (blue/solid) numerical error, (black/dotted) infinity norm, (orange/marked) training error, (green/solid) verification error.

The statements made above also apply to Fig. 35b for the IVP in Eq. (148), where

one may find some relation in the beginning, while afterwards the l_1-error is almost constant and the network errors drop and rise by several orders.

However, let us comment on the behaviour of both verification and training error, displayed in Fig. 35b. While both match, they undergo the preset of $\sigma =$ 1e-4 in some cases by several orders. Two adjacent subdomains may have a verification/training error with significant differences. Although we find a local maximum for the l_1-error in the beginning, it decreases afterwards and remains in a certain region with local minima and maxima. However, we find the decreasing behaviour to be an important characteristic compared to numerical methods, where one would expect the error to accumulate.

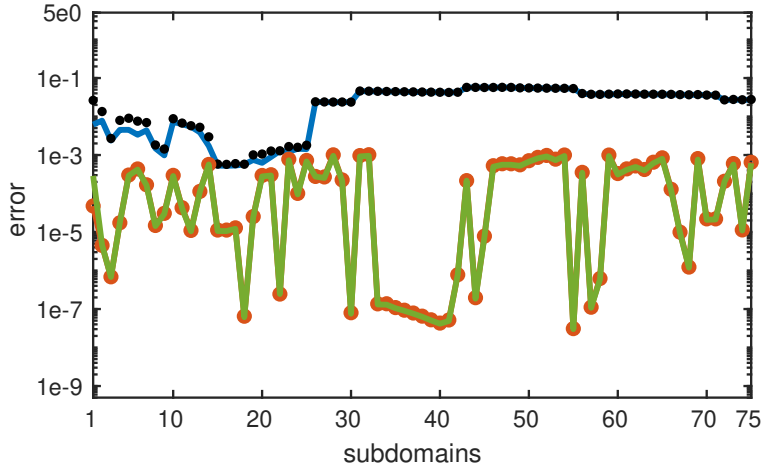


Figure 36: **IVP in Eq.** (148) Solved with ANDRe and alternative SCNF approach (mTSM) in Eq. (152) Error comparison, (blue/solid) l_1-error, (black/dotted) ℓ_∞-error, (orange/marked) training error, (green/solid) verification error.

Commenting on Fig. 35c for the IVP in Eq. (149), the possible local relations between the numerical and network errors seem to have turned into a chaotic state. For the corresponding verification error bound $\sigma =$ 1e0, even the network errors are most of the time not equal anymore. In contrast the both Fig. 35a and Fig. 35b, the numerical errors lay in between the values of the network errors, which is highly interesting. This more or less confirms, that even if the verification/training error indicate a shallow (local) minimum in the weight space, the numerical error can still be useful. Although $\Delta\omega_1$ and $\Delta\omega_\infty$ are, expect for the beginning, almost constant throughout the domain. That circumstance inherents both good and bad news. The latter connects to the apparent random behaviour, while the good news is that even though the network errors appear to be random, the IVP can still be considered to be solved.

The results for the Lotka-Volterra equations in Fig. 35d also seem to indicate a chaotic behaviour. That is, the local minima of orders around $\approx$ 1e-8 relate to arbitrary subdomains, that are not connected to, e.g., the periodic extreme points of the solution.

The diagram in Fig. 36 shows the results for a different SCNF approach (mTSM) [15, 82], combined with ANDRe. In contrast to the neural forms approach (using the initial condition to construct the neural form), now we directly combine with neural

networks with the polynomial ansatz [82, 81]:

$$\tilde{\phi}_C(t_{i,l}, \mathbf{P}_{m,l}) = N_1(t_{i,l}, \mathbf{p}_{1,l}) + \sum_{k=2}^{m} N_\kappa(t_{i,l}, \mathbf{p}_{k,l})(t_{i,l} - t_{0,l})^{k-1} \tag{152}$$

Since the initial condition is not included in Eq. (152), it appears as an additional term directly in the cost function. Here, we use

$$g(t) = \frac{1 + \frac{1}{1000} e^t \cos(t)}{1 + t^2} \tag{153}$$

as a shortcut for:

$$\begin{aligned} E_l[\mathbf{P}_{m,l}] = \frac{1}{2(n+1)} \sum_{i=0}^{n} \quad & \left\{ \dot{\tilde{\phi}}_C(t_{i,l}, \mathbf{P}_{m,l}) + g(t) + \frac{2t}{1+t^2} \tilde{\phi}_C(t_{i,l}, \mathbf{P}_{m,l}) \right\}^2 + \\ & \frac{1}{2} \left\{ N_1(t_{0,l}, \mathbf{p}_{1,l}) - \tilde{\phi}_C(t_{0,l}, \mathbf{P}_{m,l}) \right\}^2 \end{aligned} \tag{154}$$

Hence, the initial condition is learning by the first neural network. The cost function construction concept is very similar to physics-informed neural networks [64, 65]. However, the polynomial approach ansatz is different in this context. Let us recall, that the initial values follow $\tilde{\phi}_C(t_{0,1}, \mathbf{P}_{m,1}) = \phi(0)$ in the leftmost subdomain and $\tilde{\phi}_C(t_{0,l}, \mathbf{P}_{m,l}) = \tilde{\phi}_C(t_{n,l-1}, \mathbf{P}_{m,l-1})$ elsewhere. This concept avoids possible difficulties in constructing a suitable neural form. Since both approaches (Eq. (111) and Eq. (152)) only differ in their cost function construction, it appears natural to compare them. That is, the results in Fig. 36 compare to Fig. 35b. The behaviour of both verification/training error does not seem to be connected to the numerical error, for both methods. Fig. 36 shows less accurate results for the l_1-error. The loss of accuracy possibly relates to the fact, that here the new initial condition for the next subdomain is not fixed by adding it to the neural form. It rather has to be learned again, which in practice may harm the usefulness of this approach. The gap between both l_1-error and l_∞-error closes at a certain point.

When turning to Fig. 37, we observe that the parameter adjustment brought the Adam learning rate (coloured) up to various values in order to finish learning the subdomains. Additionally, the necessary number of hidden layer neurons also heavily increases towards higher subdomains.

Although the results in terms of the numerical error are not better than in Fig. 35b, we find here a confirmation of the automatic parameter adjustment. With this feature, ANDRe was able to solve the IVP. That is, we see our approach to enable the parameter adjustment when necessary, to be justified by the results. However, this does not support the overall usage of this alternative SCNF approach in context of ANDRe.

4.3.6 Method and parameter evaluation

In this paragraph we investigate and evaluate different parts of the method.

In Fig. 38, the sizes of the learned subdomains for the IVP in Eq. (147) are shown. The general trend points towards smaller subdomains throughout the computation. However, we witness that there are local differences with bigger or smaller subdomains

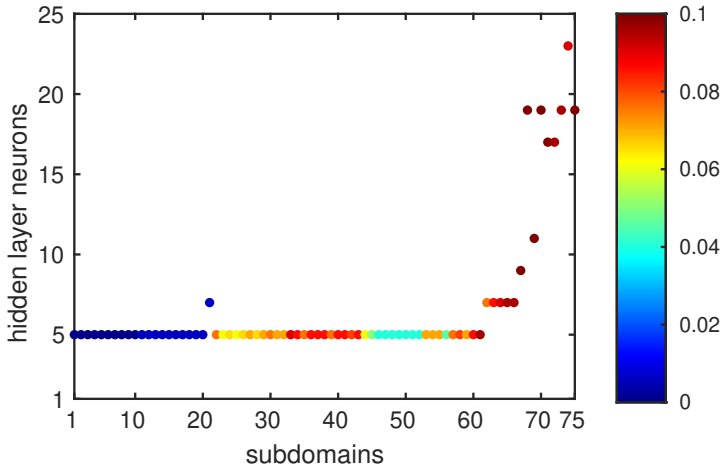


Figure 37: **IVP in Eq.** (148) Solved with ANDRe and alternative SCNF approach (mTSM) in Eq. (152) Visualisation of the automatic parameter adjustment (hidden layer neurons and learning rate) over the subdomains, (coloured) Adam learning rate α.

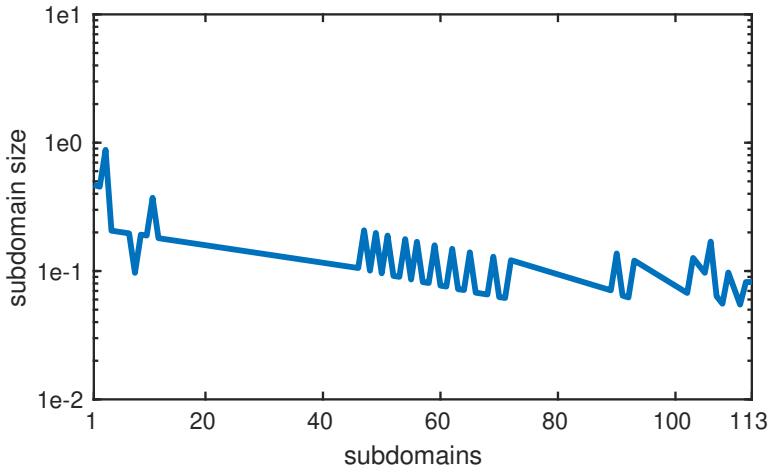


Figure 38: **IVP in Eq.** (147) Visualisation of the learned subdomain sizes.

and this is what we expect from ANDRe. The subdomain size is reduced until it is sufficiently small and that can be individual for each part of the solution. Nonetheless, let us compare both the numerical error in Fig. 35a and the subdomain sizes in Fig. 38. In the first ten subdomains there seems to be a certain correlation, a larger size in this range results in a larger numerical error. A smaller verification error bound σ to deal with the discrepancy between verification and training error in Fig. 35a may have resulted in another size reduction with better results. However, the statement that a smaller (local) subdomains size implies a better numerical error does not hold here. Although one of the complex conditions employed each subdomain to not become smaller than 0.1, we can see in Fig. 38 that certain subdomains towards the end undergo this preset condition. This is related to the fact, that the subdomain size is first reduced and then checked for its size. Therefore it is still possible for a subdomain, to slightly undergo the size of 0.1. However, and as the results confirm, a further size reduction is not possible. Turning to Fig. 39, the (learned) neural network outputs are displayed for

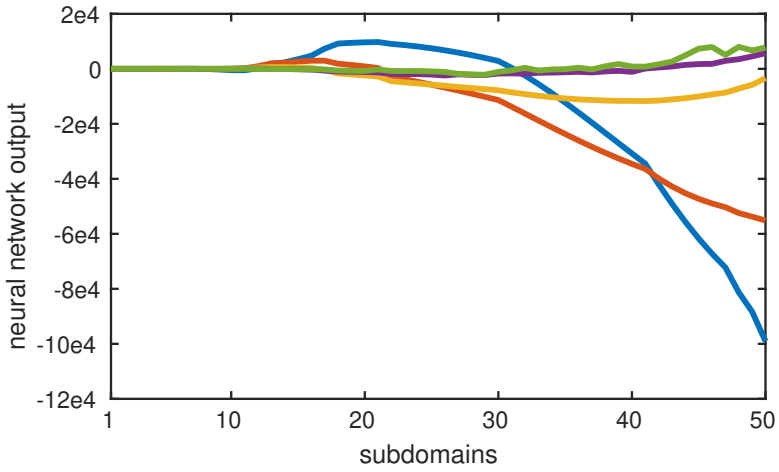


Figure 39: **IVP in Eq.** (148) Output of the incorporated SCNF neural networks, (blue) N_1, (orange) N_2, (yellow) N_3, (purple) N_4, (green) N_5.

the incorporated SCNF (cf. Eq. (111)) order $m = 5$ of the IVP in Eq. (148) (cf. Fig. 32). That is, the five displayed graphs each represent one neural network output over the subdomains. We see the first and second SCNF orders to dominate the results for higher numbers of subdomains. However, higher orders also contribute to the solution, making our approach adaptive in the approximation order, indirectly. Let us recall, that the factors $(t_{i,l} - t_{0,l})^\kappa$ for the different neural networks $N_\kappa(t_{i,l}, \mathbf{p}_\kappa), \kappa = 1, \dots, 5$, dictate the impact of each neural network since they act as a scaling factor. Hence, subdomains with a size below 1 imply a smaller influence of higher SCNF orders. This circumstance can be challenging for an IVP solution with large values. Nonetheless, we see that our SCNF algorithm was able to solve the IVP in Eq. (148), even tough it incorporates fairly large values.

t_{25}	t_{26}	$\Delta\psi_1$	E_{25}^{TP}	E_{25}^{VP}	α
5.5953	15.000	1.3557	22.888	23.598	1e-3
5.5953	10.298	2.3780	1.7622	22.695	1e-3
5.5953	7.9465	1.3774	6.8529	34.448	1e-3
5.5953	7.9465	0.3868	10.640	9.9730	6e-3
5.5953	6.7709	3.4447e-2	3.2209e-2	3.9242e-2	6e-3
5.5953	6.1831	1.1881e-2	3.4791e-2	3.5682e-2	6e-3
5.5953	5.8892	3.3488e-4	1.0875e-4	1.0779e-4	6e-3
5.5953	5.7423	1.6882e-4	2.4565e-6	2.3747e-6	6e-3

Table 15: **IVP in Eq.** (147) Results for a complete learning procedure for one subdomain.

In Tab. 15, quantitative results for the entire learning process of one subdomain of the IVP in Eq. (147) are displayed. The left subdomain boundary t_{25} remains constant while the right subdomain boundary t_{26} is adjusted as in Eq. (139). The verification error values E_{25}^{VP} demonstrate the appearance of non-uniform learning during the solution process and show how important the verification error and the parameter adjustment are. While E_{25}^{TP} decreases (as intended) for the first two subdomain size reductions, it

increases for the third one, which leads to a growth of the initial learning rate α. Now for the same subdomain size, E_{25}^{VP} decreased significantly (while E_{25}^{TP} has increased again). That circumstance enables ANDRe to continue reducing the subdomain size until it is sufficiently small.

σ	h	$\Delta\phi_1$	$E^{VP}[\mathbf{P}_{m,l}]$	$E^{TP}[\mathbf{P}_{m,l}]$
1e-1	37	5.5512e-2	1.8907e-2	1.8537e-2
1e-2	39	1.0749e-2	1.5714e-3	1.6255e-3
1e-3	47	6.2122e-3	1.1582e-4	1.1917e-4
1e-4	50	6.8152e-4	2.6581e-5	2.7788e-5
1e-5	59	3.0005e-4	1.9260e-6	2.2057e-6
1e-6	74	1.1870e-4	2.1157e-7	2.2076e-7

Table 16: **IVP in Eq.** (148) Results for different σ, on domain $t \in [0, 25]$.

From the perspective of employing a condition for minimising the cost function, the question arises how the algorithm outcome is affected by different error bound values σ. Tab. 16 shows the overall l_1-error, verification error and training error for different σ regarding the IVP in Eq. (148). The choice of σ has a direct impact on each error value, as they all decrease the smaller σ gets. However, an experiment for $\sigma = 1e$-7 did not finish learning the subdomains. We terminated the computation after the number of hidden layer neurons crossed fifty one. In this subdomain, the smallest verification error was 1.9881e-7 but the optimisation did not manage to go below σ = 1e-7. This phenomenon may again relate to the complexity of the cost function energy landscape. Either such a local minimum could not be found by the optimiser for various reasons, or even the global minimum is still too shallow for that error bound.

Results in Tab. 16 confirm the results from [82], where an increasing number of subdomains shows a decreasing numerical error.

δ	h	$\Delta\omega_1$
0.9	5	6.2398e-3
0.8	5	7.0250e-4
0.7	4	3.4629e-3
0.6	5	9.1571e-2
0.5	4	6.9541e-4
0.4	5	6.2569e-3
0.3	5	4.4258e-3
0.2	7	1.1200e-3
0.1	13	2.7217e-4

Table 17: **IVP in Eq.** (149) Results for different domain size reduction parameter values δ, on domain $t \in [0, 5]$.

Last but not least we discuss experimental results for different domain resize parameter values δ in Tab. 17 for the IVP in Eq. (149). The higher this value is set, the more aggressive each subdomain is reduced in size. On the other Hand, the smaller δ is, the more careful the subdomains are reduced in size. However, one would expect the necessary amount of subdomains to increase, the higher the resize parameter is.

But in reality the results and the amount of subdomains are comparable for all δ, if we exclude $\delta = 0.1$. On the l_1-error side, except for $\delta = 0.6$, all the results are comparable. Although the results are highly problem specific and may change with a larger domain size, we find $\delta = 0.5$ to provide the best mix with $h = 4$ and $\Delta\omega_1 = 6.9541\text{e-}4$. This domain size parameter was used for all the computations.

4.3.7 Comparison with numerical methods

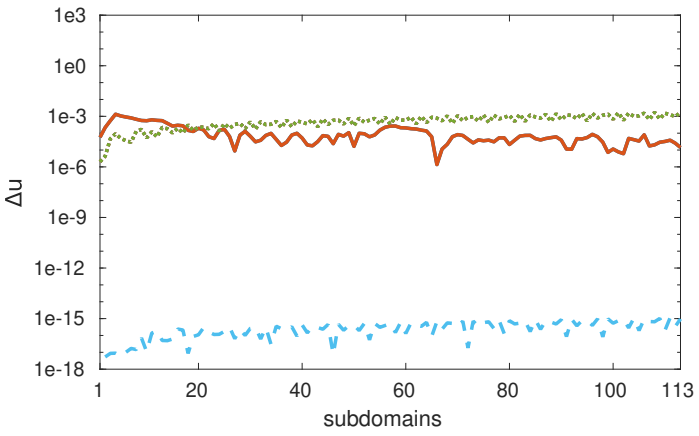
Again we want to put the current results in correlation with the Euler method and Runge-Kutta 4. The qualitative results in Fig. 40 show various interesting properties. In general, the relation between the neural forms approach and the numerical methods are very similar to the findings for SCNF in Sec. 3.4.6, where RK4 outperformed the other approaches. Expect for Fig. 40(c), the Euler method is accuracy-wise behind ANDRe with TSM. In Fig. 40(a), we observe that the overall trend for ANDRe throughout the subdomains is slightly increasing in accuracy, while Euler and RK4 show a somewhat decreasing behaviour. We have already stated in previous sections, that a subsequent subdomain only depends on the provided initial conditions from the last subdomain. That is, we see evidence that the optimisation is flexible enough to damp inaccuracies in these initial conditions. However, although we see a slight decrease in accuracy for Euler and RK4 here, this characteristic seems to depend on the problem in Eq. (147). Namely in Fig. 40(c), both Euler and RK4 show the opposite behaviour we an overall increasing accuracy. In this figure, as mentioned before, Euler even outperforms ANDRe with TSM.

Prior to discussing the highly interesting results in Fig. 40(b), let us comment on Tab. 18 in this context. Again and with no doubt, RK4 provides superior results. Nonetheless, we see that for the IVP in Eq. (147) and Eq. (148), TSM ranks above Euler. However, both Euler and RK4 show for the IVP in Eq. (149) their best accuracy, while ANDRe with TSM has the worst here. So in general, we again see evidence that finding a general relation between the characteristics of all methods is actually very difficult.

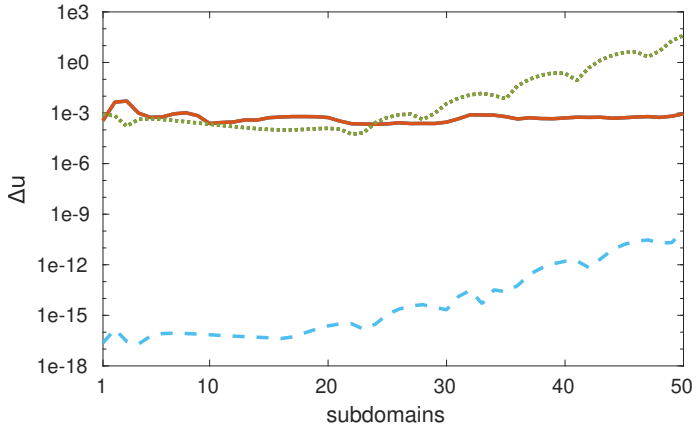
IVP	TSM $\mathbf{p}^{init}_{deter}$	Euler	RK4
IVP in Eq. (147)	1.4499e-4	7.2374e-2	3.9164e-9
IVP in Eq. (148)	6.8152e-4	82.1303e0	8.6055e-7
IVP in Eq. (149)	4.6545e-3	2.4318e-3	4.5611-13

Table 18: Comparison between TSM, mTSM with deterministic/random weight initialisation (100 subdomains) and both Euler method and RK4 (1 subdomain, 500 grid points).

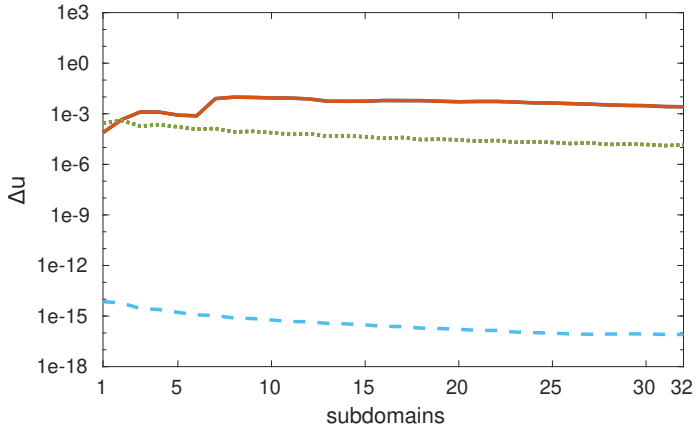
Nonetheless, now we want to comment on Fig. 40(b). Here we see that ANDRe with TSM has an overall decent accuracy, even in the region with large function values (towards larger domain parts). Although Euler is slightly better in accuracy in the beginning, towards the larger domain parts (with large function values), the methods starts diverging. The behaviour of RK4 is very similar, but the accuracy remains on a highly useful level. However, further enlarging the solution domain may favour TSM over RK4 in that region. This diverging behaviour may indicate that we have to



(a) **IVP in Eq.** (147)



(b) **IVP in Eq.** (148)



(c) **IVP in Eq.** (149)

Figure 40: Comparison of neural forms methods and numerical methods (500 grid points) with an equal amount of subdomains, (orange/solid) TSM $\mathbf{p}_{deter}^{init}$, (green/dotted) Euler method, (light blue/dashed) RK4.

deal with stiffness phenomenon here. Another indication for this assumption is that increasing the number grid points for Euler to 1e7, than the overall numerical error (computed over the entire domain [1 subdomain]) reduces to 4.0093e-3. Requiring an unexpected high amount of grid points (extremely small step size) for a useful solution is a basic characteristic for Euler when dealing with stiff differential equations. We

see that the neural forms approach can, although the accuracy is far away from being state-of-the-art, deal with stiffness and appears to be unaffected by this phenomenon. In general, investigation second order optimisation in this context could reveal additional interesting behaviour and perhaps compete with numerical methods.

4.3.8 Conclusion

The proposed ANDRe is based on two components. First, the resulting verification error arising from the participating subdomain collocation neural form (SCNF) acts as a measurement metric and refinement indicator. The second component is the proposed algorithm which refines the solution domain in an adaptive way. We find ANDRe to be a dynamic framework adapting the complexity of a given problem. We have shown that the approach is capable of solving time-dependent differential equations of different types, incorporating various interesting characteristics, in particular including large domains and extensive variations of solution values.

In contrast to numerical solution methods for solving initial value problems, the numerical error does not inevitable accumulate over the subdomains. It can rather decrease again due to the flexibility of the neural forms approach. A significant advantage of ANDRe is the verification step to make sure that the solution is also useful outside of the chosen training points. All this makes ANDRe a unique and conceptually useful framework.

However, several questions remain open for future work. While there seems to be a certain and natural correlation between the neural network and the numerical error, in reality this correlation appears to be sometimes a sensitive issue. It is unclear yet, whether some minima in the cost function energy landscape contribute better to the numerical error, or not. However, we find the verification error to already serve as a useful error indicator in ADNRe. In addition, we would like the numerical error to proportionally correspond to the neural network verification. If we could manage to achieve an improvement in the correlation between both errors or understand the relation more in detail on the theoretical level, we think that the ANDRe approach can perform even better in the future.

We also find relevant to further investigate the computational parameters and fine tuning the parameter adjustment part of ANDRe. The verification step may be considered as a part in the optimisation process, to predict early, whether a further optimisation in the corresponding subdomain is useful or a size reducing is mandatory. This could lower the computational cost but has to be incorporated and tested carefully to not lose any information during the optimisation process.

Since ANDRe represents an additional discretisation in time, the approach should also work for PDEs with both time and spatial components and it appears natural to extend in future work the method to multidimensional differential equations.

5 Discussion

With the intention of further investigating approaches from a numerical point of view, the focus began to lay on two methods (TSM [13], mTSM [14, 15]) employing and incorporating small neural network architectures and intuitive procedures to approach a given differential equation with neural forms. Besides the solution approaches, different first order optimisation methods (Adam [44], backpropagation [43]) have also been taken into account.

The first steps for the research in that direction, were based on the question, how both TSM and mTSM work and how stable these methods are with respect to parameter changes. That being said, the computational study in Section 2.4 investigated the effects of varying parameters related to a stiff initial value problem and the used feedforward neural network. At this point it is important to recall the focus on first order optimisation methods (backpropagation and Adam) in this thesis. The quantity of interest was set to be the numerical error. For most experiments, and separate from testing neural network architecture configurations itself, a basic neural network with one hidden layer with five neurons was incorporated. As an important result, it turned out that the numerical error is highly dependent on the chosen parameters and methods. That is, when testing the impact of the domain size on the solution, TSM provided useful results on smaller domains, while failing to approximate the solution on larger domains. The behaviour of mTSM was completely different and the domain size did almost not affect the numerical error in a bad way. However, especially the neural network weight initialisation turned out to be a sensitive topic. The differences between the initialisation with deterministic values ($\mathbf{p}_{deter}^{init}$) and random values ($\mathbf{p}_{rnd}^{init}$) were significant. While TSM did not provide any useful results with $\mathbf{p}_{deter}^{init}$ and the used first order optimisation methods, mTSM has shown reliable results with both initialisation methods. Nonetheless, experiments on random weights with several computations and otherwise unchanged parameter configurations revealed how sensitive this topic is. Although $\mathbf{p}_{rnd}^{init}$ only has been altered in a small range, the resulting numerical error was in some cases far away from being useful. This is a strong indication that the weight space, or energy landscape, is very difficult in shape. Since the initial weights dictate the starting point for the optimisation in the energy landscape, even minor changes may significantly affect the gradient direction for the first order methods. Therefore the trained neural network may provide results that can be considered as useful or not, depending on the final location of the weights. However, in all the corresponding results, $\mathbf{p}_{rnd}^{init}$ did provide better results than $\mathbf{p}_{deter}^{init}$. The overall best performance, in terms of the lowest numerical error, came with mTSM, Adam and $\mathbf{p}_{rnd}^{init}$. Nonetheless, when considering additional computational numerical analysis, e.g. sensitivity, condition or stability, one may want the outcome to remain unchanged in several computations and not to depend on a good or poor initialisation with $\mathbf{p}_{rnd}^{init}$. Therefore, after the computational study, the next steps in research are dedicated to improve the numerical results with the use of deterministic initial weights. Nonetheless, the comparison with numerical methods like the Euler method, Runge-Kutta 4 or BFGS (second order optimisation) with Wolfe-Powell line search especially revealed the difficulty of combining TSM and first order optimisation. Although BFGS was still behind Runge-Kutta 4, the provided results were above both TSM and mTSM. So when making conclusions based on the results from Section 2.4, the statements most likely only hold for the employed

first order optimisation methods.

Further research in that direction was documented in Section 3.3. While focusing on IVPs, the classic neural form proposed by the authors of TSM [13] resembles a first order polynomial in the domain variable. The incorporated neural network in this context may remind of a non-constant polynomial coefficient. Therefore it arises naturally to try to increase the polynomial order, including several (small) neural networks as the function coefficients. The extension to collocation polynomial neural forms (CNFs) incorporates, depending on the neural forms order, several neural networks of the same architecture with unique sets of weights (in the trained state). However, although the number of hidden layer neurons in total were now increased, the behaviour was completely different to just increasing the number of neurons in the classic neural form, as experiments in the previous experiments section have shown. Whereas the flexibility now raised from the neural networks in combination with the domain variable and the chosen polynomial order. Results have shown a significant increase in the reliability of $\mathbf{p}_{deter}^{init}$ for TSM with a raise of the polynomial order. It turned out that one neural network is not capable of having its weights adjusted in an adequate way to solve the model IVP with TSM and first order optimisation. Now, several networks incorporated in the CNF were capable of finding a suitable minimum with $\mathbf{p}_{deter}^{init}$. Turning to mTSM, expanding the approach to a polynomial representation was also possible but did barely make any profit out of more then two neural networks. However, with one more neural network incorporated, the numerical error was lowered for $\mathbf{p}_{deter}^{init}$ in some cases as well. Deterministic initial weights in some cases even outperformed the random initialisation regarding mTSM. However, even with the CNF extension, TSM still struggled to solve the model problem on larger domains and a further improvement without more and more complex neural network architectures was still a goal.

The unique construction characteristic of the TSM classic neural forms not only resembles a first order polynomial for IVPs, but also directly embeds the given initial value. Since the previous discussed results show that TSM works fine on smaller domains, the provided solution at the last grid point of the domain may identify as a new initial value for an adjacent domain. Therefore it appeared to be possible to split the entire domain into subdomains with the domain segmentation approach. As mentioned above, each subdomain then provided a new initial value for the neighbouring subdomain. That is, the subdomain collocation neural form (SCNF) was also introduced in Section 3.3 and solved separately in each domain segment, from left to right. Setting a computed value to be the starting point for the approximation in another optimisation cycle may cause doubts at a first glance. From a numerical point of view, a possible computation-related numerical error may accumulate over time, especially when the initial value is already disturbed. However, a major strength of the neural forms approach in general is that finding the optimal weights in each subdomain can be independent to the already learned weights in the previous subdomain. Although the constructed cost function did not change its appearance in different subdomains, the different grid points incorporated may change the corresponding energy landscape. Therefore the error in one subdomain can be unsatisfying, while in an adjacent one it appears to be useful again. After initialising the neural networks with $\mathbf{p}_{deter}^{init}$ in each domain fragment, the optimiser may find and follow a different and unique path in the weight space. The results have shown that the CNF is not be able to approximate the investigated example IVP on the given domain. Whereas the SCNF provided useful

results for both TSM, mTSM and $\mathbf{p}_{deter}^{init}$, $\mathbf{p}_{rnd}^{init}$. However, the domain segmentation approach may be more suitable for TSM than mTSM, since the latter has to learn the new initial value in a supervised way. The results showed that directly incorporating the initial value in the SCNF tended to work better. Nonetheless, for both approaches, a very important result was that increasing the number of subdomains can lower the numerical error significantly, especially for TSM. The latter result also showed that $\mathbf{p}_{deter}^{init}$ now competes with $\mathbf{p}_{rnd}^{init}$ is several cases. Therefore, choosing the number of subdomains may replace the fine tuning of several other parameters, like the number of hidden layer neurons or the number of training points. The final results for the domain segmentation approach demonstrated another strength of the neural forms approach. The cost function is able to take certain characteristics into account, simply by adding them as additional terms. Therefore, e.g., systems with quadratic invariant expressions can also be dealt with. This only requires additional terms in the cost function, but no general rework in the approach.

If the numerical error decreases with an increasing number of subdomains, one may consider to reverse this statement. That is, the research in Section 4.3 investigated whether a certain, user defined, error bound can determine the necessary amount of subdomains for the TSM SCNF. The results in this section in general lead to a framework, combining the advantages of two well-known numerical concepts: adaptive mesh refinement and domain decomposition. Therefore, the resulting framework was called adaptive neural domain refinement (ANDRe), which feature two main components: *(i)* an error indicator and *(ii)* an algorithm to manipulate the subdomains. The entire solution domain can be split into domain fragments and certain subdomains can be reduced in size until the error indicator meets a predefined error bound. ANDRe was tested with different IVPs with interesting, yet difficult, characteristics and has shown that the solution of IVPs not necessarily requires the subdomains to be distributed in an equidistant way. The results indicated on a qualitative level a very good approximation.

Going further into detail on ANDRe, the error indicator was first related to the training error, which is defined as the value returned by the cost function. The smaller the training error, the better the learning problem may be solved. However, it turned out that the numerical error and the training error may not be proportional to each other. Moreover, details on the relation are yet unclear. Another cost function related error was introduced, the verification error which evaluated the learned SCNF at grid points intermediate to the training points. This is also a strategy to detect overfitting. In Section 2.4, the behaviour of both errors are directly compared for a stiff model IVP. Several local minima found by the optimiser share error values that do not lay in the exact same range. Although the locations appear to be most of the time similar, the training error often appears to be much smaller than the numerical error. In a next step, the training error was eliminated as the main measurement metric and completely replaced by the verification error in the ANDRe algorithm. However, the relation between the verification error and the numerical error also remains unclear. Sometimes a very small verification error lead to a very small numerical error, other times this statement did not hold. Right now this seems to limit the effectiveness of ANDRe, although the framework has shown its capabilities. However, in its current state one may simply choose a sufficiently high number of subdomains to receive useful approximations. Therefore, a further understanding of the energy landscape and therefore the relation between both error values may significantly increase the reliability of ANDRe.

References

[1] M. Hanke-Bourgeois: *Grundlagen der Numerischen Mathematik und des Wissenschaftlichen Rechnens*, 2nd edition, Teubner Verlag / GWV Fachverlage GmbH, Wiesbaden (2006). ISBN:3-8351-0090-4

[2] H.M. Antia: *Numerical methods for scientists and engineers*, 3rd edition, Hindustan Book Agency, New Delhi (2012). doi:10.1007/978-93-86279-52-1

[3] E. Hairer, S.P. Nørsett, G. Wanner: *Solving ordinary differential equations 1: non-stiff problems*, 2nd edition, Springer Series in Computational Mathematics, Springer-Verlag, Berlin Heidelberg (1993). doi:10.1007/978-3-540-78862-1

[4] E. Hairer, G. Wanner: *Solving ordinary differential equations 2: stiff and differential-algebraic problems*, 2nd edition. Springer Series in Computational Mathematics. Springer-Verlag, Berlin Heidelberg (1996). doi:10.1007/978-3-642-05221-7

[5] R. Courant, D. Hilbert: *Methods of mathematical physics volume II: partial differential equations*, 2nd edition. Wiley-VCH Verlag GmbH, Weinheim (1962). ISBN-13: 978-0-471-50439-9

[6] L.C. Evans: Partial differential equations, 2nd edition, American Mathematical Society, Rhode Island (2010). ISBN:978-0821849743

[7] O.C. Zienkiewicz, R.L. Taylor, J.Z. Zhu: *The finite element method: Its Basis and Fundamentals*, 7th edition, Elsevier Butterworth-Heinemann, Oxford (2013). doi:10.1016/B978-1-85617-633-0.00019-8

[8] W. Bangerth, R. Rannacher: *Adaptive finite element methods for differential equations*, Lectures in Mathematics, Springer Basel AG, Basel (2003). doi:10.1007/978-3-0348-7605-6

[9] M.J. Berger, J. Oliger: *Adaptive mesh refinement for hyperbolic partial differential equations*, Journal of Computational Physics, 53(3), pp. 484–512 (1984). doi:10.1016/0021-9991(84)90073-1

[10] R. Verfürth: *A posteriori error estimation and adaptive mesh-refinement techniques*, Journal of Computational and Applied Mathematics, 50(1), pp. 67–83 (1994). doi:10.1016/0377-0427(94)90290-9

[11] K.E. Atkinson: An introduction to numerical analysis,2nd edition, John Wiley & Sons, New York (1989). ISBN:978-0-471-62489-9

[12] R.L. Burden, J.D. Faires: Numerical analysis, 10th edition, Cengage Learning, Boston (2015). ISBN:978-0-538-73351-9

[13] I.E. Lagaris, A.C. Likas, D.I. Fotiadis: *Artificial neural networks for solving ordinary and partial differential equations*, IEEE Transactions on Neural Networks, 9.5, pp. 987–1000 (1998). doi:10.1109/72.712178

[14] I.E. Lagaris, A.C. Likas, D.G. Papageorgiou: *Neural-network methods for boundary value problems with irregular boundaries*, IEEE Transactions on Neural Networks, 11.5, pp. 1041–1049 (2000). doi:10.1109/72.870037

[15] M.L. Piscopo, M. Spannowsky, P. Waite: *Solving differential equations with neural networks: Applications to the calculation of cosmological phase transitions*, Physical Review D, 100.1, pp. 016002 (2019). doi:10.1103/PhysRevD.100.016002

[16] R.S.T. Lee: Artificial intelligence in daily life, Springer Nature Singapore Pte Ltd., Singapore (2020). ISBN:978-981-15-7694-2

[17] M. Koch: *Artificial intelligence is becoming natural*, Cell, 173.3, pp. 531–533 (2018).

[18] T.M. Brill, L. Munoz, R.J. Miller: *Siri, Alexa, and other digital assistants: a study of customer satisfaction with artificial intelligence applications*, Journal of Marketing Management, 35.15-16, pp. 1401–1436 (2019). doi:10.1080/0267257X.2019.1687571

[19] V.T. Minh, R. Khanna: *Application of artificial intelligence in smart kitchen*, International Journal of Innovative Technology & Interdisciplinary Sciences, 1.1, pp. 1–8 (2019). doi:10.15157/IJITIS.2018.1.1.1-8

[20] N. Goksel-Canbek, M.E.. Mutlu: *On the track of artificial intelligence: learning with intelligent personal assistants*, Journal of Human Sciences, 13.1, pp. 592–601 (2016).

[21] H. Chung, M. Iorga, J. Voas,S. Lee: *Alexa, can I trust you?*, IEEE Computer, 50.9, pp. 100–104 (2017). doi:10.1109/MC.2017.3571053

[22] N. Yadav, A. Yadav, M. Kumar: *An introduction to neural network methods for differential equations*, SpringerBriefs in Applied Sciences and Technology, Heidelberg, Berlin (2015). doi:10.1007/978-94-017-9816-7

[23] Y. Wu, J. Feng: *Development and application of artificial neural network*, Wireless Personal Communications, 102, pp. 1645–1656 (2018). doi:10.1007/s11277-017-5224-x

[24] B. Müller, J. Reinhardt, M.T. Strickland: *Neural networks: an introduction*, 2nd edition, Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg (1995). ISBN:3-540-60207-0

[25] K.L. Priddy, P.E. Keller: *Artificial neural networks: an introduction*, SPIE press, Bellingham (2005). ISBN:0-8194-5987-9

[26] W.S. McCulloch, W. Pitts: *A logical calculus of the ideas immanent in nervous activity*, Bulletin of Mathematical Biophysics, 5, pp. 115–133 (1943). doi:10.1007/BF02478259

[27] T.J. Sejnowski: *The book of hebb*, Neuron, 24.4, pp. 773–776 (1999).

[28] D.O. Hebb: *The organization of behaviour: a neuropsychological theory*, John Wiley & Sons, New York (1949). doi:10.2307/1418888

[29] B. Widrow, M.E. Hoff: *Adaptive switching circuits*, Stanford Electronic Laboratory, Technical Report No. 1553-1, (1960).

[30] F. Rosenblatt: *The Perceptron: a probabilistic model for information storage and organization in the brain*, Psychological Review, 65.6, pp. 386–408 (1958). doi:10.1037/h0042519

[31] M.L. Minsky, S.A. Papert: *Perceptrons: an introduction to computational geometry*, The MIT Press, Cambridge (Massachusetts) (1969). ISBN:978-0-262-53477-2

[32] J.J. Hopfield: *Neurons with graded response have collective computational properties like those of two-state neurons*, Proceedings of the National Academy of Sciences, 81.10, pp. 3088–3092 (1984). doi:10.1073/pnas.81.10.3088

[33] M.M. Flood: *The traveling-salesman problem*, Operations Research, 4.1, pp. 61–75 (1956). doi:10.1287/opre.4.1.61

[34] J.J. Hopfield, D.W. Tank: *"Neural" computation of decisions in optimization problems*, Biological Cybernetics, 52, pp. 141–152 (1985). doi:10.1007/BF00339943

[35] D.E. Rumelhart, G.E. Hinton, R.J. Williams: *Learning representations by back-propagating errors*, Nature, 323, pp. 533–536 (1986). doi:10.1038/323533a0

[36] G. Cybenko: *Approximation by superpositions of a sigmoidal function*, Mathematics of Control, Signals, and Systems, 2, pp. 303–314 (1989). doi:10.1007/BF02551274

[37] K. Hornik: *Approximation capabilities of multilayer feedforward networks*, Neural Networks, 4.2, pp. 251–257 (1991). doi:10.1016/0893-6080(91)90009-T

[38] G.E. Hinton, S. Osindero, Y.W. Teh: *A fast learning algorithm for deep belief nets*, Neural Computation, 18.7, pp. 1527–1554 (2006). doi:10.1162/neco.2006.18.7.1527

[39] S. Leijnen, F. van Veen: *The neural network zoo*, MDPI Proceedings, 47.1, pp. 1–6 (2020). doi:10.3390/proceedings2020047009

[40] K. Suzuki (Ed.): *Artificial neural networks: architectures and applications*, InTech, Rijeka (2013). doi:10.5772/3409

[41] B. Mehlig: *Machine learning with neural networks*, arXiv:1901.05639, (2021).

[42] S. Sharma, S. Sharma, A. Athaiya: *Activation functions in neural networks*, International Journal of Engineering Applied Sciences and Technology, 4.12, pp. 310–316 (2020).

[43] S.I. Amari: *Backpropagation and stochastic gradient descent method*, Neurocomputing, 5.4, pp. 185–196 (1993). doi:10.1016/0925-2312(93)90006-O

[44] D.P. Kingma, J. Ba: *Adam: a method for stochastic optimization*, arXiv:1412.6980, (2017).

[45] S.S. Rao: *Engineering Optimization: Theory and Practice*, 5th edition, John Wiley & Sons, Hoboken (2019). ISBN:978-1119454717

[46] M. Al-Baali, E. Spedicato, F. Maggioni: *Broyden's quasi-Newton methods for a nonlinear system of equations and unconstrained optimization: a review and open problems*, Optimization Methods & Software, 29.5, pp. 937–954 (2014). doi:10.1080/10556788.2013.856909

[47] T. Mikolov, G. Zweig: *Context dependent recurrent neural network language model*, 2012 IEEE Spoken Language Technology Workshop (SLT), pp. 234–239 (2012). doi:10.1109/SLT.2012.6424228

[48] J.A. Leonard, M.A. Kramer: *Radial basis function networks for classifying process faults*, IEEE Control Systems Magazine, 11.3, pp. 31–38 (1991). doi:10.1109/37.75576

[49] I. Rocco, R. Arandjelović, J. Sivic: *Convolutional neural network architecture for geometric matching*, Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 6148–6157 (2017).

[50] W. Rawat, Z. Wang: *Deep convolutional neural networks for image classification: a comprehensive review*, Neural Computation, 29.9, pp. 2352–2449 (2017). doi:10.1162/neco_a_00990

[51] H. Lee, I.S. Kang: *Neural algorithm for solving differential equations*, Journal of Computational Physics, 91.1, pp. 110–131 (1990). doi:10.1016/0021-9991(90)90007-N

[52] J.J. Hopfield: *Neural networks and physical systems with emergent collective abilities*, Proceedings of the National Academy of Sciences, 79.8, pp. 2554–2558 (1982). doi:10.1073/pnas.79.8.2554

[53] A.J. Meade Jr, A.A. Fernandez: *The numerical solution of linear ordinary differential equations by feedforward neural networks*, Mathematical and Computer Modelling, 19.12, pp. 1–25 (1994). doi:10.1016/0895-7177(94)90095-7

[54] A.J. Meade Jr, A.A. Fernandez: *Solution of nonlinear ordinary differential equations by feedforward neural networks*, Mathematical and Computer Modelling, 20.9, pp. 19–44 (1994). doi:10.1016/0895-7177(94)00160-X

[55] J.C. Butcher, G. Wanner: *Runge-Kutta methods: some historical notes*, Applied Numerical Mathematics, 22, pp. 113–151 (1996). doi:10.1016/S0168-9274(96)00048-7

[56] P.L. Lagari, L.H. Tsoukalas, S. Safarkhani and I.E. Lagaris: *Systematic construction of neural forms for solving partial differential equations inside rectangular domains, subject to initial, boundary and interface conditions*, International Journal on Artificial Intelligence Tools, 29.5, pp. 1–10 (2020). doi:10.1142/S0218213020500098

[57] S. Mall, and S. Chakraverty: *Chebychev neural network based model for solving Lane-Emden type equations*, Applied Mathematics and Computation, 247, pp. 100–114 (2014). doi:10.1016/j.amc.2014.08.085

[58] S. Mall, and S. Chakraverty: *Application of Legendre neural network for solving ordinary differential equations*, Applied Soft Computing, 43, pp. 347–356 (2016). doi:10.1016/j.asoc.2015.10.069

[59] I.T. Famelis, V. Kaloutsa: *Parameterized neural network training for the solution of a class of stiff initial value systems*, Neural Computing and Applications, 33, pp. 3363—3370 (2021). doi:10.1007/s00521-020-05201-1

[60] T. Schneidereit, M. Breuß: *Solving ordinary differential equations using artificial neural networks - a study on the solution variance*, Proceedings of the Conference Algoritmy, pp. 21–30 (2020).

[61] C. Flamant, P. Protopapas, D. Sondak: *Solving differential equations using neural network solution bundles*, arXiv:2006.14372, (2020).

[62] T. Schneidereit, M. Breuß: *Polynomial neural forms using feedforward neural networks for solving differential equations*, Artificial Intelligence and Soft Computing, ICAISC 2021. Lecture Notes in Computer Science, 12854, pp. 236-245–30 (2021). doi:10.1007/978-3-030-87986-0_21

[63] Y. Shirvany, M. Hayati, R. Moradian: *Multilayer perceptron neural networks with novel unsupervised training method for numerical solution of the partial differential equations*, Applied Soft Computing, 9.1, pp. 20–29 (2009). doI:10.1016/j.asoc.2008.02.003

[64] M. Raissi, P. Perdikaris, G.E. Karniadakis: *Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations*, Journal of Computational Physics, 378, pp. 686–707 (2019). doi:10.1016/j.jcp.2018.10.045

[65] A.D. Jagtap, E. Kharazmi, G.E. Karniadakis: *Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems*, Computer Methods in Applied Mechanics and Engineering, 365, pp. 113028 (2020). doi:10.1016/j.cma.2020.113028

[66] R.T.Q. Chen, Y. Rubanova, J. Bettencourt, D. Duvenaud: *Neural ordinary differential equations*, arXiv:1806.07366, (2018).

[67] L. Ruthotto, E. Haber: *Deep neural networks motivated by partial differential equations*, Journal of Mathematical Imaging and Vision, 62, pp. 352-364 (2020). doi:10.1007/s10851-019-00903-1

[68] Z. Long, Y. Lu, X. Ma, B. Dong: *PDE-Net: learning PDEs from Data*, Proceedings of the 35th International Conference on Machine Learning, 80, pp. 3208–3216 (2018).

[69] M. Lin, Q. Chen, S. Yan: *Network in Network*, arXiv:1312.4400, (2013).

[70] Y.J. Wang, C.T. Lin: *Runge-Kutta neural network for identification of dynamical systems in high accuracy*, IEEE Transactions on Neural Networks, 9.2, pp. 294–307 (1998). doi:10.1109/72.661124

[71] K. Rudd, S. Ferrari: *A constrained integration (CINT) approach to solving partial differential equations using artificial neural networks*, Neurocomputing, 155, pp. 277–285 (2015). doi:10.1016/j.neucom.2014.11.058

[72] V. Thomée: *Galerkin finite element methods for parabolic problems*, 2nd edition, Springer-Verlaf Berlin Heidelberg, Berlin, Heidelberg (2007). ISBN:3-540-33121-2

[73] T. Alt, K. Schrader, M. Augustin, P. Peter, J. Weickert: *Connections between numerical algorithms for PDEs and neural networks*, arXiv:2107.14742, (2021).

[74] J. Xie, L. Xu, E. Chen: *Image denoising and inpainting with deep neural networks*, Advances in Neural Information Processing, pp. 341–349 (2012).

[75] T. Falk, D. Mai, R. Bensch, Ö. Çiçek,A. Abdulkadir, et al.: *U-Net: deep learning for cell counting, detection and morphometry*, Nature Methods, 16, pp. 67–70 (2019). doi:10.1038/s41592-018-0261-2

[76] S. Alfonzetti: *A finite element mesh generator based on adaptive neural network*, IEEE Transactions on Magnetics, 34.5, pp. 3363–3366 (1998). doi:10.1109/20.717791

[77] J. Bohn, M. Feischl: *Recurrent neural networks as optimal mesh refinement strategies*, Computers and Mathematics with Applications, 97, pp. 61–76 (2021). doi:10.1016/j.camwa.2021.05.018

[78] L. Manevitz, A. Bitar, D. Givoli: *Neural network time series forecasting of finite-element mesh adaptation*, Neurocomputing, 63, pp. 447–463 (2005). doi:10.1016/j.neucom.2004.06.009

[79] M. Breuß, D. Dietrich: *Fuzzy numerical schemes for hyperbolic differential equations*, KI 2009: Advances in Artificial Intelligence, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 5803, pp. 419–426 (2009). doi:10.1007/978-3-642-04617-9_53

[80] C. Anitescu, E. Atroshchenko, N. Alajlan, T. Rabczuk: *Artificial neural network methods for the solution of second order boundary value problems*, Computers, Materials and Continua, 59.1, pp. 345–359 (2019). doi:10.32604/cmc.2019.06641

[81] T. Schneidereit, M. Breuß: *Computational characteristics of feedforward neural networks for solving a stiff differential equation*, Neural Computing and Applications, 34, pp. 7975–7989 (2022). doi:10.1007/s00521-022-06901-6

[82] T. Schneidereit, M. Breuß: *Collocation polynomial neural forms and domain fragmentation for initial value problems*, Neural Computing and Applications, 34, pp. 7141–7156 (2022). doi:10.1007/s00521-021-06860-4

[83] T. Schneidereit, M. Breuß: *Adaptive neural domain refinement for solving time-dependent differential equations*, Under review in: Advances in Continuous and Discrete Models, arXiv:2112.12517, (2021).

[84] D. Nguyen, B. Widrow: *Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights*, 1990 IJCNN International Joint Conference on Neural Networks, 3, pp. 21–26 (1990). doi:10.1109/IJCNN.1990.137819

[85] V.V. Phansalkar, and P.S. Sastry: *Analysis of the back-propagation algorithm with momentum*, IEEE Transactions on Neural Networks, 5.3, pp. 505–506 (1994). doi:10.1109/72.286925

[86] Y. Kaneda, Q. Zhao, Y. Liu, Y. Pei: *Strategies for determining effective step size of the backpropagation algorithm for on-line learning*, 7th International Conference of Soft Computing and Pattern Recognition (SoCPaR), pp. 155–160 (2015). doi:10.1109/SOCPAR.2015.7492800

[87] J. Nocedal, S.J. Wright: *Numerical optimization*, 2nd edition, Springer Science+Business Media, New York (2006). ISBN:978-0387-30303-1

[88] A. Frommer: *Numerische Methoden der nichtlinearen Optimierung (engl. Numerical methods of nonlinear optimisation*, lecture notes, Wuppertal (2004). LINK: `http://www2.math.uni-wuppertal.de/%7Efrommer/manuscripts/NichtLinOpt.pdf` (last visited 01.02.2023)

[89] R. Reemtsen: *Einführung in die nichtlineare Optimierung (engl. Introduction to nonlinear optimisation*, lecture notes, Hagen. LINK: `https://www.fernuni-hagen.de/mi/studium/module/pdf/Leseprobe-komplett_01221.pdf` (last visited 01.02.2023)

[90] X. Glorot, Y. Bengio: *Understanding the difficulty of training deep feedforward neural networks*, Proceedings of the 13$^{\text{th}}$ International Conference on Artificial Intelligence and Statistics, 9, pp. 249–256 (2010).

[91] W.F. Schmidt, M.A. Kraaijveld, R.P.W. Duin: *Feed forward neural networks with random weights*, International Conference on Pattern Recognition, IEEE Computer Society Press, 9, pp. 1–4 (1992).

[92] R. Martí, J.A. Lozano, A. Mendiburu, L. Hernando: *Multi-start methods*, In: Martí, R., Pardalos, P., Resende, M. (eds) Handbook of Heuristics, Springer Nature, (1992). doi:10.1007/978-3-319-07124-4_1

[93] Y. Shang, B.W. Wah: *Global optimization for neural network training*, Computer, 29.3, pp. 45–54 (1996). doi:10.1109/2.485892

[94] R.G.. Regis, C.A. Shoemaker: *A stochastic radial basis function method for the global optimization of expensive functions*, INFORMS Journal on Computing, 19.4, pp. 497–509 (2007). doi:10.1287/ijoc.1060.0182

[95] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov: *Dropout: a simple way to prevent neural networks from overfitting*, Journal of Machine Learning Research, 15, pp. 1929-1958 (2014).

[96] D.M. Hawkins: *The problem of overfitting*, Journal of Chemical Information and Computer Sciences, 44.1, pp. 1–12 (2004). doi:10.1021/ci0342472

[97] X. Ying: *An overview of overfitting and its solutions*, Journal of Physics: Conference Series, 1168.2, pp. 022022 (2019). doi:10.1088/1742-6596/1168/2/022022

[98] G.G. Dahlquist: *G-stability is equivalent to A-stability*, BIT Numerical Mathematics, 18.4, pp. 384–401 (1978).

[99] P. Nakkiran, G. Kaplun, Y. Bansal, T. Yang, B. Barak, et al.: *Deep double descent: where bigger models and more data hurt*, Journal of Statistical Mechanics: Theory and Experiment, 2021.12, pp. 124003 (2021).

[100] L. Prechelt: *Automatic early stopping using cross validation: quantifying the criteria*, Neural Networks, 11.4, pp. 761–767 (1998). doi:10.1016/S0893-6080(98)00010-0

[101] L. Prechelt: *Early Stopping — But When?*, Neural Networks: Tricks of the Trade, Lecture Notes in Computer Science, 7700, pp. 53–54 (2012). doi:10.1007/978-3-642-35289-8_5

[102] M. Fernández-Redondo, C. Hernández-Espinosa: *Weight initialization methods for multilayer feedforward*, ESANN, pp. 119–124 (2001).

[103] D.F. Griffiths, D.J. Higham: *Numerical methods for ordinary differential equations*, Springer-Verlag London Limited 2010, London (2010). doi:10.1007/978-0-85729-148-6

[104] E. Hairer, C. Lubich, G. Wanner: *Geometric numerical integration: structure-preserving algorithms for ordinary differential equations*, 2nd edition, Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg (2006). ISBN:3-540-30663-3

[105] E. Celledoni, R.I. McLachlan, D.I. McLaren, B. Owren et al.: *Energy-preserving Runge-Kutta methods*, ESAIM: Mathematical Modelling and Numerical Analysis, 43, pp. 645–649 (2009). doi:10.1051/m2an/2009020

[106] M. Woźniak, D. Połap: *Hybrid neuro-heuristic methodology for simulation and control of dynamic systems over time interval*, Neural Networks, 93, pp. 45–56 (2017). doi:10.1016/j.neunet.2017.04.013

[107] M.C. Anisiu: *Lotka, Volterra and their model*, Didáctica Mathematica, 32, pp. 9–17 (2014).